UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Loïc Petit

Thèse dirigée par **Claudia Lucia Roncancio**
et codirigée par **Cyril Labbé**

préparée au sein **d'Orange Labs,**
du Laboratoire d'Informatique de Grenoble,
et de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Gestion de flux de données pour l'observation de systèmes

Application à la gestion du réseau domestique

Thèse soutenue publiquement le **10 décembre 2012**,
devant le jury composé de :

Jean-Marc Petit
Professeur, INSA Lyon, Rapporteur
Chantal Taconet
Maître de conférences (HDR), Télécom SudParis, Rapporteur
Laurence Duchien
Professeur, INRIA, Université de Lille 1, Examinateur
François-Gaël Ottogalli
Responsable de Recherche, Orange Labs, Examinateur
Claudia Lucia Roncancio
Professeur, Grenoble INP, Directrice de thèse
Cyril Labbé
Maître de conférences (HDR), Université Joseph Fourier, Co-Directeur de thèse

« How about y'all take a little break?
I got some fine apple juice waitin' for ya!
I can't thank you enough for this help. »

Applejack

Remerciements

Mes remerciements vont avant tout en direction de *Claudia Roncancio* et *Cyril Labbé*, mes parents académiques. Ça fait cinq ans maintenant qu'ils me supportent et croyez-moi, ce n'est pas facile tous les jours. Vous m'avez soutenu depuis le début de ma carrière scientifique et cette thèse était un vrai cadeau de votre part.

Je voudrais remercier *François-Gaël Ottogalli* qui m'a encadré chez Orange. J'ai énormément appris avec toi et tu étais la personne dont j'avais besoin pour structurer l'amas difforme que constitue mon cerveau. Merci à *Vincent Olive* de m'avoir fait confiance et d'avoir soulevé ciel et terre pour m'avoir permis de faire cette thèse.

Merci en particulier à *Jean-Marc Petit* et à *Chantal Taconet* de m'avoir fait l'honneur de juger ce manuscrit conséquent avec un timing très serré. Et merci à *Laurence Duchien* d'avoir accepté d'être examinatrice de ce travail.

Un grand merci à l'Orange Team pour votre accueil pendant ces trois ans ! À ceux qui m'ont particulièrement supporté après 18h : *Stéphane*, *Charbel* et *Rémi*$_D$. Aux thésards qui sont passés dans nos rangs pour ces bons moments : *Maxime*, *Rémi*$_M$ et *Érick*. À ceux avec qui ont lutté avec moi pour mettre en œuvre mes idées : *Julien* avant tout, mais aussi *Serge* et *Sébastien*. Et à tous ceux avec qui j'ai partagé de très bons moments : *Xavier*, *Nordine*, *Matthieu*, *Jacques*, *Marc*$_D$, *Marc*$_E$, *JD*, *Anne*, et j'en passe évidemment...

Un grand merci aussi à l'équipe SIGMA, que ce soit les permanents ou les doctorants ! Ces personnes sont tellement conviviales qu'une réunion d'équipe est un vrai plaisir avec eux. Merci de votre accueil parmi vos rangs.

Je n'aurais pas pu aller jusqu'ici sans le soutien de mes amis. Je ne sais pas si on a uniquement ceux que l'on mérite, mais vous êtes bien trop géniaux pour ma petite personne. Donc, merci aux anciens imag (*Jilouc*, *ToTo*, *Ugh*, *Gugli*, *Gnesson*, *Noémie*, *Laurent*...), à mon professeur fou *Alban*, à mon bassiste à moi *Jérémie*, à mon RH misanthrope *Jéjé*, à la famille *Paradisio*, aux cubeurs, et à tous ceux que je n'ai pas pu citer par manque de place.

Merci au jus de pomme, mon carburant ; à la musique de m'avoir fait tenir le coup ; et aux poneys pour avoir fourni les citations des chapitres de ce manuscrit.

Un immense merci à ma famille pour leur soutien sans faille. En particulier à mon père, ma mère et mon frère pour leur amour sans condition et leur confiance en moi. On ne choisit certes pas sa famille, mais je ne vous changerais pour rien au monde.

Je dédie enfin cette thèse à deux personnes qui nous ont malheureusement quittés cette année. À mon grand-père, *Papy Gérard*, qui m'a appris très tôt ce que voulait dire *passion*. Et à mon arrière-grand-mère, *Mamie Brun*, qui sera toujours pour moi ma définition du mot *gentillesse*. J'aurais tant voulu vous dire : regardez, j'ai réussi.

Table des matières

« Doing things didn't work, not doing things didn't work, and I couldn't predict the future either, so I only had one other choice. ***Monitor everything****. »*

Twilight Sparkle

1 Contexte et objectifs de la thèse

Ce premier chapitre présente brièvement le contexte de cette thèse en section 1.1. Puis, nous détaillons les problématiques de recherche en section 1.2. Nous présentons les principales contribution de ce travail en section 1.3. Ensuite, nous présentons notre cadre applicatif en section 1.4 et enfin, nous détaillons le plan de ce manuscrit en section 1.5.

1.1 De l'importance de l'observation des systèmes

L'informatique a évolué de façon drastique au cours de ces dernières années. Grâce à l'amélioration des technologies de la microélectronique, du réseau, et des techniques logicielles, il est désormais possible de concevoir des systèmes complexes et repartis tels que :

– des réseaux de capteurs [Akyildiz 2002, Szewczyk 2004]. Composé de nombreux microdispositifs autonomes, il est capable de transmettre sans-fil des informations sur des mesures physiques. Ensemble, les capteurs permettent de créer des systèmes de surveillance dans des applications telles que l'agriculture de précision [Jurdak 2008].

– des environnements domestiques intelligents [Harper 2003, Chan 2008, Coyle 2006]. Grâce aux technologies sans fil ou courants-porteur, des dispositifs sont capables de communiquer afin de fournir des services de haut niveau. De tels services peuvent être multimédia comme le partage de flux vidéos [Kang 2005] ou la visiophonie [Vilei 2006]. Ils peuvent être orientés sur l'amélioration du confort de vie de l'usager comme la gestion automatique des lumières ou encore l'assistance aux personnes malades ou handicapées [Korhonen 2003].

Ces systèmes reposent sur des dispositifs qui interagissent via un réseau pour fournir un ou des services de haut niveau. Ces interactions sont nombreuses et potentiellement volatiles. Afin de mieux comprendre ces systèmes, il est nécessaire de les observer.

L'**observation** d'un système est un processus en charge de collecter, traiter, et éventuellement archiver les données d'un système en fonctionnement pour en vérifier son bon déroulement. Cette surveillance peut être faite en temps réel afin d'être réactif aux événements importants, ou par analyse a posteriori sur l'ensemble des données collectées. Ainsi, lorsqu'un problème surgit sur un système, ce procédé est au cœur de sa résolution. En effet, le traitement en temps réel peut détecter l'anomalie et peut transmettre l'information à qui est capable d'y remédier (l'utilisateur ou un système tiers). Si les données sont archivées, alors il devient possible de retracer l'origine du problème par le parcours des données collectées afin d'établir un diagnostic.

Dans le domaine de l'administration des systèmes, le terme de *supervision* est souvent évoqué. Ce processus permet de surveiller le système et d'agir en conséquence en modifiant par exemple la configuration de chacun des dispositifs et services. L'observation est une base précieuse d'informations pour l'aide à la gestion du système, car cela permet à l'administrateur d'avoir une vue détaillée du fonctionnement de son système.

De plus, la construction de services de haut niveau a révélé plusieurs problèmes dans la conception d'intergiciels ubiquitaires [Geihs 2001]. Un des aspects critiques de ce domaine est la capacité à se reconfigurer à la demande. Afin de pouvoir s'adapter aux différents environnements, il est nécessaire que l'intergiciel soit conscient d'un ensemble de paramètres qui lui permette de prendre la bonne décision. Ces paramètres sont fournis par un système d'observation.

Il existe de nombreux produits commerciaux ou académiques pour permettre la supervision ou l'observation. Ces solutions permettent la surveillance d'infrastructures réseaux (d'un point de vue du réseau [Zabbix 2012] ou applicatif [Zoho Corp. 2012]), la gestion de processus opérationnels d'entreprises [Systar 2012] ou encore l'administration d'un parc de dispositifs [Case 1990]. Toutefois, le système d'observation doit être capable de répondre à la diversité et la grandissante complexité des systèmes informatiques. La section suivante présente plus en détail la problématique traitée dans cette thèse.

1.2 Présentation des problématiques

Dans cette thèse, nous abordons l'observation de systèmes par l'observation de *ses données*. Dans la section 1.2.1, nous présentons les systèmes que nous souhaitons

observer. La section 1.2.2 détaille les caractéristiques des données de ces systèmes. Enfin, la section 1.2.3 décrit les besoins du système d'observation.

1.2.1 Systèmes observés

Nous considérons qu'un **système** est défini comme un ensemble d'équipements ou d'applications connectés entre eux. Ses composants peuvent être hétérogènes et en grande quantité. Par exemple, un système peut être un réseau de capteurs et les mesures relevées font partie des données (en plus des données descriptives). Dans cette thèse, nous supposons que nous pouvons dialoguer d'une manière ou d'une autre avec le système observé pour récolter ses données. La figure 1.1 présente un exemple de système exposant les sources de données hétérogènes Sn. Ces données sont interrogées par des utilisateurs par le système d'observation que nous souhaitons concevoir.

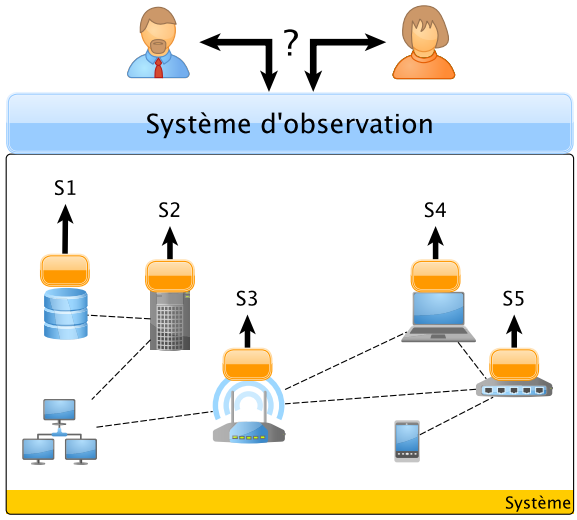


FIGURE 1.1 – Représentation d'un système observé par deux utilisateurs

1.2.2 Hétérogénéité des données des systèmes observés

Nous nous focalisons sur l'observation des données du système. Dans notre contexte, nous identifions deux types d'hétérogénéité de données : l'hétérogénéité structurelle des données, et l'hétérogénéité en terme de dynamisme.

Descriptions hétérogènes des systèmes

Tout système peut être décrit par un **schéma conceptuel** fixant une sémantique. Ce schéma s'appuie sur un **modèle de données** théorique tels que : le modèle hiérarchique, le modèle réseaux sémantiques, le modèle relationnel ou encore le modèle objet. Ainsi, un système observé peut s'abstraire par un schéma qui sera différent du choisi pour un autre système. L'observation doit pouvoir s'adapter à chacun d'eux.

De plus, le système observé expose un ensemble de sources de données hétérogènes, car chacune possède son propre schéma. Il est nécessaire d'**intégrer** ces données pour que celles-ci puissent être manipulées par le schéma du système.

Évolution des données

Le système observé évolue au fur et à mesure du temps. Ce dynamisme introduit le problème de la gestion de grandes **catégories de données** : les données **persistantes** et les données **temps réel**. Les données persistantes sont à évolution lente et les données temps réel sont des relevés instantanés et très volatiles indiquant l'état du système (sous forme de flux de données).

Néanmoins, chacune de ces catégories possède des modèles de données et des traitements propres. Or, ces données proviennent toutes du même système, qu'elles soient temps réel ou persistantes. Il existe des liens entre ces deux catégories de données. L'expérience montre que l'archivage d'un flux permet d'extraire de nouvelles connaissances par analyse de son historique. À l'inverse, une donnée stockée peut subir des modifications déclenchant la production d'événements, ce qui est assimilable à un flux de données.

Ces processus de traitement de données sont assimilables à des requêtes d'interrogation que pose l'utilisateur sur l'ensemble des données. Il existe différent **modes d'interrogations** (ou requêtes) qui, d'une certaine manière, reflètent l'hétérogénéité de l'évolution des données.

– **Interrogation instantanée** : C'est la manière usuelle d'interroger dans les applications de gestion de base de données. L'utilisateur pose une question sur un ensemble de données considérées figées, du moins le temps du calcul de la réponse. Le système fournit une réponse représentative d'un état à un instant donné. Par exemple : « *quel est l'ensemble actuel des équipements actifs de mon système* ». La réponse à cette requête pourrait être « *à cet instant, les équipements Box, PC1 et STB sont connectés et actifs* ». La mention « *à cet instant* » est très importante, car si un nouvel équipement arrive dans le système, une nouvelle évaluation donnerait une réponse différente. Ainsi, cette interrogation correspond à la consultation ponctuelle de l'ensemble des données disponibles.

– **Interrogation continue** : Les données sont considérées en constante évolution. L'utilisateur obtient ainsi une réponse qui évolue au cours du temps, sous forme de flux ou de mise à jour d'état. La durée de vie de ce processus peut être indéfini. Un exemple pouvant être : « *toutes les minutes, le flux de la charge moyenne sur une minute du processeur du PC1* ». La réponse forme un flux continu d'information qui, toutes les minutes, reporte une nouvelle valeur moyenne pour ce capteur. La formation de processus de collecte ou de formation d'alerte suit ce type d'interrogation.

Il est important de noter que ces deux méthodes d'interrogation peuvent se combiner pour exécuter une requête hybride. Par exemple, il est possible d'effectuer un appel à une interrogation instantanée à l'intérieur d'un processus continu. De façon similaire, l'appel régulier d'une interrogation instantanée forme une réponse continue. Il est nécessaire que le système d'observation soit capable de manipuler naturellement ces deux types d'interrogation pour refléter correctement le dynamisme du système observé.

1.2.3 Caractéristiques du système d'observation

La capacité à s'adapter à l'application finale est un point important de la mise en œuvre d'un système d'observation. Plus la portée de l'observation est générique plus ce critère est critique. Ainsi, il est important que le nombre et la complexité des procédures nécessaires à l'adaptation au système visé soient faibles. Car, si un système est complet, mais nécessite une adaptation longue et complexe, il devient difficile à appliquer.

Langage de manipulation de données

Afin de pouvoir intégrer et interroger les données, le système d'observation doit se doter de capacités de traitements des données. L'expression des traitements possibles se fait à travers un **langage**. Pour plus de flexibilité, nous souhaitons une solution qui soit la plus indépendante par rapport aux algorithmes de traitement de données, ainsi les **paradigmes** déclaratifs sont privilégiés. Le langage utilisé peut avoir un **pouvoir d'expression** limité. Il est important d'être capable d'énumérer ce qui est possible d'exprimer. Les classes de logiques ou les équivalences à d'autres langages permettent de caractériser ces limitations.

Pour que le système d'observation perturbe le moins possible le système observé, il est nécessaire que l'évaluation des requêtes soit **performante**. De plus, ce critère améliore la qualité des réponses en terme de latence de traitement. Le critère se mesure sur la capacité à traiter la charge d'un système en terme de nombre de sources ou en terme de débit supporté.

Adaptabilité aux besoins

Afin de pouvoir s'adapter aux besoins des utilisateurs, il est nécessaire que le système d'observation soit capable d'utiliser des **routines spécifiques** de traitement de données fournies par l'utilisateur. Cette extensibilité permet aussi bien l'intégration de tous types de besoins que l'amélioration des performances de traitements récurrents.

Enfin, il existe plusieurs **perspectives** possibles à un système. En fonction de son expérience, de ses connaissances ou de ses intérêts, chaque utilisateur perçoit le système à sa manière. Cet angle de vue définit les données surveillées, en établissant des préférences sur ce qui est observé. Mais cela peut influencer aussi la représentation du système observé. Il est nécessaire que le système d'observation s'adapte aux perspectives dans lesquelles se placent les utilisateurs.

En conclusion, la problématique de cette thèse est d'observer un système générique en étant capable de s'adapter à sa dynamique, à son hétérogénéité, tout en restant ouvert aux traitements spécifiques d'une application.

1.3 Contributions de cette thèse

Nous proposons une approche dirigée par les données pour l'observation de systèmes. Dans le domaine de la gestion des flux de données, nous proposons de créer un langage algébrique pour interroger les données persistantes et temps réel. À l'aide

de ce support théorique, nous développons un intergiciel capable de gérer les données issues de l'observation. Cette section présente brièvement ces contributions.

1.3.1 Un langage unique d'interrogation

La gestion de flux de données permet de manipuler les données temps réel. Cette approche permet une grande souplesse, car il est possible d'évaluer des requêtes complexes via un langage déclaratif. Toutefois, les langages existants manquent de fondations théoriques pour maîtriser ces données.

Nous avons spécifié *Astral*, une algèbre permettant d'interroger de manière unifiée les données sous forme de flux et de relations persistantes. Cette algèbre manipule les deux modes d'interrogation (continue et instantanée) sur des données persistantes ou temps réel. Les définitions d'opérateurs d'Astral sont dotées d'une sémantique précise. Ainsi, lors de l'expression d'une requête par cette algèbre, il n'existe qu'une interprétation du résultat, ce qui permet une gestion plus claire des données.

1.3.2 Un intergiciel extensible d'évaluation de requêtes

Il existe plusieurs intergiciels [Arasu 2004a, Sybase 2010] capables d'évaluer des requêtes continues sur les flux de données. Toutefois, leur implémentation n'est pas précise concernant les sémantiques d'évaluations de requête [Jain 2008, Botan 2010]. Or, il existe peu d'approches permettant de spécifier formellement la sémantique des opérateurs d'un évaluateur.

Nous proposons *Astronef*, un intergiciel capable d'évaluer des requêtes exprimées dans le langage algébrique que nous avons développé. Cette algèbre sert aussi pour la spécification des composants de l'intergiciel. En effet, chaque module implémentant un opérateur doit spécifier sa sémantique selon une ou plusieurs règles se basant sur des opérateurs algébriques. Ce principe couplé avec les notions architecturales de composants orientés services offre une grande flexibilité et permet aux utilisateurs de personnaliser au mieux cette solution générique à leurs besoins.

De plus, cet intergiciel possède un optimiseur pour construire des plans d'exécutions correspondants aux expressions de requêtes *Astral*. Notre approche est à base de règles ce qui permet une approche générale d'optimisation des requêtes. Il est ainsi possible d'appliquer une optimisation logique, puis physique des requêtes. Ces optimisations sont possibles avec les équivalences de requêtes découvertes avec *Astral*.

1.3.3 Une intégration des supports persistants

Dans la littérature, plusieurs rapprochements existent entre les supports persistants et la gestion de flux, de manière ad hoc ou implicite [Balazinska 2007, Reiss 2007]. Étant donné qu'*Astral* permet d'interroger de manière unifiée les données persistantes et temps réel, nous proposons l'extension *Asteroid* capable d'intégrer un SGBD à *Astronef*. Ainsi, l'utilisateur peut spécifier le schéma du système observé, ainsi que les requêtes permettant l'intégration des données des sources du système. L'utilisateur peut interroger les données persistantes et temps réel via l'algèbre et la requête sera évaluée de façon efficace.

1.3.4 Personnalisation des résultats

Enfin, afin de pouvoir gérer les différentes perspectives du système, nous avons introduit deux nouveaux opérateurs *Best* et *KBest*, capables de personnaliser les résultats des requêtes de notre système d'observation. L'utilisateur peut établir un profil de préférences qui lui est propre, ensuite, ces opérateurs adaptent les résultats en fonction de ce profil. Le premier, sélectionne les données préférées tandis que le second sélectionne les k meilleures. Ces opérateurs peuvent s'appliquer sur toute requête Astral ce qui rend cette approche très générique.

De plus, l'ajout de ces opérateurs permet de démontrer l'extensibilité de notre approche, autant au niveau de l'algèbre que de l'intergiciel.

1.4 Domaine d'application : le réseau local domestique

Cette thèse a été développée dans un milieu industriel chez *Orange Labs*. Les propositions de cette thèse sont génériques, mais grâce à ce contexte de travail, nous avons pu appliquer notre approche dans un domaine applicatif qui pose actuellement de nombreux problèmes : le réseau domestique (ou *Digital Home*).

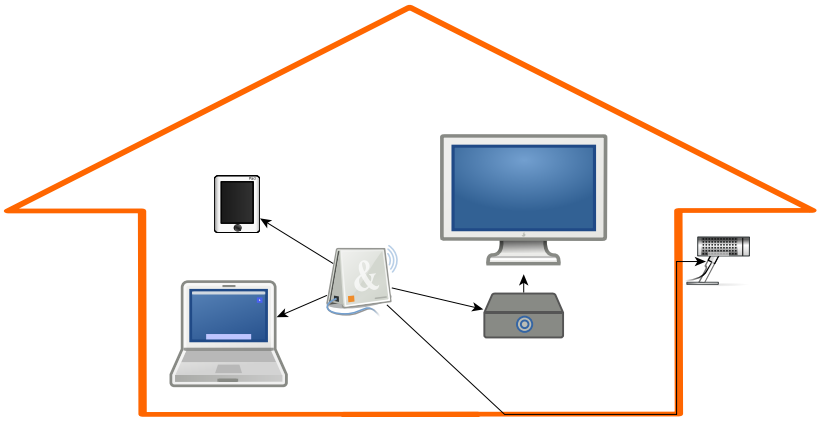


FIGURE 1.2 – Exemple de réseau domestique

Le réseau domestique est formé par l'ensemble des appareils se trouvent dans une maison. Ce réseau est déjà capable de fournir des services tels que le partage de contenus entre un disque dur en réseau (ou *NAS*) sur la télévision ou sur les ordinateurs. L'étape suivante du développement de cette approche est l'introduction de capteurs et d'actuateurs (domotique).

Les opérateurs télécoms qui fournissent des équipements et des services sont bien souvent en difficultés pour dépanner les utilisateurs. Une des raisons de ce problème est notamment le manque d'informations sur le système. Le service après-vente ne possède pas de moyen de connaître la topologie du réseau (utilisation de *wifi* ou courant porteur), la configuration de certains équipements et encore moins l'état de fonctionnement des équipements, des liens réseau ou des services.

Ce cadre applicatif a fourni les cas d'études et d'expérimentation à cette thèse. Ils réunissent les caractéristiques que nous avons introduites en 1.2.

– Le réseau domestique est composé de dispositifs hétérogènes.

– Chaque équipement fournit ses propres données sous des schémas que nous ne maîtrisons pas.

– Les paramètres de configuration et d'autres données accessibles via des services spécifiques sont des données persistantes. Il est possible de récupérer des données de métriques pour mesurer la santé du réseau.

– Des requêtes hybrides sont nécessaire pour former des alarmes basées sur des données persistantes (catalogue, agrégats historiques).

– Plusieurs types d'utilisateurs avec des métiers différents observent ce réseau.

Les travaux développés dans cette thèse contribuent à un système d'observation applicable sur tout système. Nous validerons notre approche par son application sur la compréhension du réseau local domestique. Ceci pourra être utilisé pour aider les diagnostics faits au service après-vente.

1.5 Plan de thèse

Ce manuscrit est découpé en cinq grandes parties :

1. **L'état de l'art**. Le chapitre 2 analyse les solutions actuelles capables de faire de l'observation de systèmes génériques. Ensuite, nous détaillons les travaux du domaine plus spécifique de la gestion des flux de données dans le chapitre 3.

2. **Modèle algébrique**. Le chapitre 4 présente les définitions d'*Astral*, notre algèbre de gestion de données. Le chapitre 5 analyse en détail l'expressivité offerte par *Astral*, en la comparant à l'état de l'art, puis en démontrant des propriétés non triviales d'équivalence de requêtes.

3. **Mise en œuvre et couplage relationnel**. Le chapitre 6 décrit *Astronef*, un intergiciel capable d'interpréter et d'évaluer une requête *Astral*. Nous détaillons dans le chapitre 7 l'extension *Asteroid* capable de manipuler un support persistant relationnel, complétant ainsi notre gestion de données pour l'observation.

4. **Expérimentations**. La mise en œuvre de notre solution d'observation dans le cadre du réseau local domestique est décrite dans le chapitre 8. Puis, nous présentons dans le chapitre 9 des éléments d'évaluation de performances afin de démontrer que notre optimiseur permet d'évaluer de manière efficace des requêtes continues et des requêtes couplées avec un SGBD.

5. **Gestion des préférences**. Le chapitre 10 présente une extension à notre contribution introduisant des opérateurs permettant de personnaliser les résultats en fonction de l'utilisateur. Nous présentons leur formalisation dans *Astral* ainsi que deux implémentations que nous intégrons à *Asteroid*. Des expérimentations permettent de comparer leurs performances.

Finalement, le chapitre 11 conclut ce travail et présente quelques perspectives de recherche.

Partie I

État de l'art

Dans nos recherches sur la gestion de flux de données pour l'observation des systèmes, nous avons été amenés à étudier de nombreux travaux. Sans vouloir être exhaustifs, nous consacrons ici un chapitre à diverses solutions orientées*observation et analyse de systèmes* sans que forcément elles intègrent des flux de données. Chacune des solutions présentées permet de répondre à une partie des problématiques. En complément, nous présentons un deuxième chapitre consacré à la *gestion de flux de données* sans que forcément elle soit intégrée à une solution d'observation des systèmes.

« – Honestly, Spike, don't you know better than to sneak up on ponies?
– Oh, sorry, but, um, well, isn't that what you're doing?
– No! I'm doing scientific research. »

Twilight Sparkle & Spike

2

Gestion de données pour l'observation

L'observation est un domaine très actif grâce à l'impact que cela peut avoir dans des situations concrètes (compréhension, diagnostic, autoconfiguration). En dehors des systèmes ad hoc, plusieurs solutions ont été développées pour gérer les données de systèmes. Ce chapitre présente quatre grandes approches que nous avons pu identifier comme intéressantes pour notre problématique.

La section 2.1 présente les systèmes d'administration, ceux-ci sont déployés depuis plusieurs années pour gérer des parcs de dispositifs à grande échelle et fournissent un grand nombre de données. Afin d'exploiter l'ensemble des données que nous pouvons extraire, nous détaillons la gestion de contexte en section 2.2. Ces systèmes fournissent à une application des informations de contexte pour que celle-ci s'adapte en conséquence. Puis afin d'extraire et d'analyser les données d'historiques collectées, nous développons l'approche par entrepôts de données en section 2.3, maintenant largement répandus. Enfin, afin de gérer les données temps réel, les systèmes de gestion de flux de données seront présentés en section 2.4. Ceux-ci permettent de manipuler des flux de données de manière déclarative. Le chapitre 3 est consacré entièrement à ce thème.

2.1 Systèmes d'administration

Depuis le début des années 80, grâce aux premières mises en réseau d'équipements, les systèmes d'administration permettent de gérer des parcs de ressources. Le principe est de surveiller et surtout contrôler un système afin qu'il satisfasse les demandes des utilisateurs et les contraintes du propriétaire [Sloman 1994]. Dans cette thèse, nous nous intéressons à l'observation, le contrôle (par rétroaction par exemple) est en dehors du périmètre de cette étude. Pour permettre la surveillance, chaque objet d'un système va représenter son état dans un modèle. Ces informations sont consultable via un protocole.

Cette section présente les systèmes d'administration déployés pour exploiter des parcs de dispositifs à grande échelle. Ces systèmes sont spécifiés au travers de divers consortiums ou forums. Les principaux acteurs sont : le *BroadBand Forum* (BBF) (porté par les opérateurs télécoms du monde de l'accès internet résidentiel), le *Forum Universal Plug'n'Play* (UPnP) (porté par l'électronique grand publique), ou encore *Distributed Management Task Force* (DMTF), l'*Institute of Electrical and Electronics Engineers* (IEEE) et l'*Internet Engineering Task Force* (IETF), organisations ouvertes où participent entreprises, laboratoires et indépendants. Ces ententes permettent la spécification des standards autant au niveau des protocoles de communications que sur la structure des données manipulées en terme de modèle ou de schéma.

Cette section présente d'abord la structure et la gestion des données issues des ressources administrées. Ensuite, nous analysons les fonctions fournies par les systèmes d'administration. Enfin, une synthèse est présentée à l'aide d'une grille d'analyse correspondant aux problématiques présentées en 1.2.

2.1.1 Gestion des données pour l'administration

La gestion des données dans les systèmes classiques d'administration est principalement fondée sur des gestionnaires « agents » [CCITT 1992] (voir fig. 2.1). Cette approche est celle utilisée de nos jours dans les protocoles d'administration tels que TR-069 [BroadBand Forum 2011a], UPnP Device Management [UPnP Forum 2012b], mais aussi sur des protocoles plus anciens tels que SNMP [Case 1990]. Le principe est qu'un module logiciel est présent sur les dispositifs devant être administrés. Celui-ci comporte un agent capable de maintenir une petite « base de données » sous une forme particulière représentant les données et états du système. Un gestionnaire est capable par la suite de transmettre les informations de l'agent à un système d'administration global. Ce dernier agrège ainsi l'ensemble des informations.

Les données fournies par les agents

Il existe plusieurs structures de données dans le cadre des systèmes d'administration. La structure la plus répandue reste la **structure hiérarchique**. La première apparition d'un tel modèle dans ce domaine remonte à la spécification de SNMP [Case 1990] qui décrit le concept de *Management Information Base* (MIB) [Rose 1990]. Une *MIB* est une base d'information où les données sont regroupées sous forme d'arbre. Chaque information possède un chemin unique (l'*object identifier*) décrit par une suite de chiffres.

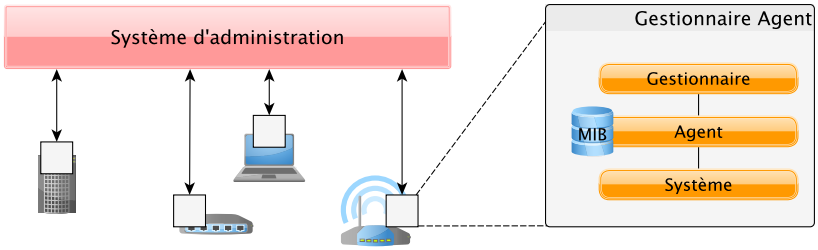


FIGURE 2.1 – Agent d'administration sur un dispositif

Des catalogues répertorient l'ensemble des *MIB* existantes (standardisées ou non) [1].

Les protocoles récents permettent aussi de manipuler des données hiérarchiques. Par exemple, le *BroadBand Forum* décrit dans le document technique *TR-106* [BroadBand Forum 2011b] son modèle de données en arbre. De même, le forum *UPnP* décrit dans son service de gestion de configuration (CMS) [UPnP Forum 2012a] un modèle très similaire.

Un exemple de ce type de structure de données est présenté en figure 2.2. Une donnée est définie de manière unique, tout comme dans une *MIB*, grâce à son chemin complet. Dans le vocabulaire du domaine de l'administration, cette donnée est appelée *paramètre*. Le *chemin* d'un paramètre est la concaténation des noms des nœuds qui le sépare de la racine, avec pour séparateur « / » dans UPnP ou « . » dans TR-069. L'implémentation de l'agent permet de créer et de remplir cette base d'information.

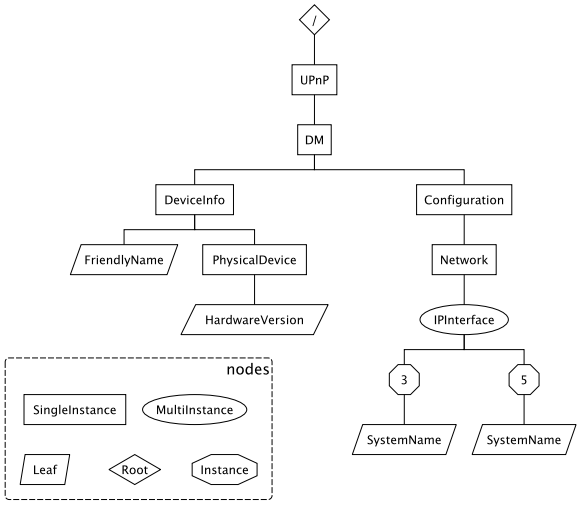


FIGURE 2.2 – Structure hiérarchique du modèle de données d'UPnP-DM

Toutefois, tous les modèles ne sont pas structurés sous forme de hiérarchie. Par exemple, la DMTF a adopté le modèle objet. En effet, dans les protocoles tels que WS-MAN [DMTF 2010], le schéma conceptuel *CIM* (*Common Information Model*) [DMTF 2012a] est décrit par un diagramme de classe UML comme présenté en

1. www.mibsearch.com ou www.mibdepot.com par exemple

figure 2.3. Ces protocoles sont notamment utilisés pour l'administration de web-services et applications entreprises.

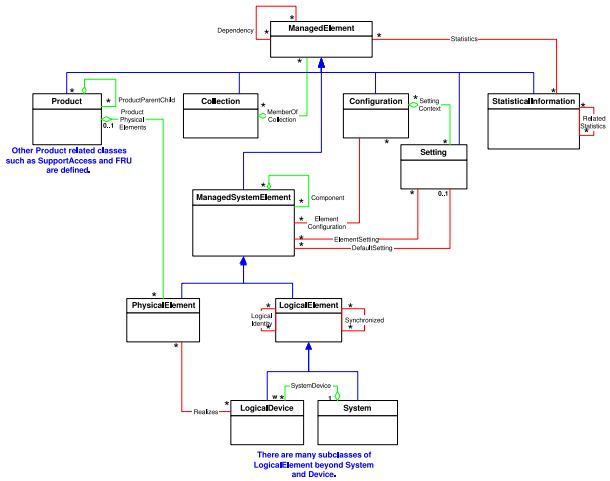


FIGURE 2.3 – Modèle de classe de la partie *Core* de *CIM*

Plusieurs extensions standardisées existent pour représenter les différents types d'objets observables : Dispositifs, Réseau, Sécurité, Utilisateur, Applications… Ainsi, le modèle de données est un modèle d'objet respectant l'ensemble des spécifications et des extensions. L'inconvénient majeur d'un tel choix reste la complexité de *CIM* qui se trouve souvent réduite pour être applicable [López De Vergara 2001].

Pour accéder aux données, le système se compose de multiples agents qui permettent d'accéder à chaque objet. Nous pouvons noter que ce principe reste le même pour l'administration de modules Java dans le protocole JMX [Sun Microsystems 2002] où des objets particuliers (les *MBean*) sont exposés et peuvent être consultés et manipulés via ce protocole.

Gestion de l'évolution des données

Pour les différentes solutions présentées, les agents ont toujours trois modes principaux pour fournir des données :

Consultation indirecte: Ce fonctionnement est le plus simple. La donnée est stockée dans une base d'information. Au moment de la consultation, l'agent lit directement dans la base et renvoie l'information.

Consultation active: Au moment de la consultation de la donnée via une primitive comme « *get* », l'agent va effectuer le relevé actif de la donnée à la source. Ce relevé peut potentiellement prendre plus de temps qu'une simple lecture dans une base d'information.

Événement: La plupart des systèmes d'administration supportent un mécanisme de *publish/subscribe* permettant de créer des canaux d'événements. La création d'événement se fait à partir du changement de valeur d'un paramètre ou à la création d'objets (ou de chemins).

Toutefois, la consultation et les canaux d'événements sont deux approches et deux mécanismes différents qui sont rarement intégrés.

Gestion globale et unifiée

Le système d'administration est capable de se connecter à une multitude d'agents. Ainsi, il peut constituer la vue globale du système par l'intégration des données.

Il est important de noter que ce système peut être réparti pour mieux amortir la charge en la présence de grandes quantités d'agents à administrer. Toutefois, pour l'administration de ses dispositifs, *France Telecom* utilise un point central d'administration matérialisé sous la forme d'un *ACS* (*Auto-configuration Server*, serveur d'administration en TR-069). En effet, la fréquence d'émission des données en fonctionnement normal est lente (pour chaque dispositif, un rapport par jour typiquement). Pour les 10 millions d'équipements à gérer, une capacité de 500 réceptions par seconde côté serveur est suffisante. Cette capacité est atteinte sur des *ACS* récents tels que *EDGE* [Motorola Mobility, Inc. 2012] de *Motorola* sur des serveurs de puissance moyenne.

Pour permettre de supporter la charge des informations remontées, plusieurs systèmes d'administration proposent des architectures décentralisées (principalement hiérarchiques) pour la gestion à plus grande échelle [Kessis 2010]. La hiérarchie peut se découper par lieu géographique, ou par domaine d'activité.

Nous venons de voir comment les systèmes d'administration sont structurés et notamment comment ces systèmes modélisent leurs données. La section suivante présente les capacités de traitements de ce type de systèmes.

2.1.2 Possibles traitements de données

Que ce soit au niveau de l'agent comme au niveau du gestionnaire global, les données peuvent être traitées. Cette section détaille les différentes possibilités. Tout d'abord, nous voyons comment l'hétérogénéité des systèmes est traitée et comment l'intégration des multiples agents se fait. Par la suite, nous détaillons l'ajout de nœuds particuliers comme calcul de statistiques ou l'ajout de fonctions tierces.

L'hétérogénéité par l'uniformisation des modèles

La standardisation est un enjeu majeur pour les systèmes d'administration à grande échelle. Comme présentés précédemment, la spécification les protocoles d'administration se fait au sein de consortiums. Ainsi, la gestion de l'hétérogénéité s'appuie sur le respect des standards. Suivant les dispositifs observés, différents profils existent. Que ce soit pour les profils hiérarchiques ou à objet, des spécifications sont écrites pour décrire le modèle de données.

La gestion de l'hétérogénéité se base sur l'adoption de profils standardisés. L'intégration de protocoles équivalents est un domaine à part entière dans lequel plusieurs approches ont été proposées [El Kaed 2012]. Enfin, remarquons que plus le nombre de domaines d'intérêt augmente, plus le nombre de spécifications à considérer augmente.

Intégration de sources

L'avantage principal d'utiliser des modèles standards est l'intégration des sources de données. En effet, comme chacune des entités du système répond à un profil prédé-

fini, il est possible de faire l'union des données par catégorie pour avoir toutes les entités répondants aux différents profils. Ainsi, les données sont naturellement intégrées dans un modèle commun, ce qui permet aux concepteurs de systèmes d'observations de fournir des fonctions très avancées sans pour autant connaître l'instance du système. De plus, plusieurs protocoles et standards peuvent être utilisés dans un même système, comme l'a proposé WBEM [DMTF 2012b], dans lequel des objets SNMP, JMX et autres sont intégrés à un modèle commun CIM.

Traitement de données

Une fois les données accessibles à un niveau global, il devient possible de traiter ces données afin de les analyser, ou former des alertes. Pour cela, les systèmes d'administration ne fournissent pas tous les mêmes capacités. Par défaut, la seule capacité que fournit l'agent est la récupération de son modèle de données (ou une sous-partie). Cependant, plusieurs systèmes permettent à l'utilisateur de définir des processus plus complexes pour permettre un traitement de plus haut niveau.

L'approche la plus utilisée est le *scripting*. Le système d'observation fournit à l'utilisateur un langage **impératif** qui lui permet de définir des routines. Par exemple, EDGE [Motorola Mobility, Inc. 2012] fournit une interface *Javascript*, et l'*ACS* d'*Alcatel-Lucent* permet d'utiliser des programmes *Python*. Ces routines peuvent par la suite être intégrées dans des réponses aux événements, ou dans des procédures de diagnostics ou encore de configuration.

Il est notable que les standards WBEM et WS-MAN définissent un langage **déclaratif** de manipulation de modèles CIM, le *CQL* (*CIM Query Language*) [DMTF 2007]. Ce langage est très similaire à *SQL* utilisé dans un cadre relationnel-objet. Dans sa spécification, il permet toutes les fonctionnalités de *SQL* (sélection, projection, jointure, agrégation, imbrication de requêtes). Il est aussi utilisé afin de définir des filtres plus précis sur les événements (en remplacement du langage par défaut *XPath*). Il est intéressant de noter que ce langage permet aussi la définition de *politiques de gestions*, assimilables à des routines événements-condition-action. Toutefois, il reste peu implémenté dans la pratique.

Sur l'agent : paramètres calculés

Dans chacune des solutions présentées, il existe des parties du schéma de données consacrées à la présentation de statistiques. En effet, pour un paramètre dont la valeur représente une mesure, il peut être intéressant de fournir des extremums ou moyennes calculées à la volée. Plusieurs standards intègrent un tel calcul principalement sur un échantillon précis (N données) avec un ensemble fixé d'opérateurs.

Enfin, pour permettre de rendre cette approche flexible, tous les modèles présentés sont extensibles. Par exemple, sous UPnP-DM et TR-069, il est autorisé de rajouter des branches à l'arbre de données sous l'appellation `X_{ORGANISATION}` (par exemple `X_ORANGE_COM`). Ainsi, les développeurs peuvent fournir des données non prévues dans les standards ou rajouter des fonctionnalités métiers de traitement.

2.1.3 Synthèse

Le tableau 2.1 résume l'ensemble de l'analyse menée sur les systèmes d'administration. L'ensemble permet effectivement beaucoup de fonctionnalité pour les utilisateurs. Le choix de s'appuyer sur des standards fait que cette approche est actuellement très répandue pour gérer des systèmes de tous types. Ce qui en fait un bon système pour collecter les données sur les ressources du système. Avec cette approche, nous pouvons extraire un grand nombre de données, statiques ou événementielles.

Toutefois, l'interrogation des données est majoritairement faite dans un langage non déclaratif. Il est difficile de créer de la cohérence entre l'ensemble de ces données récoltées, surtout s'ils viennent d'agents et de standards différents. De plus, les canaux événementiels et la consultation des bases d'informations sont gérés dans des approches et mécanismes très différents. Ceci rend une observation intégrée difficile.

Représentation des données	
Modèle de données	Principalement modèle **hiérarchique** sous forme d'arbres. Quelques systèmes d'administration utilisent des modèles objets avec *CIM*.
Schéma conceptuel	Les différentes entités du système sont des nœuds du modèle. Pas de contraintes ou d'inférences exprimables.
Gestion de l'évolution	Le dynamisme est géré par le mode d'accès. Certaines données peuvent être notifiables. Les mécanismes d'interrogations sont séparés.
Traitement des données	
Modes d'interrogation	Instantanée et continue sur certaines données. Pas d'hybride possible vu que les procédés sont très séparés.
Intégration de sources	Standardisation des modèles. Toutes les entités sont structurées dans le même formalisme. Intégration par union des données pour chaque profil lorsque cela est possible.
Langage et paradigme	Appel de méthodes standards pour récupérer un sous-arbre du modèle. Code impératif (scripting) principalement pour manipuler les données au niveau du gestionnaire. Utilisation de langage déclaratif (similaire SQL ou XPath) possible.
Pouvoir expressif	Procédures à écrire en *script*. Projection, sélection et union principalement. Certains nœuds particuliers permettent de calculer des statistiques.
Adaptation à l'application	
Adaptation au système	Pas d'adaptation spécifique, car les dispositifs doivent implémenter des standards.
Gestion de perspectives	Pas de perspectives métiers en dehors de la sélection sur les branches du modèle.
Extensibilité	Nœuds particuliers pour le calcul. Fonctions métiers intégrées dans le gestionnaire.
Performances	Très efficace et utilisé pour gérer des parcs de millions de dispositifs.

TABLE 2.1 – Synthèse des systèmes d'administration

2.2 Informatique contextuelle

Au centre des systèmes pervasifs et de l'informatique ubiquitaire, l'informatique contextuelle prend une importance de plus en plus grande. Sa définition a fait l'objet de plusieurs débats au sein de la communauté scientifique. La définition la plus couramment utilisée est : « L'informatique contextuelle (*context-aware computing* a pour but de permettre aux équipements de fournir de meilleurs services aux utilisateurs par l'utilisation d'informations de contexte »[Han 2008]. Ainsi, le point important est de former un ensemble d'information pour que des applications puissent adapter leur fonctionnement. L'instanciation de cette notion de contexte est un processus d'observation de l'environnement dans lequel se trouve l'application.

La section 2.1 a permis de voir que le système observé peut fournir de grandes quantités de données. Grâce à l'informatique contextuelle, nous souhaitons pouvoir donner de la cohérence à ces données. Ainsi, nous pouvons mieux exploiter leur sémantique et envisager d'effectuer de l'observation de plus haut niveau afin d'établir un diagnostic.

Cette section présente d'abord les définitions et applications de l'informatique contextuelle. Par la suite, nous détaillons la façon dont le contexte est capturé et modélisé. Nous présentons les capacités de traitement sur celui-ci. Enfin, nous analysons différents systèmes pervasifs afin de percevoir la mise en application de cette approche. Nous concluons par une synthèse, détaillant l'adéquation aux critères de notre problématique.

2.2.1 Définitions et applications

La définition de contexte a été elle aussi au cœur de nombreux débats. Après analyse des travaux sur le sujet, le rapport de recherche [Dey 1999] propose la définition suivante : « *Un contexte est toute information pouvant être utilisée pour caractériser la situation d'une entité. Cette entité pouvant être une personne, un lieu, ou un objet considéré comme pertinent à l'interaction entre l'utilisateur et le système.* ».

Il est important de noter que cette définition est orientée par l'utilisation qui en est faite. Une donnée quelconque peut faire partie d'un contexte si elle est utilisée comme tel. Ainsi, il est nécessaire de voir l'ensemble des utilisations de ce contexte. Celles-ci sont rassemblées dans sept catégories principales [Soylu 2009] :

1. Sélection et recommandations d'informations ou de services.
2. Présentation et accès à l'information et aux services.
3. Recherche d'information ou de service.
4. Adaptation de l'exécution de processus séquentiels.
5. Modification et reconfiguration d'applications.
6. Conseil d'actions semi-automatique.
7. Allocations de ressources.

Les utilisations du contexte sont directement reliées à l'observation, car c'est elle qui permet la construction du contexte.

2.2.2 Modélisation et capture du contexte

Le modèle utilisé pour créer et manipuler le contexte peut être de différentes formes : basé sur des principes d'intelligence artificielle (représentations de connaissances, réseaux bayésiens), le génie logiciel (UML), les bases de données (ER : Entité-Relation) ou d'autres moyens applicatifs (arbres, entrées clefs-valeurs). L'UML et l'ER atteignent rapidement leurs limites d'expressivité. Il devient difficile de manipuler les données dans le cadre de contextes larges et hétérogènes à cause de leur rigidité. Ces modèles permettent d'abstraire une partie du monde ou de la logique pour un usage restreint. Par opposition, la gestion de données issues d'ontologies est moins soumise à cette rigidité.

Un modèle sous forme de triplet

Pour représenter l'ensemble des connaissances sur le système, autant en terme de structure que de données, les contextes sont la plupart du temps modélisés comme un réseau sémantique. Le langage communément utilisé pour cela est le RDF (*Resource Description Framework*) [W3C 2004], un standard répandu. Son principe est à la fois simple et puissant. Son expression est simple, car toute sa structure est orchestrée par des triplets : Objet, Relation, Valeur. Par exemple, *la télévision* **est située dans** *le salon*. De même, *le salon* **est une** *pièce*. L'ensemble des triplets forme un graphe où les objets et valeurs sont des nœuds et les relations sont des liens, d'où l'appellation de *graphe sémantique* [Minsky 1974]. Pour étendre l'expressivité des graphes sémantiques et y apporter la notion de classes, les ontologies se sont développées.

Les ontologies

Tel que l'a défini Kalfoglou [Kalfoglou 2001], une ontologie est *une représentation explicite d'une compréhension commune de concepts importants appartenant à un domaine d'intérêt*. Elle permet de capturer et de représenter une vue simplifiée d'un domaine à travers des concepts prédéfinis. Cela permet un langage commun et une taxonomie des concepts, mais aussi, l'ontologie est capable de représenter leurs liaisons. Ainsi, il est possible de modéliser la sémantique propre des différentes données.

Les ontologies sont toutefois structurées dans un langage qui permet de définir les grandes catégories de relations ou d'objets. Plusieurs langages existent, mais tous définissent les entités suivantes :

Concepts Décrivent les classes et sous-classes de toutes les choses du monde.

Instances Ce sont les individus correspondant aux concepts.

Relations Permet de lier les concepts et instances entre eux. Une des relations principales est la relation *Est-Un* (*Is-A* en anglais) qui lie une instance à un concept.

Types de données Types syntaxique d'une donnée : entier, chaîne de caractères, booléen.

Valeurs Valeur qu'un concept ou instance peut avoir.

Par la suite, les langages permettent différentes manières de lier les concepts entre eux. Cette capacité va permettre de montrer l'expressivité de la structure. Par

exemple, une des relations des plus importantes est la relation de hiérarchie. Un *chien* est une sous-classe d'*animal*, et *labrador* est une sous-classe de *chien*. La *relation de hiérarchie* est une *relation transitive*. Ainsi, le langage vérifie naturellement que le *labrador* est une sous-classe d'*animal*.

La logique permettant d'exprimer ces contraintes et inférences structurelles est une logique de description. Suivant la classe de la logique de description sous-jacente, le langage est plus ou moins complexe à traiter par la suite [2]. Par exemple, le langage le plus utilisé reste *OWL Lite*, équivalent à la logique $\mathcal{SHIF}^{(\mathcal{D})}$. Cette logique permet lors de la description des concepts l'utilisation des constructeurs suivants : quantification universelle, négation [3], transitivité de relation, inversion de relation, hiérarchie de relations. De plus, l'usage de propriétés fonctionnelles et de données est possible. En revanche, la restriction d'une collection à un nombre d'éléments donné par exemple n'est pas possible. Son utilisation est répandue, car les inférences sont calculables dans la pratique contrairement à des logiques plus expressives.

Comparées à des structures telles qu'UML, les ontologies jouissent de la flexibilité des réseaux sémantiques. Il est toutefois important de noter que cette puissance et cette liberté rendent sa manipulation délicate. En effet, il est supposé que toutes les sources de connaissances s'appuient sur une ontologie commune. Il est important d'être minutieux dans la manipulation de cette structure pour éviter par exemple des duplications de concepts, voire des conflits de définitions.

Capture du contexte

La capture du contexte est la manière de récupérer une information et de la représenter sous la forme choisie lors de la modélisation. Par exemple, un capteur de température peut insérer un ensemble de triplets pour indiquer qu'à 10h25 le lundi 26 avril, il faisait 25.256°C sur la source T75896. Cet ensemble de triplets dépend de la modélisation des concepts qui forme le contexte. Lors de la capture, les données sont issues du système observé ou par l'extraction de nouvelles connaissances construites à partir d'informations déjà capturées.

La gestion de l'hétérogénéité est rarement mentionnée puisque les sources de données sont censées être conformes au schéma commun. Toutefois, comme présenté dans [El Kaed 2012], il est possible d'aligner les modèles des sources avec un schéma commun. Le formalisme des ontologies permet en effet de décrire les équivalences logiques entre différents concepts ce qui permet l'intégration de données.

2.2.3 Capacités de traitement

Un intérêt des réseaux sémantiques est le pouvoir de raisonnement logique. Le but ici est de pouvoir inférer de nouveaux triplets à partir des connaissances accumulées. Pour cela, il existe plusieurs langages permettant de spécifier ces inférences. Le plus courant est le langage associé à *RDF* : *SPARQL*. Ce langage a la particularité d'être

2. Dans certains cas, comme OWL-Full, son expressivité est tellement large qu'il devient indécidable de vérifier si un concept appartient à une classe.

3. Si la négation et la quantification universelle existent, alors le prédicat d'existence est autorisé, car $\not\forall \equiv \exists$

aussi expressif que l'algèbre relationnelle [Angles 2008] [4]. Ainsi, il est possible de faire des inférences du premier ordre sur ces données.

Ces inférences sont de trois types :

Association directe Une information bas-niveau est associée à une information haut-niveau

Fusion de contexte Un ensemble de données infère un nouvel état

Fission de contexte Une donnée infère un ensemble d'informations

Dans l'informatique contextuelle, il est important de distinguer plusieurs espaces de données [Padovitz 2008] :

L'espace de valeur Par exemple, pour une personne, son age est compris entre 0 et 125.

L'espace applicatif L'ensemble des données atomiques qui représentent le contexte dit de bas-niveau.

L'espace de situation Reflète les situations pouvant être extraites de l'espace applicatif.

À un instant donné, il est possible de définir un **état de contexte** en tant que collection d'attributs de l'espace de contexte. Cet état peut par la suite inférer un ensemble de situations. La figure 2.4 résume la structure abstraite du raisonnement sur les contextes.

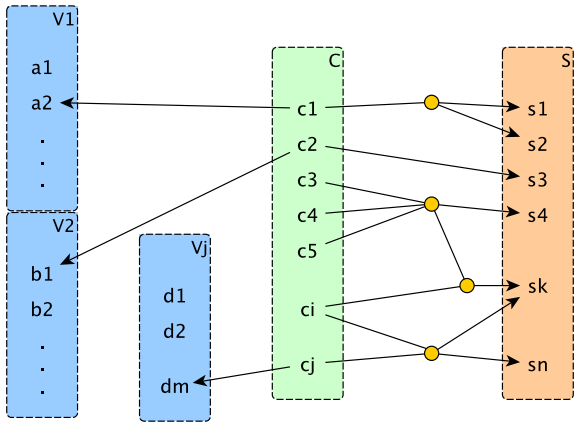


FIGURE 2.4 – Structure abstraite du raisonnement sur contexte. À un élément du contexte c_i est associée une valeur du domaine V_i. À partir de l'espace du contexte C, on effectue des associations, des fissions ou fusions pour inférer l'espace des situations S.

La qualité de contexte

La qualité du contexte [Buchholz 2003] est définie comme toute information permettant de décrire la qualité des informations contenues dans le contexte. Le point

4. Plus exactement, SparQL est équivalent au *datalog* non récursif avec négation, ce qui est équivalent à l'algèbre relationnelle.

crucial est que cette qualité n'est pas liée à un quelconque processus ou matériel. Elle peut se décliner en plusieurs *paramètres* : précision, fiabilité, confiance, fraicheur,...

La qualité des données a des impacts lors du diagnostic. Supposons que nous constatons un problème de pixélisation TV. Cette situation a été inférée par la valeur actuelle d'une métrique indiquant le nombre d'images non décodées. Premièrement, si la source de donnée remontant la métrique n'est pas assez fiable, nous pouvons mettre en doute les relevés. De plus, il est possible que l'information ne reflète pas l'état actuel. Ainsi, la situation indiquant un problème est vraie à la qualité des sources près.

De plus, si nous souhaitons effectuer un diagnostic, nous pouvons indiquer au système que ce type de panne est dû à un problème de lenteur de réseau interne. Toutefois, ce peut être dû à des dysfonctionnements plus rares, comme une panne matérielle. Ainsi, certaines recherches permettent d'introduire une part de probabilités dans les raisonnements pourtant déterministes a priori [Padovitz 2008].

Dans cette thèse, nous ne détaillons pas le concept de qualité des données. Néanmoins, nous pouvons voir que cela peut avoir des impacts majeurs. L'utilisation de ces travaux serait intéressant pour des améliorations futures.

2.2.4 Analyse de systèmes d'observation à base de contextes existants

Dans cette partie, nous analysons des systèmes pervasifs à base de contexte existant. Ceux-ci sont en général très utilisés dans le réseau domestique qui est l'environnement de développement le plus courant dans le domaine de l'informatique ubiquitaire. Cette étude nous permet de voir comment est mise en pratique l'approche présentée jusqu'ici.

De la représentation du système

DogOnt [Bonino 2008], a pour objectif de pouvoir modéliser les objets des environnements domotiques intelligents. Ainsi, en plongeant l'ensemble des équipements au niveau conceptuel des ontologies, il est possible de résoudre les problèmes d'interopérabilité et d'hétérogénéité des données. *DogOnt* est capable de répondre à des requêtes telles que : la position de l'équipement, ses capacités, ses moyens de communication, comment l'environnement est composé (notamment architecturalement parlant, ce qui permet de représenter la maison).

Ainsi, une représentation de haut niveau permet de poser tous les concepts afin de représenter le réseau domestique au sens large. D'une manière plus concrète, un équipement est représenté en tant que « *Controllable* » (et ses sous-classes). Cette ontologie est représentée dans la figure 2.5.

Pour pouvoir observer, et contrôler, les instances de ces concepts, il est nécessaire de rajouter des fonctionnalités et des variables d'états. Ceci se fait par l'introduction de relations sémantiques telles que *hasControl*, *hasFunctionality*, et *hasState*. En combinant ces associations ainsi que l'ensemble complet des instances, il est possible de représenter l'ensemble des périphériques et leurs capacités.

Plusieurs autres projets ont utilisé ce type de modélisation pour des applications pervasives. Par exemple, Amigo [Ben Mokhtar 2007] se focalise plus sur la modéli-

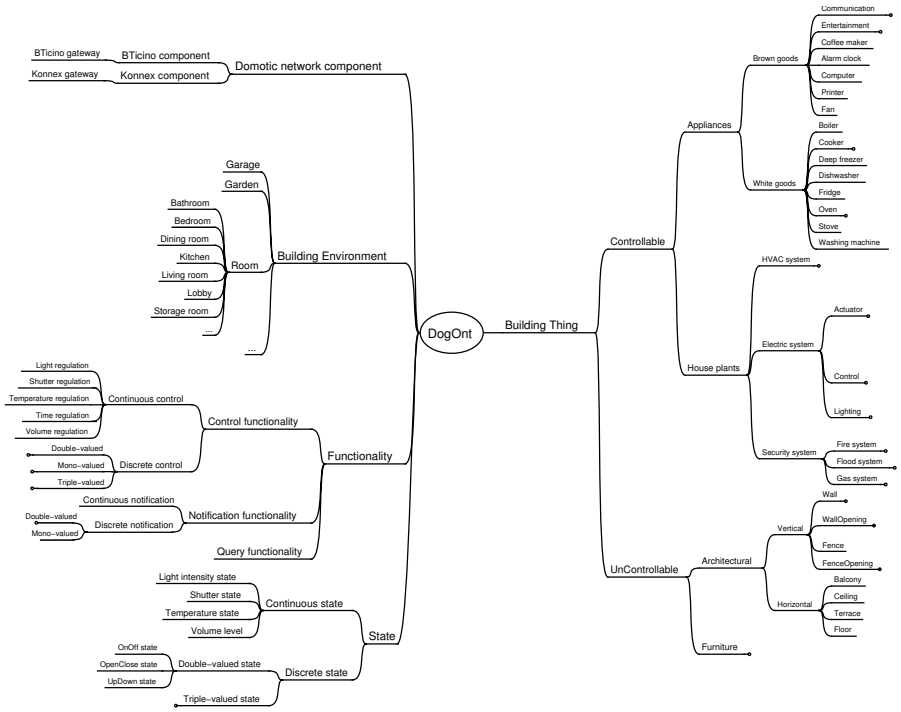


FIGURE 2.5 – L'ontologie de DogOnt

sation des services et des capacités. A contrario, *MATCH* [Docherty 2009] met l'accent sur la hiérarchie des ontologies pour que chaque domaine puisse apporter ses connaissances en utilisant des concepts communs (*dispositifs*, *réseau*,...).

Nous remarquons que la séparation des domaines permet de gérer les perspectives utilisateurs en fonction de leurs intérêts. De plus, nous notons que l'hétérogénéité des schémas conceptuels est gérée par l'utilisation d'une ontologie commune. La nécessité d'une ontologie commune est récurrente dans ces travaux. Toutefois, dans des travaux récents [Niang 2011, Niang 2012], il existe des approches semi-automatiques pour intégrer des données de sources hétérogènes via la génération d'une ontologie commune et d'alignements.

SOCAM : De l'utilité du raisonnement logique

SOCAM (Service-Oriented Context-Aware Middleware) [Gu 2005] propose un intergiciel générique pour permettre aux développeurs de créer des applications pervasives par contexte. Cet intergiciel supporte l'acquisition, la découverte, l'interprétation et l'accès aux contextes. Comme les autres solutions présentées jusqu'ici, il s'appuie sur une ontologie conceptualisée comme celle de *MATCH* afin de pouvoir être générique et y apporter les connaissances de chacun des domaines.

L'architecture de SOCAM, représentée en figure 2.6 qui se décrit en trois 4 composants principaux.

Fournisseurs de contexte Qui permet d'abstraire l'hétérogénéité des données issues des différentes sources (internes, tels que les capteurs ou autres dispositifs), ou externe (services météo ou autres) pour les convertir en *OWL*.

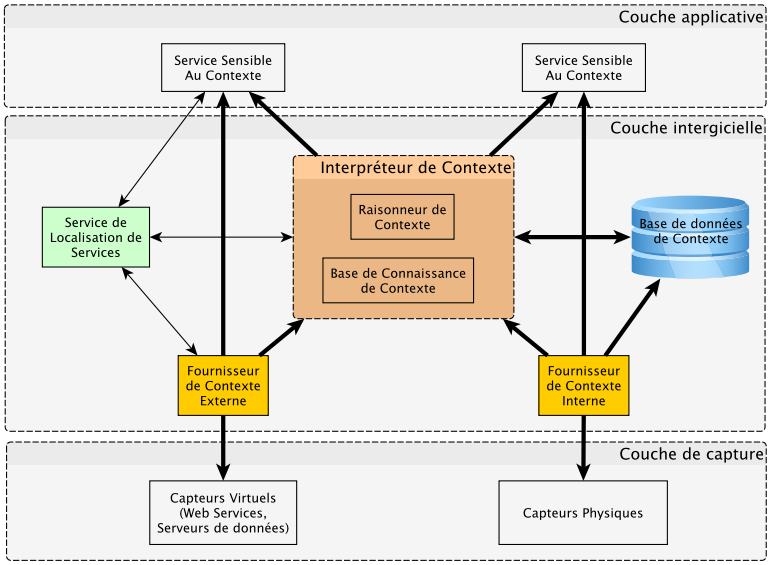


FIGURE 2.6 – Architecture de SOCAM

Interpréteur de contexte Fournit la logique de raisonnement sur le contexte.

Base de données de contexte Stocke les ontologies de contexte et l'historique des contextes pour chaque sous-domaine.

Service de localisation de services Sert de catalogue des services externes disponibles. Sa sélection peut se faire sur un type de service, mais il peut aussi faire une comparaison sur le contexte que le service fournit.

Les services applicatifs utilisant la notion de contexte vont ainsi utiliser les fonctionnalités que fournit SOCAM pour gérer cet ensemble de données. Une manière classique de construire ces services est de spécifier des actions déclenchées par un ensemble de règles au moment où le contexte change.

SOCAM implémente deux manières de faire de l'inférence logique de prédicats. La première est l'inférence structurelle. En effet, l'application d'une propriété transitive à un concept nous donne plusieurs informations. Par exemple, nous pouvons inférer que l'appareil *Livebox* est une passerelle internet. Or, tous les équipements de cette catégorie ont des propriétés telles que la capabilité à gérer les règles de routage. La deuxième manière d'inférer des informations est un ensemble de règles utilisateurs. En utilisant un moteur similaire aux moteurs *Prolog*, il est possible d'induire des informations qui constituent l'ensemble des situations. Par exemple, les auteurs présentent l'inférence du triplet *(user socam:status 'SLEEPING')* par la détection de la position allongée dans la chambre avec la lumière éteinte.

Nous remarquons ici la présence d'une couche de traitement de données entre les sources et les services qui les utilisent. Cette couche se base sur des inférences logiques qui utilisent un support persistant pour constituer son contexte.

2.2.5 Synthèse

En conclusion, le traitement contextuel des données permet de gérer l'hétérogénéité sémantique des données. Les capacités de traitements sont liées à l'inférence logique pour enrichir la base des connaissances accumulées. Ainsi, il est possible de construire un espace de contexte et d'en inférer des situations de haut niveau. Cette capacité d'abstraction est nécessaire pour l'informatique ubiquitaire qui a besoin de manipuler des concepts humains.

Pour notre problématique d'observation, il reste difficile de gérer l'évolution des données au cours du temps. La liberté d'expression des bases de connaissances fait que l'inférence est complexe à manipuler. Plusieurs travaux s'attellent à permettre aux connaissances et aux règles d'introduire la dimension dynamique [Weikum 2010, Hellerstein 2010]. L'apport de cette dimension est toutefois difficile d'un point de vue théorique.

Représentation des données

Modèle de données	Structure sémantique à base de triplet.
Schéma conceptuel	Utilisation d'ontologies. Gestion des contraintes et de l'inférence structurelles variable suivant l'expressivité du langage. En général, *OWL Lite* est utilisé.
Gestion de l'évolution	Pas de gestion explicite du dynamisme en dehors d'annotations.

Traitement des données

Modes d'interrogation	Instantané uniquement.
Intégration de sources	Les sources s'intègrent en se conformant à un modèle commun. Si tel n'est pas le cas de façon native, des règles d'alignements d'ontologies sont à fournir.
Langage et paradigme	Paradigme déclaratif en général dérivé de programmation logique allant de *Prolog* à *SPARQL*.
Pouvoir expressif	Cela part de la logique du premier ordre complète à la capacité de l'algèbre relationnelle (*datalog* non récursif avec négation)

Adaptation à l'application

Adaptation au système	Nécessité de spécifier des ontologies de domaines pour donner la structure des concepts du système.
Gestion de perspectives	La séparation des domaines et le rattachement des données aux domaines permettent de clairement spécifier les différentes perspectives.
Extensibilité	Pas d'extensibilité possible sur le traitement d'inférence. Par contre, il est possible suivant l'architecture du système final de créer des capteurs de contextes pour fournir des données logiques de haut niveau.
Performances	La complexité de l'inférence peut aller jusqu'en *EXPTIME*. Toutefois, plusieurs implémentations optimisées permettent de traiter des millions de triplets en quelques secondes.

TABLE 2.2 – Synthèse de l'informatique contextuelle

2.3 Entrepôts de données

Pour l'informatique décisionnelle (*Business Intelligence*), l'approche par entrepôt de données (*Data Warehouses*) est très largement répandue. Son apport est originellement utilisé dans les applications pour entreprises. Toutefois, les outils de traitement et d'analyse des données sont maintenant très largement utilisés notamment pour des données liées aux réseaux [Gilbert 2001] ou à l'agriculture [Abdullah 2009]. Des procédés ont été développés pour faire l'intégration de données hétérogènes, ainsi que des structures d'entrepôts prêts à répondre aux besoins d'analyses.

L'utilisation des entrepôts dans le cadre de l'observation est pertinente lors de la persistance des données relevés. Ainsi, nous obtenons un grand ensemble d'historiques sur lesquels nous pouvons effectuer de l'analyse. Or, les entrepôts de données fournissent les outils nécessaires pour effectuer ces opérations. De plus, vu cette approche permet l'intégration de données, cela est à considérer pour la gestion de l'hétérogénéité des sources du système observé.

Par la suite, cette section détaille en premier lieu l'architecture globale des entrepôts de données. Nous détaillons ensuite comment l'intégration est faite via les processus *ETL*. Ensuite nous décrivons les structures de données utilisées pour permettre différentes analyses. Enfin, nous présentons brièvement quelques outils d'analyses pouvant raisonner sur les données.

2.3.1 Architecture globale

La définition fonctionnelle d'un entrepôt de donnée est « *une collection de données orientées sujet, intégrées, non volatiles et historisées, organisées pour le support d'un processus d'aide à la décision* » [Inmon 2002]. Au sens large, un entrepôt est une base de données avec une organisation qui s'oriente avant tout sur l'application finale (l'analyse) plutôt que sur l'intégrité des données, et qui néglige la suppression ou la modification de données afin de garder une trace. La figure 2.7 présente une architecture abstraite des

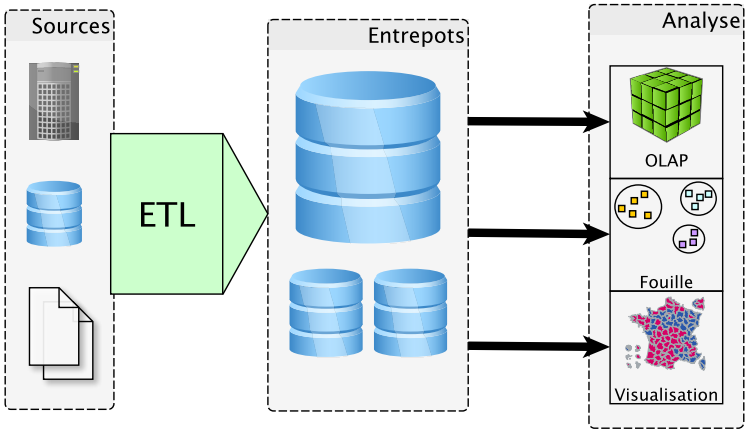


FIGURE 2.7 – Architecture globale d'une solution d'entrepôts de données

approches à entrepôts pour intégrer des sources hétérogènes. Afin d'effectuer l'intégration des données à l'intérieur d'un (ou des) entrepôt(s), il est nécessaire de traiter

cette hétérogénéité et charger les données. Les processus *ETL* ont été conçus pour cette tâche. Ces procédés sont découpés en trois phases : l'extraction (***E**xtract*), le traitement (***T**ransform*) et enfin le chargement (***L**oad*). Cet aspect est détaillé en section 2.3.2.

Par la suite, des outils d'analyses tels que *OLAP* [Codd 1993] ou d'autres algorithmes de *fouilles de données* sont utilisés pour aider l'utilisateur à prendre une décision. Ainsi, les entrepôts adoptent des modélisations permettant ce type d'analyses telles que les schémas relationnels en étoile ou en flocons [Levene 2003] pour représenter des données multidimensionnelles [Gray 1997].

2.3.2 L'intégration par ETL

Afin de charger les données provenant de sources hétérogènes dans un entrepôt, les données sont d'abord extraites des sources. Cette extraction peut être faite par des bases de données intermédiaires. Puis elles sont transformés grâce à une composition d'opérateurs variés [Vassiliadis 2009]. Enfin, les données sont chargées dans l'entrepôt.

Chargement

Plusieurs considérations de performances rentrent en compte pour savoir quand la publication est effective. En effet, lors d'une mise à jour de base de données relationnelle, le système transactionnel requiert que les données soient à jour à la fin de la transaction. Ici, l'écriture effective peut-être retardée pour permettre un chargement des données moins coûteux [Petit 2009].

De plus, à cette étape, il est courant de générer des rapports compilant les données transformées vers un document qui est transmis à l'utilisateur.

Le temps réel : un nouveau challenge

La fraicheur des données à l'intérieur d'un entrepôt de donnée n'a pas été une contrainte critique jusqu'à présent. Les latences usuellement visées par ce type d'approche sont au alentour de l'heure voire du jour [Oracle 2010a]. Récemment, des travaux se sont intéressés à améliorer la qualité du traitement en terme de fraicheur de données (pour passer à l'ordre de la minute voire de la seconde).

Comme présentée en section 1.2, la gestion de l'évolution des données est importante dans notre cadre. Ici, la gestion du dynamisme n'est pas explicite, car il n'y a pas de mécanisme d'événement. Ainsi, il est nécessaire d'implémenter un *CDC* (*Change Data Capture*) capable de transmettre les nouvelles données via un canal de communication dédiée.

Plusieurs systèmes commencent à supporter la gestion en temps réel de l'import de données [Thomsen 2008, Oracle 2010b]. Toutefois, il n'y a pas de résultats concrets concernant le traitement des données dynamiques en tant que tel. Par exemple, la définition d'agrégat sur une fenêtre de temps doit se faire à la main via une base de données intermédiaire.

2.3.3 Synthèse

En conclusion, les entrepôts de données représentent une grande capacité de traitement des données. Ils nous permettent de voir les possibilités que nous pouvons exploiter lors de la persistance de données. Ainsi, la prise en charge de données historique et les capacités d'analyses sont précieuses pour l'observation de système. Nous notons aussi la structure du schéma conceptuel en étoile (ou flocon) séparant métadonnées (données descriptives) et faits (archives de flux).

Toutefois, le processus d'intégration est trop lent pour pouvoir gérer des alertes. De plus, il existe une ambiguïté sémantique sur les opérateurs disponibles dans les ETL qui dépendent de leur implémentation.

Représentation des données	
Modèle de données	Multidimensionnel
Schéma conceptuel	Modélisation en étoile ou en flocon.
Gestion de l'évolution	Faible gestion du dynamisme en dehors des mécanismes CDC dans les entrepôts temps réels.
Traitement des données	
Modes d'interrogation	Instantané principalement avec les opérations OLAP. Les ETL peuvent être apparentés à des interrogations continues.
Intégration de sources	Intégration complexe via les ETL.
Langage et paradigme	Paradigme déclaratif pour l'exploration de données. Algorithmique pour la fouille de données. Principalement procédural pour les ETL.
Pouvoir expressif	En dehors des algorithmes et des procédures de gestion de syntaxe ou de règles métiers, le traitement des données est dérivé de l'algèbre relationnelle.
Adaptation à l'application	
Adaptation au système	Spécification du schéma de l'entrepôt. Description des processus ETL. Si les besoins d'analyses sont poussés, il est nécessaire de concevoir les outils d'analyse aussi.
Gestion de perspectives	La gestion de données multidimensionnelles apporte cette gestion de perspective de façon native. Il est possible de créer différents sous-entrepôts pour chacun des métiers avec des données agrégées (alimentés par ETL).
Extensibilité	Il est très souvent nécessaire de rajouter des opérateurs, ou des logiciels d'analyses plus poussés. Les implémentations permettent en général de rajouter des procédures personnalisées à tous les niveaux de la chaîne.
Performances	Les procédés sont lourds et complexes et la durée de réactivité peut se compter en minutes ou en heures. Toutefois, ces systèmes manipulent des Tera de données.

TABLE 2.3 – Synthèse des entrepôts de données

2.4 Gestion de flux de données

Devant la multiplication des applications à base de flux de données telles que : la gestion des données de capteurs ou la surveillance réseau, les *Systèmes de Gestion de Flux de Données* (*SGFD*) ont été conçus pour mieux maîtriser les données de ce type [Madden 2002, Yao 2002, Cranor 2003]. L'idée principale est de permettre l'interrogation des flux de données via un langage déclaratif (tout comme le *SQL*) avec un grand pouvoir d'expression.

La gestion de flux est le socle fondamental des systèmes capables de traiter les événements. Les travaux récents sur les *CEP* temps réel (*Complex Event Processing*) permettant de faire de la détection d'événements complexes [Brenna 2007] peuvent être vu au final comme une extension des SGFD avec des opérateurs spécifiques et optimisés. L'expression des motifs d'événements pourront être complexes à écrire et à évaluer mais le concept reste d'évaluer des requêtes continues sur des flux de manière déclaratives. Cet aspect conforte nos besoins en terme d'extensibilité du système d'observation.

2.4.1 Approche des SGFD

Un flux de données est une série de données qui s'accumulent au fur et à mesure du temps [Golab 2003]. De façon générale, un flux n'est pas considéré comme régulier, par exemple « *une donnée toutes les 5 secondes* ». L'idée de faire une interrogation sur ces flux de données au sens « gestion de base de données » du terme n'a pas de sens, car il y aurait confusion entre les modes d'interrogations continues et instantanées présentés en section 1.2. En effet, le paradigme des requêtes est fondamentalement différent, car les requêtes sont de longue durée et persistantes [Chen 2000] comme illustré dans la figure 2.8. Nous détaillons ici les différences conceptuelles entre SGBD et SGFD.

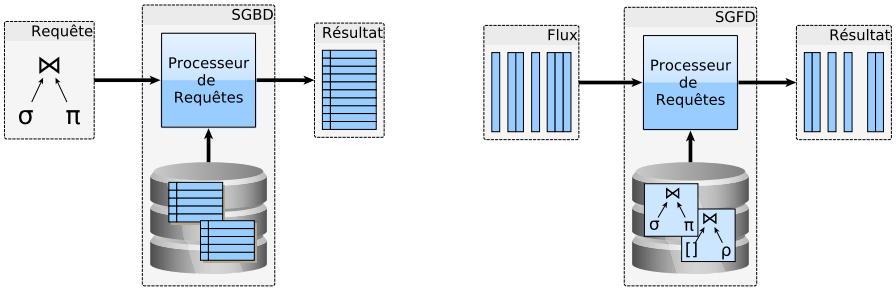


FIGURE 2.8 – SGBD : Requêtes transitoires, Données persistantes vs SGFD : Données transitoires, Requêtes persistantes [Gürgen 2007]

Base de données : Une requête est une question posée sur un ensemble de relations figées et persistantes (principe transactionnel). La réponse est un ensemble de n-uplets. Une fois la requête traitée elle n'existe plus.

Flux de données : Une requête est un ensemble d'opérateurs considéré comme persistant. Le ou les flux de données sont appliqués sur cet ensemble d'opérateurs pour en produire un nouveau flux. La particularité d'un flux de données est qu'une

fois la donnée *consommée*, elle n'est plus considérée comme présente dans le flux d'entrée. Les données fournies en entrée de la requête sont transitoires.

Ainsi, le terme *requête* a un tout autre sens. Toutefois, beaucoup de concepts sont applicables dans ce contexte. En effet, plusieurs éléments du modèle relationnel sont appliqués dans ce domaine [Arasu 2002]. Historiquement, les premières requêtes continues [Terry 1992] étaient une exécution périodique de requêtes *SQL*. Le traitement était entièrement basé sur les opérateurs du modèle relationnel standard. Par la suite, les modèles ont évolué pour supporter le dynamisme propre aux flux.

2.4.2 Intérêt pour l'observation de systèmes

La gestion de flux de données a été créée pour mieux gérer le dynamisme des données. Le résultat montre qu'effectivement, il est possible de faire des requêtes continues sur des flux de données de manière déclarative et avec des performances très efficaces, ce qui correspond à nos exigences pour un système d'observation.

Toutefois, la complexité qui en ressort comparé à celle de l'algèbre relationnelle est plus importante. De plus, les données sont considérées comme uniquement volatiles. Pour notre problématique, il est nécessaire de pouvoir gérer les données persistantes et les requêtes instantanées. Cette approche semble être la plus proche de notre objectif. Ainsi, nous consacrons le chapitre 3 à son analyse détaillée.

2.5 Conclusion

Ce chapitre passe en revue des différents systèmes capables d'offrir une solution d'observation de système. Il en ressort qu'aucun système ne résout l'ensemble des problématiques que nous avons relevés en section 1.2. Le tableau 2.5 résume les éléments d'analyse en colorant les différents points suivant leurs conformités.

Il en ressort que les systèmes d'administration sont avant tout des systèmes qui donnent accès à de nombreuses données grâce aux standards. L'architecture avec agents permet une grande flexibilité pour s'adapter aux cas d'usages. De son côté, l'informatique contextuelle fournit des outils permettant de modéliser et manipuler proprement les concepts du système grâce aux ontologies et aux raisonnements logiques. Il en sort une claire séparation des domaines de compétences, mais une complexité de traitement importante. Les entrepôts de données quant à eux, se distinguent par des capacités d'analyses très poussées, ainsi qu'un procédé d'intégration, très complexe et lourd. Enfin, la gestion de flux de données est une base solide pour traiter les données dynamiques. L'intégration et l'adaptation à l'application sont faites de manière déclarative, ce qui en fait une solution performante et viable pour établir un système d'observation.

À la vue de l'état de l'art, voici les principaux points qui serviront à l'établissement de notre contribution :

– Aucune des approches ne permet de constituer un système d'observation complet, notamment en l'absence de langage permettant d'exprimer des requêtes instantanées, continues ou hybrides.

Représentation des données

Modèle de données	Modèle dérivé du relationnel, mais où le contenu est variable dans le temps.
Schéma conceptuel	Ensemble de sources indépendantes sauf représentation ad hoc.
Gestion de l'évolution	Toutes les données sont dynamiques et événementielles a priori.

Traitement des données

Modes d'interrogation	Continue.
Intégration de sources	Il est supposé que chaque source produit un flux de données. Le fait de traiter ces flux participe à l'intégration de sources.
Langage et paradigme	Langages de requêtes similaires aux SQL supportant la dynamique des données
Pouvoir expressif	À un instant donné, les opérateurs sont semblables au relationnel. L'expressivité du support de l'évolution des données reste inconnue encore.

Adaptation à l'application

Adaptation au système	Création de source pour fournir les flux de données. Comme il n'y a pas de schéma conceptuel, l'adaptation est l'écriture de requêtes continues.
Gestion de perspectives	Pas de perspectives métiers.
Extensibilité	Les sources et puits sont en général adaptés par les utilisateurs. Un développement peut être fait dessus, mais il est rarement possible de rajouter des opérateurs.
Performances	Très rapide, car la latence de traitement doit être contrôlée pour supporter des hauts débits.

TABLE 2.4 – Synthèse des systèmes de gestion de flux de données

– La gestion de flux est un bon socle pour gérer les données dynamiques grâce aux requêtes continues.

– Les entrepôts et les SGBD sont capables de répondre aux requêtes instantanées.

– Les ETL sont peu adaptés pour intégrer les données en flux, alors que les SGFD sont plus déclaratifs.

Les systèmes de gestion de flux de données forment une bonne approche pour l'observation de systèmes. Nous allons approfondir l'état de l'art technique sur ce domaine pour modeler notre contribution. En particulier, en clarifiant et en augmentant les capacités des langages de gestion de flux de données pour accéder aux données persistantes, nous obtenons un outil plus apte à répondre à notre problématique car il intégrera données persistantes et temps réel. De plus, l'utilisation d'un support relationnel, cela permet de gérer un schéma conceptuel du système observé, ainsi que des capacités d'analyses plus évoluées (via les *OLAP* relationnels). Le chapitre suivant détaille l'état de l'art technique de la gestion de flux de données.

Adéquation par rapport aux problématiques : correcte utilisable mauvaise

Critère	Système d'administration	Gestion de contexte	Entrepôts de données	Gestion de flux de données
Modèle de données	Hiérarchique	Triplets	Multidimensionnel	Relationnel dérivé
Schéma conceptuel	Structure hiérarchique sans contraintes	Ontologies	Étoile ou Flocon	Pas de structure
Gestion de l'évolution	Notifications	Ajout du temps en propriété	CDC	Flux natif
Modes d'interrogation	Instantanée, continu en ad-hoc	Instantané principalement	Instantané. ETL en pseudo-continu	Continu uniquement
Intégration de sources	Standardisation, union de modèles	Fusion d'ontologies non standards	Processus ETL (complexe)	Union et jointures de flux
Langage et paradigme	Impératif principalement	Logique	Déclaratif (SQL) et Procédural (ETL)	Déclaratif principalement
Pouvoir expressif	Procédures à écrire soi-même	Logique du premier ordre	Relationnel multidimensionnel et Algorithmie dédiée	Relationnel avec support du dynamisme
Adaptation au système	Support des standards	Spécification longue des domaines	Spécification du schéma, des ETL, autre (complexe)	Écriture de requêtes
Gestion de perspectives	Aucune	Séparation par les domaines	Données multidimensionnelles	Aucune
Extensibilité	Modèle extensible, fonctions métiers dans le gestionnaire	Capteurs virtuels	Opérateurs ETL, procédures SQL, algorithmes	Sources et puits mais pas les opérateurs
Performances	Large échelle	Complexité très haute	Réactivité lente, Support de grande quantité	Support de haut débits

TABLE 2.5 – Récapitulatif de l'état de l'art des systèmes d'observation

« Now that I know what they all do
I have to find my place
And help with all of my heart
Tough task ahead I face »
Twilight Sparkle, Winter Wrap Up

3 Systèmes de Gestion de Flux de Données

Nous avons présenté dans le chapitre précédent les concepts de la gestion de flux de données. Celle-ci permet de répondre à l'interrogation continue des données temps réel. Dans ce chapitre, nous explorons ce domaine d'un point de vue plus technique afin d'identifier nos points de contributions pour exploiter ce domaine dans le cadre de l'observation de systèmes.

Ce chapitre est organisé en quatre sections. Tout d'abord, la section 3.1 présente les formalisations théoriques pour analyser les capacités d'interrogation des SGFD. Par la suite, nous présentons les infrastructures de traitement en section 3.2. Cette section analyse l'impact des modèles théoriques et les limitations de leur mise en œuvre. Ainsi, il devient possible d'analyser l'intégration des supports persistants aux infrastructures de traitements en section 3.3. Enfin, nous observons des aspects liées aux performances en présentant les recherches sur les optimisations en section 3.4. Nous concluons en présentant une synthèse et une description des points techniques de contributions sur lesquelles nous intervenons dans les chapitres suivants.

3.1 Formalisations théoriques

De nombreuses propositions ont été faites pour construire un modèle sur les flux en tentant de réutiliser les connaissances développées sur les systèmes de gestion de

base de données relationnels depuis 40 ans.

Observer l'évolution des modèles au fur et à mesure des années permet en effet de cerner les problématiques théoriques liées à la gestion de flux. En 3.1.1, nous présentons les premiers modèles. En 3.1.2, nous présentons la sémantique abstraite mélangeant flux et relations. Puis, nous présentons en 3.1.3 les initiatives de clarification et reformalisation des modèles existants.

3.1.1 La genèse de la gestion de flux

L'idée de créer des requêtes continues a été présentée en 1992 dans le système Tapestry [Terry 1992]. Dans ce système une requête continue est avant tout une requête instantanée exécutée périodiquement. Ces requêtes s'appliquent sur un ensemble de données sans suppression ou mise à jour (*append-only*). Le résultat est représenté par le flux des nouvelles données calculées de manière incrémentale. Pour cela, les relations sont considérées comme des variables et elles deviennent dépendantes du temps.

En 1995, les flux de données ont été explorés tout d'abord par le modèle séquentiel $\mathcal{SEQ}$ [Seshadri 1995]. Dans ce modèle, une « *séquence* » est un ensemble d'enregistrements (n-uplets relationnels) avec un ordre positionnel. Dans le modèle relationnel originel [Codd 1970], ceci n'est pas autorisé, car l'ordre des n-uplets est dit « *irrelevant* ». La définition des opérateurs relationnels utilise cette liberté d'ordre pour être consistante et obtenir des propriétés intéressantes (voir chapitre 4). Le formalisme de $\mathcal{SEQ}$ montre qu'il est toujours possible d'exploiter les opérateurs algébriques relationnels classiques et d'en créer de nouveau comme les premières notions de regroupements de n-uplets appelés*collapse* [1]. Ce formalisme a été fondateur pour plusieurs des travaux futurs qui s'inspirent explicitement de ce modèle [Gürgen 2007, Babcock 2002].

Par la suite, les premières opérations sur les flux en tant que telles ont pu apparaître avec le système Chronicle [Jagadish 1995]. Toutefois, les flux étaient considérés soit comme un ensemble de données historiques, soit comme une donnée constamment mise à jour (fenêtres complètes, ou instantanées). La notion de fenêtre a été présentée pour la première fois dans Tribeca [Sullivan 1996, Sullivan 1998]. Sa spécification est basée sur sa taille positionnelle (nombre de données) ou temporelle (durée). Toutefois, ce système ne peut gérer qu'un seul flux à la fois. La notion de fenêtre est désormais intégrée à SQL:1999 [Melton 2002] ainsi que dans SQL:2003 [Eisenberg 2004] pour des opérations OLAP.

Les requêtes continues ont été aussi développées pour interroger les bases de données dont le contenu est régulièrement mis à jour. À la fin des années 90, les bases de données contenant des pages et flux web dynamiques ont connu de graves problèmes de performances lors de l'analyse de flux d'événements. Ceci a été l'objet de projets tels que OpenCQ [Liu 1999] et NiagaraCQ [Chen 2000] qui permet de faire des opérations de jointures.

Le début des années 2000 a marqué l'avènement des systèmes de gestion de flux de données à part entière. L'arrivée d'applications réseaux [Cranor 2003] et capteurs [Madden 2002, Yao 2002] a permis le développement des SGFD, car les besoins en performances et en puissance d'expression devenaient de plus en plus grand. À

1. Cette primitive représente ce qui est devenu l'opérateur de fenêtrage depuis

quelques mois d'intervalles, les premiers SGFD ont fait leur apparition : TelegraphCQ [Chandrasekaran 2003], STREAM [Babu 2001] et Aurora [Carney 2002]. Ce dernier apporte de plus : SQuAl [Abadi 2003], la première algèbre complète sur les flux de données. Un flux est considéré comme un ensemble de n-uplets strictement ordonnés avec un schéma prédéfinit $(\mathbf{TS}, A_1, \ldots, A_n)$, où **TS** correspond au *timestamp*. L'algèbre décrit le comportement d'opérateurs simples comme la sélection (*Filter*), la projection-évaluation (*Map*) et l'union (*Union*). Mais aussi d'opérateurs complexes (i.e. utilisant une fenêtre) tels que le tri à la volée (*BSort*), l'agrégat (*Aggregate*), la jointure sur bande (*Join*) et un opérateur de synchronisation de *timestamp* (*Resample*). Il est important de noter que ces opérateurs complexes s'utilisent grâce à des fenêtres définies à l'avance.

TelegraphCQ [Chandrasekaran 2003] quant à lui propose un langage de requête beaucoup plus axé sur le relationnel avec notamment une définition de fenêtre générique. En effet, ces dernières étaient décrites par une séquence de type *boucle for*.

```
for(t=initial_value; continue_condition(t); change(t)) {
    WindowIs(StreamA, left_end(t), right_end(t));
    WindowIs(StreamB, left_end(t), right_end(t)); ?
}
```

Cependant, depuis le début des requêtes continues, la spécification des opérateurs était avant tout dirigée par l'implémentation. Cela a amené à des limitations d'expressivité (voire des divergences d'interprétation) qui ont été traitées dans le SGFD générique STREAM.

3.1.2 La sémantique abstraite à deux concepts

Le système STREAM [Babu 2001, Arasu 2004a] se démarque des autres systèmes de son époque pour avoir décrit une sémantique abstraite qui distingue les concepts de flux et de relation [Arasu 2002] :

Un flux est un ensemble potentiellement infini de n-uplets conformes à un schéma commun possédant un *timestamp*.

Une relation est une fonction qui associe le temps à un ensemble fini de n-uplets conformes à un schéma commun.

Le point crucial de cette approche est que les opérateurs permettent de passer d'un concept à l'autre (voir figure 3.1). Les opérateurs capables de traiter des relations pour en fournir une nouvelle sont les opérateurs relationnels (sélection, projection, jointure). Ceux-ci sont une adaptation de l'algèbre relationnelle. Les opérateurs transformant un flux en relation sont les opérateurs de fenêtres. Les opérateurs transformant les flux en relations sont des *streamers*.

Afin de mieux illustrer la sémantique abstraite utilisée, nous allons détailler un exemple plus concret.

Exemple 3.1 :
Soit S un flux de capteurs (temperature, $\boldsymbol{T}$) *avec* $\boldsymbol{T}$ *pour timestamp et temperature pour sa mesure. Nous souhaitons faire un agrégat sur 5 min. La composition des opérateurs est la suivante :*

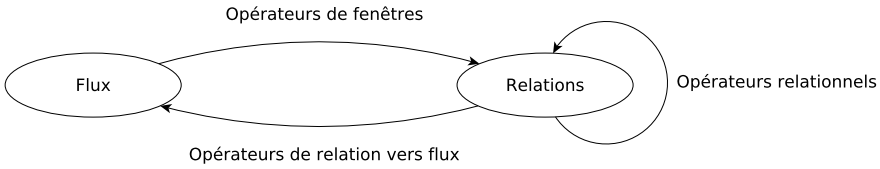


FIGURE 3.1 – Transformations appliquées dans la sémantique abstraite de STREAM

– D'abord un opérateur de fenêtrage $[5min]$ *est appliqué. Il transforme le flux en la relation « groupement des 5 dernières minutes » qui évolue au cours du temps.*

– Ensuite un opérateur d'agrégation[2] $\mathcal{G}_{avg(temp)}$ *produit une relation avec un seul n-uplet.*

– Enfin, un opérateur de streaming *est appliqué. Celui-ci peut avoir plusieurs sémantiques. Prenons le plus simple ISTREAM qui permet de créer un flux à partir des insertions dans la relation. Une mise à jour est considérée comme une suppression suivie d'une insertion.*

La requête finale est :

$$ISTREAM(\mathcal{G}_{avg(temp)}(S[5min]))$$

Le point clé et novateur de cette approche est que les opérateurs de flux vers flux **n'existent pas**. En particulier, la jointure de deux flux n'existe pas. Seule la jointure de relations est possible et la création des relations se fait à partir de fenêtres sur des flux. Les auteurs ont justifié cette approche, car l'écriture de requête est plus intuitive et que cela permet de généraliser l'utilisation des vues matérialisées dans le traitement des flux (introduit auparavant dans Chronicle [Jagadish 1995]). La spécification de cette sémantique a permis par la suite de décrire le langage associé CQL [Arasu 2006b] (*Continuous Query Language*) dérivé du *SQL* qui est désormais utilisé dans de nombreux produits académiques et commerciaux [Witkowski 2007, SQLstream 2010].

Exemple 3.2 :
La requête CQL de l'exemple précédent a la forme suivante :

ISTREAM(SELECT AVG(temp) as avg FROM S [RANGE 5min])

La sémantique formelle du langage CQL est présentée dans la thèse d'Arvind Arasu [Arasu 2006a]. L'algèbre *ACO* (*Algebra of Continuous Operators*) y est décrite. Les définitions élémentaires sont les suivantes :

Instant (τ) : élément de l'ensemble $\mathcal{T}$, discret et ordonné[3] et représenté par $\mathbb{N}$.

Relation : Fonction associant un instant à un multiensemble de n-uplets avec un schéma commun.

2. notation simplifiée de l'opérateur d'agrégation sans groupement dont le résultat est une relation avec un seul attribut *avg*

3. Il est en réalité définit comme un ensemble satisfaisant les axiomes de Peano. En particulier la présence d'un unique successeur. L'instant de départ de l'exécution est le plus petit instant observé 0.

Flux : Multi-ensemble d'éléments $\langle s, \tau \rangle$ où s est un n-uplet respectant un schéma commun et $\tau \in \mathcal{T}$ son *timestamp*. Pour un τ donné, il existe un nombre fini (mais non borné) d'éléments.

Des opérateurs sont définis sur ces éléments. Les opérateurs relationnels sont simples car pour une *relation* R, et un instant τ : la *relation instantanée* $R(\tau)$ est une relation au sens SGBD[4]. Les opérateurs classiques peuvent être utilisés. Les équivalences suivantes sont vérifiées : $\forall \tau \in \mathcal{T}$,

$$\begin{aligned} (\sigma_c R)(\tau) &= \sigma_c(R(\tau)) \\ (R_1 \bowtie R_2)(\tau) &= R_1(\tau) \bowtie R_2(\tau) \end{aligned}$$

Les opérateurs les plus importants sont ceux de classe *S2R* (*stream-to-relation*) et *R2S* (*relation-to-stream*). La table 3.1 liste les opérateurs présentés.

Opérateur	Classe	Description
$S[N]$	*S2R*	Fenêtre positionnelle glissante de taille N n-uplets
$S[W]_T$	*S2R*	Fenêtre temporelle glissante de taille W unités de temps
$S[1]_T$	*S2R*	Fenêtre représentant l'instant présent
$S[\infty]$	*S2R*	Fenêtre accumulative
$\mathcal{IS}(R)$	*R2S*	Flux d'insertion
$\mathcal{DS}(R)$	*R2S*	Flux de suppression
$\mathcal{RS}(R)$	*R2S*	Flux de présence

TABLE 3.1 – Opérateurs de flux de l'algèbre *ACO*

Pour les *streamers*, leurs définitions sont dérivées de l'état de la relation à l'instant présent ainsi qu'à l'instant précédent. Leurs définitions dépendent de la discrétisation du temps. Soit R une relation,

$$\begin{aligned} \mathcal{IS}(R) &= \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\}) \\ \mathcal{DS}(R) &= \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\}) \\ \mathcal{RS}(R) &= \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\}) \end{aligned}$$

La définition des fenêtres est quant à elle plus technique. L'idée est de regrouper les n-uplets selon un critère particulier. Soit S un flux,

$$\begin{aligned} S[W]_T &= \{s | \langle s, \tau' \rangle \in S \wedge (\tau' \leq \tau) \wedge (\tau' \geq \max\{\tau - W + 1, 0\})\} \\ S[N] &= \{s_i \in S | \max\{1, n(\tau) - N + 1\} \leq i \leq n(\tau)\} \end{aligned}$$

où la suite (s_n) correspond à la suite des n-uplets ordonnés par *timestamp* et $n(\tau)$ le nombre d'éléments de S possédant un *timestamp* $\leq \tau$.

Le langage *CQL* et l'algèbre *ACO* ont été démontrés comme plus expressifs [Arasu 2006b] que les autres solutions que nous avons mentionnées précédemment (Chronicles, Tribeca, Tapestry, Gigascope, Aurora et TelegraphCQ).

4. D'un point de vue strict, les multi-ensembles sont utilisés dans les SGBD mais ne font pas parti du modèle relationnel.

3.1.3 Formalisation de la sémantique des opérateurs

Depuis 2005, plusieurs travaux se sont intéressés à la formalisation de la sémantique des opérateurs. Le plus étudié reste l'opérateur de fenêtrage qui est une des principales raisons de la complexité des systèmes de gestion de flux de données [Maier 2005, Patroumpas 2006, Patroumpas 2011]. Une meilleure compréhension de cet opérateur permet de mieux maîtriser la sémantique implantée, mais aussi des optimisations de l'évaluation telles que le partage de résultats (voir section 3.4).

En 2008, les auteurs à l'origine de *STREAM*, d'*Aurora*, de *StreamBase* et d'*OracleCQ* ont présenté [Jain 2008] pour montrer l'ambiguïté sémantique des langages de gestion de flux. En effet, l'exécution des opérateurs complexes tels que les fenêtres n'étaient pas identiques entre l'interprétation de *StreamBase* et l'interprétation d'*Oracle*. Le phénomène s'observe lorsque les modes d'exécutions sont différents.

– Le mode basé sur le temps groupe tous les n-uplets qui possèdent le même *timestamp*. Le flux est formé, comme présenté sur la figure 3.2, de n-uplets groupés par *timestamp*. Ainsi, les traitements sont exécutés à chaque *timestamp*.

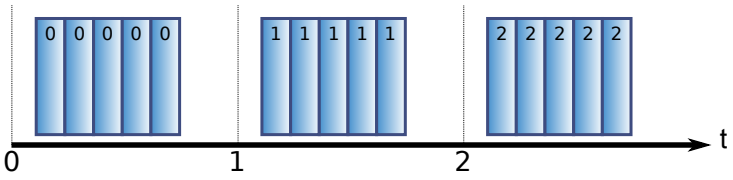


FIGURE 3.2 – Mode d'exécution à base temporelle

– Le mode basé sur les n-uplets au contraire considère que chaque n-uplet est une donnée à part entière. Le *timestamp* n'est qu'une information du n-uplet pouvant être éventuellement utilisé par un opérateur de fenêtre, comme présenté sur la figure 3.3.

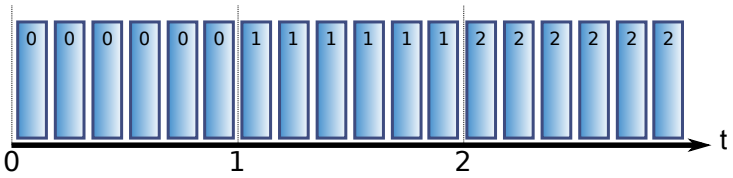


FIGURE 3.3 – Mode d'exécution basé n-uplet

Or, ces deux sémantiques rentrent effectivement en conflit sur certaines opérations, en voici un exemple :

Exemple 3.3 :
Soit S un flux dont le contenu est, dans le formalisme ACO, $\{\langle s_1, 0\rangle, \langle s_2, 0\rangle, \langle s_3, 1\rangle\}$. Nous souhaitons observer le résultat de la requête $S[1]$ (fenêtre de 1-tuple).

Si le modèle d'exécution est basé sur l'arrivée des n-uplets. Alors lorsque le n-uplet s_1 entre dans le système. Le système remplit une nouvelle fenêtre, ce qui donne $S[1](0) = \{s_1\}$.

Le n-uplet s_2 est fournit et le système produit une nouvelle fenêtre : $S[1](0) = \{s_2\}$. Cette interprétation permet de considérer tous les n-uplets. Néanmoins, au même instant, la fenêtre a deux états différents, ce qui n'est pas correct d'un point de vue formel.

Si le modèle d'exécution est à base temporelle. Alors à l'instant 0, l'opérateur va obtenir les deux n-uplets. Puis, conformément à sa spécification algébrique, va sélectionner l'un d'eux, pour produire $S[1](0) = \{s_1\}$ ou $\{s_2\}$. Ceci est correct d'un point de vue formel, toutefois cet opérateur va perdre des données, ce qui peut être problématique.

Le problème est que les deux sémantiques ont du sens a priori. Pour clarifier, les auteurs introduisent la notion de *batch*. Le *batch* est une généralisation de ces modes puisqu'il considère que l'ensemble des n-uplets simultanés est partitionné en sous-ensembles ordonnés, comme représentés dans la figure 3.4. Ainsi, l'unité de temps d'exécution n'est pas le *timestamp* ou l'n-uplet, c'est le *batch*.

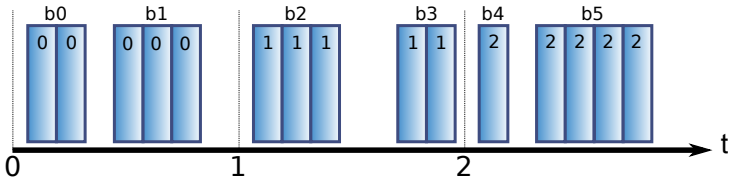


FIGURE 3.4 – Mode d'exécution basé *batch*

Le *batch* correspond à un envoi groupé de la part de la source. Deux événements peuvent se produire au même *instant* (à l'unité de temps près)[5]. Ainsi, lorsque la source effectue deux actions de production de données, ces actions correspondent à deux *batchs* différents.

Des opérateurs sont développés pour manipuler les *batchs* pour passer d'un mode d'exécution à l'autre. Par exemple, il est possible de structurer les *batchs* d'un *timestamp* donné à partir d'un autre attribut (identifiant par exemple).

De façon similaire, nous pouvons remarquer que dans l'algèbre *ACO* la gestion de l'ordre positionnel est floue. Le flux est ordonné par *timestamp*, mais en cas de n-uplets dits *simultanés*, aucun ordre strict n'est définit. Dans la définition de la fenêtre positionnelle, *ACO* introduit la définition d'une suite (s_n) strictement ordonné. A. Arasu note sur ce point [Arasu 2006a] : « *Comme nous ordonnons arbitrairement la séquence s_1, s_2,..., les fenêtres positionnelles sont non déterministes lorsque les timestamps ne sont pas uniques, ce qui peut ne pas être approprié* ». L'introduction des *batchs* apporte une généralisation de l'ordre temporel, ce qui permet de mieux maîtriser la sémantique d'exécution. Toutefois, s'il est nécessaire de faire un choix d'un nombre limité de n-uplets par exemple pour remplir une fenêtre, ce choix n'est pas spécifié.

Suite à l'identification des problèmes de modes d'exécution, le modèle *SECRET* [Botan 2010] a été développé. Le principe est de décomposer l'opérateur de fenêtre en trois parties. Tout d'abord, le modèle décrit sa portée et son contenu (*scope & content*) pour savoir quels n-uplets sont inclus dans les fenêtres. Ensuite, les auteurs

5. Ces cas apparaissent facilement dans les cadres de grande échelle ou lorsqu'un événement est cause d'un autre, ou encore lors de la jointure d'un n-uplet avec plusieurs autres.

détaillent son mode de consommation de n-uplet (*tick*), et enfin son mode de notification à l'utilisateur. Cette séparation claire des concepts permet d'avoir une meilleure compréhension des sémantiques possibles sur les fenêtres. Ainsi, ce modèle permet de décrire les modes de fonctionnement de systèmes existants pour mieux permettre leur intégration par la suite.

Les travaux de Krämer [Krämer 2009] détaillent une algèbre de manipulation des flux. Le point novateur est la séparation des flux dits*bruts*, *logique* et *physique*.

– Les flux tels que nous les avons décrit dans cette section sont des flux *bruts* composés de couples $(\text{n-uplet}, timestamp)$.

– Les flux logiques sont quant à eux un ensemble de triplets $(\text{n-uplet}, timestamp, n)$. La multiplicité n permet de compter le nombre d'occurrences du n-uplet a été observée dans le flux originel.

– Les flux physiques utilisent une définition très similaire à ce qui peut se trouver dans les bases de données temporelles avec un intervalle de validité. Ses éléments sont des couples $(\text{n-uplet}, [t_s, t_e[)$.

Deux algèbres sont ainsi décrites. La première manipule les flux logiques. Elle est utilisable pour correctement comprendre la sémantique des différents opérateurs. La seconde manipule les flux physiques qui sont utilisés par les implémentations des opérateurs. Des équivalences d'expressions sont ensuite définies pour passer du domaine logique au physique.

Enfin, les travaux sur SoCQ [Gripay 2010] permettent d'exprimer une algèbre pouvant modéliser des extensions aux relations classiques de base de données. Ces extensions apportent des attributs pouvant être issus de flux ou des propriétés virtuelles fournies par des services logiciels[6]. L'avantage d'une telle approche est de pouvoir intégrer rapidement les services issus de l'implémentation avec une gestion des données. Ainsi, nous avons une modélisation des sources mélangeant relations classiques et flux de données.

3.2 Infrastructure de traitement des flux de données

Les architectures des systèmes de gestions de flux de données diffèrent sur plusieurs points. Toutefois, plusieurs concepts fondateurs sont communément utilisés pour l'évaluation des requêtes. Tout d'abord, en 3.2.1, nous présentons les principes des infrastructures des SGFD. Ensuite, en 3.2.2, nous détaillons les principes architecturaux développés particulièrement pour permettre aux infrastructures de supporter les grandes quantités de données. Enfin, nous présentons, l'intégration de systèmes hétérogènes en 3.2.3.

3.2.1 Éléments d'architecture de l'évaluation de requête

Le modèle théorique permet de formaliser la sémantique exacte des expressions de requêtes. L'évaluation de ces requêtes soulève des difficultés, notamment en ce qui concerne la gestion de la mémoire.

6. Au sens architecture à service du terme

Les résultats intermédiaires

Comme présentée précédemment en section 2.4, la gestion de flux de données se décompose en trois composants principaux : les sources, les opérateurs et les puits. Les opérateurs consomment une ou plusieurs entités (un flux par exemple) et produisent une nouvelle entité. Ces entités sont des composants représentant les résultats intermédiaires représentés par A, B, X et Y dans la figure 3.5.

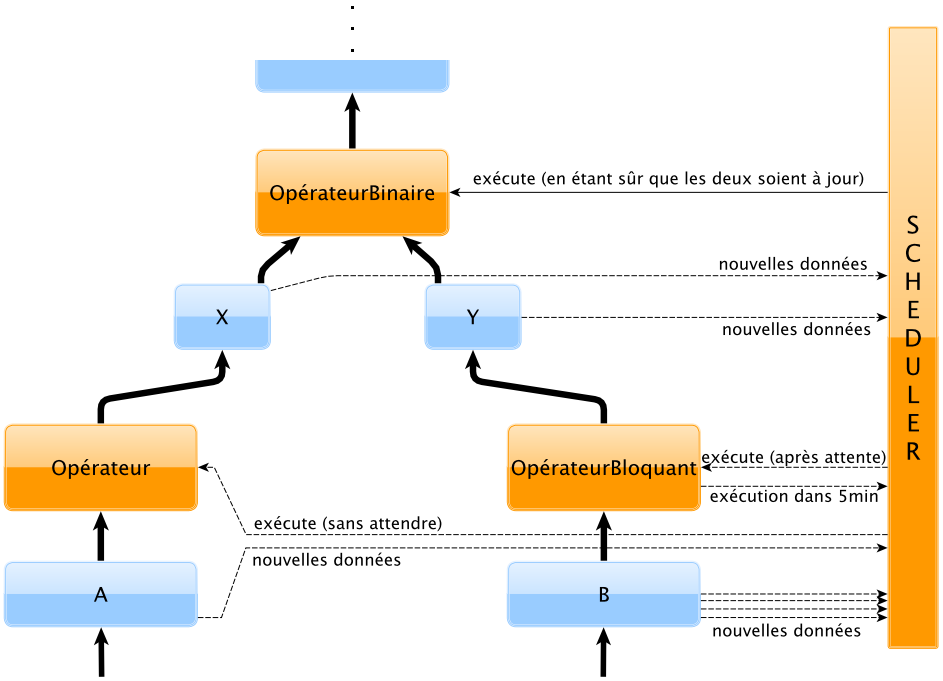


FIGURE 3.5 – Structure de l'évaluation d'une requête continue

Ces composants utilisent des mémoires tampons et occasionnellement des supports persistants [Abadi 2003]. Ces composants sont manipulés par une interface d'appel très simple en général basée sur les primitives *push* et *pop* telle une pile. Or, lors de l'évaluation de la requête, il est nécessaire de vider le plus rapidement possible ces mémoires pour éviter les saturations. Certains travaux introduisent des opérations plus complexes dans ces composants tels que des primitives d'ordres ou de sélection comme dans les *SweepArea* de *PIPES* [Krämer 2009]. Toutefois, il n'est pas toujours possible de consommer directement ces mémoires tampons.

L'exécution

Deux types d'exécutions sont référencés : les non bloquantes et les bloquantes [Babcock 2002]. Originellement, un opérateur *bloquant* est « *un opérateur qui n'est pas capable de produire son premier n-uplet tant que l'ensemble de son entrée n'a pas été consommé* ». Les opérateurs souvent cités sont les agrégats. Par la suite, grâce aux recherches sur les fenêtres [Maier 2005], le terme est généralisé aux opérations de fenêtrages dont le décalage est différent de *1 n-uplet*.

Ainsi, les opérateurs bloquants ne sont pas seulement cadencés par les données de son entrée. Ils ont leur propre cycle de vie, ce qui les rend plus compliqués à exécuter. Le module d'ordonnancement (*scheduler*) [Carney 2003] est présent dans la plupart des implémentations supportant ce type d'opérateur. Ce composant a pour tâche de décider quel opérateur exécuter et quand.

Dans la figure 3.5, le *scheduler* contrôle l'exécution des trois opérateurs. Le premier *Opérateur* est un opérateur non bloquant, ainsi lors de l'arrivée d'une nouvelle donnée dans A, il acquiert le droit de s'exécuter. L'*OpérateurBloquant* quant à lui décide de lui même qu'il veuille s'exécuter 5 minutes plus tard (pour une fenêtre par exemple). Ainsi, B accumule des données et envoie d'un bloc l'ensemble de ses données au moment où l'opérateur décide de finalement les consulter. L'*OpérateurBinaire* est non bloquant, néanmoins, le *scheduler* doit bien vérifier que X et Y sont à jour avant de lancer l'exécution. Il est en effet possible que X soit à jour, mais qu'un traitement soit encore en attente pour Y.

Quant à la communication entre les composants résultats et les opérateurs, les mécanismes de souscriptions (*publish/subscribe* ou *push*) [Eugster 2003] ne sont pas toujours adoptés. En effet, au lieu d'appliquer strictement l'approche événementielle, l'infrastructure peut nécessiter de privilégier une approche par appel (*pull*).

Par exemple, le moteur MXQuery [Botan 2007] est un SGFD dont la particularité est de traiter des documents *XML* tels que *RSS* (flux d'actualités). Ce moteur ne supporte que l'approche *pull* en entrée. A la différence, un SGFD plus généraliste tel que STREAM doit supporter un mécanisme événementiel. La gestion de l'hétérogénéité de l'évolution des données s'applique ainsi au niveau de l'infrastructure d'évaluation des requêtes.

3.2.2 Passage à l'échelle

Les travaux sur l'architecture des SGFD se sont beaucoup concentrés sur le passage à l'échelle en terme de volume de données. Le volume de données se décline en deux dimensions : le débit d'un flux et le nombre de flux sources. Nous analysons dans un premier temps le *load-shedding* prévu pour gérer des débits trop importants pour le SGFD. Puis, nous détaillons le procédé de *désignation* permettant de maîtriser le nombre de flux sources. Enfin, nous présentons les choix d'architectures distribuées pratiquées pour subvenir aux grands volumes de données en général.

Load-shedding

Contrairement aux approches classiques de gestion de base de données, il est important de voir que dans la gestion de flux certaines optimisations dégradent la qualité des résultats. L'objectif du *shedding* est de fournir les résultats les plus pertinents possible avec des contraintes d'espace mémoire, de temps de réponse ainsi que de charge réseau. En effet, comme la gestion de flux fonctionne de manière continue, une congestion dans le processus de traitement implique des retards et des complications pouvant aller jusqu'à une saturation du système.

L'idée du *load-shedding* est de pouvoir abaisser ou limiter le taux de n-uplet pour pouvoir les traiter sans introduire de retard. Dans les travaux fondateurs [Tatbul 2006,

Tatbul 2003], l'idée est de pouvoir surveiller le traitement des requêtes afin de reconnaitre les points d'engorgements et d'y implanter des opérateurs de *shedding*. Afin de garantir un certain taux de réussite, la politique utilisée est dirigée par des indications de qualité de services déterminant quelles données sont utiles. En effet, en prenant en entrée un graphe représentant *valeur* $\mapsto$ *taux de suppression acceptable*. Par exemple, dans le cadre d'une surveillance d'incendie, le taux de *shedding* pour les températures inférieures à 20°C pourrait être très fort. Le critère de sélection afin d'autoriser la suppression d'une données peut se faire de façons différentes :

– Par un histogramme et un choix sur le nombre de tuple produit [Han 2007]

– Par un calcul probabiliste pour privilégier un flux plutôt qu'un autre pour sa productivité [Han 2007]

– Par des raisonnements sur l'age des n-uplets [Srivastava 2004]

Désignation

Plus le nombre de sources de données augmente, plus il devient difficile de savoir lesquelles sont concernées par la requête que l'utilisateur souhaite déployer. Une requête par désignation est une requête utilisant une autre requête pour désigner les sources de données à lire. Par exemple : « *Le flux des mesures des capteurs de température du bâtiment A* ». Dans cette requête, il est nécessaire de d'abord faire une interrogation sur l'expression « *Quels sont les capteurs situés dans le bâtimentA et de type température* » avant de déployer la requête.

Ce procédé est implanté dans notamment SStreamWare, HiFi et GSN [Aberer 2007]. Le passage à l'échelle en nombre de sources de données est particulièrement critique dans *GSN* qui vise un internet de capteurs. Son architecture est naturellement pair-à-pair. L'interrogation continue sur ce système peut se faire par le déploiement de nouveaux capteurs virtuels. La désignation est centrale, car la formation d'un capteur virtuel à partir d'autres sources est possible. Toutefois, du fait de l'absence de méta-données la puissance d'expression de cette requête de désignation est limitée. Dans SStreamWare et HiFi, au contraire, le support des méta-données telles que leur position, leur type ou leur fréquence d'émission facilite la sélection des sources concernées.

Architecture distribuée

La distribution des SGFD pour mieux supporter les volumes de données fait partie des techniques classiques. La structure même des requêtes en forme d'arbre d'opérateurs permet la mise en œuvre naturelle de structures hiérarchiques. En effet, des systèmes tels que *Borealis* [Abadi 2005] (historiquement, l'évolution d'*Aurora* [Abadi 2003]) permettent de distribuer le calcul d'une requête sur plusieurs nœuds.

La distribution d'un calcul apporte des problèmes de synchronisation, de répartition de charge, des tolérances aux fautes et de latence de traitement. Des recherches [Hwang 2005, Tucker 2003] ont été faites sur ces points pour permettre une évaluation des requêtes exacte avec une latence faible.

L'approche hiérarchique permet de faire du traitement à plusieurs niveaux. Ces niveaux peuvent avoir un rôle bien défini dans le traitement des données. C'est le

cas pour *HiFi* [Franklin 2005] présentant cinq niveaux de traitement des données. À chaque niveau correspond un type d'opération et à une localisation bien définie :

Nettoyage : Application de filtres sur les données pour enlever les anomalies. Traitement situé sur le capteur.

Lissage : Interpolation des mesures perdues. Utilisation d'agrégats sur les fenêtres. Traitement situé sur la station d'accueil des capteurs.

Arbitrage : Consolidation des données. Suppression de données dupliquées, agrégations des données nécessaires au niveau suivant. Traitement situé au niveau d'un bâtiment.

Validation : Applications de règles métiers pour valider les données. Utilisation principale de jointures. Traitement situé à un niveau régional.

Analyse : Support de décisions tels que vu dans la section 2.3. Traitement situé au centre de commandement.

Cette approche a été généralisée dans *SStreamWare* [Gürgen 2007] avec les *sites de contrôle* (plus haut niveau hiérarchique) et les *passerelles*. Ces dernières sont capables d'accueillir des capteurs (virtuels ou non), ainsi que d'exécuter une requête de flux sur les capteurs accueillis ou sur des flux externes venant d'une autre passerelle. Le site de contrôle est distribué et déclenche l'exécution des requêtes sur les différentes passerelles.

3.2.3 Intégration de systèmes de gestion de flux de données

Les implémentations de SGFD diffèrent en terme d'architecture, de modèle de données, de langage de requête et même de paradigme d'exécution. En ce sens, le système Exoengine [Duller 2011] virtualise les éléments qui composent un SGFD quelconque. L'implémentation de celui-ci se fait avec une approche orientée service. Ainsi, les divers composants représentant les résultats intermédiaires exposent deux interfaces de services, l'entrée et la sortie. Ces composants sont appelés *canaux*. Il devient désormais possible de relier naturellement les composants entre eux afin de gérer le flot des données formées par les *canaux* pour les unir, les chaîner ou les disperser. Ainsi, tout traitement est virtualisé par un *slet* qui possède un ou plusieurs ports en entrée pouvant se connecter à des *canaux* et fournit un port de sortie qui est connectée à un autre *canal*.

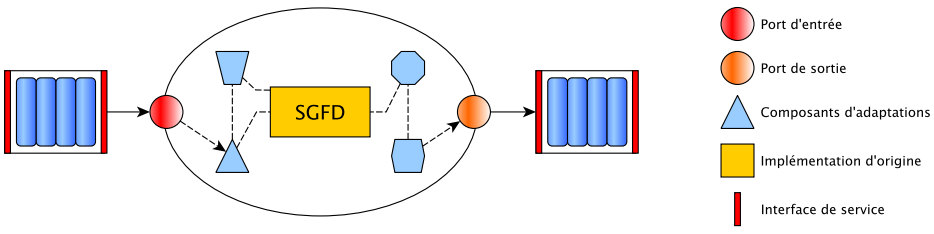


FIGURE 3.6 – Un *slet* virtuel permettant d'abstraire le fonctionnement d'un SGFD

La figure 3.6 montre un tel *slet* virtuel. L'intérêt d'une telle approche est non seulement de pouvoir gérer l'hétérogénéité des SGFD mais aussi de :

– Faciliter la distribution du système, en implémentant des canaux réseau, dont l'entrée est sur une machine et la sortie sur une autre. Il devient ainsi possible d'avoir une communication entre deux systèmes distribués. L'utilisation du même système sur plusieurs nœuds réseau permet la distribution du calcul bien que le SGFD originel ne le supporte pas.

– Convertir un système *pull* en système *push* et inversement. L'hétérogénéité des mécanismes de récupération des données est gérée grâce à l'abstraction de service, car les interfaces de services des canaux supportent les deux modes de fonctionnements.

– Utilisation de plusieurs modèles de données. Les canaux transmettent des objets. Les objets peuvent être la représentation de données relationnelles ou semi-structurées. Des *slets* de conversions permettent de faire fonctionner deux moteurs avec des données hétérogènes.

– Remplacement d'un composant à chaud. L'utilisation de l'approche à service permet aussi une bonne gestion du cycle de vie des composants. Ainsi, l'administration des requêtes et des composants (*slets* ou *canaux*) peut se faire pendant que le système est en fonctionnement.

Toutefois, l'intégration de SGFD n'est valable que d'un point de vue de l'infrastructure. En effet, la mise en œuvre entre les différents systèmes suppose que l'utilisateur est capable de traduire la sémantique de sa requête dans le langage utilisé par chaque système. Or, nous avons vu qu'il n'existe pas de langage universel permettant de traduire la sémantique de chacun.

3.3 Intégration de supports persistants à la gestion de flux

Les SGFD utilisent principalement l'interrogation continue, alors que les systèmes de gestions de données persistantes (SGBD par exemple) utilisent l'interrogation instantanée. Bien que les SGBD et SGFD ont des similarités dans leurs fondations théoriques, le problème réside dans la difficulté à combiner les deux types de requêtes. Dans cette section, nous analysons le problème des requêtes hybrides mélangeant flux et données persistantes. D'autres utilisations de supports persistants existent, par exemple la matérialisation des mémoires tampons [Abadi 2003]. Cette application n'est toutefois qu'un point particulier d'optimisation de traitement n'altérant pas la sémantique d'interrogation.

3.3.1 Requêtes dépendantes d'un contexte

Nous avons déjà vu les requêtes par désignation dans la section 3.2.2. La requête sous-jacente utilisée pour désigner les sources d'information est posée sur un ensemble de données de manière instantanée. L'ensemble des données pouvant être interrogé par la requête de désignation est assimilable à un contexte tel que présenté en section 2.2.

Les catalogues de meta-données sont souvent utilisés en tant que contexte dans les applications d'observation à base de flux. Ceci a été pratiqué dans les projets SStreamWare et plus récemment sur SmartCIS [Liu 2010]. Dans ce dernier, un SGBD relationnel est utilisé pour recenser les différents dispositifs logiques et physiques. Au moment du déploiement de la requête, l'optimiseur interroge cette base de données pour sélectionner les sources. D'un point de vue de la modélisation, les méta-données sont considérées comme des attributs spéciaux des flux. Lors de la réécriture de la requête, les attributs de méta-données sont identifiés pour effectuer une l'interrogation sur le catalogue. Il est intéressant de noter que ces méta-données ont une évolution lente.

Toutefois, le contexte peut être mis à jour. Une telle mise à jour influe sur l'évaluation de la requête. Par exemple, si l'utilisateur déploie la requête précédemment citée : « Flux des températures du bâtiment A » et qu'un nouveau capteur arrive dans le bâtiment. Doit-il être pris en considération ? Cette situation ouvre la voie au concept de transaction dans le cadre des requêtes continues. Le phénomène a d'abord été cerné et a été résolu par le protocole de mise à jour de SStreamWare [Gürgen 2006]. Si une mise à jour est effectuée, le protocole modifie les nœuds et les requêtes concernées pour changer le déploiement si nécessaire.

De façon proche, l'idée de transaction pour les requêtes continues a ensuite été présentée dans [Botan 2012]. Les transactions sont définies comme un ensemble restreint d'opérations de lectures ou d'écritures appliquées sur les flux ou relations avec un ordre d'exécution défini. De cette façon, l'exécution de requête continue est un ensemble de microtransactions, garantissant une exécution sans conflit. Il est notable que cette conception permette de réutiliser les transactions des SGBD.

D'un point de vue opérationnel, la désignation consiste en une semi-jointure (sur les identifiants typiquement) avec le résultat d'une requête sur un support persistant. Ainsi, soit le système applique une procédure de désignation comme vue précédemment, et ouvre la possibilité à des problèmes potentiels de sémantiques. Soit le système applique une opération de semi-jointure avec le support persistant qui a une vue à jour des données. Cependant, cette dernière mise en œuvre nécessite une uniformisation des langages de requêtes entre le support persistant et SGFD.

3.3.2 Analyses sur historique

La deuxième utilisation importante des supports persistants est l'ajout à un flux des données des analyses faites sur des historiques d'autres flux. Par exemple :

Flux des charges processeurs des différents équipements domestiques dont la charge actuelle est anormale.

Dans ce cas, afin de comparer la charge actuelle à une *charge normale*, il est nécessaire de calculer la moyenne historique à tout moment. L'implémentation d'une telle procédure doit considérer que les données persistantes sont constamment mises à jour et que l'opération d'agrégation soit coûteuse.

La première mise en œuvre de cette jointure avec agrégat sur l'historique a été faite dans les travaux joints de FastBit et TelegraphCQ [Reiss 2007]. Pour cela, deux requêtes sont spécifiées : celle exécutée sur les flux de données et celle exécutée sur

une base de données. Puis, un *contrôleur* exécute les deux requêtes séparément dans le but d'ajouter les données calculées sur la base de données aux n-uplets des flux.

Une opération plus explicite a été définie depuis dans *Moirae* [Balazinska 2007] : l'opérateur *Rappel* (*Recall*). Pour un identifiant en particulier, l'opérateur regroupe les agrégats d'un historique H. Afin d'absorber la charge, la gestion des *opérations de rappel* peut être distribuée. Enfin, la matérialisation de contextes plus généralement est aussi utilisée dans ce projet.

Il est intéressant de noter que ces requêtes peuvent être assimilées à des désignations sur des données de contextes consolidées par des analyses historiques.

3.3.3 Une première approche intégrée

Enfin, une approche unifiée a été présentée par Oracle [Witkowski 2007]. Cette extension du SGBD relationnel permet de manipuler les requêtes continues. Le même système intègre : flux, relations et historiques. Toutefois, le système manque d'un modèle commun pour formaliser les intégrations entre les deux paradigmes comme nous l'avons présenté précédemment. En effet, l'utilisateur doit décrire comment les vues matérialisées [7] doivent être créé et mises à jour.

Un point novateur est que le système explicite clairement différentes sémantiques de mises à jour de la vue des données persistantes utilisée par la requête continue. L'approche reste toutefois dirigée par les mécanismes et l'implémentation. Néanmoins, l'infrastructure permet d'exécuter les requêtes présentes dans cette section.

3.4 Optimisations de l'évaluation de requêtes

Dans le cadre des SGFD plusieurs aspects sont importants. En premier lieu, nous présentons les points spécifiques d'optimisation du traitement des flux qui viennent compléter les techniques de *load-shedding* présentées en 3.2.2. Enfin, nous détaillons les optimisations des algorithmes des opérateurs.

3.4.1 Optimisation du traitement des flux

L'infrastructure de traitement des flux présente des spécificités par rapport aux SGBD. Pour l'optimisation cela concerne : le mode de traitement des requêtes, le partage des requêtes, l'ordonnancement et enfin le routage pour les infrastructures distribués. Nous pouvons remarquer que tous ces aspects ont été explorés premièrement dans le domaine des systèmes de base de données. Toutefois, ils prennent plus d'ampleur dans ce contexte.

Calcul incrémental

Le calcul incrémental permet d'évaluer une requête continue grâce aux traitements effectués sur les changements des entités manipulées. Par exemple, pour une fenêtre

7. Les vues matérialisées servent de passerelles comme dans *Chronicle* (c.f. 3.1.1)

glissante dont le changement est d'un seul n-uplet à chaque évaluation, il n'est peut-être pas nécessaire d'appliquer les traitements sur la relation complète. Comme présenté dans la section 3.1, historiquement, les premiers traitements de flux [Terry 1992] étaient considérés comme des traitements particuliers sur les n-uplets qui ont été ajoutés à une relation.

De façon plus générale, en se plaçant dans l'algèbre *ACO* : Soit R une relation, au lieu de calculer une requête sur $R(\tau)$, il est possible de considérer les *delta* de cette relation : $\Delta_R^+(\tau) = R(\tau) - R(\tau-1)$ et $\Delta_R^-(\tau) = R(\tau-1) - R(\tau)$. Comme le traitement des fenêtres, par exemple, peut fournir directement ces différences : il n'y a pas de surcoût à l'utilisation d'un tel procédé. Comme la cardinalité des Δ est souvent minime face à celle de la relation totale. Il devient intéressant de travailler avec ces données.

Multi-Query Optimization (MQO)

Aussi connu sous le nom de *Global Query Optimization*, cette optimisation profite des exécutions parallèles pour éviter la redondance de traitement et une économie de ressources. Elle est issue du monde des SGBD [Sellis 1988] et permet de répondre à plusieurs requêtes en même temps en utilisant des parties communes (classiquement, lorsque seules les clauses de sélection sont différentes). L'idée est toutefois peu exploitée dans les SGBD, car il faut pouvoir soumettre plusieurs requêtes en même temps et les gestions de caches permettent d'obtenir de bons résultats pour ce genre de requêtes.

Dans le monde des flux de données, cette optimisation est largement reconnue comme importante. Les requêtes durent dans le temps, potentiellement indéfiniment. Le nombre de requêtes similaires peut être important. Ainsi, partager les ressources des requêtes devient un enjeu. Plusieurs points interviennent ici.

– Tout d'abord l'existence des *m-op* (multioperators) [Hong 2009]. Ces opérateurs permettent de regrouper plusieurs opérateurs en un seul permettant d'éviter des duplicatas de n-uplets. Un exemple classique est de grouper deux conditions de sélection sur un flux commun en une seule. Ainsi, si un n-uplet vérifie les deux conditions, un seul n-uplet est fourni avec l'indication qu'il appartient aux requêtes 1 et 2. Cette optimisation permet de faire des requêtes en utilisant les définitions de flux fragmentés.

– La grande disponibilité des ressources. Chaque opérateur utilise des ressources et peut potentiellement les partager avec d'autres. Cependant, ces ressources ne sont peut-être pas utiles à partager, car elles sont trop précises. Dans plusieurs travaux [Arasu 2004c], le calcul des agrégations sur fenêtres peut être partagé. En découpant la mémoire par bloc de façon adaptée, il est possible de partager les ressources afin d'économiser la mémoire.

Actuellement, les propositions de partage de plans de requêtes sont construites manuellement au déploiement de la requête. Seuls des travaux comme RUMOR [Hong 2009] recherchent à fusionner le nouveau plan de requête avec un autre (sous conditions encore très strictes). Un problème ouvert reste de savoir si les optimisations locales (optimisation algébrique) gênent les optimisations globales. Une requête est optimisée au début de son traitement, mais la stratégie d'exécution peut être modifiée pendant son évaluation. En effet, si une nouvelle requête arrive et qu'en

changeant légèrement la structure de la première, il est possible d'obtenir un partage, alors il est probable que ce nouveau plan soit le préféré. Ainsi, l'adaptabilité de l'exécution d'une requête est importante.

Parallélisation & Ordonnancement

Comme les requêtes peuvent être exécutées en parallèle, leur ordonnancement est important. Afin d'éviter les engorgements, ou pour donner plus de priorité à une partie de la requête, l'ordonnanceur doit cadencer les unités de traitement pour fournir des résultats conformes à la qualité attendue. Par exemple, une requête importante d'alerte peut exprimer des contraintes fortes de *latence*, contrairement à une requête d'observation passive. Le problème est que l'exécution d'une requête dans le cadre d'un SGFD n'est pas triviale, comme présenté est section 3.2. Ceci implique que la gestion des contraintes d'ordonnancement est plus complexe à mettre en œuvre. Plusieurs stratégies ont été proposées [Babcock 2003, Jiang 2004] afin de garantir le meilleur temps de réponse en utilisant le moins de ressources possible.

Placement des opérateurs & Routage

Le placement des opérateurs est important dans le cadre de l'évaluation distribuée d'une requête. Dans le cadre des topologies de réseaux complexes, cet aspect implique aussi des problèmes de routages. Pour le traitement en flux de grandes quantités de données, des optimisations de routage rentrent en compte sur le calcul de jointure par exemple [Zhou 2006, Palma 2009]. En effet, pour optimiser le plan de requête, il faut savoir s'il vaut mieux effectuer une opération lourde répartie sur plusieurs nœuds, avec des surcoûts de communication, ou centralisée, mais plus lourde en traitement. Dans le cadre des réseaux de capteurs, le coût d'un plan de requête (notamment énergétique) est analysé afin de fournir un routage efficace au moment du déploiement [Galpin 2009] ou à l'exécution [Madden 2005].

Nous avons vu les optimisations propres au traitement des requêtes dans le cadre de la gestion de flux. Nous détaillons maintenant les optimisations algorithmiques des opérateurs.

3.4.2 Optimisation des opérateurs

Tout comme dans les SGBD, les SGFD possèdent plusieurs implémentations et plusieurs algorithmes pour calculer des opérateurs. Nous détaillons dans cette section deux opérations qui ont reçu l'attention de la communauté du fait de leur usage fréquent dans la gestion de flux : la jointure et l'opérateur d'agrégation sur fenêtre.

Jointures

Comme présentée précédemment, la jointure en flux n'est pas aussi simple qu'une jointure de relations, car elle nécessite la gestion de fenêtres. Beaucoup de travaux [Han 2007, Srivastava 2004, Law 2007] se sont portés sur les jointures similaires à $I_S(S_1[W_1] \bowtie ... \bowtie S_n[W_n])$ et bien souvent limités par $W_1 = ... = W_n$. L'implémentation de ceci requiert a priori une quantité non bornée de mémoire. De plus, le traitement

d'un n-uplet peut nécessiter beaucoup du temps si les cardinalités sont grandes ce qui peut entrainer des congestions. Les travaux se concentrent sur des résultats approximatifs en établissant des modèles statistiques inspirés du *load-shedding* sur les attributs de jointures.

L'utilisation d'index, comme dans les SGBD, est désormais plus délicate, car les données sont constamment actualisées. Le point crucial de l'index est que le surcoût introduit au moment de l'insertion d'une donnée est amorti par le nombre de fois où la donnée est interrogée. Si les relations sont constamment mises à jour, ce surcoût a plus de difficultés à être amorti. Il est toutefois important de noter que les optimisations présentes dans le calcul de jointures relationnelles en mode *pipeline* [Gajski 1984] telles que le *symetric hash join* [Wilschut 1991] sont applicables aux requêtes continues. En effet, le mode *pipeline* est similaire au calcul incrémental dans le cadre des SGFD.

Fenêtrage et agrégations

Nous nous intéressons maintenant particulièrement aux traitements de requêtes d'agrégations de classe $I_S(G(S[W]))$. Les travaux les plus conséquents se focalisent sur le calcul approximatif des opérations de comptage et de quantile en mémoire limité [Arasu 2003]. Ce comptage permet d'effectuer les autres statistiques en quantité restreinte [Datar 2002], par exemple : minimums, maximums, sommes, moyennes, histogrammes et nombre de valeurs distinctes. L'approche est principalement mathématique et probabiliste. Par exemple, en se fixant une tolérance ε, il est possible d'obtenir un résultat dans une quantité de mémoire prévisible.

L'évaluation exacte et efficace de ces opérations reste possible grâce à l'utilisation de *Pane* [Li 2005]. Le principe est qu'une fenêtre sur 4 minutes décalée de 1 minute peut être vue comme 4 blocs de 1 minute. Ces blocs peuvent être agrégés. Ainsi, lors du décalage de la fenêtre, il devient possible de réutiliser les résultats de chacun de ces blocs pour calculer le résultat de la fenêtre entière. Afin de faire ces opérations, il est nécessaire de catégoriser les fonctions agrégations.

Les agrégats *holistiques* sont définis par deux fonctions L et S. Le calcul d'un tel agrégat F sur un ensemble X partitionné en blocs (X_n) est fait par :

$$F(X) = S(\{L(X_i), i \in \mathbb{N}\}).$$

Dans ce cas, il devient évident que le calcul de l'agrégat sur les fenêtres est fait par l'application de la fonction S sur les évaluations des blocs par L. Par exemple, l'agrégation *max* est *holistique* car les définitions $L = S = \max$ permettent ce calcul.

Les *différentiables* sont elles définies par trois fonctions H, J et L permettant la suppression et l'ajout de données dans l'agrégat :

$$\begin{cases} F(X - Y) &= H(L(Y), L(X)) \\ F(X \cup Y) &= J(L(Y), L(X)) \end{cases}$$

Par exemple, le comptage et la moyenne sont des agrégats *différentiables*. Le comptage peut être fait avec $L = count$, $H = -$ et $J = +$. Son application dans le cadre de l'évaluation de fenêtres est comme suit : reprise du résultat de la fenêtre précédente, soustraction du bloc qui a été enlevé de la fenêtre et ajout du nouveau bloc.

3.5 Conclusion

Après 20 années de recherche, la gestion de flux de données devient désormais suffisamment mature pour être appliquée massivement. Plusieurs produits commerciaux sont d'ailleurs maintenant utilisés en production. Toutefois, nous pouvons nous rendre compte que la complexité théorique de ces systèmes a été sous-estimée. De nombreux modèles ont été décrits pour représenter les flux de données et leurs traitements. Ces modèles sont encore remis en questions aujourd'hui au fur et à mesure du déploiement d'applications concrètes. Toutefois, les modèles sont bien souvent liés au système d'implémentation et certaines sémantiques d'exécutions ne sont pas claires. Ainsi, l'exécution de la même requête sur deux systèmes différents peut provoquer des interprétations divergentes.

Nous avons vu que l'infrastructure des SGFD a reçu beaucoup d'attention en terme d'architecture, de tolérance aux fautes, d'intégration et de support du passage à l'échelle. Néanmoins, la compréhension de la sémantique des composants du SGFD est encore limitée. Ainsi, il est difficile d'intégrer deux SGFD, car ils ne partagent pas le même langage et que l'interprétation des langages peut être conflictuelle.

L'intégration des supports persistants reste ad hoc et souvent assistée par l'utilisateur. Les expressions de requêtes hybrides entre flux et relations persistantes sont faites par l'écriture de deux requêtes pour chaque système. Il est nécessaire d'avoir un langage capable d'exprimer les deux types d'interrogations, mais aussi de faire la jointure entre les deux mondes.

Les contributions sur l'optimisation de traitement des requêtes sont nombreuses. Mais peu de travaux [Galpin 2009, Krämer 2009] proposent des optimisations de plan de requêtes similaires aux SGBD (optimisation logique puis physique). Dû au manque de connaissances sur les équivalences de requêtes, seules les règles simples telles l'application de projections au plus près des sources sont faites dans l'optimisation logique. Il est nécessaire d'avoir une *recherche d'optimisation plus approfondie* sans intervention de l'utilisateur, ce qui est limité en l'état.

Présentation des contributions

Cette thèse adresse le problème de l'observation de systèmes. Mais ces systèmes produisent des données hétérogènes en terme de schéma ainsi qu'en terme de dynamisme. Nous développons un système d'observation flexible afin de s'adapter aux besoins des utilisateurs.

Notre contribution se focalise sur trois axes :

Modélisation : Création d'Astral, algèbre de traitement des requêtes continues sur flux et relations temporelles. Il est à noter que les définitions sont indépendantes du système d'implémentation ce qui permet l'optimisation et la médiation de systèmes. Cette algèbre est présentée dans le chapitre 4. Son expressivité ainsi que la démonstration d'équivalences de requêtes sont présentées dans le chapitre 5.

Exécution : Mise en œuvre de l'intergiciel Astronef pour construire et exécuter efficacement les requêtes exprimées avec l'algèbre Astral. Ce moteur intègre un constructeur de plan de requête sélectionnant un assemblage de composants efficace pour exécuter une expression algébrique. Cette mise en œuvre est développée dans le chapitre 6.

Persistance : Conception de l'extension Asteroid permettant l'intégration des requêtes continues sur flux et des requêtes sur support relationnel persistant. Ceci permet de gérer la représentation du système observé ainsi que l'historisation des données dynamiques. Il devient possible de former des requêtes hybrides utilisant les données temps réel et persistantes. Le support formel de cette intégration est effectué par Astral et sa mise en œuvre par Astronef. Ces travaux sont présentés dans le chapitre 7.

Personnalisation : Proposition d'extension permettant d'adapter les résultats des requêtes en fonction de l'utilisateur. Ainsi, face à la masse de données auquel l'utilisateur est confronté, il sera en mesure de réduire le volume de données à restituer au regard de ses besoins et préférences. Cette personnalisation est formalisée en Astral et a été implémentée en tant qu'extension Asteroid. Le chapitre 10 présente cette proposition.

Ces contributions sont validées par une mise en œuvre sur un système réel dans le chapitre 8 et par des évaluations de performances dans le chapitre 9.

Grâce à l'ensemble de ces contributions, il devient possible de mettre en œuvre un système d'observation générique applicable sur un large ensemble de données. L'utilisateur exprime des requêtes dans le langage algébrique Astral. Les requêtes sont ensuite exécutées par Astronef-Asteroid.

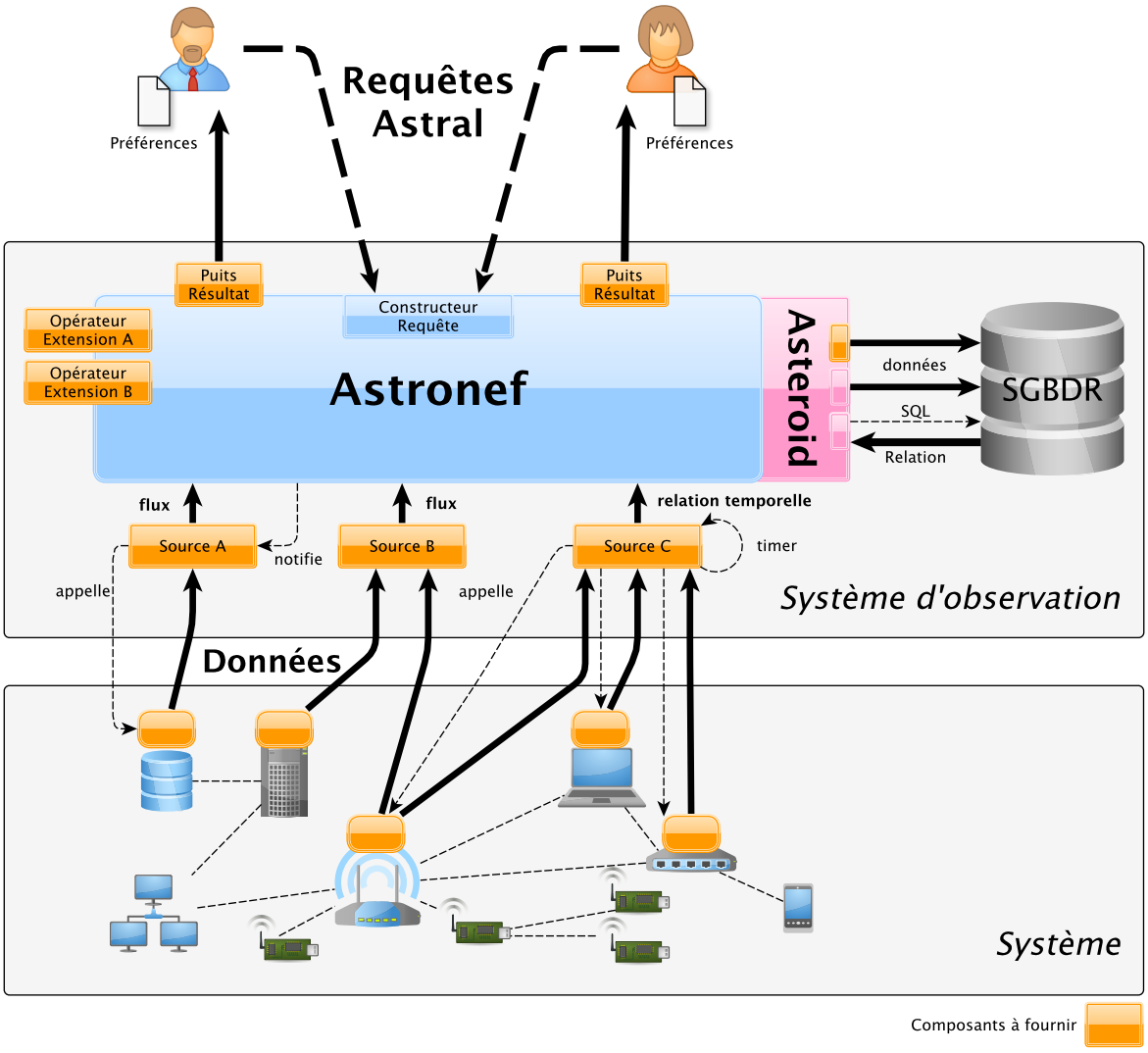


FIGURE 3.7 – Présentation des contributions de cette thèse

Partie II

Modèle algébrique

Pour manipuler toutes les données des systèmes observés, nous avons choisi de décrire un langage d'interrogation unique. Afin d'avoir des définitions précises des sémantiques des requêtes, nous avons décidé de faire le langage algébrique Astral. Dans cette partie, nous détaillons ce langage et ses capacités. Nous consacrons un chapitre à l'ensemble de ses définitions (des n-uplets jusqu'aux opérateurs). Puis, nous comparons cette algèbre avec les approches existantes et nous présentons quelques résultats d'équivalence de requêtes.

« Don't you use your fancy mathematics to muddle the issue! »

Applejack

4

Astral : Modèle théorique de requêtes continues

La gestion de flux de données est un domaine dérivé de la gestion de bases de données relationnelles. Afin de définir un langage de requête uniforme pour pouvoir interroger autant les flux que les relations, il est nécessaire de définir un modèle algébrique précis. Une grande importance est accordée au fait que les définitions soient séparées du système d'implémentation. Dans certaines formalisations, différentes évaluations d'une même requête peuvent retourner des résultats différents. Nous souhaitons formaliser exactement les bases de l'algèbre afin de pouvoir fédérer les modèles existants.

Nous développons dans ce chapitre les fondations de notre algèbre : *Astral* [1]. Cette algèbre reprend des bases théoriques développées dans *STREAM* [Arasu 2004a] (lui-même basé sur le modèle relationnel) tout en précisant des concepts clés pour en améliorer l'expressivité et la clarté. La section 4.1 définit les concepts primitifs tels que les n-uplets, les flux, relations ainsi que la notion de requête. Par la suite, nous présentons les opérateurs : tout d'abord ceux issus de l'algèbre relationnelle (section 4.2), puis ceux plus spécifiques propre à la gestion de flux (section 4.3). Enfin, nous analysons l'influence de l'instant de début d'une requête sur son fonctionnement en section 4.4.

1. *Advanced Stream Algebra*

Puis, nous concluons par une synthèse de ce modèle en section 4.5.

4.1 Définitions générales

Avant de présenter les définitions plus spécifiques à la gestion des flux de données, définissons les notions les plus élémentaires de notre algèbre. Nous présentons en premier les notions de n-uplets et d'identifiants, puis celles de flux et relations. Nous continuons par la présentation d'un exemple fil rouge, qui permet d'illustrer les définitions de ce chapitre. Enfin, nous présentons les notions fondamentales de l'algèbre : les ordres, les requêtes et les équivalences de requêtes.

4.1.1 N-uplets et identifiants

La définition 4.1.1 présente un n-uplet de manière similaire au modèle relationnel avec une fonction partielle.

Définition 4.1.1 **n-uplet**

Un n-uplet est une fonction partielle de l'ensemble des attributs vers l'espace des valeurs. Le domaine de cette fonction est appelé le schéma du n-uplet.

Comme toute fonction partielle, un n-uplet peut être représenté comme un ensemble de couples $Attribut \times Valeur$.

Ajoutons désormais la définition 4.1.2 décrivant l'identifiant physique d'un n-uplet. Cet attribut particulier a deux buts : identifier un n-uplet et définir la position (définition 4.1.4) dans une séquence de n-uplets (définition 4.1.3).

Définition 4.1.2 **Identifiant physique**

Nous appelons un Φ-espace, un ensemble dénombrable totalement ordonné.

L'identifiant physique d'un n-uplet s est l'attribut φ dont la valeur $s(\varphi)$ est un élément d'un Φ-espace.

La gestion des flux de données nécessite de plus le concept de *timestamp* (def 4.1.5) qui est associé aux n-uplets d'un flux. Nous adoptons un modèle continu pour le temps, car cela nous permet de ne pas faire d'hypothèse sur la plus courte distance entre deux *timestamps* consécutifs.

4.1.2 Flux et relations

Comme présentée dans le chapitre 3, la notion de batch est introduite pour gérer les n-uplets simultanés (égalité de timestamp). La définition 4.1.7 définit un flux de donnée comme une séquence de n-uplets organisés en une séquence de batchs. Tous

Définition 4.1.3 **Séquence de n-uplet**

Un ensemble dénombrable de n-uplets TS est une séquence si et seulement si :

- Tout n-uplet de TS partage le même schéma A contenant φ.
- L'ensemble $\mathbb{I}_{TS} = \{s(\varphi), s \in TS\}$ forme un Φ-espace.
- $\forall s, s' \in TS^2, s = s' \Leftrightarrow s(\varphi) = s'(\varphi)$

Une séquence est naturellement totalement ordonnée par son identifiant physique.

Définition 4.1.4 **Position d'un n-uplet**

La position d'un n-uplet s dans une séquence TS correspond au nombre de n-uplets ayant un identifiant physique plus petit que s.

$$\underset{TS}{\text{pos}}(s) = \#\{s' \in TS,\ s'(\varphi) \leq s(\varphi)\}$$

Définition 4.1.5 **Timestamp**

L'espace-temps $\mathbb{T}$ est un corps totalement ordonné, isomorphique à $\mathbb{R}$. Un *timestamp* est un élément de $\mathbb{T}$ et son attribut est noté τ.

les n-uplets simultanés sont partitionnés en *batchs*. Ceux-ci sont identifiés (def 4.1.6) par un entier identifiant le batch en plus du *timestamp*.

Définition 4.1.6 **Identifiant de batch**

Un identifiant de batch est un élément de l'ensemble $\mathbb{T} \times \mathbb{N}$ totalement ordonné par l'ordre naturel du produit cartésien.

Définition 4.1.7 **Flux**

Un flux est un couple $(S, \mathcal{B}_S)$ tel que :

- S est une séquence de n-uplets potentiellement infinie possédant un schéma contenant l'attribut spécial τ.
- $\mathcal{B}_S$ est une fonction $S \rightarrow \mathbb{T} \times \mathbb{N}$ définissant le batch d'appartenance d'un n-uplet

Par mesure de commodité, sauf mention contraire, nous écrivons S pour désigner le couple $(S, \mathcal{B}_S)$.

Notre algèbre définit les deux concepts comme présentés dans la section 3.1. Nous définissons une relation temporelle (def 4.1.8 comme une fonction associant le temps

à un ensemble de n-uplets. Dans notre cas, le temps est défini comme un identifiant de batch, et l'ensemble de n-uplets est en réalité une séquence. L'évaluation d'une

Définition 4.1.8 **Relation temporelle**

Une relation temporelle R est une fonction en escalier associant un identifiant de batch $(t, i) \in \mathbb{T} \times \mathbb{N}$ à une séquence de n-uplet $R(t, i)$

relation temporelle à un instant donnée est une séquence (ou *relation instantannée*) avec un ordre strict sur les n-uplets. Par abus de langage, nous utilisons le terme *relation* pour désigner une relation temporelle.

La définition des relations temporelles et des flux implique qu'il existe une quantité dénombrable de *batchs* différents dans le flux ou dans une relation temporelle. Ainsi, pour tout *batch* (t, i), il existe un autre *batch* proche de (t, i) sans y être égal même si t est assimilable à un réel [2]. Ce type de *batch* est noté $(t, i)^-$. Par exemple, pour une relation temporelle dont l'état change au *batch* b, nous pouvons comparer les états $R(b)$ et $R(b^-)$ correspondants à avant et après le changement.

Nous appelons **entité** tout objet pouvant être une relation ou un flux.

4.1.3 Exemple

Dans la suite de ce chapitre, nous développons plusieurs exemples. Ces exemples se basent sur les entités suivantes issues de l'application du réseau local domestique.

CPU Flux(appId, cpu, τ) : flux de relevé des charges processeurs. Un n-uplet de ce flux indique qu'au timestamp τ, l'application *appId* indique que sur son équipement hôte, la charge processeur est égale à *cpu*.

Applications Relation(appId, appName, deviceId) : catalogue des applications. Un n-uplet de cette relation indique que l'application *appId* appelée *appName* est déployée sur le l'équipement *deviceId*.

Devices Relation(deviceId, deviceName, deviceType, deviceStatus) : catalogue des équipements. Un n-uplet de cette relation indique que l'équipement *deviceId* appelé *deviceName* est de type *deviceType* et est actuellement dans le statut *deviceStatus* (1 ou 0 pour allumé et éteint).

4.1.4 Fondations de l'algèbre

Hypothèse fondamentale de l'ordre

La notion d'ordre est cruciale dans la gestion de flux de données. Nous avons introduit l'ordre naturel dans la séquence de n-uplet. Ceci est l'ordre positionnel. Mais pour un flux, nous pouvons aussi définir l'ordre des *batchs* et l'ordre temporel. La définition 4.1.9 spécifie l'expression formelle de ces ordres.

Nous supposons maintenant que nos flux sont correctement ordonnés avec l'hypothèse 4.1. Si un n-uplet est positionné avant un autre, alors son *batch* est inférieur ou égal. Et par conséquent son *timestamp* est lui aussi inférieur ou égal.

2. Au final, $(t, i)^- = (t, i - 1)$ si $i \neq 0$ et $(t^-, 0)$ sinon

Définition 4.1.9 **Ordres d'un flux**

Soit S un flux, trois ordres sont naturellement définis : $\forall s, s' \in S^2$,

$$\begin{array}{lrcl} \text{L'ordre positionnel :} & s \leq_\varphi s' & \Leftrightarrow & s(\varphi) \leq s'(\varphi) \\ \text{L'ordre des batch :} & s \leq_\mathcal{B} s' & \Leftrightarrow & \mathcal{B}_S(s) \leq \mathcal{B}_S(s') \\ \text{L'ordre temporel :} & s \leq_\tau s' & \Leftrightarrow & s(\tau) \leq s'(\tau) \end{array}$$

Hypothèse 4.1 **Cohérence temporelle**

L'ordre positionnel implique l'ordre des batchs.

$$\forall s, s' \in S^2, \qquad s <_\varphi s' \Rightarrow s \leq_\mathcal{B} s'$$

Requêtes

Tout d'abord, il est nécessaire de définir que dans la gestion de flux de données, une entité n'a pas d'existence avant la création de son instance à t_0. En effet, lorsqu'une requête est déployée sur un système, sauf opération explicite, le flux arrivant n'a pas d'informations sur les données passées. D'un point de vue algébrique, la définition 4.1.10 affirme qu'une entité est initialisée à un *timestamp* donné si et seulement si toutes ses données sont accessibles pour un *timestamp* supérieur à celui-ci.

Définition 4.1.10 **Entité initialisée**

Une entité E initialisée à un timestamp t_0 est une entité vérifiant :

- Si E est une relation : $\forall b \in \mathbb{T} \times \mathbb{N}$, tel que $b < (t_0, 0)$, alors $E(b) = \emptyset$.
- Si E est un flux : $\forall s \in E, s(\tau) \geq t_0$.

La notion de requête s'appuie directement sur cette notion d'entité initialisée. Ainsi, la requête (definition 4.1.11) est une fonction transformant ces entités initialisées en une nouvelle. Il est important de voir que cette fonction soit potentiellement dépendante du *timestamp* d'initialisation.

Définition 4.1.11 **Requête**

Soient $E = (E_1, ..., E_n)$, n entités,
Une requête continue sur E démarrée au temps t_0 est définie comme une fonction Q appliquée sur les éléments de E initialisés à t_0 et produisant une nouvelle entité. Cette requête est notée $(Q(E), t_0)$.

Équivalences de requêtes

Avec la définition 4.1.12, nous définissons l'inclusion de séquences de manière similaire à la notion de suites extraites. La notion d'**équivalence** de deux séquences est

quant à elle définie par la double inclusion.

Définition 4.1.12 **Inclusion et équivalences de séquences de n-uplets**

Soient TS_1 et TS_2 deux séquences possédant le même schéma A,
TS_1 est inclue dans TS_2 ($TS_1 \subseteqq TS_2$) si et seulement si :
$\exists f : \mathbb{I} \mapsto \mathbb{I}$ strictement croissante telle que

$$\forall s \in TS_1, \exists s' \in TS_2, \text{ tel que } \forall a \in A, s'(a) = \begin{cases} f(s(\varphi)) & \text{si } a = \varphi \\ s(a) & \text{sinon} \end{cases}$$

L'équivalence ($TS_1 \equiv TS_2$) est définie par $TS_1 \subseteqq TS_2$ et $TS_1 \supseteqq TS_2$.

Exemple 4.1 :
La table 4.1 représente quatre exemples de séquences d'une instance figée de la relation temporelle ***Device****. Ces séquences pourraient être équivalentes d'un point de vue algèbre relationnelle (en ne tenant pas compte de φ).*

(a)

φ	id	name	type
1	1	Livebox	Passerelle
5	4	hecate	PC
12	3	iPad	Tablette

(b)

φ	id	name	type
7	1	Livebox	Passerelle
9	4	hecate	PC
10	3	iPad	Tablette

(c)

φ	id	name	type
3	4	hecate	PC
6	1	Livebox	Passerelle
11	3	iPad	Tablette

(d)

φ	id	name	type
7	4	hecate	PC
9	1	Livebox	Passerelle
10	3	iPad	Tablette

TABLE 4.1 – Quatre séquences représentant une instance figée de **Device**

Les séquences (a) et (b) sont équivalentes, car leurs identifiants physiques sont différentes mais représentent le même ordre des n-uplets. Selon la définition, nous pouvons effectivement trouver une fonction f strictement croissante pour transformer les φ de (a) en (b) et inversement.

Les séquences (a) et (c) ne sont pas équivalentes, car l'ordre des n-uplets est inversé. Il est en effet impossible de trouver une transformation f de φ ***croissante*** *telle que l'égalité des n-uplets soit exacte. Nous remarquons que la valeur de φ n'a que peut d'importance, car même les séquences (b) et (d) ne sont pas équivalentes malgré l'égalité des identifiants. Bien évidemment, (c) et (d) sont équivalentes.*

La notion d'inclusion et d'équivalence s'étend naturellement aux flux et aux relations. Nous pouvons maintenant définir proprement la notion d'équivalence de requêtes comme l'équivalence de l'entité résultante quelques soient les entités d'entrées. Il est important de noter que l'équivalence suppose que les requêtes ont démarré au même *timestamp* t_0. La section 4.4 présente les conséquences du changement de *timestamp*.

Définition 4.1.13 **Équivalences de requêtes**

Soient $E = (E_1, ..., E_n)$, n entités,
Les requêtes Q et Q' démarrées à t_0 sont équivalentes si et seulement si pour toutes entités E' de mêmes types et schéma, les entités $Q(E')$ et $Q'(E')$ sont équivalentes.

Nous venons de poser les bases fondamentales pour notre algèbre de gestion de données. Il est nécessaire de construire des opérateurs pour pouvoir manipuler nos concepts. Nous allons voir que les contraintes que nous nous sommes fixées sur l'ordre, la nature du temps.

4.2 Héritages du modèle relationnel

La définition de relation temporelle que nous avons proposée repose sur la notion de séquence de n-uplets. Cette notion est certes proche des relations classiques, mais diffère par un point majeur : l'ordre. Dans cette section, nous voyons comment les opérateurs classiques de l'algèbre relationnelle peuvent être définis sur les relations temporelles.

4.2.1 Opérateurs unaires simples

Tout d'abord, explorons le domaine des opérateurs unaires relationnels : sélection, projection et renommage. Comme ces opérateurs sont indépendants de l'ordre des n-uplets, le principe est d'appliquer les définitions sur la relation temporelle à un instant donné.

Par exemple, notons la sélection relationnelle classique Σ. Alors, pour un batch b quelconque, l'expression suivante : $\Sigma(R(b))$, exprime bien la sélection des n-uplets. Ainsi, l'application de l'opérateur relationnel standard sur le batch présent permet de définir la sélection (def 4.2.1). L'identifiant physique n'est pas modifié par l'opérateur, ainsi l'ordre ne l'est pas non plus.

Définition 4.2.1 **Sélection**

Soit R une relation temporelle,
Soit c une expression booléenne applicable sur tout n-uplet de R,
Alors, la sélection sur une relation temporelle est définie comme suit :

$$\sigma_c(R) : b \mapsto \{s \in R(b), c(s)\} = \Sigma_c(R(b))$$

Nous pouvons remarquer d'ores et déjà que la définition d'inclusion de requête est directement applicable à la sélection (en prenant pour fonction d'extraction l'identité).

La projection et le renommage se définissent de façon similaire. Toutefois, il existe des cas pouvant modifier l'identifiant physique. Par exemple, la projection sur des

Proposition 4.2.1 **Inclusion de la sélection**

Soit R une relation temporelle, et c une condition de sélection, alors

$$\sigma_c R \subseteq R$$

attributs ne comprenant pas φ le supprimerait. Nous instaurons des règles supplémentaires (def 4.2.2) pour éviter ces cas. De façon similaire, nous pourrions définir l'opérateur d'évaluation d'expressions e_f^a permettant d'évaluer une expression f dont le résultat serait placé dans l'attribut a par exemple e_{x+y}^z.

Définition 4.2.2 **Projection et renommage**

La projection Π_p et le renommage $\rho_{b/a}$ sont défini par extension de l'algèbre relationnelle à l'exception de ces deux règles :

- Une projection Π_p est strictement égale à $\Pi_{p \cup \{\varphi\}}$
- Le renommage $\rho_{b/\varphi}$ correspond a une copie de φ dans b.

Nous avons réussi à appliquer les définitions des trois premiers opérateurs de l'algèbre relationnelle dans notre contexte. Il est nécessaire maintenant d'explorer les opérateurs binaires, en commençant par le produit cartésien.

4.2.2 Produit cartésien

La contrainte de l'ordre est plus délicate à gérer dans le cadre des opérations binaires. En effet, il est nécessaire d'établir un ordre strict sur la séquence de n-uplets résultants du produit des deux relations. Il est important de noter que cette notion de séquence de n-uplets est primordiale même pour les relations temporelles (voir notamment la définition 4.3.8 des *streamers*). Voyons désormais le problème sur un exemple concret.

Exemple 4.2 :

*Soit TS la relation groupant les 60 dernières secondes du flux **CPU**. Voici un exemple de données instantanées pour la relation temporelle **Applications** et de TS :*

	deviceId	*appId*
	1	*2*
TS	*2*	*23*
	3	*23*
	4	*12*

	appId	*cpu*	τ
	12	*12*	*21*
CPU	*2*	*11*	*32*
	2	*14*	*48*
	12	*13*	*54*

Supposons que l'utilisateur souhaite obtenir la charge cpu des équipements. L'opération demandée est une jointure entre ces deux relations. Toutefois, deux solutions sont envisageables.

	deviceId	*cpu*	τ
	1	*11*	*32*
Solution 1	*1*	*14*	*48*
	4	*12*	*21*
	4	*13*	*54*

	deviceId	*cpu*	τ
	4	*12*	*21*
Solution 2	*1*	*11*	*32*
	1	*14*	*48*
	4	*13*	*54*

Dans le premier cas, les n-uplets sont listés par dispositifs, puis par timestamp. Dans le second cas, les n-uplets sont listés par timestamp. Si ce résultat est transformé en flux, il peut y avoir des impacts sémantiques lourds : fenêtres positionnelles ou load-shedding différents. Mais de plus, le coût d'agrégation éventuel est lui aussi impacté (tri par groupement déjà effectué).

Ainsi, il est important de clarifier l'ambiguïté latente à la gestion de l'ordre dans les opérations binaires. Intéressons-nous au produit cartésien qui est au centre des opérations binaires les plus utilisées. L'utilisation de l'identifiant physique force la définition de l'ordre à tout niveau. Ainsi, la définition 4.2.3 du produit cartésien est similaire au produit classique nonobstant l'utilisation d'une application $\Phi^\times$ à définir permettant la création du nouvel identifiant.

Définition 4.2.3 **Produit Cartésien**

Soient R_1 et R_2 deux relations temporelles telles que $Attr(R_1) \cap Attr(R_2) = \{\varphi\}$, soit b un identifiant de *batch*,
Soient $\mathbb{I}^\times$ un Φ-espace et $\Phi^\times$ une application de $\mathbb{I}_{R_1} \times \mathbb{I}_{R_2}$ vers $\mathbb{I}^\times$,
Le produit cartésien de R_1 par R_2 au *batch* b est : $(R_1 \times R_2)(b) =$

$$\bigcup_{\substack{r \in R_1(b) \\ s \in R_2(b)}} \{(\varphi, \Phi^\times(r(\varphi), s(\varphi))) \cup r[Attr(R_1)\backslash\varphi] \cup s[Attr(R_2)\backslash\varphi]\}$$

Sauf mention contraire, dans Astral, nous considérons que

$$\Phi^\times : \varphi_1, \varphi_2 \mapsto (\varphi_1, \varphi_2) \in \mathbb{I}^\times = \mathbb{I}_{R_1} \times \mathbb{I}_{R_2}$$

avec $\mathbb{I}^\times$ lexicographiquement ordonné (d'abord R_1 puis R_2). Aucun critère évident ne permet d'affirmer que cette fonction est meilleure qu'une autre. Ce choix est dirigé par son caractère intuitif et par son analogie avec le comportement de l'algorithme usuel de boucles imbriqués (itération sur R_1 puis pour chaque n-uplet itération sur R_2). Les caractéristiques de cette fonction supplémentaire ont des implications concrètes sur les propriétés du produit cartésien comme : la propriété d'asymétrie (théorème 4.1).

Théorème 4.1 **Asymétrie du produit cartésien**

Le produit cartésien ne peut être symétrique dans le cadre général.

Démonstration du théorème 4.1 :

Tout Φ-espace est isomorphe à $\mathbb{N}$, par mesure de simplification, nous travaillons dans cet espace. Supposons qu'il existe un ordre total $<^2$ sur $\mathbb{N}^2$, qui de plus est conservateur par symétrique du couple d'entiers.

Soient $a, b \in \mathbb{N}^2$ tels que $a \neq b$. Puisque l'ordre est total, alors $(a, b) <^2 (b, a)$ (ou inversement). Puisque l'ordre est conservateur par symétrie du couple alors, $(b, a) <^2 (a, b)$ ce qui est absurde.

Ainsi, dans l'exemple 4.2.2, nous avions défini deux réponses à l'opération de jointure. Ces résultats correspondent aux opérations $Applications \bowtie TS$ et $TS \bowtie Applications$, ce qui illustre bien le problème d'asymétrie. Ce premier résultat est **important**, car le choix de l'ordre des jointures est déterminant pour l'optimisation de requête. Toutefois, il est possible de redéfinir les ordres de jointures pour deux produits cartésiens $\times^1$ et $\times^2$ et obtenir $R_1 \times^1 R_2 = R_2 \times^2 R_1$ mais la définition de chacun des produits cartésiens n'est pas la même du fait d'un choix de $\Phi^\times$ différent. En pratique, cela peut se concrétiser par un tri a posteriori, ce qui peut introduire un surcout.

Nous notons que la jointure dite naturelle $\bowtie$ entre deux relations temporelles est définie comme le produit cartésien avec sélection sur l'égalité des attributs communs. Toute jointure est un opérateur composite centré sur le produit cartésien et sur des projections-renommage-sélection.

4.2.3 Union

La définition 4.2.4 de l'union de relations temporelles est aussi complexe à cause de l'identifiant physique. En effet, lors de l'union de deux séquences de n-uplets, il n'est pas directement possible d'extraire une nouvelle séquence. Il est nécessaire réécrire sa définition. Le principe réside encore une fois dans une application particulière de réécriture $\Phi^\cup$ qui a pour but de réordonner.

Définition 4.2.4 **Union relationnelle**

Soient R_1 et R_2 deux relations temporelles avec le même schéma A,
Soient $\mathbb{I}^\cup$ un Φ-espace et $\Phi^\cup$ une application de $(\mathbb{I}_{R_1} \cup \{\emptyset\}) \times (\{\emptyset\} \cup \mathbb{I}_{R_2})$ vers $\mathbb{I}^\cup$, soit b un identifiant de *batch*,
L'union de R_1 et R_2 au *batch* b est définie par : $(R_1 \times R_2)(b) =$

$$\bigcup_{r \in R_1(b)} \{r[A \backslash \varphi] \cup (\varphi, \Phi^\cup(r(\varphi), \emptyset)\}$$
$$\bigcup_{s \in R_2(b)} \{s[A \backslash \varphi] \cup (\varphi, \Phi^\cup(\emptyset, s(\varphi))\}$$

Deux sémantiques principales peuvent s'appliquer dans le cadre de l'union. Tout d'abord, la sémantique générique que nous appliquons par défaut dans l'algèbre Astral : nous sélectionnons d'abord les n-uplets de la séquence de gauche et ensuite ceux de la séquence de droite. $\mathbb{I}^\cup$ est égal à $(\mathbb{I}_{R_1} \cup \{\emptyset\}) \times (\mathbb{I}_{R_2} \cup \{\emptyset\})$ avec un ordre naturel lexicographique ($\emptyset$ est la valeur la plus petite possible) et l'application $\Phi^\cup$ est définie par :

$$\Phi^\cup(\varphi_1, \varphi_2) = (\varphi_1, \varphi_2)$$

Nous remarquons que cette définition de l'union est elle aussi **asymétrique**, car ce n'est pas une simple union ensembliste. Dans certains cas, il est possible d'avoir une union symétrique. Si l'union de $\mathbb{I}_{R_1}$ et $\mathbb{I}_{R_2}$ forme naturellement un Φ-espace $\mathbb{I}$ et que pour tout *batch* b, $R_1(b)$ et $R_2(b)$ ne partagent pas d'identifiants physiques, alors il est possible de définir une union naturelle qui conserve les identifiants physiques :

$$\Phi^{\cup}(\varphi_1, \varphi_2) = \begin{cases} \varphi_1 & \text{, si } \varphi_1 \neq \emptyset \\ \varphi_2 & \text{, si } \varphi_2 \neq \emptyset \end{cases}$$

Ce cas se présente souvent lorsqu'une entité est partagée en multiples sous-entités (souvent appelé partitionnement). Ainsi, les identifiants physiques proviennent du même Φ-espace et sont répartis sur plusieurs relations dont l'union est naturelle. Nous remarquons l'importance de ces définitions lors de la définition des fenêtres partitionnées en section 4.3.

Note : L'opérateur de différence entre deux relations temporelles est délicat, car pour être capable de retrancher un n-uplet d'une séquence, il est nécessaire de l'identifier exactement. L'identifiant physique φ est effectivement présent pour ce point. La différence est une différence **ensembliste** pure. Toutefois, il est nécessaire de correctement gérer les identifiants physiques de la séquence à retrancher pour que les Φ-espaces des deux séquences soient identiques et que les identifiants soient pertinents. Par exemple, $(R_1 \cup R_2) - R_2$ donne $R_1 \cup R_2$, car les identifiants sont de natures différentes. Par contre, $(R_1 \cup R_2) - (\Omega \cup R_2) = R_1$ (avec Ω la relation temporelle vide).

4.2.4 Agrégation

L'agrégation est une opération qui n'a pas été définie dans l'algèbre relationnelle classique. Toutefois, vu son utilisation fréquente notamment dans le contexte des flux de données, il nous semble pertinent d'en exposer la définition précise. L'opération consiste en l'application de fonctions d'agrégations décrites dans la définition 4.2.5 sur des sous-groupes formés grâce à un regroupement par attributs égaux. Par exemple, la moyenne des valeurs de charge processeur calculée pour chaque identifiant d'équipement.

Définition 4.2.5 **Fonction d'agrégation**

Une fonction d'agrégation f est une application associant : une séquence de n-uplet S et un attribut A à une valeur agrégée $f(S, A)$.

Exemple 4.3 :

La fonction d'agrégation de moyenne (avg) *est définie par*

$$\text{avg}(S, A) = \frac{\sum_{s \in S} s(A)}{\#S}$$

À nouveau, il convient de définir un nouvel identifiant physique. Celui-ci est défini par un agrégat particulier aux séquences de n-uplets :

$$\text{last}(S, A) = s(A) \qquad \text{avec } s \in S \text{ tel que } \underset{S}{\text{pos}}(s) = \#S - 1$$

Nous pouvons désormais écrire la définition 4.2.6 de l'opérateur d'agrégation comme l'application de fonctions d'agrégations aux sous-groupes créés et dont l'identifiant physique est calculé par l'application de la fonction max.

Définition 4.2.6 **Opérateur d'agrégation**

Soit R une relation temporelle de schéma A,
Soient $a_1, ..., a_n$, n attributs de A,
Soient $f_1, ..., f_m$, m fonctions d'agrégats, $b_1, ..., b_m$, m attributs de A, et $c_1, ..., c_m$, m attributs,
L'opérateur d'agrégat ${}_{a_1,...,a_n}\mathcal{G}_{f_1{}^{c_1}_{b_1},...,f_m{}^{c_m}_{b_m}}$ au **batch** b est égal à :

$$\{g(\sigma_{a_1=v_1,...,a_1=v_n}R(t,i), v_1, ..., v_n), v_1 \in \text{Dom}(a_1), ..., v_n \in \text{Dom}(a_n)\}$$

$$\text{avec } g(S, v_1, ..., v_n) = \{(\varphi, \max(S, \varphi))\} \cup_{i=1}^{n} \{(a_i, v_i)\} \cup_{i=1}^{m} \{(c_i, f_i(S, b_i))\}$$
$$\text{si } A \neq \emptyset, \emptyset \text{ sinon}$$

Exemple 4.4 :
La requête continue calculant les valeurs moyennes sur la relation temporelle CPU est ${}_{id}\mathcal{G}_{\text{avg}_{cpu}^{moyenne}}(CPU)$.

Nous avons adapté tous les opérateurs classiquement définis dans l'algèbre relationnelle pour être applicables dans notre formalisme. Nous allons désormais développer les opérateurs applicables sur les flux.

4.3 Opérateurs de flux

Astral repose sur deux concepts : les flux et les relations temporelles. Ainsi, il est nécessaire définir les opérateurs de flux vers relation (fenêtres) et de relation vers flux (streamers). Puis, nous définissons des opérateurs spécifiques : la gestion des modifications des relations temporelles et des *batchs*.

4.3.1 Fenêtres

Comme nous l'avons vu dans la section 3.1, l'opérateur de fenêtre est un des opérateurs les plus étudiés dans la littérature. Toutefois, son comportement est encore flou sur certains points. La formalisation de son fonctionnement permet une meilleure spécification.

Association position-*batch*

Avant de définir formellement l'opération de fenêtrage, nous avons besoin d'un outil pour gérer l'association entre la position d'un n-uplet et de son *batch*. La fonction τ_S définit cette association (def 4.3.1).

Définition 4.3.1 **Fonction position-*batch***

Soit S un flux,
La fonction $\tau_S : \mathbb{N} \to \mathbb{T} \times \mathbb{N}$ est la fonction qui par une position (dans $\mathbb{N}$) donne l'identifiant du batch (dans $\mathbb{T} \times \mathbb{N}$) du seul n-uplet ayant cette position.
Par convention, $\tau_S(0) = (t_0, 0)$.

En corollaire de l'hypothèse des ordres 4.1, la fonction τ_S est croissante non-stricte. Ainsi, il est possible de définir une pseudo inverse (cor 4.1) τ_S^{-1} capable de donner une position (la maximale) pour un *batch* donné.

Corollaire 4.1 **Fonction pseudo-inverse de τ**

Soit S un flux,
La pseudo-inverse $\tau_S^{-1} : \mathbb{T} \times \mathbb{N} \to \mathbb{N}$ existe et correspond à la plus grande position du *batch* donné en entrée. Si aucun *batch* n'existe, le plus proche est utilisé. Formellement,

$$\forall b \in \mathbb{T} \times \mathbb{N}, \qquad \tau_S^{-1}(b) = \sum_{n=0}^{+\infty} n \mathbb{1}_{[\tau_S(n), \tau_S(n+1)[}(b)$$

Description de séquences de fenêtres

Afin de se rapprocher le plus possible d'un aspect déclaratif, il est nécessaire de décomposer l'opérateur de fenêtre en deux objets mathématiques : la description de son évolution et la séquence de fenêtres. Cette dernière prend une description en argument pour représenter la relation temporelle résultante. Le principe des descriptions de séquences de fenêtres tel que décrit dans la définition 4.3.2) est assez simple puisqu'il suffit de décrire deux bornes (α et β) évoluant de manière discrète ainsi qu'un taux d'évaluation de ces bornes (r).

Exemple 4.5 :
Nous souhaitons connaître tous les 100 *relevés de charge processeur, les* 10 *derniers relevés. Dans ce cas, nous souhaitons obtenir une séquence de fenêtres positionnelles générées tous les* 100 *n-uplets* ($r = 100 \in \mathbb{N}$). *Nous appliquons des bornes positionnelles* $\alpha, \beta \in (\mathbb{N} \to \mathbb{N})^2$. *La première fenêtre contient du* $91^{ème}$ *n-uplet au* $100^{ème}$. *Ainsi :* $\alpha(0) = 91$ *et* $\beta(0) = 100$. *Sachant que l'évolution des bornes est linéaire, nous*

Définition 4.3.2 **Description de Séquence de Fenêtre (DSF)**

Soient $\mathcal{D}$ et $\mathcal{D}'$ pouvant être $\mathbb{T}$ ou $\mathbb{N}$, une description de séquence de fenêtre (DSF) est un triplet (α, β, r) tel que :

– $r \in \mathcal{D}$ est le taux d'évaluation des bornes de la fenêtre

– α et β sont deux fonctions de $\mathbb{N} \rightarrow D'$ représentant l'évolution des bornes.

$\alpha(j)$ et $\beta(j)$ définissent les $j^{\text{ème}}$ valeurs des bornes. La première borne est donnée pour $j = 0$. Ces fonctions se doivent de vérifier les propriétés suivantes (en considérant $\mathcal{D} = \mathcal{D}' = \mathbb{T}$) :

$$\forall j \in \mathbb{N}, \begin{cases} \alpha(j) \leq \beta(j) & \text{le début est avant la fin} \\ \alpha(j) \geq t_0 & \text{le début existe} \\ \beta(j) \leq jr + \beta(0) & \text{la fin est accessible} \end{cases}$$

Les conditions pour les autres cas pour $\mathcal{D}$ et $\mathcal{D}'$ se déduisent par application des fonctions τ_S et τ_S^{-1}.

avons :

$$\begin{aligned} \alpha(j) &= 100j + 91 \\ \beta(j) &= 100j + 100 \\ r &= 100 \end{aligned}$$

La création de fenêtres nécessite d'associer les n-uplets du flux et le numéro de fenêtre décrit dans la *DSF*. Pour cela, nous définissons en 4.3.3 une *fonction d'attente* utilisant les identifiants de *batchs*. Cette fonction donne le rang de la dernière fenêtre complète au moment du batch. Le terme *attente* est lié au fait que l'évaluateur de fenêtre doit attendre le prochain changement de γ. Nous retrouvons dans cette fonction le caractère *bloquant* des fenêtres.

Définition 4.3.3 **Fonction d'attente γ**

Soit S un flux, soit (α, β, r) une DSF,
La fonction d'attente de la DSF est une fonction $\mathbb{T} \times \mathbb{N} \rightarrow \mathbb{N}$ permettant de trouver pour un identifiant de *batch*, l'identifiant de la dernière fenêtre complétée.

– Si $r \in \mathbb{T}$, cette fonction est définie par $\gamma : (t, i) \mapsto \left\lfloor \frac{t-\beta(0)}{r} \right\rfloor$.

– Si $r \in \mathbb{N}$, cette fonction est définie par $\gamma : (t, i) \mapsto \left\lfloor \frac{\tau_S^{-1}(t,i)-\beta(0)}{r} \right\rfloor$.

Exemple 4.6 :

En reprenant l'exemple précédent, après simplification nous obtenons :

$$\gamma(b) = \left\lfloor \frac{\tau_S^{-1}(b)}{100} \right\rfloor - 1.$$

Si nous supposons que le flux produit un n-uplet par seconde (ainsi, $\tau_S^{-1}(t,i) = \lfloor t/1s \rfloor$) : alors $\gamma(1024s, 0) = \left\lfloor \frac{1024}{100} \right\rfloor - 1 = 9$. Nous avons bien la $10^{ème}$ fenêtre ($j = 9$) comme la dernière fenêtre créée à l'instant 1024.

L'opérateur

Il devient désormais possible de définir un opérateur permettant de générer une relation temporelle à partir d'un flux donné. Cette relation temporelle change d'état avec l'instant défini par la fonction γ. De manière générale, une DSF peut être ramenée à une expression plus générale (α, β, γ) ce que nous utilisons pour la définition 4.3.4 de séquence de fenêtres.

Définition 4.3.4 **Opérateur de Séquence de Fenêtres**

Soit S un flux et (α, β, γ) une description de séquence,
L'opérateur de séquence de fenêtres est défini par : $\forall (t,i) \in \mathbb{T} \times \mathbb{N}$,

- Si $\gamma(t,i) \geq 0$,
  - Si la description possède des bornes temporelles :

$$S[\alpha,\beta,\gamma](t,i) = \{s \in S,\ (\alpha(\gamma(t,i)),0) \leq \mathcal{B}_S(s) \leq (\beta(\gamma(t,i)),i)\}$$

  - Si la description possède des bornes positions :

$$E(t,i) = \{s \in S,\ \tau_S(\alpha(\gamma(t,i))) \leq \mathcal{B}_S(s) \leq \tau_S(\beta(\gamma(t,i)))\}$$

$$S[\alpha,\beta,\gamma](t,i) = \{s \in E(t,i) / (\#E(t,i) - \underset{E(t,i)}{\text{pos}}(s)) \leq \beta(\gamma(t,i)) - \alpha(\gamma(t,i))\}$$

- Si $\gamma(t,i) < 0$ alors $S[\alpha,\beta,\gamma](t,i) = \emptyset$

Plusieurs remarques peuvent être formulées sur cette définition. Tout d'abord, les expressions sont différentes si les bornes sont positionnelles ou temporelles. Pour les fenêtres temporelles, l'opérateur inclut les n-uplets dont l'identifiant de *batch* s'étend :

depuis le premier *batch* de la fenêtre : $(\alpha(\gamma(t,i)),0)$, i.e. ceux dont le *timestamp* est supérieur à la borne inférieure.

jusqu'au dernier *batch* de la fenêtre : $(\beta(\gamma(t,i)),i)$. Ce qui correspond au $i^{ème}$ *batch* ayant le *timestamp* inférieur ou égal à la borne.

Il est important de voir que $S[\alpha, \beta, \gamma]$ peut changer à l'arrivée d'un nouveau *batch*, même si le *timestamp* ne change pas. Ne pas inclure ces modifications ferait perdre des données à la fenêtre. Nous retrouvons les problématiques explorées dans la section 3.1.

Pour les fenêtres positionnelles, la gestion est plus délicate. Si nous considérons que le flux réparti ses *batchs* (un n-uplet par *batch*), alors $E(b) = S[\alpha, \beta, \gamma](b)$. Mais dans le cadre général, $E(b)$ contient l'ensemble des n-uplets potentiels et la séquence $S[\alpha, \beta, \gamma](b)$ en est un sous-ensemble dont la taille est exactement celle décrite dans la DSF (sélections des n-uplets les plus récents). De plus, nous remarquons que γ en positionnel est défini en fonction de τ_S^{-1} qui fournit la position maximale en cas d'égalité de *batchs*, ce qui nous garanti de couvrir l'ensemble des n-uplets concernés.

Exemple 4.7 :

La figure 4.1 montre l'évolution d'une séquence où la fenêtre glisse de 2 secondes toutes les 2 secondes ($r = 2$) avec une taille constante de 3 secondes. $t_0 = 0$ par simplicité ici. La première fenêtre possède les bornes $\alpha(0) = t_0 + 0s$ et $\beta(0) = t_0 + 3s$. Sachant que le glissement est de $2s$ la description de fenêtre est

$$\forall j \in \mathbb{N}, \begin{cases} \alpha(j) & = j * 2s + t_0 \\ \beta(j) & = j * 2s + 3s + t_0 \end{cases}$$

La relation temporelle générée par cette DSF peut être notée $S[2js, 2js + 3s, 2s]$. Le calcul de son état à un instant est simple. Prenons le batch $(t_0 + 5.5s, 0)$. La fenêtre à calculer est la fenêtre numérotée $\gamma(t_0 + 5.5s, 0) = \left\lfloor \frac{t_0 + 5.5s - \beta(0)}{r} \right\rfloor = 1$. Ainsi : $S[2js, 2js + 3s, 2s](t_0 + 5.5s, 0) = F_1 = \{s_4, s_5, s_6, s_7, s_8\}$.

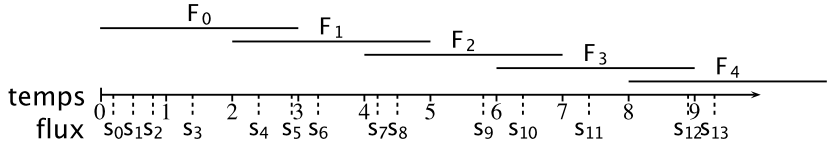


FIGURE 4.1 – Séquence de fenêtre de taille $3s$ glissante de $2s$

Définition 4.3.5 **Exclusions de bornes de fenêtres**

Soit S un flux et (α, β, γ) une description de séquence de fenêtre,
Les notations $S]\alpha, \beta, \gamma]$, $S[\alpha, \beta, \gamma[$ et $S]\alpha, \beta, \gamma[$ désignent les définitions de l'opérateur classique de séquence de fenêtre permettant d'exclure respectivement les bornes inférieures, supérieures ou les deux.

Exemple 4.8 :

En reprenant la définition de fenêtre sur 5 secondes : $(j * 5s + t_0, j * 5s + 5s + t_0, 5s)$. Nous remarquons que les fenêtres 0 et 1 contiennent les n-uplets dont le timestamp est égal à $t_0 + 5s$. Ainsi, l'opérateur $S]jr + t_0, jr + r + t_0, r]$ avec $r = 5s$ permet de retirer les n-uplets à ce timestamp de cette fenêtre.

Fenêtres partitionnées

L'opérateur de fenêtres partitionnées est très utilisé pour appliquer la même séquence de fenêtres à des sous-flux. Les opérateurs partitionnés sont tous décrits de la même manière. Le principe, illustré dans la figure 4.2 est de diviser le flux suivant un (ou des) attribut A donné. Sur chacun de ces sous-flux est appliqué un opérateur quelconque. Le résultat est l'union des résultats de ces opérateurs.

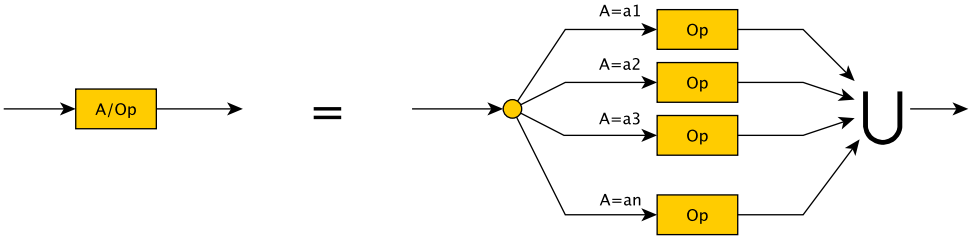


FIGURE 4.2 – Principe d'un opérateur partitionné

Ainsi, nous décrivons dans la définition 4.3.6 de séquence de fenêtre partitionnée l'application de ces principes de partitionnement sur l'opérateur de séquence de fenêtre.

Définition 4.3.6 **Séquence de fenêtre partitionnée**

Soient S un flux, $a_1, ..., a_k$ un ensemble d'attributs du schéma de S, et (α, β, γ) une DSF,
Soit $\cup^*$ l'union relationnelle conservant les identifiants physiques,
Alors, la séquence de fenêtres (α, β, γ) partitionnées par $a_1, ..., a_k$ est définie par :

$$S[a_1...a_k/\alpha, \beta, \gamma] = \bigcup_{a \in Dom(a_1,...,a_k)}^{*} (\sigma_{(a_1,...,a_k)=a} S)[\alpha, \beta, \gamma]$$

Nous remarquons que nous utilisons la définition de l'union relationnelle conservant les identifiants physiques présentés dans la section précédente, ainsi l'ordre naturel décrit dans le flux d'entrée peut être retrouvé dans la relation de sortie.

Exemple 4.9 :

L'exemple le plus courant est la représentation de l'état actuel d'un système à partir d'un flux. Supposons un flux d'entrée $CPU(appId, cpu, \tau)$, nous donnant les relevés de charge de processeur. Soit la description de fenêtre rapportant le dernier n-uplet d'un flux : $(1j, 1j, 1)$. Nous pouvons obtenir la relation temporelle représentant pour chaque application $appId$, la dernière valeur connue de cpu ainsi que le timestamp τ de mesure :

$$CPU[appId/1j, 1j, 1]$$

*Cet exemple illustre comment nous pouvons passer d'un flux brut à une représentation (dynamique) d'un **contexte**.*

La définition du partitionnement par l'union conservant les identifiants physiques permet d'avoir un ordre clair et intuitif sur la relation temporelle résultante. Lorsqu'un nouveau n-uplet entre dans cette relation (ou remplace l'ancien n-uplet), alors il est mis en bout. Avec l'union classique, il est nécessaire de définir l'ordre d'application des unions. Et les n-uplets auraient changé d'identifiants à chaque nouvel état de la fenêtre, ce qui est difficile à contrôler et à manipuler.

Par la suite, plusieurs notations simplifiées sont utilisées pour désigner des descriptions de fenêtres courantes décrites dans le tableau 4.2. Notons que les équivalences présentées dans ce tableau sont non triviales et sont démontrées dans l'annexe A.

	Définition	Équivalence
$[L]$	$[j, j, 1]$ La séquence de fenêtre où chaque fenêtre ne contient que le dernier n-uplet du flux. Cette séquence est égale à $[B]$ si le flux répartit ses n-uplets avec un n-uplet par *batch*.	
$[B]$	$]\tau_S^{-1}(\tau_S(j)^-), j, 1]$ Séquence de fenêtre où chaque fenêtre contient le dernier *batch*.	$\{s \in S, \mathcal{B}_S(s) = \tau_S \circ \tau_S^{-1}(b)\}$
$[\infty]$	$[0, j, 1]$ Séquence cumulative contenant tout le flux jusqu'au *batch* courant.	$\{s \in S, \mathcal{B}_S(s) \leq b\}$
$[T\ r\ s]$	$]\max(sj - r + t_0, t_0), sj + t_0, s]$ Fenêtre temporelle de taille r se déplaçant toutes les s unités de temps.	
$[P\ r\ s]$	$]\max(sj - r, 0), sj, r]$ Fenêtre positionnelle de taille r n-uplets se déplaçant tous les s n-uplets.	

TABLE 4.2 – Liste des fenêtres courantes

4.3.2 Streamers

La classe des *streamers* est l'ensemble des opérateurs produisant un flux à partir d'une relation. Son utilisation, après l'application d'une fenêtre, permet la création d'un flux résultant. La définition des streamers est découpée en deux parties, d'abord la définition de la réécriture de *timestamp* qui permet d'ajouter le *timestamp* aux données produites, et ensuite la définition de l'opérateur.

Réécriture de *timestamp*

Il est nécessaire de gérer les identifiants physiques. En effet, si un n-uplet est envoyé plusieurs fois dans le flux résultat, il y a conflit si φ n'est pas modifié. De plus, tout flux doit avoir un attribut τ. Cet attribut doit être créé ou remplacé pour assurer l'hypothèse des ordres. Le principe est que le n-uplet $s \in R(b)$ doit être envoyé dans le flux au *batch* (t_s, i_s), alors nous devons envoyer le n-uplet $\Psi_b(s, t_s)$ décrit dans la définition 4.3.7.

Définition 4.3.7 **Fonction de réécriture de *timestamp***

Soit R une relation de schéma A, b un batch,
Soit $\mathbb{I}^S = \mathbb{T} \times \mathbb{N} \times \mathbb{I}_R$ le Φ-espace ordonné de manière naturelle,
La fonction de réécriture de *timestamp* appliquée à $R(b)$ est définie par :

$$\forall s \in R(b),\ \forall t_s \in \mathbb{T},\ \Psi_b(s, t_s) = \{(\tau, t_s), (\varphi, (b, s(\varphi)))\} \bigcup_{a \in A \setminus \{\varphi, \tau\}} \{(a, s(a))\}$$

Cette définition assure que les nouveaux n-uplets du flux vérifient les propriétés suivantes :

– Ils ont pour *timestamp* t_s le moment où il a été estampillé.

– Ils ont un identifiant physique (et du coup une position) plus grand que les identifiants précédents.

– L'ordre positionnel de $R(b)$ est préservé.

Nous pouvons définir des *streamers* dit sensibles dans la définition 4.3.8). Ces *streamers* réagissent aux changements de R et produisent un flux en conséquence.

Définition 4.3.8 ***Streamers* sensibles**

Soit R une relation,
Les *streamers* sensibles sont les opérateurs créateurs d'un flux S défini à partir des changements de R, trois types sont définis :

– Le *streamer* d'insertion, envoyant les nouveaux n-uplets : $\mathcal{I}_{\mathcal{S}}(R)$,

$$s \in R(t, i) \wedge s \notin R((t, i)^-) \Leftrightarrow s' = \Psi_{(t,i)}(s, t) \in S \wedge \mathcal{B}_S(s') = (t, i)$$

– Le *streamer* de suppression, envoyant les n-uplets disparus : $\mathcal{D}_{\mathcal{S}}(R)$,

$$s \notin R(t, i) \wedge s \in R((t, i)^-) \Leftrightarrow s' = \Psi_{(t,i)^-}(s, t) \in S \wedge \mathcal{B}_S(s') = (t, i)$$

– Le *streamer* d'envoi, envoyant tous les n-uplets présents : $\mathcal{R}^u_{\mathcal{S}}(R)$,

$$s \in R(t, i) \neq R((t, i)^-) \Leftrightarrow s' = \Psi_{(t,i)}(s, t) \in S \wedge \mathcal{B}_S(s') = (t, i)$$

Nous avons constitué la chaîne complète permettant de traiter les flux de données : flux vers relation, relation vers relation, et relation vers flux. Nous allons explorer un exemple complet pour montrer la clarté d'expression issue de l'algèbre.

Exemple 4.10 :

Le flux d'alerte avec pour attributs (deviceId, avgcpu, τ) représente que l'équipement ***deviceId*** *au timestamp τ a eu une charge moyenne avgcpu supérieure à 25%. La moyenne est calculée sur 5 min toutes les minutes.*

Pour former ce flux, il est nécessaire de joindre le flux ***CPU*** *avec* ***Applications*** *afin de pouvoir associer le bon deviceId à l'appId. Or, cette opération est interdite à moins*

de faire une fenêtre sur le flux. Nous pourrions utiliser une fenêtre telle que le dernier batch [B], mais comme il est nécessaire d'obtenir une fenêtre temporelle pour le calcul de moyenne, nous pouvons appliquer la description appropriée :

$$CPU[T\ 5min\ 1min] \bowtie Applications.$$

Appliquons l'opération d'agrégation sur cette relation pour obtenir la moyenne. Puis nous pouvons sélectionner les n-uplets la moyenne est supérieure à 25%. Enfin, le flux doit être créé à partir de la relation temporelle représentant les équipements supérieurs à 25% de charge pendant les 5 dernières minutes. Ici une ambiguïté réside dans l'énoncé. Souhaitons-nous avoir le flux des nouveaux équipements ayant ce problème ou souhaitons-nous être notifiés à chaque vérification ? Ce critère va influencer le choix du streamer : $\mathcal{I}_{\mathcal{S}}$ ou $\mathcal{R}^{u}_{\mathcal{S}}$. Prenons le premier, nous obtenons la requête suivante :

$$\mathcal{I}_{\mathcal{S}}\left(\sigma_{avgcpu\geq 25\ deviceId}\mathcal{G}_{avg_{cpu}^{avgcpu}}(CPU[T\ 5min\ 1min] \bowtie Applications)\right)$$

D'autres streamers peuvent être construits pour effectuer l'insertion de données dans le flux résultat de manière périodique comme le présente la définition 4.3.9.

Définition 4.3.9 ***Streamers* périodiques**

Soit R une relation,
Les *streamers* périodiques sont les opérateurs créateurs d'un flux S où les insertions ont lieu périodiquement. Le *streamer* $\mathcal{R}^{r}_{\mathcal{S}}$ est défini par un taux temporel r et la propriété :

$$s \in R(t,i) \wedge t - t_0 \equiv 0[r] \Leftrightarrow s' = \Psi_{(t,i)}(s,t) \in S \wedge \mathcal{B}_S(s') = (t,i)$$

Nous avons maintenant défini des opérateurs permettant de couvrir la plupart des opérations. Toutefois, deux opérateurs sont encore à définir pour certaines opérations plus particulière.

4.3.3 Manipulation temporelle

Contrairement à l'algèbre relationnelle, les relations sont temporelles ici. Elles changent au cours de l'exécution de la requête. Il n'existe pas d'opérateurs capables de contrôler ces changements. L'opérateur de manipulation temporelle permet de sélectionner, pour l'instant présent, un état passé d'une relation. Tout d'abord, définissons une transformation temporelle avec la définition 4.3.10 permettant de sélectionner l'instant voulu sur la relation temporelle. La seule contrainte de cette transformation est l'impossibilité de regarder dans le futur. Nous pouvons ainsi définir l'opérateur de manipulation temporelle (def 4.3.11) qui applique cette transformation à la relation temporelle.

Plusieurs fonctions classiques ont une utilité directe :

Définition 4.3.10 **Transformation temporelle**

Une transformation temporelle est une fonction de $\mathbb{T} \times \mathbb{N} \rightarrow \mathbb{T} \times \mathbb{N}$ telle que $\forall b \in \mathbb{T} \times \mathbb{N}, f(b) \leq b$.

Définition 4.3.11 **Opérateur de manipulation temporelle**

Soient R une relation, c une condition sur $\mathbb{T} \times \mathbb{N}$ et f une transformation temporelle,
Alors l'opérateur de manipulation temporelle est défini par

$$\mathcal{D}_c^f(R) = b \mapsto \begin{cases} R(f(b)) & \text{si } c(b) \\ \emptyset & \text{sinon} \end{cases}$$

– $\text{freeze}^{t_s}(t,i) = (t_s, 0)$ si $t \geq t_s$. Permet de figer une relation temporelle à un instant précis, plus aucun changement n'est retransmis à partir de ce point. La relation résultante de cette opération est notée R^{t_s}. Un cas particulier est lorsque $t_s = t_0$. Dans ce cas la relation n'a qu'un seul état. Grâce à cette opération, nous sommes capables de faire une **interrogation instantanée** sur des relations temporelles.

– $\text{period}^r(t,i) = \left(\left\lfloor \frac{t-t_0}{r} \right\rfloor r + t_0, 0\right)$ met à jour la relation de manière périodique avec un taux de rafraîchissement r.

– $\text{change}_S(t,i) = \tau_S \circ \tau_S^{-1}$ met à jour la relation à chaque fois qu'un *batch* est inséré dans S. Le même principe permet une mise à jour *à chaque fois que la relation R' change*.

Cette dernière fonction a un impact particulier lors de la jointure de deux relations temporelles. Dans la jointure (ou au sens large le produit cartésien) telle que nous l'avons définie, si un changement est appliqué d'un côté ou de l'autre, un nouveau calcul de jointure est effectué. Ce comportement peut ne pas être souhaité dans la pratique où nous pourrions souhaiter que seule une branche soit déclencheur de calcul et l'autre soit passive. La jointure semi-sensible permet cette opération. Dans la définition 4.3.12 le résultat $R_1 \ltimes R_2$ n'est pas mis à jour au changement de R_2 mais au changement de R_1 avec l'état de R_2 correspondant.

Définition 4.3.12 **Jointure semi-sensible**

Soient R_1 et R_2 deux relations temporelles,
Soit change_{R_1} la transformation temporelle telle que $\text{change}_{R_1}(b)$ correspond au dernier changement de R_1 inférieur ou égal à b,
La jointure semi-sensible est définie par :

$$R_1 \ltimes R_2 = R_1 \bowtie \mathcal{D}^{\text{change}_{R_1}}(R_2)$$

Exemple 4.11 :

$$DeviceCPU = \mathcal{I}_{\mathcal{S}}(\Pi_{deviceId,deviceName,cpu}(CPU[B] \bowtie Applications) \rtimes Devices)$$

Le flux résultat DeviceCPU correspond au flux CPU originel sur lequel nous avons remplacé l'attribut appId par les attributs deviceId et deviceName qui sont plus intéressants d'un point de vue de l'observation. Si Devices est mise à jour (pour changer son statut) en utilisant la jointure simple, la relation temporelle change d'état. Ce qui provoque l'envoi d'un nouveau n-uplet dans le flux. L'opérateur de jointure semi-sensible empêche ce cas puisque les mises à jour sont cadencées par celles de $CPU[B] \bowtie Applications$ c'est-à-dire $CPU[B]$ soit encore les batchs de CPU et non la relation Devices.

4.3.4 Spread

Enfin, le dernier opérateur est celui évoqué lors de la conception des *batchs* dans [Jain 2008]. Il est nécessaire d'avoir un opérateur pour manipuler les *batchs* afin de modifier la fonction $\mathcal{B}_S$ d'un flux sans toucher à ses données.

L'opérateur de réinitialisation des *batchs* (def 4.3.13, *spread all* dans la littérature) permet de rassembler les *batchs* multiples pour chaque *timestamp* en un seul. Ceci permet de réorganiser le flux.

Définition 4.3.13 **Réinitialisation des batch**

Soit S un flux,
Le flux S réinitialisé noté $\lhd S$ vérifie :

$$\forall s \in S, \text{ alors } s \in \lhd S, \text{ et } \mathcal{B}_{\lhd S}(s) = (s(\tau), 0)$$

L'opérateur *spread* décrit dans la définition 4.3.14 permet de découper les *batchs* présents dans un flux en plusieurs nouveaux. Le découpage se fait sur l'ordre induit par ses attributs. Ceci permet par exemple de partitionner selon un identifiant.

Originellement, si pour l'opérateur *spread*, aucun attribut n'est mentionné, alors tous les n-uplets sont étalés dans des batchs séparés de manière non déterministe. Dans notre algèbre, l'attribut φ et l'ordre positionnel permettent de définir l'ordre des *batchs* comme l'ordre positionnel : $\rhd = \rhd_{\varphi}$. À l'inverse, en appliquant $\lhd$, nous définissons l'ordre des *batchs* comme l'ordre temporel.

Exemple 4.12 :
Sachant le contenu du flux ***CPU****(appId, value, τ) :*

(1,0)	*(1,1)*	*(2,0)*
$(1, 20, 1)$ $(1, 25, 1)$ $(2, 22, 1)$	$(4, 10, 1)$ $(1, 25, 1)$ $(4, 12, 1)$	$(2, 23, 1)$ $(6, 64, 1)$

Définition 4.3.14 *Spread*

Soient S un flux de schéma A, et a un attribut de S,
Soit $\mathbb{I}^S = \mathbb{T} \times \mathbb{N} \times \mathbb{I}_R$ le Φ-espace ordonné de manière lexicographique,
Alors le flux S raffiné par l'attribut a noté $\rhd_a S$ vérifie : $\forall s_1, s_2 \in S^2$, alors $s'_1, s'_2 \in (\rhd_a S)^2$, tel que :

$$b_1 = \mathcal{B}_{\rhd_a S}(s'_1) < b_2 = \mathcal{B}_{\rhd_a S}(s'_2) \quad \Leftrightarrow \quad \mathcal{B}_S(s_1) < \mathcal{B}_S(s_2) \vee (\mathcal{B}_S(s_1) = \mathcal{B}_S(s_2) \wedge s_1(a) < s_2(a))$$

Ainsi que $s'_i = \{(\varphi, (b_i, s_i(\varphi)))\} \bigcup_{x \in A \setminus \{\varphi\}} \{(x, s(x))\}$, pour $i \in \{1, 2\}$

$\lhd$ *CPU : L'application de l'opérateur de réinitialisation des batchs regroupe tous les n-uplets simultanés dans le même batch.*

(1,0)	*(2,0)*
(1, 20, 1) (1, 25, 1) (2, 22, 1) (4, 10, 1) (1, 25, 1) (4, 12, 1)	(2, 23, 1) (6, 64, 1)

$\rhd_{appId}$ *CPU : L'utilisation du spread sur l'identifiant permet d'affiner les batchs originels en garantissant que chaque batch ne contienne qu'une valeur de appId.*

(1,0)	*(1,1)*	*(1,2)*	*(1,3)*	*(2,0)*	*(2,1)*
(1, 20, 1) (1, 25, 1)	(2, 22, 1)	(1, 25, 1)	(4, 10, 1) (4, 12, 1)	(2, 23, 1)	(6, 64, 1)

$\lhd \rhd_{appId}$ *CPU : La combinaison de la réinitialisation et du spread permettent d'avoir un batch par timestamp affecté à une valeur d'appId.*

(1,0)	*(1,1)*	*(1,2)*	*(2,0)*	*(2,1)*
(1, 20, 1) (1, 25, 1) (1, 25, 1)	(2, 22, 1)	(4, 10, 1) (4, 12, 1)	(2, 23, 1)	(6, 64, 1)

Nous avons maintenant présenté l'ensemble des opérateurs de l'algèbre Astral. Nous sommes désormais capables d'exprimer une requête continue (et instantané avec l'aide de la manipulation temporelle) avec une grande clarté et sans ambiguïté sémantique. Nous allons maintenant, définir l'équivalence de requête avec des *timestamps* de départs t_0 différents.

4.4 Transposabilité

La définition de l'équivalence de requêtes (def 4.1.13) spécifie que les entités sont toutes deux initialisées à un *timestamp* t_0. Ce type d'équivalence permet de réécrire une requête avant de l'exécuter tout en conservant exactement le même comportement. Toutefois, cela ne permet pas de comparer avec une requête déjà en exécution, puisque la requête n'est pas synchronisée au même instant. Pour intégrer des données provenant de différentes sous-requêtes, cet aspect a une grande importance.

4.4.1 Équivalence de requêtes générale

Pour illustrer que l'idée d'effectuer des équivalences de requêtes à différents moments n'est pas triviale, prenons un exemple. Soit la requête $CPU[B]$. Cette requête représente la relation contenant le dernier *batch* du flux **CPU**. Dans cette requête, durant la période $[t_0, \tau_S(0)[$, par définition, la relation est vide : il est nécessaire d'attendre le premier n-uplet pour former le résultat. Si nous prenons une autre requête ayant démarré au timestamp $t_1 \ll t_0$. Alors pendant la période $[t_0, \tau_S(0)[$, il contient son dernier *batch*. Ceci constitue un exemple de ce que nous appellerons le phénomène d'élaboration défini en 4.4.1.

Définition 4.4.1 **Phénomène d'élaboration**

Le phénomène d'élaboration correspond à une période initiale de la vie d'une requête durant laquelle le résultat n'est pas calculable. Cette période transitoire constitue l'élaboration d'une requête.

Ainsi, nous sommes capable d'établir l'équivalence entre deux requêtes par la définition 4.4.2 qui précise que les résultats sont équivalents à partir d'un certain moment. Pour cela, nous définissons qu'il existe un *timestamp* pour lequel, l'initialisation des entités (grâce à σ et $\mathcal{D}$) à ce *timestamp* implique une équivalence des résultats.

Définition 4.4.2 **Équivalence de requêtes générale**

Soient $(Q_1(E_1), t_1)$ et $(Q_2(E_2), t_2)$ deux requêtes quelconques,

Soit $\mathcal{E}_t$ l'opérateur égal à $\begin{cases} \sigma_{t \geq \tau} & \text{si les requêtes sont des flux} \\ \mathcal{D}^{(t,i)}_{t \geq \tau} & \text{si les requêtes sont des relations} \end{cases}$

Alors, l'inclusion de requêtes entre ces requêtes est définie par

$$\exists t \in \mathbb{T}, \text{ tel que } \quad (\mathcal{E}_t\ Q_1(E_1), t_1) \subseteq\!\!\!\!= (\mathcal{E}_t\ Q_2(E_2), t_2)$$

L'équivalence de requêtes est définie par double inclusion.

Nous avons défini une notion générique d'équivalence. Voyons désormais les conséquences du changement de *timestamp* d'initialisation pour une requête donnée.

4.4.2 Transposabilité

La transposabilité telle que définie par la définition 4.4.3 est de changer le *timestamp* de départ d'une requête.

Définition 4.4.3 **Transposabilité de requête**

Soit (A, t_0) une requête,
A est transposable par B sur $T \subseteq \mathbb{T}$ si et seulement si :

$$\forall t \in T, \quad (A, t_0) \equiv (B, t)$$

A est dite *naturellement* transposable si $B = A$.

Toutefois, cette définition ne permet pas de conclure sur l'équivalence de requête, car nous ne pouvons pas a priori calculer la transposition de la requête sur son nouveau *timestamp*. Pour permettre la résolution de ce problème, nous allons raisonner par récurrence sur les opérateurs. Comme le défini la définition 4.4.4, chaque opérateur peut se transposer en un autre sur un ensemble de *timestamp* calculé.

Définition 4.4.4 **Transposabilité d'opérateur**

Soit O un opérateur unaire,
Soit (Q, t_0) une requête transposable par Q' sur E,
O est un opérateur transposable par O' sur T si et seulement si :

$$(OQ, t_0) \text{ est naturellement transposable par } O'Q' \text{ sur } T \cap E$$

O est dit *naturellement* transposable si $O' = O$.

Il est nécessaire d'initialiser la récurrence en supposant que pour toute expression algébrique, il est possible de trouver un ensemble d'entité source naturellement transposable. Du point de vue de l'implémentation, les sources de données produisent les mêmes flux ou relations quelque soit le moment où elles sont exploitées. Cette affirmation reste toutefois à travailler, car lors du déploiement d'une requête, nous instancions aussi le processus d'acquisition, qui est lui dépendant du moment de démarrage. Ces aspects sont détaillés lors de l'exploration de l'expressivité d'Astral dans le chapitre 5.

Hypothèse 4.2 **Transposabilité native**

Soit $(Q(E), t_0)$ une requête,
Alors, il existe une expression $Q'(E')$ telle que :

$$\forall A \in E', \qquad A \text{ est naturellement transposable sur } \mathbb{T}$$

Les éléments de E' sont appelés sources de la requête.

Afin d'illustrer les propriétés de transposabilité, nous allons montrer la transposabilité d'une requête simple.

Exemple 4.13 :

Supposons que nous souhaitons obtenir la transposabilité de la requête permettant d'obtenir toutes les 5 secondes la liste des dispositifs actuellement connectés dans la maison :

$$\mathcal{R}^{5s}_{\mathcal{S}}(\sigma_{deviceStatus=1}Devices).$$

Ici, nous supposons par l'hypothèse des transposabilités des sources que Devices est naturellement transposable à tout instant. La sélection est un opérateur qui a la particularité de ne traiter que l'instant présent et est naturellement indépendant de t_0. Ainsi, $\sigma_{deviceStatus=1}Devices$ est naturellement transposable à tout instant.

Par contre, $\mathcal{R}^r_{\mathcal{S}}$, lui n'est pas transposable à tout moment. En effet, dans sa définition la condition d'appartenance au flux produit est la suivante : $s \in R(t,i) \wedge t - t_0 \equiv 0[r]$. Ainsi, si nous transposons à t_1, pour obtenir l'équivalence des requêtes, il est nécessaire que $\forall t \geq t_1$, $t - t_0 \equiv t - t_1 \equiv 0[r]$. Ce qui nous conduit à montrer que $t_1 = t_0 + kr$ avec $k \in \mathbb{Z}$.

Supposons que t_1 vérifie cette condition. Nous arrivons très facilement à voir que $\forall R$, en prenant $t = \max(t_1, t_0)$, nous avons bien[3] *que $(\mathcal{E}_t\mathcal{R}^r_{\mathcal{S}}(R), t_0) \equiv (\mathcal{E}_t\mathcal{R}^r_{\mathcal{S}}(R), t_1)$.*

$\mathcal{R}^{5s}_{\mathcal{S}}\sigma_{deviceStatus=1}Devices$ est naturellement transposable sur $\{t \in \mathbb{T} / \ t \equiv t_0[5s]\}$

Nous avons désormais montré comment nous pouvions faire des équivalences de requêtes à travers le temps. Nous pouvons manipuler ces concepts pour en extraire des propriétés. Dans le chapitre 5, nous explorons des cas généraux et plus complexes de transposabilité pour démontrer la puissance d'expression d'Astral.

4.5 Conclusion

Dans ce chapitre, nous avons présenté un nouveau modèle de gestion de flux de données. Ce modèle permet d'obtenir une spécification claire et précise du résultat attendu pour une requête. Ainsi, une implémentation de système d'évaluation de requête peut décrire exactement son exécution grâce à ce modèle théorique.

Nous avons clarifié la gestion de l'ordre des n-uplets dans les flux. Ce point nous a conduits à reformuler certaines définitions classiques (telles que la jointure ou l'union) et à découvrir des comportements encore non détaillés dans la littérature (telle que l'asymétrie de ces opérateurs). Nous voyons que grâce à l'opérateur de manipulation temporelle, nous sommes capable d'exécuter des requêtes instantanées en figeant les relations temporelles. Enfin, nous avons défini strictement la notion d'équivalence de requête, autant à *timestamp* de départ fixe, que différent. Ce qui nous permet par la suite de formuler mathématiquement des preuves.

Le chapitre suivant détaille l'expressivité de cette algèbre par la confrontation aux formalisations existantes et par la présentation de théorèmes d'équivalences de requêtes.

3. en suivant naturellement les définitions, nous obtenons de plus une égalité stricte même sur les φ

« If you will remember back,
The words I spoke were quite exact. »
Zecora

5 Expressivité d'Astral

Dans le chapitre 4, nous avons présenté Astral, une algèbre pour exprimer des requêtes continues sur flux et relations. Nous avons jusqu'ici présenté principalement des définitions, mais nous n'avons pas encore listé de propriétés formelles. Or, ces propriétés ont des impacts forts dans le cadre de l'optimisation de requêtes.

Ce chapitre permet d'analyser plus en profondeur cette algèbre et d'en explorer ses capacités et limitations. Afin de valider nos choix, nous analysons la puissance d'expression résultante ainsi que les capacités de démonstration de propriétés que nous obtenons.

En premier lieu, nous détaillons en section 5.1 les choix concernant les définitions fondamentales en les comparant avec l'état de l'art. Par la suite, dans la section 5.2, nous comparons l'expressivité d'Astral avec les autres modèles théoriques de la littérature. Dans la section 5.3, nous explorons les capacités de cette algèbre en démontrant des théorèmes non triviaux d'équivalence de requêtes et de transpositions. Puis, nous concluons par une synthèse dans la section 5.4.

5.1 Choix des fondations de l'algèbre

Lors de l'établissement des premières définitions d'Astral dans la section 4.1, nous avons fait des choix concernant le temps, les entités et les équivalences de requêtes. Cette section discute de la validité de ces choix en montrant que des choix différents

auraient mené à des ambiguïtés sémantiques. La section 5.1.1 présente le choix de continuité du temps. La section 5.1.2 détaille l'hypothèse de cohérence temporelle. Enfin, nous analysons nos choix en terme de gestion des ordres dans la section 5.1.3.

5.1.1 Continuité du temps

La définition 4.1.5 présente un *timestamp* comme un élément d'un espace continu. Dans la littérature, le temps est souvent considéré comme un entier, ou au mieux dans un espace isomorphe à $\mathbb{N}$. Notre choix permet de mieux gérer les différences d'interprétation du temps.

En effet, chaque système informatique est limité par un *chronon* : la plus petite différence de timestamp observable. Pour certains systèmes, ce *chronon* est d'une milliseconde, d'autres d'une seconde et d'autres d'un *tick* processeur. Ne pas se restreindre à un *chronon* particulier permet de manipuler facilement les *timestamps* issus de deux systèmes de datation (synchronisés) sans ambiguïté. De façon plus formelle, nous aurions pu définir le temps comme un ensemble isomorphe à $\mathbb{N}$. Dans un tel cas, l'utilisation d'opérateurs binaires serait plus délicate. En effet, il y aurait deux notions de temps pour chacune des branches. Ainsi, il faudrait introduire des outils pour intégrer ces deux horloges en une commune.

Par mesure de simplicité, nous avons fait le choix d'avoir un seul temps universel. Dans la pratique, cela veut dire que les *timestamps* sont synchronisés au niveau des différents équipements distribués, ou que les *timestamps* sont issus du système où est évalué la requête. Ceci est un domaine de recherche complexe à part entière que nous ne détaillons pas dans cette thèse. En supposant une horloge commune, alors le choix d'avoir un *timestamp* réel nous permet d'établir une précision indépendante du *chronon*.

Nous remarquons aussi que ce choix permet de clarifier l'opérateur $\mathcal{RS}$ de STREAM [Arasu 2004a]. Cet opérateur est décrit par la phrase « *à chaque timestamp, envoyer l'ensemble de la relation* ». Cet opérateur est dépendant du système, car si un système possède un chronon de $1s$ alors $\mathcal{RS}$ produit des n-uplets chaque seconde. Pour un autre système possédant un chronon de $1ms$ alors le streamer produit plus de n-uplets. Cet opérateur est remplacé par $\mathcal{R}^r_{\mathcal{S}}$ dans Astral avec r une période de temps explicite.

5.1.2 Hypothèse de la cohérence temporelle

L'hypothèse de la cohérence temporelle 4.1 affirme que les n-uplets doivent arriver dans un flux de manière ordonnée selon leurs *timestamps*. Cette hypothèse a suscité beaucoup d'interrogation de la part de la communauté. La question est de savoir s'il est nécessaire que d'autres opérateurs existent dans l'algèbre pour garantir cette propriété, ou est-ce à l'implémentation de le garantir d'une manière ou d'une autre ? Par exemple, Aurora [Abadi 2003] définit des opérateurs spécifiques au tri d'un flux grâce à une mémoire tampon de n n-uplets.

Dans Astral, si l'hypothèse n'est pas vérifiée, nous avons la fonction position-*batch* (def 4.3.1) qui n'est plus croissante. Ainsi, sa pseudo-inverse n'existe plus, ce qui fait qu'il devient impossible de définir les séquences de fenêtres telles que nous les avons

faites. Par exemple, pour les fenêtres positionnelles contenant les 50 derniers n-uplets, la notion de *dernier* est sémantiquement lié aussi à son *timestamp* ce qui induit des confusions. Du point de vue de l'implémentation, la croissance du temps fait qu'une fois qu'un *timestamp* t est traité, il est nécessaire de garantir que toutes les données inférieures à t ont été traités. Ainsi le *scheduler* peut décider d'exécuter un opérateur en garantissant la sémantique d'Astral.

Notre constat est le suivant : pour avoir une algèbre sans ambiguïté sémantique, nous devons supposer que les *timestamps* sont croissants. C'est à l'implémentation de garantir cette contrainte, et si elle n'est pas vérifiée, il est très difficile de prévoir les conséquences.

5.1.3 Sémantiques d'ordres

L'évaluation d'une relation temporelle à un *batch* donné n'est pas un ensemble de n-uplets, c'est une séquence de n-uplets. Ce choix est central, car il intervient dans la majorité des définitions. Que ce soit dans la définition du produit cartésien 4.2.3 ou des *streamers* 4.3.8, où la sémantique que nous choisissons pour l'ordre des n-uplets est explicitée.

Dans la littérature, les flux sont souvent étendus du modèle $\mathcal{SEQ}$ [Seshadri 1995] décrivant les manipulations de séquences. Il est acquis que les flux sont des ensembles totalement et strictement ordonnés. Or, lorsque nous transformons ce flux en relation temporelle, cet ordre est rarement étudié. Pourtant, nous avons vu dans la définition des *streamers* 4.3.8 qu'il est nécessaire assigner un ordre strict aux flux produits.

Dans Astral, nous considérons que les relations instantanées doivent avoir un ordre pour obtenir un flux correctement ordonné après l'application d'un *streamer*. Ainsi, nous n'avons pas d'ambiguïté sémantique liée à cet aspect. Cette formalisation plus stricte nous a amenés à découvrir le théorème 4.1 qui montre l'asymétrie du produit cartésien sur les relations temporelles.

Il est important de noter que ce théorème est vrai à cause du choix de l'équivalence de requête (def 4.1.13). Cette équivalence inclut deux notions, celle des entités initialisées qui prend tout son sens dans le cadre des transpositions, et celle des inclusions et équivalences de séquences. Nous avons en effet défini l'inclusion des relations instantanées comme une définition d'inclusion de suite, ce qui nécessite une équivalence de l'ordre. Une définition différente de l'équivalence de requête permettrait d'obtenir une symétrie du produit. Dans la pratique, il est en effet possible que l'utilisateur souhaite obtenir ses n-uplets dans un ordre quelconque, ce qui permet des optimisations supplémentaires.

Ces choix pour les définitions fondamentales de l'algèbre permettent d'avoir une sémantique claire et sans ambiguïté. Nous présentons maintenant l'expressivité d'Astral vis-à-vis d'algèbres existantes.

5.2 Comparaison de l'expressivité d'Astral

L'algèbre Astral a été conçue pour permettre d'exprimer des requêtes continues sans ambiguïté, mais aussi pour introduire de nouveaux opérateurs afin d'étendre

la puissance d'expression des requêtes continues. Dans cette section, nous analysons en détail son expressivité. En premier lieu, nous détaillons qu'Astral couvre naturellement un large ensemble des approches actuelles. Ensuite, nous analysons les opérateurs que nous avons particulièrement remodelés : les séquences de fenêtres et la manipulation temporelle.

5.2.1 Expressivité générale

Astral est inspiré de l'algèbre *ACO* de *STREAM*. Nous avons repris la majeure partie des idées, notamment celle d'avoir des flux et des relations, et avons appliqué des définitions plus strictes et plus précises (c.f. section 5.1). Ainsi, une requête exprimée dans *ACO* peut être exprimée sans difficulté en Astral.

Dans [Arasu 2004a], il est démontré que l'expressivité de *STREAM* couvre les approches de précédentes comme *Chronicles*, *Tribeca*, *NiagaraCQ* ou *Aurora*. Nous pouvons considérer qu'Astral couvre aussi ces travaux.

Notre formalisation à base de *batchs* permet de supporter les sémantiques exposées par [Jain 2008]. Nous avons de même présenté notre interprétation des opérateurs *SPREAD* en remplaçant les choix non déterministes par des choix sur l'ordre positionnel.

Ainsi, Astral couvre l'expressivité de plusieurs approches actuelles. Afin d'étudier ceci en détail, nous nous concentrons maintenant sur l'opérateur le plus étudié de la littérature : les séquences de fenêtres.

5.2.2 Fenêtres

Dans cette section, nous analysons l'expressivité des séquences de fenêtres. Afin de voir les capacités de notre formalisation, nous revisitons les expressions de fenêtres présentes dans l'état de l'art. Nous commençons par la plus largement utilisée : la fenêtre glissante *RANGE/SLIDE*.

RANGE x **SLIDE** y

Il existe deux descriptions possibles en Astral pour cette fenêtre glissante. Ces définitions permettent d'exprimer la *phase* d'initialisation. La première définition correspond à celle présente dans plusieurs travaux comme [Jain 2008] :

$r = y$	$\beta(j) = yj + t_0$	$\alpha(j) = \max(yj - x, 0) + t_0$

Les premiers états des bornes ont une largeur de fenêtre plus petite que x. Dès que $j \geq \frac{x}{y}$, alors la largeur temporelle devient égale à x. Nous pouvons noter que la fonction $\alpha(j) = yj - x + t_0$ n'est pas valide dans notre contexte. La seconde condition des DSF (def 4.3.2) nécessite $\alpha \geq t_0$, ce qui n'est pas vrai pour $i = 0$. L'insertion de la fonction max rend la description valide et décrit la sémantique des premières phases, ce qui n'est pas explicite dans la description textuelle et dans les descriptions de la littérature.

La description suivante est aussi valide, mais ne possède pas de phase initiale. La relation temporelle est vide jusque $\beta(0)$ afin d'assurer que la fenêtre couvre toujours une largeur temporelle de x.

$r = y$	$\beta(j) = yj + x + t_0$	$\alpha(j) = yj + t_0$

Afin d'illustrer les différences entre les deux sémantiques possibles, la figure 5.1 représente les différences d'évaluations pour une fenêtre glissante avec $y = 2$ et $x = 4$. Dans le premier cas (max), il y a deux autres évaluations à $i = 0$ et $i = 1$. Après cette partie, les deux modèles sont identiques.

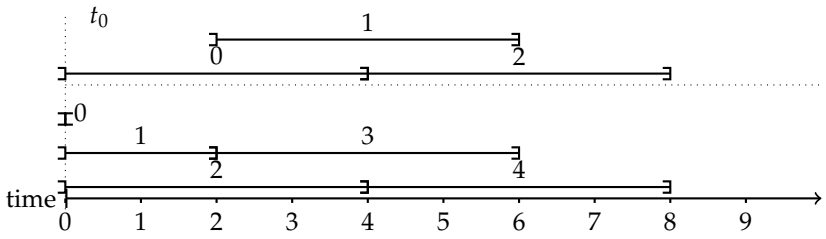


FIGURE 5.1 – Différence entre les sémantiques lors de la phase d'initialisation de la fenêtre RANGE 4 SLIDE 2

Nous pouvons remarquer que nous avons aussi introduit la notion d'inclusion de bornes. Ce qui permet de clarifier si la borne inférieure est incluse dans le contenu de la fenêtre. Dans le cadre des fenêtres glissantes, ce n'est pas le cas [1].

RANGE x

Dans ses définitions usuelles, la séquence de fenêtre RANGE x est similaire à RANGE x SLIDE 1 ce qui est dépendant du système d'implémentation. Dans Astral, le temps n'est pas discret et une telle définition n'a pas de sens. Une définition similaire est possible en supposant que le *chronon* du système d'implémentation est ε, alors : RANGE x = RANGE x SLIDE ε.

Une approche plus propre serait d'utiliser une DSF générique en basant ses instants d'évaluations (fournis par γ, pour rappel) sur les arrivées et départs de n-uplets dans la fenêtre. Toutefois, une connaissance de α^{-1} et β^{-1} est nécessaire et rend son expression plus complexe. Il est intéressant de voir que ce comportement est similaire aux implémentations existantes. En effet, un opérateur n'est pas efficace s'il vérifie à chaque *timestamp* système si un n-uplet doit sortir de la fenêtre. Il est plus efficace de planifier (via le *scheduler*) ses instants d'évaluations. Ainsi, à la réception d'un n-uplet à $\tau = 42$, l'opérateur planifie une exécution à $42 + x$ pour faire sortir le n-uplet de la fenêtre.

1. Il est intéressant de noter un erratum dans la spécification de cet opérateur dans [Jain 2008] où la définition précise que la borne inférieure est incluse alors que les exemples et les expérimentations pratiques indiquent le contraire.

Descriptions à bornes linéaires

Dans SStreamWare [Gürgen 2007], les fenêtres sont définies par un 5-uplet (start, end, rate, start_adv, end _adv) :

– start (resp. end) décrit la borne inférieure (resp. supérieure) de la première fenêtre produite par la séquence

– rate est la fréquence d'évaluation

– start_adv (resp. end_adv) décrit la quantité de glissement de la borne inférieure (resp. supérieure) à chaque évaluation.

Dans Astral, ce comportement est facilement représentable sous forme de DSF :

$r = \text{rate}$	$\beta(j) = \text{end\_adv} * j + \text{end}$	$\alpha(j) =$ $\text{start\_adv} * j + \text{start}$

Descriptions procédurales

Dans les premières versions de TelegraphCQ [Chandrasekaran 2003], les descriptions de fenêtres étaient faites par une boucle *for* procédurale sur un *timestamp* :

```
1 for(t = init ; continue(t) ; t = evolution(t))
2     WindowIs(S, begin(t), end(t))
```

Ceci peut être formalisé grâce à une DSF générique. Considérons la suite :

$$u_n = \begin{cases} \text{init} & \text{si } n = 0 \\ \text{evolution}(u_{n-1}) & \text{si continue}(u_{n-1}) \\ u_{n-1} & \text{sinon} \end{cases}$$

$\gamma(t,i) = \sum_{i=0}^{+\infty} u_i \mathbb{1}_{[u_i, u_{i+1}[}(t)$	$\beta = \text{end}$	$\alpha = \text{begin}$

La fonction γ est définie par une liste de points. Cette définition est courante créer des fonctions en escaliers. Il est notable que pour des fenêtres avec un taux d'évaluation constant et une position initiale usuelle, une DSF simplifiée est suffisante.

Description multidomaines

Dans certains systèmes d'observations [Jurdak 2008], il est courant de récupérer les données par vagues. Ainsi, dans les interfaces de ces systèmes il est courant de trouver des séquences de fenêtres telles que « les n derniers n-uples toutes les m secondes ». Ces descriptions ont pour particularité d'être temporelles et positionnelles. Pour formaliser une telle séquence, il n'est pas nécessaire d'utiliser une DSF générique, car l'évaluation reste périodique. Voici la description de cette séquence de fenêtre :

$r = m$	$\beta(j) = \tau^{-1}(mj + t_0)$	$\alpha(j) =$ $\max(\tau_S^{-1}(mj + t_0) - n, 0)$

Le taux est temporel et les bornes sont positionnelles. Le pont entre les domaines est assuré grâce aux fonctions τ_S et τ_S^{-1}. Dans ce cas, l'expression $\tau_S^{-1}(mi + t_0)$ donne la position à un *timestamp* donné. La remarque concernant le max que nous avions faite plus tôt reste valable ici.

Introduction d'un délai

Dans les travaux de Patroumpas et Sellis [Patroumpas 2006], l'introduction d'un délai de traitement a été formalisée. En effet, si un n-uplet possède un *timestamp* légèrement déphasé par rapport à son temps réel, alors la fenêtre peut tout de même l'inclure.

La formalisation dans notre cas revient à légèrement changer notre fonction γ par γ' qui décale l'évaluation de δ. Par exemple, nous obtenons pour une description temporelle :

$$\gamma'(t,i) = \gamma(t-\delta,i) = \left\lfloor \frac{t-\beta(0)-\delta}{r} \right\rfloor$$

Modèle d'exécution de SECRET

Enfin, SECRET [Botan 2010] est un modèle permettant de généraliser les sémantiques d'exécutions des fenêtres. L'approche est de découper l'exécution d'une fenêtre en quatre concepts que nous pouvons retrouver dans Astral.

– Les *Ticks* décident du moment où le système doit réagir au flux (sémantique basée n-uplet, basé temps ou *batch*).

– Le *Content* défini le contenu global de la fenêtre par son *Scope*, c'est-à-dire sa description.

– Enfin, un *Report* envoie le résultat final.

Bien que les approches soient différentes : nous retrouvons des notions similaires. Le *Scope* est similaire à α, β. Le *Content* est assimilable à une fenêtre en particulier. Les *Ticks* et les *Reports* sont faits grâce à la fonction γ et le contrôle du mode d'exécution par les opérateurs *SPREAD*. L'avantage de l'approche de SECRET est de pouvoir qualifier rapidement le comportement de l'exécution d'un SGFD en particulier, alors qu'Astral décrit la sémantique exacte du résultat pour l'utilisateur.

Nous avons détaillé le positionnement de l'opérateur de séquence de fenêtre par rapport à la littérature. Nous constatons que l'opérateur permet de couvrir l'ensemble des sémantiques. Nous avons clarifié plusieurs descriptions non triviales telles que les fenêtres RANGE strictes, les fenêtres multidomaines et l'introduction du délai. Nous présentons maintenant l'analyse de l'opérateur de manipulation temporelle.

ROWS x **SLIDE** y

Nous avons présenté dans le tableau 4.2 que la séquence de fenêtre positionnelle décrivant les x derniers n-uplets tous les y n-uplets est une description linéaire. Cette description reflète la sémantique de *ROWS* de dans la plupart des cas. Toutefois, lors que le nombre de n-uplets par *batch* est plus grand que y et non proportionnel à y, alors les descriptions diffèrent.

Soit un flux S ayant les batchs suivants $@b_1 = \{s_1, s_2, s_3\}$, $@b_2 = \{s_4\}$, $@b_3 = \{s_5\}$. Nous souhaitons appliquer l'opérateur $]P\ 2\ 2]$. Nous obtenons pour b_1 la sélection des deux n-uplets les plus récents du batch contenant les n-uplets 1 et 2. Pour b_2, nous sélectionnons les n-uplets 3 et 4. Selon [Jain 2008], la sémantique de [ROWS x SLIDE y] est pour b_1 : deux n-uplets du dernier batch, et pour b_3 les deux n-uplets suivants. Un décalage a été introduit pour permettre de ne pas utiliser deux fois le même *batch* ce qui rend la description non linéaire. Le tableau suivant compare les deux descriptions :

batch	$]P\ 2\ 2]$	[ROWS x SLIDE y]
b_1	$\{s_2, s_3\}$	$\{s_2, s_3\}$ ou $\{s_1, s_3\}$ ou $\{s_1, s_2\}$
b_2	$\{s_3, s_4\}$	comme b_1
b_3	comme b_2	$\{s_4, s_5\}$

Il est toutefois possible de retrouver cette description grâce à l'introduction d'un décalage suivant le *modulo* des cardinalités des *batchs* rencontrés à $n * y$:

$$\delta_j = \left(\sum_{n=0}^{j} \tau_S^{-1} \circ \tau_S(yn) \right) \bmod y$$

Il suffit d'ajouter ce δ_j aux expressions d'α et β et nous retrouvons le même comportement que la description [ROWS x SLIDE y]. Enfin, la sémantique de ROWS x est équivalente à $y = 1$.

5.2.3 Manipulation temporelle

L'opérateur de manipulation temporelle (définition 4.3.11) permet de donner à une relation temporelle un état qu'elle a eu précédemment. Ceci permet de transformer le moment d'évaluation de la relation temporelle. Dans l'état de l'art, à notre connaissance, il n'existe pas d'opérateur capable d'exprimer explicitement cette opération.

Cet opérateur nous a permis notamment d'introduire la jointure semi-sensible $\bowtie\!\!\!\blacktriangleleft$ (définition 4.3.12). Cette jointure permet de refléter un comportement qui se retrouve dans plusieurs SGFD : ne réagir que sur les mises à jour d'une seule branche d'une opération de jointure.

Une autre application intéressante de l'opérateur de manipulation temporelle est le calcul de changement d'une relation temporelle. Comme nous l'avons vu, l'opérateur $\mathcal{I}_S$ est capable de créer un flux composé des nouveaux n-uplets d'une relation. Dans le cadre l'observation de système, nous pouvons réaliser l'opération suivante : « A partir d'une relation temporelle $R(id,v)$, fournir le flux des changements (id,v,v_{old}) où v_{old} est l'ancienne valeur v pour l'identifiant id ». Cette requête s'écrit ainsi :

$$\mathcal{I}_S(R \bowtie_{v \neq v_{old}} (D_{t>t_0}^{(t,i)^-} \rho_{v_{old}/v} R))$$

L'opérateur $D_{t>t_0}^{(t,i)^-}$ « retarde » la relation temporelle d'un *batch*. Il devient possible d'interroger à un instant donné, une relation temporelle et son état précédent.

Nous avons présenté l'ensemble des aspects novateurs des définitions d'Astral. Nous détaillons maintenant les propriétés de cette algèbre avec un ensemble de propositions et de théorèmes.

5.3 Propositions et théorèmes

Dans cette section, nous explorons les équivalences de requêtes possibles grâce à Astral. Dans la littérature, quelques résultats sont connus, comme les commutativités classiques. Nous démontrons ici que nous pouvons prouver ces propriétés avec nos définitions et que nous pouvons en démontrer d'autres. Premièrement, nous montrons comment le *timestamp* et les *batchs* sont conservés lors de l'utilisation de *streamers*, et leurs conséquences. Ensuite, nous explorons les relations classiques de commutativité et d'associativité des opérateurs, ce qui a des impacts pour l'optimisation logique. Enfin, nous présentons les résultats de calculs de transposabilité. Les démonstrations des propositions et théorèmes non-triviaux sont présentés dans l'annexe A.

5.3.1 Transmission du temps

La définition 4.3.7 de la réécriture des n-uplets des *streamers* implique le changement de son *timestamp* et de son *batch*. Ainsi, lors de l'application successive d'un opérateur de séquence de fenêtre et d'un *streamer* : il n'est pas trivial de voir comment ces propriétés sont conservées. Le théorème 5.1 de transmission temporelle permet d'avoir une condition suffisante pour garantir cette transmission.

Théorème 5.1 **Transmission temporelle des *streamers***

Soit S un flux,
Soit $[\alpha, j+k, 1]$ une DSF positionnelle avec α croissante, k un entier avec une borne inférieure incluse ou non,
Considérant S' le flux formé par la requête $\mathcal{I}_{\mathcal{S}}(S[\alpha, i+k, 1])$,
Si un n-uplet réécrit de S' a pour *batch* (t,i) alors, ce n-uplet avait originellement pour *batch* (t,i) dans S. Formellement :

$$\Psi_{(t,i)}(s,t) \wedge \mathcal{B}_{S'}(\Psi_{(t,i)}(s,t)) = (t,i) \Rightarrow s \in S \wedge \mathcal{B}_S(s) = (t,i)$$

Cette propriété est aussi valable pour $\mathcal{R}^u_{\mathcal{S}}(S[B])$.

Le corolaire 5.1 à ce théorème permet d'interpréter $\mathcal{I}_{\mathcal{S}}$ comme l'opération inverse de la fenêtre « dernier batch » $[B]$, entre autres.

Corollaire 5.1 **Équivalence de la composition fenêtre-*streamer***

Soit une composition de fenêtre-streamers respectant les conditions du théorème 5.1,
Si α est telle que la séquence de fenêtre contient $[B]$, alors,

$$S \equiv \mathcal{I}_{\mathcal{S}}(S[\alpha, i+k, 1n]) \equiv \mathcal{I}_{\mathcal{S}}(S[B]) \equiv \mathcal{I}_{\mathcal{S}}(S[\infty]) \equiv \mathcal{R}^u_{\mathcal{S}}(S[B])$$

Exemple 5.1 :

Reprenons l'exemple vu dans la section 4.1.3 avec le flux ***CPU****(appId, cpu, τ). En prenant la fenêtre [B], le corollaire nous assure que $\mathcal{I}_S(S[B]) = S$. L'insertion d'un n-uplet dans le flux est effectuée à un batch égal au batch du n-uplet initial. Ainsi, l'opérateur $\mathcal{I}_S$ écrase le timestamp avec celui du batch. Voici la suite des états par lesquels passe la relation temporelle $CPU[B]$, ainsi que le flux résultant de $\mathcal{I}_S(CPU[B])$:*

$$CPU(id, cpu, \tau) = \{(1, v1, 3); (2, v2, 9); (1, v3, 10); (3, v4, 12); ...\}$$

$CPU[B](3)$:

id	*cpu*	τ
1	$v1$	3

$CPU[B](9)$:

id	*cpu*	τ
2	$v2$	9

$CPU[B](10)$:

id	*cpu*	τ
1	$v3$	10

$\mathcal{I}_S(CPU[B])$:

id	*cpu*	τ
1	$v1$	3
2	$v2$	9
1	$v3$	10
...	...	...

Voyons maintenant un contre-exemple avec une fenêtre ne respectant pas les conditions du théorème. Nous considérons maintenant l'utilisation de la fenêtre temporelle glissante de 2 secondes : $[T\ 2s\ 2s] = [W]$. Comme la production d'un n-uplet dans un streamer sensible est dirigée par les changements de fenêtres, alors le timestamp affecté aux n-uplets produits est le moment où le contenu de la fenêtre change. Dans le cas d'une fenêtre changeant toutes les 2 secondes, cela ne correspond pas au timestamp original.

Voici la suite des états par lesquels passe la relation temporelle $CPU[W]$, ainsi que le flux résultant de $\mathcal{I}_S(CPU[W])$:

$CPU[W](4)$:

id	*cpu*	τ
1	$v1$	3

$CPU[W](10)$:

id	*cpu*	τ
2	$v2$	9
1	$v3$	10

$CPU[W](12)$:

id	*cpu*	τ
3	$v4$	12

$\mathcal{I}_S(CPU[W])$:

id	*cpu*	τ
1	$v1$	4
2	$v2$	10
1	$v3$	10
3	$v4$	12
...	...	...

Ainsi : $I_S(CPU[T\ 2s\ 2s]) \not\equiv CPU$.

Ce résultat est assimilable aux implémentations courantes des opérateurs flux→flux. En effet, la lecture d'un flux correspond dans la pratique à la récupération du dernier *batch* et à l'envoi de ces n-uplets dans un nouveau flux. De plus, lorsque le dernier *batch* est récupéré, il est possible d'y appliquer des opérations simples avant de le renvoyer. Nous définissons les sélections, les projections et le renommage sur flux par l'application de ces opérations sur la fenêtre *batch*.

Afin de montrer que cette définition est viable, nous démontrons grâce au théorème 5.1 que la sélection sur un flux correspond à la sémantique intuitive suivante :

$$\sigma_c(S) = \mathcal{I}_S(\nu(S[B])) = \{s \in S, c(s)\}$$

Corollaire 5.2 **Définition de la sélection, projection et renommage sur flux**

Soit S un flux,
Soit ν un opérateur pouvant être σ, Π ou ρ,
Son application sur un flux est définie par :

$$\nu S = \mathcal{I}_{\mathcal{S}}(\nu(S[B]))$$

5.3.2 Commutativité et associativité

Nous voyons désormais les règles de commutativité et d'associativité. Nous détaillons premièrement le cas des projections.

Comme la projection ne perturbe pas les cardinalités ou les ordres, il n'y a pas d'impact important. De plus, l'opérateur permute avec l'opérateur de séquence de fenêtres et les *streamers*. Le tableau 5.1 liste l'ensemble des règles de transformations permettant de pousser les projections au plus proche des sources lors d'un processus d'optimisation logique.

L'opérateur de sélection est plus délicat. En effet, il perturbe la cardinalité du flux. Ainsi, l'opérateur de fenêtre positionnel ne peut pas permuter avec un opérateur de sélection. Il est possible de fournir un contre-exemple interdisant cette permutation.

Exemple 5.2 :

Soit le flux exemple CPU, considérons les requêtes suivantes :

1. $\sigma_{cpu>50}(CPU[N\ 10\ 1])$ *est l'ensemble des relevés supérieurs à 50% des 10 dernières mesures à chaque nouvelle mesure.*

2. $(\sigma_{cpu>50}CPU)[N\ 10\ 1])$ *est l'ensemble des 10 dernières valeurs des relevés supérieurs à 50% chaque nouvelle mesure >50%.*

Les tailles des fenêtres de ces requêtes sont différentes. Dans la seconde requête, la largeur est constante à 10 n-uplets. Alors que dans le premier cas, s'il existe des n-uplets $\leq 50\%$ *alors la taille est inférieure à* 10 *n-uplets.*

Toutefois, dans le cadre particulier des fenêtres temporelles et de la fenêtre accumulative $[\infty]$ la propriété de commutativité est vraie [2]. De façon similaire, la commutativité avec $\mathcal{R}^u_{\mathcal{S}}$ n'est pas autorisé. Les autres règles autorisant la commutativité sont pour la plupart issues de l'algèbre relationnelle. La table 5.2 liste l'ensemble des règles pour pousser les sélections.

Enfin, une dernière commutativité est notable : la manipulation temporelle. Sa commutativité est intuitive à appréhender. En effet, la manipulation temporelle ap-

Proposition 5.3.1 **Commutativité de la manipulation temporelle**

L'opérateur de manipulation temporelle est commutatif avec tous les opérateurs relationnels.

2. Car ces définitions de fenêtres ne dépendent pas de la fonction τ_S

(E = entité, R = relation temporelle, S = flux)

Hypothèse	Condition	Résultat
$\Pi_a E$	$a = attr(E)$	E
$\Pi_a \Pi_b E$		$\Pi_a E$
$\Pi_a \sigma_c E$		$\Pi_a \sigma_c \Pi_{a \cup attr(c)} E$
$\Pi_a e^c_{f(b)} E$		$\Pi_a e^c_{f(b)} \Pi_{(a \setminus c) \cup b} E$
$\Pi_a \rho_{y/x} E$	$y \in a$	$\rho_{y/x} \Pi_{a \setminus \{y\}, x} E$
	$y \notin a$	$\rho_{y/x} \Pi_a E$
$\Pi_a (R_1 \bowtie R_2)$		$\Pi_a (\Pi_{Attr(R_1) \cap a} R_1 \bowtie \Pi_{Attr(R_2) \cap a} R_2)$
$\Pi_a S[\alpha, \beta, \gamma]$		$(\Pi_{a \cup \tau} S)[\alpha, \beta, \gamma]$
$\Pi_a \mathcal{I}_{\mathcal{S}}(R)$		$\mathcal{I}_{\mathcal{S}}(\Pi_{a \setminus \tau} R)$
$\Pi_a \mathcal{D}_{\mathcal{S}}(R)$		$\mathcal{D}_{\mathcal{S}}(\Pi_{a \setminus \tau} R)$
$\Pi_a \mathcal{R}^u_{\mathcal{S}}(R)$		$\mathcal{R}^u_{\mathcal{S}}(\Pi_{a \setminus \tau} R)$
$\Pi_a \mathcal{R}^r_{\mathcal{S}}(R)$		$\mathcal{R}^r_{\mathcal{S}}(\Pi_{a \setminus \tau} R)$
$\Pi_a \mathcal{D}^f_c(R)$		$\mathcal{D}^f_c(\Pi_a R)$
$\Pi_{a\ b} G_{f(c)} R$		$\Pi_{a\ b} G_{f(c)} \Pi_{b \cup c} R$
$\Pi_a (R_1 \cup R_2)$		$(\Pi_a R_1) \cup (\Pi_a R_2)$

TABLE 5.1 – Table des règles de commutativité de la projection Π

(E = entité, R = relation temporelle, S = flux)

Hypothèse	Condition	Résultat
$\sigma_c \sigma_{c'} E$		$\sigma_{c \wedge c'} E$
$\sigma_c e^a_{f(b)} E$	$a \notin attr(c)$	$e^a_{f(b)} \sigma_c E$
$\sigma_c \rho_{y/x} E$	$x \in attr(c)$	$\rho_{y/x} \sigma_{\text{replace}(x,y,c)} E$
	$x \notin attr(c)$	$\rho_{y/x} \sigma_c E$
$\sigma_c (R_1 \bowtie_d R_2)$	$attr(c) \subseteq attr(R_1) \setminus attr(R_2)$	$(\sigma_c(R_1)) \bowtie_d R_2$
	$attr(c) \subseteq attr(R_2) \setminus attr(R_1)$	$R_1 \bowtie_d (\sigma_c(R_2))$
	sinon	$R_1 \bowtie_{d \wedge c} R_2$
$\sigma_c S[\alpha, \beta, \gamma]$	$[\alpha, \beta, \gamma] = [\infty]$	$(\sigma_c S)[\alpha, \beta, \gamma]$
	α, β, γ temporels	
$\sigma_c \mathcal{I}_{\mathcal{S}}(R)$		$\mathcal{I}_{\mathcal{S}}(\sigma_c R)$
$\sigma_c \mathcal{D}_{\mathcal{S}}(R)$		$\mathcal{D}_{\mathcal{S}}(\sigma_c R)$
$\sigma_c \mathcal{R}^r_{\mathcal{S}}(R)$		$\mathcal{R}^r_{\mathcal{S}}(\sigma_c R)$
$\sigma_c \mathcal{D}^f_c(R)$		$\mathcal{D}^f_c(\sigma_c R)$
$\sigma_{c\ b} G_{f(c)} R$	$attr(c) \subseteq b$	$\sigma_{c\ b} G_{f(c)} \Pi_{b \cup c} R$
$\sigma_c (R_1 \cup R_2)$		$(\sigma_c R_1) \cup (\sigma_c R_2)$

TABLE 5.2 – Table des règles de commutativité de la sélection σ

plique une fonction f au *batch* utilisé pour récupérer la relation instantanée d'une relation temporelle. Or, les opérateurs relationnels sont décomposables en $\nu(R)(b) = \nu' R(b)$. L'application d'une fonction f donne : $\nu(R)(f(b)) = \nu' R(f(b))$, avec $R(f(b))$ correspondant à l'application de la manipulation temporelle.

Nous abordons maintenant la question de l'associativité. Dans le cadre de nos définitions, les propriétés d'associativités issues de l'algèbre relationnelle sont toujours vraies.

Proposition 5.3.2 **Associativité des jointures et unions**

Les opérations de jointure et d'union sont associatives pour les définitions de $\Phi^{\times}$ et $\Phi^{\cup}$ données en section 4.2.

Ceci est dû au fait que l'ordre lexicographique est lui aussi un ordre associatif. Les ordres lexicographiques induits par $\mathbb{I} \times (\mathbb{I} \times \mathbb{I})$ ou $(\mathbb{I} \times \mathbb{I}) \times \mathbb{I}$ sont équivalents (démonstration simple). Si une autre fonction est utilisée, il est nécessaire de vérifier que les ordres fournis sont toujours équivalents.

Nous avons vu plusieurs propriétés utilisables par les implémentations et optimiseurs. Nous avons désormais plusieurs outils pour construire une optimisation logique suffisamment efficace. Nous explorons désormais la transposabilité, qui peut avoir un impact dans le cadre des partages de requêtes.

5.3.3 Transposabilité des opérateurs

Dans cette section, nous étudions la propriété de transposabilité des opérateurs telle que présentée dans la définition 4.4.4. Ces propriétés ont des conséquences concrétées dans le cadre du partage des requêtes. En effet, lors du déploiement d'une requête, il peut être utile de voir qu'une sous-requête est déjà en cours de traitement. Si tel est le cas, comment réutiliser ses résultats et à quels instants ?

Tout d'abord, nous remarquons que les opérateurs issus de l'algèbre relationnelle sont naturellement transposables (prop 5.3.3). Ce résultat est intuitif, car l'opérateur de manipulation temporelle commute avec ces opérateurs.

Proposition 5.3.3 **Transposabilité des opérateurs de Codd**

Les opérateurs issus de l'algèbre relationnelle sont naturellement transposables sur $\mathbb{T}$.

Les *streamers* sensibles ne sont pas gênés par le changement de *timestamp* de départ t_0 dû à la transposabilité. Il est intéressant de voir que les *streamers* subissent tout de même le phénomène d'élaboration (car $\mathcal{I}_{\mathcal{S}}$ à t_0 dépend du fait que $R(b) = \emptyset$ pour $b < (t_0, 0)$). Toutefois, comme les transposabilités sont calculées lorsque la requête est stable, nous obtenons une transposabilité naturelle comme précisée dans la propriété 5.3.4. Les *streamers* périodiques sont eux naturellement transposables sur des multiples du taux r comme vu dans l'exemple 4.4.2.

Nous considérons maintenant le cas plus complexe des fenêtres. Soit $Q_1 = (S, t_0)$ un flux transposable par $Q_2 = (S', t_1)$ avec $t_1 > t_0$, nous souhaitons savoir si les

Proposition 5.3.4 **Transposabilité des *streamers***

Les *streamers* sensibles $\mathcal{I}_S$, $\mathcal{D}_S$ et $\mathcal{R}^u_S$ sont naturellement transposables sur $\mathbb{T}$. Le *streamer* périodique $\mathcal{R}^r_S$ est naturellement transposable sur $\{t \in \mathbb{T}, t \equiv t_0[r]\}$.

deux DSF simplifiées $[\alpha, \beta, r]$ et $[\alpha', \beta', r]$ sont équivalentes sur ces *timestamps*. Avant de détailler les résultats, introduisons des constantes dont la description et les valeurs (dans le cas temporel ou positionnel) sont présentées dans le tableau suivant :

Notation	Description	Temporel	Positionnel
D	Décalage temporel implicite entre deux descriptions	0	$\tau^{-1}_{(S,t_0)}((t_1,0)^-)+1$
B_t	Valeur minimale autorisée pour une description avec pour *timestamp* initial t	t	0
K	Facteur de synchronisation. Correspond au nombre de fenêtres évaluées sur Q_1 lorsque Q_2 démarre	$\dfrac{\beta'(0)-\beta(0)+D}{r}$	

Ces constantes permettent d'exprimer plus aisément les résultats. Le théorème général 5.2 permet de trouver une condition suffisante pour permettre la transposition de la DSF $[\alpha, \beta, r]$ par $\sigma_{t \geq t_1}[\alpha', \beta', r]$. En effet, la transposition n'est pas naturelle, car la description de la fenêtre peut changer, mais aussi il est nécessaire de réinitialiser le flux à t_1 en supprimant les données inférieures à ce *timestamp*.

Théorème 5.2 **Théorème général de la transposabilité des DSF**

$$\text{Si} \begin{cases} K & \in & \mathbb{N} \\ \alpha'(j) & = & \max(B_{t_1}, \alpha(j+K)-D) \\ \beta'(j) & = & \beta(j+K)-D \end{cases},$$

Alors, l'opérateur $[\alpha, \beta, r]$ est transposable sur t_1 par $\sigma_{\tau \geq t_1}[\alpha', \beta', r]$.

Avoir K entier est intuitif, car cette constante décrit la synchronisation entre les requêtes. Ainsi, il est possible de prouver que $\gamma'(t,i) = \gamma(t,i) - K$. En considérant ceci, le remplacement de j par $j+K$ devient logique. Il ne reste qu'à limiter la borne inférieure par B_{t_1} et nous obtenons le résultat.

Afin de mettre ce théorème en pratique, nous présentons la proposition 5.3.5 qui présente les conditions nécessaires et suffisantes à vérifier pour entrer dans le cadre du théorème 5.2.

Exemple 5.3 :

Considérons les requêtes suivantes : $\begin{cases} Q_1 = (S[2is+3s, 5is+3s, 5s], t_0) \\ Q_2 = (S[2is+2s, 5is+8s, 5s], t_1) \end{cases}$

Nous souhaitons réutiliser les résultats de Q_1 pour Q_2. Ici, $K = \frac{(8s+t_1)-(3s+t_0)}{5s} = 1 + \frac{t_1-t_0}{5s}$. La proposition 5.3.5 nous indique que, pour faire un partage, t_1 doit être égal à $t_0 + n * 5s$ avec $n \in \mathbb{N}$.

Proposition 5.3.5 **Transposabilité des DSF linéaires**

Si les descriptions de fenêtres sont de la forme suivante :

$$\begin{array}{ll} \alpha(j) = \max(aj+b, B_{t_0}) & \alpha'(j) = \max(a'j+b', B_{t_1}) \\ \beta(j) = cj+d & \beta'(j) = c'j+d' \end{array}$$

Alors, les DSF vérifient le théorème 5.2 ssi. : $K = \dfrac{d'-d+D}{r} \in \mathbb{N}$,

$$\begin{cases} a = a' \\ c = c' \\ c = r \text{ si } K \neq 0 \end{cases} \quad \text{et} \quad \begin{cases} b' = b + aK - D & \text{si } a \neq 0 \\ \max(B_{t_1}, b') = \max(B_{t_1}, b-D) & \text{si } a = 0 \end{cases}$$

Corollaire 5.3 **Transposabilité pseudo-naturelle des DSF linéaires**

Si les descriptions sont de la forme suivante :

$$\begin{array}{lll} \alpha(i) - B_{t_0} = \alpha'(i) - B_{t_1} = & \max(ai+b, 0) \\ \beta(i) - B_{t_0} = \beta'(i) - B_{t_1} = & ci+d \end{array}$$

Alors, les DSF vérifient le théorème 5.2 si et seulement si : $c = r \wedge ((a = 0 \wedge b \leq 0) \vee a = r)$ et

$$\begin{array}{rcll} t_1 - t_0 & \in & r\mathbb{N} & \text{temporel} \\ \tau^{-1}_{(S,t_0)}((t_1,0)^-) + 1 & \in & r\mathbb{N} & \text{positionnel} \end{array}$$

Comme $K \neq 0$, c doit être égal au taux r. Et comme $a \neq 0$, b' doit être égal à $3s + t_0 + 2s * K = 3s + t_0 + 2s * (1+n)$. Ainsi, nous obtenons $b' - t_1 = 5s + n * 2s + t_0 - t_1 = 5s - n * 3s$. Le problème est maintenant réduit à une équation du premier degré pour vérifier si les deux DSF sont synchronisés. Dans notre cas, $2 = 5 - 3n \Rightarrow n = 1$ est possible. Nous pouvons partager le résultat en nous plaçant à $t_1 = t_0 = 5s$. La figure 5.2 illustre le résultat de la transposition.

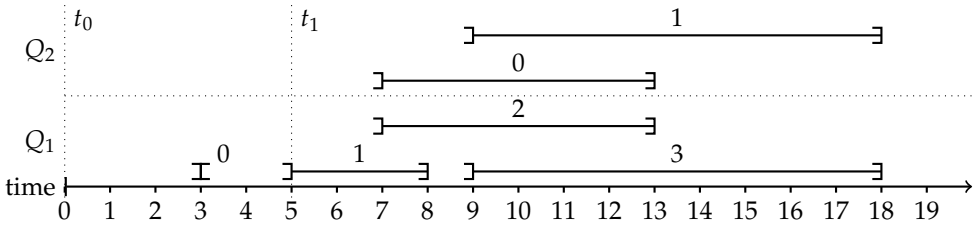


FIGURE 5.2 – Résultat de la transposition sur l'opérateur de fenêtre

Le corollaire 5.3 permet de vérifier l'équivalence des deux descriptions dans le cas linéaire et dans le cas où les coefficients sont identiques (à l'initialisation près). La synchronisation est plus claire et se réduit à une condition sur t_1 relativement à t_0. En

particulier, ce résultat permet de valider que les fenêtres usuelles présentées dans le tableau 4.2 soient toutes transposables tel quel (à l'opérateur $\sigma_{\tau \geq t_1}$ près).

5.4 Conclusion

Nous avons réussi à formaliser de nombreuses approches actuelles avec Astral. Cette algèbre permet une désambiguïsation des sémantiques des requêtes. De plus, l'expression d'une requête n'est pas attachée à des concepts liés à une implémentation particulière. Ainsi, Astral est un bon candidat comme langage d'interrogation instantané et continu.

De plus, nous avons pu prouver certains résultats fondamentaux sur les équivalences de requêtes. Tout d'abord, nous avons montré que nos définitions de fenêtres et de *streamers* étaient consistantes par le théorème de transmission temporelle. Ensuite, nous avons pu montrer des résultats exploitables pour l'optimisation logique d'une requête. Enfin, nous avons vu les résultats liés à la transposabilité qui sont exploitables pour le partage de sous-requêtes.

Toutefois, nous ne sommes pas capables de qualifier formellement l'expressivité d'Astral. Dans le domaine relationnel, le théorème de Codd [Codd 1972] permet de prouver l'algèbre relationnelle est équivalente au *datalog non récursif avec négation*. Mais dans le domaine des requêtes continues et des flux de données, il n'existe pas de classe logique permettant de qualifier la puissance d'expression d'un langage.

Astral a été présenté tout d'abord dans [Petit 2010] en mettant en avant les sémantiques des fenêtres. Elle a aussi été présenté dans [Petit 2012c] en relevant les problèmes d'ordres observés dans la gestion de flux.

Nous avons présenté l'algèbre Astral et analysé ses points forts et limitations. Nous nous servons de ce modèle comme base fondamentale pour concevoir notre système d'évaluation de requête continue que nous présentons dans la partie suivante.

Tables de l'algèbre Astral

109

Définitions

Théorèmes

Propositions

Corollaires

Partie III

Mise en œuvre et couplage relationnel

La définition d'Astral nous permet de créer des requêtes pouvant manipuler toutes les données des systèmes observés. Cette partie présente l'exécution de ces requêtes. Nous dédions un chapitre à l'intergiciel Astronef qui est un système de gestion de flux de données extensible. Ceci forme une bonne base de travail, car il est possible d'ajouter facilement des composants, ce que nous faisons pour ajouter l'utilisation d'un support persistant relationnel avec Asteroid. De plus, nous proposons un schéma conceptuel de base de données pour utiliser ce support lors de l'observation de systèmes.

« Is this thing on? I don't think this thing is on. Hello! [...] Confangled modern doohickeys. »

Grany Smith

6

Astronef : De l'expression à l'exécution

Astral est une algèbre permettant d'exécuter des requêtes continues sans ambiguïtés sémantiques. Les concepts développés au chapitre 4 ne sont pas attachés à un mode d'exécution ou à une implémentation particulière. Dans ce chapitre, nous présentons l'intergiciel *Astronef*[1] permettant l'exécution de requêtes continues Astral. Dans cette mise en œuvre, nous focalisons notre attention sur trois points en particulier :

Conformité à Astral : Nous avons développé un modèle expressif et libre de toute ambiguïté. Il est important que les composants développés dans cette mise en œuvre correspondent aux sémantiques qu'ils sont capables d'exécuter en Astral.

Extensibilité : L'utilisateur doit pouvoir adapter l'intergiciel à son utilisation. Ainsi, il doit pouvoir ajouter ou reconfigurer des composants à la volée.

Optimisation : L'intergiciel doit être capable de fournir un plan d'exécution efficace pour une requête donnée.

La section 6.1 présente l'architecture générale d'Astronef. La section 6.2 présente notre approche à base de règle pour construire un plan de requête. La section 6.3 présente la première partie de l'optimisation du plan : l'optimisation logique, permettant de réécrire une requête de manière plus efficace. Puis nous détaillons en section 6.4 la

1. Astral Optimization and Execution Framework

seconde partie de l'optimisation : l'optimisation physique permettant de sélectionner les meilleurs composants pour exécuter le plan de requête. Enfin, nous analysons en section 6.5 les impacts de l'intégration d'un nouveau composant à l'architecture. Nous concluons ce chapitre en section 6.6.

6.1 Architecture d'Astronef

Dans cette section, nous détaillons les éléments d'architecture qui permettent d'instancier un processus de traitement pour une requête Astral. Nous abordons premièrement les principes architecturaux utilisés. Ensuite, nous détaillons les différents composants utilisés dans Astronef.

6.1.1 Choix d'architecture logicielle

Avant de détailler l'architecture de notre système de traitement de requêtes continues, nous rappelons brièvement les principaux éléments des architectures à services et des composants orientés services que nous utilisons par la suite.

Architecture à service

Les architectures à services permettent aux applications d'être assemblés sous forme de blocs réutilisables : des *services*. Un *service* est défini par une spécification (ou *description*, ou *contrat*), qui décrit sa syntaxe, son comportement, sa sémantique ainsi que sa dépendance aux autres services. Dans les architectures à services, les services interagissent via un patron récurrent d'interaction (fig 6.1). Un fournisseur de service

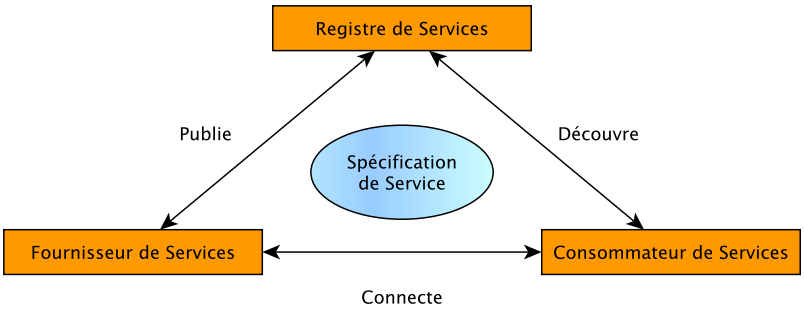


FIGURE 6.1 – Patron d'interaction de service

va publier sa spécification à un registre. Un consommateur de service découvre le service fourni par une requête sur le registre. Enfin, le consommateur et le fournisseur se connectent. Le point clé est que la résolution est faite à l'exécution.

Architecture à composants orientés services

Le modèle d'architecture à composants orientés services [Cervantes 2004] permet la mise en œuvre d'applications à base de services dans le paradigme de la programmation par composants. Le principe est de séparer les mécanismes des architectures à

services du code implémentant le comportement du service fourni. Ainsi, les principes d'un tel modèle sont :

– Un service offre une fonctionnalité.

– Un service est caractérisé par sa spécification.

– Les composants implémentent des spécifications de services, qui peuvent eux-mêmes dépendre, du fait de leurs implémentations, d'autres services.

– Les patrons d'interactions de services sont utilisés pour résoudre les dépendances de services à l'exécution.

– Les compositions sont décrites en terme de spécifications de services.

– Tout composant peut être substitué par un autre si les spécifications de services sont identiques.

Le modèle combine les idées de composants et de services. De plus, en s'inspirant des modèles tels que Fractal [Bruneton 2006], chaque composant possède un ensemble de propriétés (ou attributs) configurables. Nous obtenons aussi le pouvoir d'instancier (grâce aux fabriques) des composants à partir de configurations.

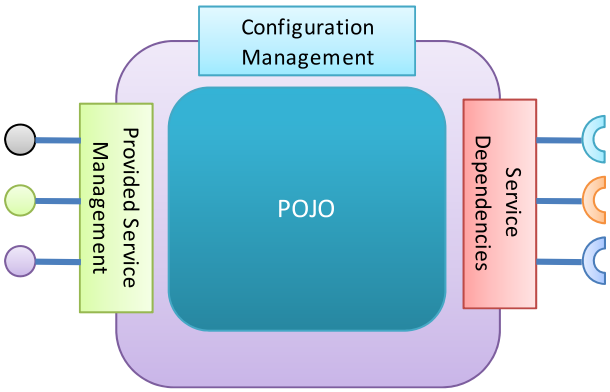


FIGURE 6.2 – Un composant iPojo

La figure 6.2 représente un composant dans l'implémentation *iPojo* [Escoffier 2007]. Le code du composant (le *POJO*, *Plain Old Java Object*) est embarqué dans un conteneur auxquels sont accrochés des gestionnaires. Les trois couramment utilisés sont les gestionnaires de dépendances, de production de service, et de configuration. Ainsi, un composant peut s'exposer sur un registre, tout en dépendant d'autres services et en étant configurable.

6.1.2 Les composants et services d'Astronef

L'architecture d'Astronef est entièrement dirigée par les approches de composants orientés services. De multiples composants sont créés et instanciés à chaque construction de requête. Afin de pouvoir exécuter les requêtes, nous définissons trois services centraux pour chacune d'elles :

Les services *EventProcessor* ont deux primitives, l'une exécute une tâche, l'autre indique les autres *EventProcessors* devant être exécutés avant sa propre exécution.

Le service *Scheduler* permet la planification. Ses primitives permettent aux différents *EventProcessor* d'exprimer leur volonté de s'exécuter. Ce service établit l'ordonnancement de ces demandes en fonction des contraintes des *EventProcessor*s.

Le service *QueryRuntime* permet d'exécuter une requête. Il a son propre *Scheduler* et utilise la primitive *next* de celui-ci pour connaître la prochaine tâche à exécuter.

Nous adoptons l'approche proposée par [Carney 2003] en gérant l'ordonnancement par un service externe aux opérateurs, comme présenté dans la section 3.2, notamment dans la figure 3.5. Maintenant que nous avons vu les différents services nécessaires à l'exécution. Nous avons plusieurs types de composants que nous pouvons instancier à l'exécution à l'aide de services de fabriques :

Les composants flux ou relations (entités) : Fournissent les primitives nécessaires à leur manipulation par Astral. Ces entités servent de résultats intermédiaires (ou de tampons). De plus, ils permettent un service de notification : en cas de changement, les *EventProcessor* abonnés sont notifiés. Ainsi, ces composants requièrent le *Scheduler* de la requête pour demander l'exécution de leurs abonnés.

Les composants sources : Une source requiert une entité qu'elle alimente grâce aux données issues d'origines diverses (protocoles réseaux, fichiers,...). Ce composant peut requérir le *Scheduler* de sa requête pour, entre autres, notifier sa fin de vie ce qui peut engendrer la fin de vie de la requête.

Les composants opérateurs : Nécessitent n entités en lecture, et une autre particulière en écriture. Ce composant doit fournir le service *EventProcessor*. Les implémentations des opérateurs bloquants peuvent faire appel au *Scheduler* pour planifier des exécutions ponctuelles.

Les composants puits : Nécessitent une entité en lecture. Ce composant fournit le service *EventProcessor* et doit être non bloquant.

Ainsi pour créer une requête : l'utilisateur de l'intergiciel doit fournir un ou plusieurs composants sources et un composant puits. Par la suite, il demande à Astronef de lui instancier ses sources et son puits en configurant les composants selon sa volonté. Enfin, il spécifie l'expression algébrique liant les sources au puits. Astronef instancie alors *automatiquement* les composants nécessaires à l'évaluation de cette requête (c.f. 6.2).

De l'importance de la réutilisation

Astronef permet une grande flexibilité grâce aux composants orientés services. Tout d'abord, chaque composant peut être configuré tout en gérant son cycle de vie, ce qui permet de réutiliser le même module de plusieurs manières. Par exemple, supposons l'existence d'une source capable de récupérer une information périodiquement sur un dispositif D via un protocole donné. Cette source peut être utilisée pour plusieurs requêtes sous différentes instances en ayant plusieurs configurations : différents dispositifs ou périodes d'acquisitions.

Cette abstraction sous forme de services permet surtout la substitution. En effet, nous pouvons remplacer tout composant par un autre supportant le même service. Nous utilisons ce principe pour sélectionner les meilleurs composants pour remplir le

plus efficacement leurs rôles. De plus, l'utilisateur peut apporter ses propres implémentations pour étendre les capacités de l'intergiciel.

6.2 De l'algèbre aux composants

À tout composant est attachée une fabrique permettant de créer les instances. Pour construire une requête, l'utilisateur d'Astronef peut faire appel à chaque fabrique pour instancier chaque source, opérateur, entité, et puits. Une fois les composants construits et liés entre eux, il peut demander au moteur de créer un service *QueryRuntime* pour exécuter la requête. Toutefois, nous souhaitons pouvoir exploiter les connaissances d'Astral pour aider à la construction de ce plan d'exécution. Ainsi, l'utilisateur n'a pas à intervenir lors de la construction de la requête.

En effet, de son point de vue, il ne doit écrire que l'expression algébrique de sa requête et il doit être garanti d'une mise en œuvre efficace. Cette section détaille notre approche. Tout d'abord, nous détaillons l'idée de la construction du plan d'exécution par règles. Ensuite, nous présentons les choix technologiques mis en œuvre. Enfin, nous voyons comment préparer la requête à l'optimisation que nous détaillons dans les sections suivantes.

6.2.1 Approche de la construction de requêtes par inférence

Pour atteindre une construction *automatique*, nous avons choisi de mettre en place une approche à base de règles. En partant de l'expression algébrique, nous itérons suivant plusieurs règles d'inférence jusqu'à l'obtention d'un plan de requête. Nous remarquons plusieurs avantages à une telle approche :

– Intégration naturelle des connaissances. Les propriétés de l'algèbre peuvent s'exprimer dans un langage déclaratif pour les exploiter.

– Expression de la sémantique des composants par l'algèbre. Comme il est nécessaire d'associer la sémantique d'Astral aux composants logiciels, il est nécessaire de spécifier à quelle opération chaque composant (et chaque paramètre de configuration) répond. Cela permet une clarification des sémantiques d'exécution.

– Extensibilité très forte. En considérant que l'ajout de nouveaux composants peut se faire via l'ajout de nouvelles règles, il devient aisé d'étendre le système pour permettre des opérateurs, ou des optimisations qui n'étaient pas prévues.

Optimisation par heuristiques

Afin de mettre en œuvre cet ensemble de règles pour obtenir un plan de requête efficace, nous introduisons une optimisation logique et physique à l'image des optimisations des SGBD. Nous restructurons la structure de l'expression algébrique dans la section 6.3 pour qu'elle soit plus optimisée. Ensuite, nous sélectionnons les meilleurs composants et les meilleures configurations pour mettre en œuvre cette nouvelle expression, dans la section 6.4.

Toutefois, dans notre contexte, nous ne pouvons pas appliquer directement certaines optimisations des SGBDs. Contrairement aux règles habituelles, nous ne réordonnons pas les jointures du fait du théorème 4.1. De plus, la notion d'entrée-sortie est réduite à une notion de résultats intermédiaires, même au niveau des sources (rendant l'utilisation des index moins pertinents que sur disque). En effet, la performance se mesure généralement en débit supporté et les débits de sources sont a priori inconnus. Au total, un gain de performance est mesuré sur un gain de consommation processeur et, à moindre mesure, sur une consommation mémoire contrôlée (ou bornée).

De plus, Astral possède plus d'opérateurs et de restrictions que l'algèbre relationnelle ce qui rend l'espace de recherche plus large. Ainsi, nous développons plusieurs heuristiques nous permettant de faire les différentes réécritures par applications de règles.

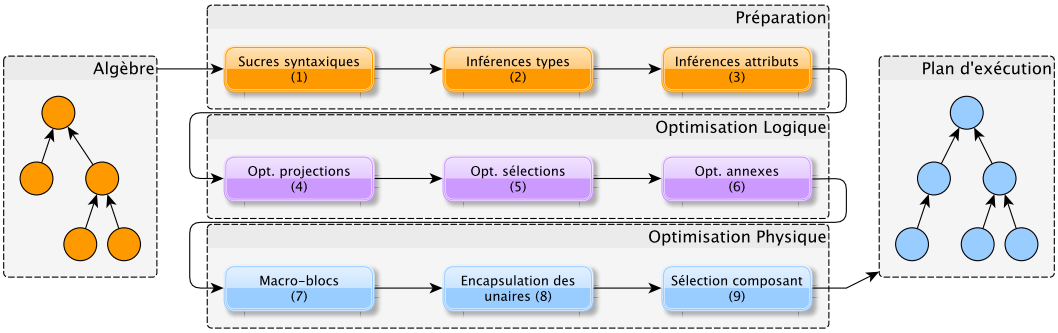


FIGURE 6.3 – Processus d'optimisation d'une requête algébrique dans Astronef

La figure 6.3 représente le processus total d'optimisation d'une requête algébrique. Nous pouvons clairement distinguer trois parties, la phase de préparation, puis l'optimisation logique et enfin physique. Chaque sous-tâche est détaillée dans la suite de ce chapitre.

6.2.2 Expression d'une requête

Nous utilisons un moteur capable d'exécuter du Prolog [2], langage de programmation logique, pour appliquer nos règles. Avant de détailler l'ensemble de ces règles, nous présentons d'abord la structure d'une expression. Toute expression est structurée sous forme d'arbre, il est possible de représenter une requête avec des nœuds de la forme [3] :

$$[\ \underbrace{A}_{\text{Nature du nœud}}\ ,\ \underbrace{B}_{\text{Ensemble de propriétés}}\ ,\ \underbrace{C}_{\text{Liste de nœuds fils}}\ ]$$

2. En réalité, le langage utilisé dans l'implémentation (*Prova* [Kozlenkov 2004]) est un dérivé de Prolog mieux adapté à l'intégration avec Java. Mais le principe reste similaire.

3. L'utilisation d'objets de propriétés fait partie du langage *Prova*, mais cela reste formalisable en Prolog standard avec une liste d'éléments [clé,valeur].

Exemple 6.1 :

Soit R une source déclarée dans le système, nous souhaitons exécuter la requête $\sigma_{id=1}R$. L'expression de cette requête en Prolog est la suivante :

```
1 [sigma, {"condition":"id=1"}, [
2     [source, {id:"R"}, []]
3 ]]
```

Nous remarquons que cette syntaxe est très similaire à *XML* : chaque nœud possède un nom, un ensemble de propriétés et un ensemble de fils, tout comme les balises *XML*. C'est pour cela que ce langage est celui utilisé en pratique pour spécifier des requêtes dans le prototype. Il est ensuite traduit en expression utilisable en Prolog. L'ensemble des nœuds possibles correspond aux différents opérateurs de l'algèbre. Nous ne détaillons pas cet aspect dans ce manuscrit. Le lecteur peut consulter le manuel sur la page web suivante http://code.google.com/p/astral/wiki/XMLSyntax pour trouver les expressions supportées.

6.2.3 Préparation de la requête

Afin de pouvoir commencer à raisonner sur l'expression algébrique, nous devons tout d'abord préparer l'arbre de requête, comme présenté dans la figure 6.3. Nous présentons l'ensemble de la phase de préparation composé du remplacement des sucres syntaxiques, de l'inférence des types et de l'inférences des attributs des résultats intermédiaires.

Sucres syntaxiques

Un sucre syntaxique permet d'aider l'utilisateur à écrire et lire son expression algébrique. Afin de pouvoir appliquer des règles sur l'arbre de requête, nous devons remplacer ces sucres syntaxiques par leurs expressions en termes d'opérateurs primitifs Astral. Par exemple, l'opérateur $\bowtie$ est une composition d'opérateur $\bowtie$ et $\mathcal{D}^c_f$, l'opérateur $[L]$ correspond à l'opérateur $[j, j, 1]$.

Règle 1 **Sucres syntaxiques**

Le remplacement des sucres syntaxiques transformant l'expression $[A1, B1, C1]$ en $[A2, B2, C2]$ se fait par le prédicat :

sugar($[A1, B1, C1]$, $[A2, B2, C2]$).

Ce prédicat est appliqué tant qu'il peut l'être (de manière itérative).

Exemple 6.2 :

Nous nous proposons de remplacer le sucre syntaxique R^{t_0} en $\mathcal{D}^{t0}(R)$. R^{t_0} est exprimé par le nœud freeze possédant le paramètre « at » et $\mathcal{D}$ est exprimé par un nœud time-transform possédant un paramètre « description ». Voici l'expression de ce remplacement :

```
sugar(
    [freeze, % Lorsqu'il existe un noeud freeze
        {at: t0},
        Children],
    [timetransform, % ... le remplacer par timetransform
        {"description": [ % Avec un parametre description
        % contenant le type de manipulation voulue
            {"type": "freeze",
            "at": t0}
        ]},
        Children]
).
```

Ce prédicat est de plus utilisé pour analyser les chaînes de caractères ce qui permet, par exemple, d'extraire la liste des attributs de conditions ou d'expressions.

Inférences des types et attributs

Afin de pouvoir traiter correctement les différents nœuds, il nous faut inférer les deux propriétés majeures de chaque nœud d'une requête qui va définir la nature de son résultat intermédiaire : son type (*flux* ou *relation*) et ses attributs. Pour cela, il faut être assuré que les sources exposent leurs attributs et types dans leurs propriétés. Par la suite, un programme applique ces règles de façon récursive. Les résultats sont stockés dans les propriétés *type* et *attributes*.

Règle 2 **Inférences des types**

L'inférence du type $Type$ de l'expression $[A, B, C]$ dont les types fils sont $TypesFils = [T1, ...]$ se fait par le prédicat :

$$\textbf{typerules}([A, B, C, TypesFils], Type).$$

Ce prédicat s'applique de manière récursive.

Règle 3 **Inférences des attributs**

L'inférence de la liste d'attributs d'un nœud $[A, B, C]$ dont les attributs fils sont $AttributsFils$ se fait par le prédicat :

$$\textbf{attribrules}([A, B, C, AttributsFils], Attributs).$$

Ce prédicat s'applique de manière récursive.

Exemple 6.3 :

Pour la définition de la jointure dans Astronef, les règles sont simples. La liste des attributs est l'union des listes d'attributs fils. Et le type de la jointure est relationnel si ses fils le sont aussi.

```
typerules([join,_,_, TypesFils], Type):-
    allequal(TypesFils,Type), % Verification que tous soient relationnels
    relation(Type), !.
attribrules([join,_,_,AttributsFils], Attributs):- !,
    union(AttributsFils,Attributs).
```

Voyons maintenant un exemple simple sur la jointure de deux relations.

Exemple 6.4 :
Soit la requête $S[B] \bowtie_{a<b} R$. *Son expression correspondante en Prolog est :*

```
[join, {condition: "a < b"}, [
    [window, {description: [{type: "B"}]}, [
        [source, {id: "S", attributes: ["a", "T"], type: "stream"}, []]
    ]],
    [source, {id: "R", attributes=["b", "c", "d"], type: "relation"}, []]
]]
```

Après la phase de préparation, nous obtenons l'expression suivante :

```
[join, {condition: "a < b", conditionAttributes: ["a", "b"],
        type: "relation", attributes: ["a", "b", "c", "d", "T"]}, [
    [window, {description: [{type: "B"}],
              type: "relation", attributes: ["a", "T"]}, [
        [source, {id: "S", attributes: ["a", "T"], type: "stream"}, []]
    ]],
    [source, {id: "R", attributes=["b", "c", "d"], type: "relation"}, []]
]]
```

Nous remarquons que les propriétés type et attributes existent sur tous les nœuds maintenant. De plus, la propriété conditionAttributes a été placée sur la jointure.

Après le remplacement des sucres syntaxiques et l'inférence des types et des attributs, l'évaluation procède à l'optimisation logique.

6.3 Optimisation logique

La section 6.2 a présenté la phase de préparation de la requête et produit une représentation sous forme d'arbre. Cette section présente la phase d'optimisation logique qui s'appuie sur les travaux présentés au chapitre 5.

6.3.1 Projection et sélection

Nous faisons l'hypothèse 6.1 qui permet d'intégrer des projections et sélections à tout endroit de l'arbre de requête. Les conséquences de cela sont de pouvoir placer des sélections et projections au plus près des sources.

Cette optimisation permet de réduire l'empreinte mémoire des résultats intermédiaires. Par conséquence, il y a moins de données à traiter ce qui implique un gain de calcul. Pour atteindre ce résultat, il est nécessaire d'appliquer les résultats que nous donne Astral. Ces résultats ont été présentés dans le chapitre 5. Voici les deux prédicats permettant une telle optimisation :

Hypothèse 6.1 **Heuristique du coût de sélection et projection**

Le coût processeur de la sélection et de la projection peuvent être considérés comme négligeables.

Règle 4 **Optimisation des projections**

La transformation d'un nœud contenant une projection afin de l'appliquer sur ses nœuds fils est gérée par le prédicat suivant :

pushprojectionrule($[pi, BPi, [[A, B, C]]], [AOut, BOut, COut]$).

Ce prédicat est appliqué de manière itérative.

Règle 5 **Optimisation des sélections**

La transformation d'un nœud contenant une sélection afin de l'appliquer sur ses nœuds fils est gérée par le prédicat suivant :

pushselectionrule($[sigma, BSigma, [[A, B, C]]], [AOut, BOut, COut]$).

Ce prédicat est appliqué de manière itérative.

Exemple 6.5 :

Détaillons la règle transformant $\Pi_A \mathcal{I}_{\mathcal{S}}(R)$ *en* $\mathcal{I}_{\mathcal{S}}(\Pi_{A \setminus \tau} R)$ *:*

```
pushprojectionrule(
    [pi, ArgPi, [
        [streamer,ArgStreamer,[C]]
    ]],
    [streamer, ArgStreamerFinal, [
        [pi, ArgPiFinal, [C]]
    ]]
):- map_get(ArgPi, "attributes", A),
    remove(A,"T",Attr), % Attr contient les nouveaux attributs
    % Les types et attributs des deux noeuds ont change...
    map_put(ArgStreamer, ['attributes', AttrA], ArgStreamerFinal),
    map_merge(ArgPi, {type: "relation", attributes: Attr}, ArgPiFinal).
```

Maintenant, pour la sélection, voyons comment nous pouvons appliquer la règle de l'algèbre relationnelle $\sigma_c(R_1 \cup R_2) = (\sigma_c R_1 \cup \sigma_c R_2)$.

```
pushselectionrule(
    [sigma, ArgSigma, [
        [union, ArgUnion, [C1,C2]]
    ]],
    [union, ArgUnion, [
        [sigma, ArgSigma, [C1]],
        [sigma, ArgSigma, [C2]]
    ]]
).
```

6.3.2 Optimisations annexes

Il peut devenir nécessaire d'introduire d'autres règles d'optimisations. Une des plus efficaces serait l'introduction de règles pour appliquer les propriétés de commutativité sur l'opérateur $\mathcal{D}_c^f$. En effet, cet opérateur est très souple puisqu'il peut commuter très facilement avec les opérateurs relationnels. Ainsi, le rapprocher au plus prêt des sources permet d'éviter de mettre à jour trop souvent les résultats intermédiaires.

De plus, du fait de l'extensibilité d'Astronef, il est possible d'introduire de nouveaux opérateurs. Si ceux-ci peuvent subir des réécritures pour optimiser structurellement la requête, l'utilisateur d'Astronef doit pouvoir soumettre de nouvelles règles.

Pour permettre l'écriture de telles règles complémentaires, nous avons prévu un autre prédicat.

Règle 6 **Optimisations logiques annexes**

La transformation d'un nœud $[AIn, BIn, CIn]$ en $[AOut, BOut, COut]$ pour l'optimisation est géré par le prédicat :

optimizationrule$([AIn, BIn, CIn], [AOut, BOut, COut])$.

Ce prédicat est appliqué de manière itérative.

6.4 Optimisation physique

Comme présentée dans la figure 6.3, l'optimisation physique travaille sur l'arbre produit par l'optimisation logique. Les composants opérateurs implémentent différentes sémantiques d'exécution. Cette sémantique peut être traduite avec Astral. Une fois ces liens entre théorie et implémentation établis, nous avons plusieurs plans d'exécutions envisageables. Dans cette section, nous présentons les règles permettant de sélectionner les composants les plus efficaces.

6.4.1 Macroblocs

Afin d'exécuter des sous-parties de la requête de façon plus optimale, nous utilisons le concept de macroblocs. Un macrobloc est une composition d'opérateurs pouvant être réunis en un seul bloc, considéré plus efficace. Par exemple, la combinaison agrégation-fenêtre $\mathcal{G}S[W]$ largement étudié dans la littérature est un macrobloc. Sachant que des composants sont capables d'exécuter cet opérateur composite, nous pouvons effectuer cette transformation. Ici, l'hypothèse 6.2 permet d'appliquer des macroblocs dès que possible.

Cette hypothèse est basée sur l'idée qu'un opérateur composite restreint ses capacités en terme de puissance d'expression. Ainsi, il devient plus spécialisé et efficace pour l'exécution de sa tâche plutôt que deux (ou plus) opérateurs génériques. De plus, moins il y a d'opérateurs, plus les travaux de planifications du *Scheduler* sont allégés.

Hypothèse 6.2 **Heuristique des macros-blocs**

> Plus le nombre de composants opérateurs utilisés pour exécuter une requête est petit, plus son exécution est efficace.

La mise en œuvre des macroblocs se fait via l'utilisation d'un prédicat spécifique pour réécrire un arbre ou un sous-arbre en un nouveau bloc. Il est important de voir que la nature du nœud peut se changer en un opérateur non standard de l'algèbre.

Règle 7 **Regroupement de macroblocs**

> La réécriture d'un groupe de nœud $[AIn, BIn, CIn]$ en un macrobloc $[AOut, BOut, COut]$ est géré par le prédicat :
>
> $$\textbf{macrobloc}([AIn, BIn, CIn], [AOut, BOut, COut]).$$
>
> L'application de ce prédicat est faite de manière itérative.

Exemple 6.6 :
*Reprenons la composition agrégation-fenêtre $\mathcal{GS}[W]$ dont il existe plusieurs implémentations. Nous allons combiner les deux opérateurs **aggregate** et **window** pour former un nouveau nœud **windowaggregate**. Celui-ci peut avoir plusieurs implémentations.*

```
1 macrobloc(
2     [aggregate, ArgAgg, [
3         [window, ArgWindow, C]
4     ]],
5     [windowaggregate, ArgWinAgg, C]
6 ):-
7     map_get(ArgWindow, "description", [D]), !,
8     map_put(ArgAgg, ["description", [D]], ArgWinAgg).
```

Si un bloc est créé, il doit exister un composant capable de l'implémenter tel qu'il a été formé. Ainsi, il est nécessaire de vérifier des conditions supplémentaires si les capacités des composants sont limitées. En reprenant notre exemple, si nous ne possédons qu'un composant capable d'exécuter le nœud **windowaggregate** avec une description temporelle. Alors, nous devons le vérifier au moment de la formation du bloc.

6.4.2 Encapsulation d'opérations de n-uplets

Les opérateurs de n-uplets tels que présentés dans la définition 6.4.1 sont des opérateurs dont l'évaluation peut se faire de manière indépendante, n-uplet par n-uplet. Plusieurs opérateurs de ce type peuvent regroupés dans un seul macrobloc.

Ces opérateurs ont la particularité de vérifier le corollaire 5.2 ce qui fait qu'ils sont applicables sur des flux ou des relations. De plus, ces opérateurs sont composables. La proposition 6.4.1 montre que l'évaluation de celle-ci est égale à la composition des traitements n-uplet par n-uplet. La démonstration de cette proposition se fait trivialement par récurrence en composant les fonctions λ.

Définition 6.4.1 **Opérateurs de n-uplets**

Soit TS une séquence de n-uplets,
Un opérateur de n-uplet Λ est défini par la fonction partielle de n-uplet vers n-uplet : λ.

$$\Lambda(TS) = \{\lambda(s)/s \in TS\}$$

Proposition 6.4.1 **Composition d'opérateurs de n-uplets**

La composition d'opérateurs de n-uplet est égale à l'opérateur de n-uplet défini par la composition de leurs fonctions partielles.

Ainsi, il est possible de regrouper tous les opérateurs de n-uplets en un seul composant efficace. La règle pour appliquer cette opération utilise le prédicat **macrobloc**. Afin de simplifier les règles de type, d'implémentation et de macrobloc, nous créons un prédicat pour déclarer les opérateurs de n-uplets.

Règle 8 **Déclaration d'opérateurs de n-uplets**

La déclaration que le nœud de nature A avec la configuration B peut se traiter n-uplet par n-uplet avec le composant $Impl$ se fait via le prédicat :

unaryimpl(A,B,$Impl$)

Une fois ces composants déclarés, une règle interne réécrira l'ensemble des opérateurs de n-uplets sous un nœud **unary** comprenant la propriété *operations* contenant la liste des opérations à appliquer.

Exemple 6.7 :

Soit la requête $\sigma_{a>30}\Pi_{id,a}R$. Son expression avant l'encapsulation est :

```
[sigma, {attributes: ["id","a"], type: "relation",
         condition: "a>30", conditionAttributes:"a"}, [
    [pi, {attributes: ["id","a"], type: "relation"}, [
        [source, {id:"R", attributes: ["id","a","b"], type:"relation"},[]]
    ]]
]]
```

En supposant qu'il existe à l'intérieur d'Astronef les déclarations suivantes.

```
unaryimpl(sigma,_, "SelectUnaryOperation").
unaryimpl(pi,_, "ProjectUnaryOperation").
```

Nous obtenons après encapsulation, le résultat suivant :

```
[unary, {
    operations: [
        {impl: "ProjectUnaryOperation", attributes: ["id","a"]},
        {impl: "SelectUnaryOperation", attributes: ["id","a"],
                condition: "a>30", conditionAttributes:"a"}],
    attributes: ["id","a"], type: "relation"}, [
```

```
7 |       [source, {id:"R", attributes: ["id","a","b"], type:"relation"},[]]
8 |]]
```

6.4.3 Sélection des composants

Nous avons maintenant un arbre dont la structure est jugée comme efficace. Nous en venons désormais à la dernière règle : la sélection du meilleur composant pour exécuter chaque nœud. Ce choix pourra être seulement basé sur la nature du nœud dans le cas où il n'existe pas d'autres alternatives. Mais dans la plupart des cas, il est nécessaire de sélectionner selon sa configuration.

Calcul incrémental

Dans la section 3.4.1, nous avons présenté le calcul incrémental et son intérêt dans l'optimisation du traitement des flux. Ce principe reste applicable naturellement dans Astral, il est toujours possible de définir à partir d'une relation temporelle R les deux Δ_R^+ et Δ_R^-. Ces deux objets indiquent les différences de R à chaque *batch* (resp. ajout et suppression).

Certains composants implémentant un opérateur particulier sont capables de traiter une relation en utilisant uniquement ces Δ_R. Et certains opérateurs fournissent sans coût supplémentaire ces Δ_R. Ainsi, il devient important de sélectionner les composants les plus capables d'exploiter ce mode. Il est important que garder à l'esprit que dans certains cas, les tailles des Δ_R sont similaires aux tailles des états de R (quand les R états de R n'ont pas d'éléments en commun). Dans ce cas, il n'est peut-être pas optimal d'utiliser le mode incrémental [4].

Le prédicat de sélection

La sélection du meilleur composant est importante, car les performances peuvent être différentes en fonction des implémentations. Par exemple, les opérateurs de séquences de fenêtres $[L]$ (dernier n-uplet) et $[B]$ (dernier batch) peuvent se définir avec une description de séquence de fenêtre générique. Toutefois, leur implémentation ne nécessite pas d'utiliser un opérateur générique, car leur calcul est très simple. Utiliser un composant qui calcule une description de fenêtre linéaire générique serait inutile et contre performant.

Les propriétés optionnelles permettent d'indiquer certaines informations au composant qui utilise ce nœud. Par exemple, il est possible d'indiquer que cette implémentation fournit une vue incrémentale des modifications.

Exemple 6.8 :

Nous souhaitons implémenter les règles pour choisir les meilleures fenêtres. Nous utilisons un prédicat capable d'identifier les fenêtres particulières $[L]$ et $[B]$. Si la description correspond à ces fenêtres, alors LastBatchWindow est sélectionnée, sinon WindowImpl est utilisé par défaut.

4. Mais ce choix est difficile à mettre en œuvre car il est souvent nécessaire d'évaluer les cardinalités, ce qui nécessite des modèles statistiques.

Règle 9 **Sélection d'implémentations**

La sélection du composant nommé *Impl* pour le nœud $[A, B, C]$ auxquels sont ajoutées les propriétés optionnelles *Props* est décidé par le prédicat :

implrules($[A, B, C]$,*Impl*,*Props*).

L'application de ce prédicat est faite de manière récursive.

```
1 implrules([window,{description: [D]},_], "LastBatchWindow",
2        {incremental: 1}):- % support de l'incremental
3    lastbatchorlasttuple(D), !.
4 implrules([window,_,_], "WindowImpl", {incremental: 1}):- !.
```

Exemple 6.9 :

Le streamer $\mathcal{I}_{\mathcal{S}}$ est capable de supporter les deux modes. Il est important de noter que le calcul incrémental pour les streamers $\mathcal{I}_{\mathcal{S}}$ et $\mathcal{D}_{\mathcal{S}}$ est très important, car leur coût devient quasi nul en utilisant directement les Δ_R. Ces deux modes sont gérés par deux implémentations différentes, ce qui se traduit par :

```
1 implrules([streamer,{stype: "Is"}, [[_,{incremental:1},_]]],
2    "DynamicIsImpl"):- !. % Si Is et incremental
3 implrules([streamer,{stype: "Is"},_], "IsImpl"):- !. %Par defaut
```

L'arbre fourni par l'application finale de la sélection des composants est ensuite donné à un service capable d'instancier les différents composants qui constituent le plan de requête. Il est intéressant de noter que ce service est lui-même écrit en *Prolog* par facilité d'implémentation.

Nous avons vu comment, par le renseignement de règles logiques, nous pouvons construire un plan de requête efficace. Ce plan est construit par sa restructuration logique et par la sélection des meilleurs composants pour mettre en œuvre cet arbre. Une évolution dans la même lignée que le calcul incrémental serait de raffiner les structures utilisées dans les résultats intermédiaires. Nous détaillons désormais les capacités de cette base de règle en terme d'extensibilité.

6.5 Intégration de nouveaux composants

Astronef est basé sur l'architecture de composants orientés services. Ainsi, nous pouvons apporter de nouveaux composants. Toutefois, l'intégration des composants opérateurs nécessite aussi l'apport de ses connaissances en terme de règles logiques. L'intergiciel expose un service *KnowledgeBase* capable d'ajouter des règles à sa base de connaissances (sous forme de fichier ou de chaînes de caractères). Ainsi, le constructeur de requête est lui aussi extensible.

Afin d'être compatible, le nouveau composant doit fournir la fabrique dans la même technologie que les opérateurs originels (en l'occurrence iPojo/OSGi). Il doit

naturellement aussi fournir les services nécessaires à son exploitation. Enfin, il doit spécifier les propriétés de configurations qu'il supporte, et évidemment respecter et correctement manipuler les services d'Astronef pour manipuler les structures de données ou le *Scheduler*.

La seule règle obligatoire pour exploiter un nouveau composant est de fournir au moins une règle **implrules** où le nom du composant (sa classe d'implémentation par défaut) est indiqué. Si ce composant implémente un macrobloc, alors il faut définir potentiellement un nouveau nom de nœud en plus des règles **macrobloc**.

Mais si ce composant implémente un nouvel opérateur que nous souhaitons utiliser dans l'expression de requête. Alors, il est strictement **nécessaire** de définir sa sémantique en terme de types supportés et d'attributs fournis. Sans ces deux règles, il est impossible de construire la requête. De plus, si nous possédons la connaissance suffisante, nous pouvons indiquer son comportement face à la projection, la sélection ou d'autres optimisations logiques.

6.6 Conclusion

Nous avons présenté dans ce chapitre l'intergiciel *Astronef*. Il est le moteur d'exécution de requêtes *Astral*. Son architecture basée sur le modèle de composants orientés services permet une grande flexibilité. Nous pouvons en effet ajouter, supprimer ou remplacer chaque composant de l'intergiciel par d'autres. De plus, nous sommes capables de décrire la sémantique des composants opérateurs grâce à Astral. Ainsi, nous pouvons aligner l'expression algébrique avec son implémentation. Toutefois, pour une expression, il existe de nombreuses possibilités pour son évaluation : différentes expressions algébriques sont possibles, différents composants peuvent être utilisés. Il devient intéressant de sélectionner le meilleur plan d'exécution.

Notre approche à base de règle a permis une mise en œuvre intuitive et efficace. Tout d'abord, nous réécrivons l'expression algébrique pour qu'elle soit plus performante. Ces réécritures sont basées sur les connaissances théoriques accumulées avec Astral que nous pouvons directement traduire en terme de règles. Puis nous sélectionnons les meilleurs composants pour exécuter cette nouvelle requête. L'évaluation des performances d'Astronef est abordée dans le chapitre 9.

De plus, l'intégration de nouveaux composants se fait rapidement par les règles et par l'architecture à composants orientés services. En effet, il nous suffit de spécifier la sémantique algébrique du composant selon plusieurs règles et le composant peut être exploité. Nous abordons cet aspect lors de l'extension de notre approche dans le chapitre 10 sur la personnalisation des résultats.

Dans le chapitre suivant, nous profitons de l'extensibilité de cet intergiciel pour mettre en œuvre le couplage avec un SGBD relationnel à l'intérieur d'Astronef.

*« I was elected because I can think outside the box. Which means... ***bunk*** I can also think inside a chimney! »*

Pinkie Pie

7

Asteroid : Intégration des supports relationnels persistants

Nous sommes désormais en possession d'un intergiciel capable d'exécuter une requête exprimée en Astral. Or, cet intergiciel a pour propriété d'être extensible et Astral est capable de supporter l'hétérogénéité des données en terme de mode d'interrogation. Ainsi, l'algèbre nous sert de fondement théorique pour coupler un SGBD relationnel et Astronef. Dans ce chapitre, nous allons présenter *Asteroid* [1] qui étend l'intergiciel Astronef pour mettre en œuvre ce couplage.

La section 7.1 présente les fondements théoriques qui servent au couplage et détaillent l'influence de l'évolution des données sur le schéma utilisé dans le SGBD couplé. Ensuite, nous détaillons en section 7.2 les composants Astronef qui nous permettent d'instancier ce couplage. Enfin, en section 7.3, nous présentons les règles de réécriture nécessaires pour restructurer le plan de requête suivant ces différents composants de couplage. Puis, nous concluons en section 7.4.

1. Astronef Extension for Relations In Databases

7.1 Intégration théorique

Dans cette section, nous présentons les fondements théoriques qui permettent l'interrogation de relations persistantes dans *Astral*. Tout d'abord, nous présentons en détail les différents motifs d'évolution des données persistantes ou temps réel. Ceci est important, car ces motifs ont des impacts sur le schéma de la base de données utilisé pour la persistance que nous présentons par la suite. Enfin, nous décrivons comment une relation persistante est représentée en tant que relation temporelle afin de pouvoir l'interroger avec *Astral*.

7.1.1 Dynamique des données

Nous nous intéressons à des systèmes où les données peuvent être persistantes ou sous forme de flux volatile. Nous remarquons aussi que les données évoluent suivant des motifs différents qui vont influencer la manière de les manipuler par la suite. Notamment, cela a un impact sur le schéma de la base de données relationnelle utilisée comme persistance.

Par définition, la persistance d'une donnée implique le stockage sur un support. Sa mise à jour sur ce support est une opération considérée comme lente [2]. Ainsi, il est difficile de supposer possible l'utilisation d'un SGBD seul pour gérer toutes les données du système. Il est nécessaire de séparer les intérêts de chacun des systèmes. Les données du système sont considérées en quatre dynamiques divisées en deux catégories : les meta-données et les volatiles.

Tout d'abord, celles que nous qualifions de meta-données, rassemblées dans des catalogues (typiquement, des relations persistantes). Celles-ci sont décomposées en deux classes de dynamiques :

Statique correspond à des méta-données qui ne changent jamais de par leur nature. Comme leur valeur est constante, leur utilisation en interrogation continue est similaire à une relation temporelle R figée : R^{t_0}. Par exemple, le numéro de série d'un équipement est une information qui par nature est immuable.

Stable correspond à des méta-données considérées la plupart du temps comme figée. Elles ne sont toutefois pas immuables et peuvent subir des modifications. Bien que leur utilisation soit avant tout une interrogation instantanée, en interrogation continue, elles sont manipulables par une relation temporelle R sans manipulation temporelle. Par exemple, un paramètre de configuration d'un équipement du réseau local est considéré comme stable.

La deuxième catégorie rassemble les données volatiles évoluant en temps réel. Elles prennent la forme de flux de données et peuvent posséder deux dynamiques :

Périodique rassemble les données dont l'historique forme un flux régulier. Leur interrogation continue passe par l'application d'une fenêtre dont le contenu n'est pas limité à un *batch*. En effet, la régularité induite par cette donnée implique qu'il est plus important d'observer son évolution que sa valeur présente. Le relevé des débits d'une carte réseau constitue une donnée périodique.

2. Cette lenteur a motivé la création des SGFD à la fin des années 90.

Imprévisible rassemble les données sans motifs d'évolution particuliers. Leur comportement fait que chaque nouvelle donnée du flux a son importance. Leur utilisation en requêtes continues est faite par l'application d'une fenêtre $[B]$ décrivant le dernier *batch*. La notification de l'arrivée d'un équipement sur le réseau est imprévisible.

Le principe important est que ces classes de dynamiques sont manipulables grâce à l'algèbre. Il est possible de figer une donnée à un instant grâce à la manipulation temporelle ou de former un flux de changement à partir d'une relation stable. De plus, elle traduit une certaine qualité de la donnée, car si nous utilisons une donnée d'une classe comme une autre alors nous perdons des informations quant à son évolution.

Exemple 7.1 :

Si nous récupérons les notifications d'arrivée des équipements sur le réseau de manière périodique, nous perdons de la qualité en terme de ponctualité. De même si nous considérons un paramètre de configuration comme statique. À l'inverse, nous introduisons du bruit si nous interrogeons de manière périodique la configuration du routeur de la passerelle d'accès à internet.

Il est important de voir que l'identification des classes de dynamiques nous permet d'imaginer les mécanismes les plus adaptés pour collecter les données. Toutefois, si un mécanisme n'est pas disponible et qu'un autre est utilisé[3], cela nous permet d'en analyser rapidement les conséquences. La figure 7.1 montre des transformations possibles entre les dynamiques grâce à l'algèbre Astral. Les données persistantes sont représentées par des relations temporelles R et les données volatiles sont des flux non partitionnables S.

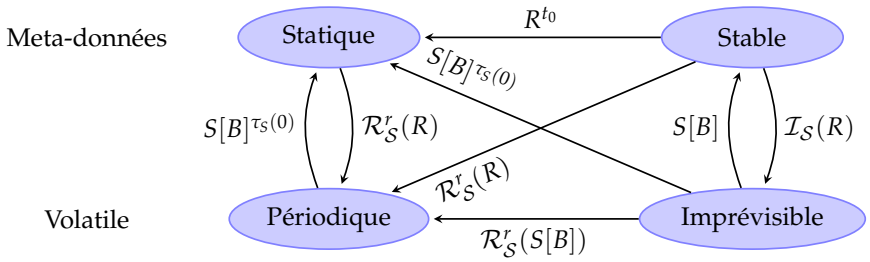


FIGURE 7.1 – Transformations des différentes dynamiques en Astral

7.1.2 Schéma physique de la persistance

La dynamique des données impacte directement la structure du schéma de la base de données. En l'occurrence, la persistance permet de stocker et conserver les deux catégories de données : les méta-données par le *modèle descriptif* et les données volatiles par les *historiques*. Dans la suite de cette section, nous détaillons ces deux parties de la base de données.

3. *push* absent ⇒ remplacement par un *pull* régulier

Le schéma descriptif

Les méta-données forment une description du système observé. Cette description sert de catalogue lors de son interrogation. Elle contient l'ensemble des concepts du système, leurs relations, ainsi que leurs propriétés. D'un point de vue conceptuel, celui-ci peut-être structuré comme un schéma entité relation qui peut par la suite être traduit en schéma physique normalisé.

Le point important est le choix d'une classe *Monitorable* pour représenter tous les concepts dits observables du système. Cette classe permet d'identifier de manière unique chaque objet du système (clé artificielle unique) et de les manipuler de façon uniforme. Les sous-classes de *monitorables* représentent des objets observables spécifiques.

Exemple 7.2 :

Dans le cadre du réseau local domestique, nous observons un système composé d'équipements. Ces équipements sont hôtes d'applications pouvant avoir un statut allumé ou éteint. Ils peuvent posséder un numéro de série unique. Ces équipements embarquent une ou plusieurs interfaces réseau. Celles-ci possèdent une adresse IP et MAC unique. Le réseau est composé de liens physiques entre les interfaces réseau.

Nous avons les relations Devices, Applications, Interfaces et Link toutes filles de Monitorable. Nous obtenons ainsi le schéma physique présenté en figure 7.2

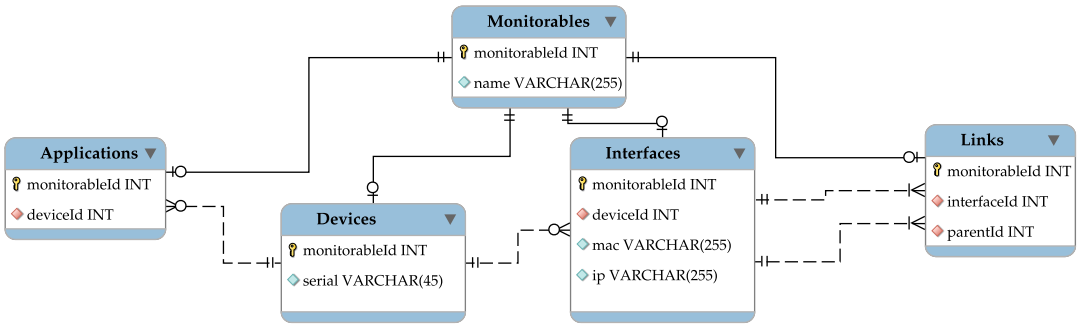


FIGURE 7.2 – Exemple de schéma descriptif du réseau local domestique

La différence entre une donnée stable et une donnée statique est qu'il est intéressant de surveiller la modification d'une donnée (par des mécanismes tels que les *triggers*) et en former un flux, qui peut être potentiellement archivé.

Remarque : Les clés artificielles ont une grande importance dans le cadre des applications d'observation. En effet, du fait du caractère incomplet de l'observation [4], il est possible d'obtenir des informations sur un équipement sans avoir pu obtenir son numéro de série. Des moyens alternatifs d'identification doivent être mis en place. Les clés artificielles permettent de tisser les relations entre les instances en ayant des données partielles. Toutefois, des vérifications d'intégrités sont nécessaires pour garder un modèle cohérent. Nous détaillons des cas applicatifs de ce problème dans nos expérimentations au chapitre 8.

4. soit parce que la donnée n'est pas parvenue au système, soit il n'existe pas de moyen technique pour y accéder

Les historiques

Supposons qu'une donnée *volatile* est produite dans le flux S. Son historique est matérialisé par la relation temporelle $S[\infty]$. Il est nécessaire de faire persister la liste des historiques archivés. Nous modélisons cela par la création de l'association *HasVolatile* donnant à partir d'un objet *monitorable* et d'un *volatile*, la relation contenant son historique.

D'un point de vue du schéma physique, les SGBD supportent rarement l'implémentation d'une relation dans un attribut. Pour contourner ce problème, nous créons une relation *Pointers* indiquant un nom de relation historique et un attribut contenant la donnée *volatile*. Ainsi, l'association *HasVolatile* fournit un identifiant de pointeur au couple (*monitorable*, *volatile*) comme montré dans la figure 7.3.

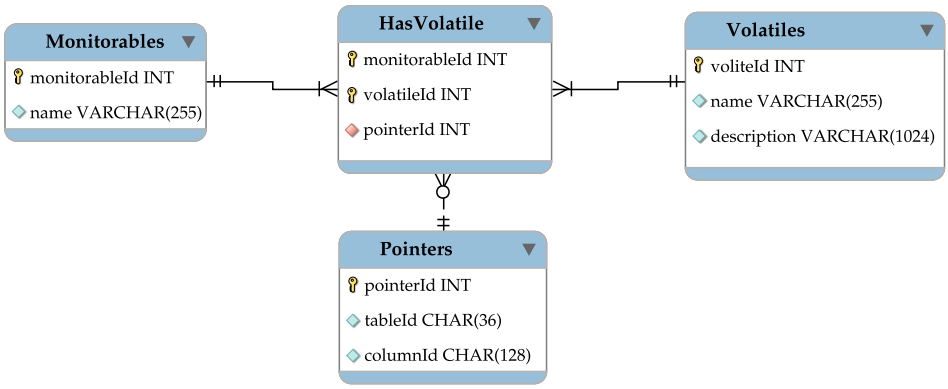


FIGURE 7.3 – Représentation physique de l'enregistrement des historiques

Exemple 7.3 :

Lors de l'enregistrement d'un historique de status concernant un équipement par exemple, il est nécessaire d'insérer un n-uplet dans la relation Pointers, qui permet de déclarer l'historique. Un autre est inséré dans HasVolatile pour chaque monitorable rencontré dans l'historique. Si la donnée status n'avait pas été déclarée, il est nécessaire de le faire en insérant un n-uplet dans la relation Volatiles.

De façon similaire, les relevés de charges processeurs cpu peuvent être attachés à un équipement ou à une application. Alors que les relevés de débits n'ont de sens que sur les interfaces (ou les liens suivant la modélisation voulue par l'utilisateur).

Il est intéressant de voir que puisque les données temps réel ont tendances à être de fortes densités, il n'est peut-être pas possible de garantir une fraicheur des données au niveau du support de persistance pour les historiques.

7.1.3 Représentation d'une relation persistante dans Astral

Les relations issues d'un SGBD relationnel sont manipulables dans Astral du fait de leurs fondations théoriques similaires. Ainsi, une relation qui change au cours du temps devient assimilable à une relation temporelle. Deux différences majeures subsistent : l'ordre des données et le mode d'interrogation.

L'ordre d'une relation

Lors de l'utilisation d'un SGBD, l'ordre n'a pas d'importance jusqu'au moment de l'envoi des données à un utilisateur qui utilise le mot clé *ORDER BY*. Le SGBD peut en profiter pour modifier l'ordre lors d'opérations de jointures par exemple. Dans notre cas, lors de la représentation en relation temporelle, il est important de garder la même notion d'ordre au long de la requête continue. L'identifiant physique de la relation temporelle induite est par défaut la clé primaire de la relation persistante. Ainsi, nous gardons le même identifiant, et l'ordre est conservé lors des opérations.

Le mode d'interrogation

Pour représenter une relation temporelle dans *Astronef*, il est nécessaire de surveiller les changements effectués dans la relation persistante. Ceci peut toutefois être coûteux ou impossible suivant les moyens techniques disponibles dans le SGBD manipulé. Ainsi, pour représenter la relation telle que nous l'obtenons dans Astronef par la suite, nous utilisons un opérateur de manipulation du temps. Ce qui nous permet d'obtenir, comme indiqué dans la section 4.3.3, une mise à jour effective, périodique, à la demande ou inexistante.

Nous avons vu comment le SGBD peut être utilisé pour aider à la gestion de données d'observation. Nous avons aussi présenté comment représenter les relations persistantes dans Astral. Nous détaillons maintenant l'extension d'Astronef pour pouvoir manipuler le SGBD grâce à Astral.

7.2 Extension d'Astronef pour l'intégration d'un SGBD

Dans cette section, nous présentons les composants développés pour permettre le couplage entre Astronef et un SGBD relationnel. Nous présentons trois types de composants :

- Une source capable de représenter une relation temporelle à partir d'une relation issue d'un SGBD relationnel.
- Un opérateur permettant de faire une opération de jointure Astronef par le SGBD pour exploiter ses capacités (macroopération de jointure).
- Les puits en charge de la persistance des données.

7.2.1 Le SGBD comme source d'interrogation

Comme présentée dans la section 7.1.3, une relation persistante peut être présentée comme une relation temporelle Astral. Pour cela nous introduisons un composant source de données *dbsource* capable de représenter une telle relation temporelle. Cette source doit suivre l'évolution de la relation suivant différents modes. Son implémentation est décrite par la séquence suivante : lorsque la relation temporelle doit être mise à jour, le composant interroge le SGBD avec une requête *SQL* telle que `SELECT * FROM Relation`.

Afin de pouvoir réutiliser *dbsource*, nous pouvons le rendre configurable en permettant la spécification de paramètres. Ainsi, la source est capable de supporter la

représentation Astral de toute requête du type $\mathcal{D}^f Q$ où Q désigne une relation temporelle exprimable en algèbre relationnelle sur toute relation persistante. Notons que $\mathcal{D}^f$ permet de refléter une dynamique de mise à jour comme présentée dans la section précédente. De ce fait, nous pouvons produire une relation temporelle avec une dynamique statique, stable, périodique ou imprévisible (mise à jour suite à un événement extérieur).

La liste des paramètres de ce composant est disponible dans la table 7.1 et la sémantique des modes de mise à jour est disponible dans la table 7.2.

Paramètre	Description
query	Requête *SQL* à exécuter sur le SGBD
mode	Mode de mise à jour

TABLE 7.1 – Paramètres obligatoires du composant *dbsource*

Mode	Sémantique	Dynamique	Paramètre supplémentaire
oneshot	$\mathcal{D}^{(t_s,0)}_{t \geq t_s}$	statique	*at* : *timestamp* correspondant à t_s
trigger	$\mathcal{D}^{\text{change}_{R_1,\ldots,R_n}}$	stable	*tables* : liste des relations (R_i) à surveiller
periodic	$\mathcal{D}^{\text{period}^r}$	périodique	*rate* : période en seconde (r)
notify	$\mathcal{D}^{\text{change}_E}$	imprévisible	*dependentRId* : numéro du service de E

TABLE 7.2 – Modes de mise à jour supportés par le composants *dbsource*

Il est important de voir que l'ensemble de la relation est représenté dans la relation temporelle fournie. Or, dans Astronef, toute relation temporelle est placée en mémoire. Ainsi, si la séquence produite par l'interrogation du SGBD est de plusieurs millions de n-uplets, la taille des données est à prendre en compte.

Comme cette source est capable de représenter toute requête *SQL*, il est possible de déporter des opérations dans cette requête pour exploiter les capacités du SGBD. Nous détaillons dans la section 7.3 cette réécriture de façon automatique grâce à l'optimisation par règle d'Astronef.

7.2.2 Macroopération de jointure

Le second composant d'Asteroid est un opérateur de jointure : *dbjoin*. Son rôle est le suivant : lors de la réception d'un nouvel état d'une relation temporelle, interroger le SGBD et effectuer une jointure entre l'état courant de la relation et une requête *SQL*. Cette opération est souvent utilisée lors qu'il est nécessaire d'ajouter à la relation courante des informations issues du SGBD (données du catalogue, agrégats de l'historique).

L'idée principale de cet opérateur est d'exploiter au maximum le SGBD. L'utilisation d'index précalculés sur le disque permet d'effectuer la jointure de façon efficace, car l'optimiseur sélectionne un plan performant grâce à ceux-ci.

D'un point de vue Astral, l'opérateur *dbjoin* permet d'effectuer l'opération suivante $R \bowtie_c Q$, où R est la relation d'entrée et Q est une relation temporelle exprimable en

algèbre relationnelle sur les relations de la base de données, et c une condition de jointure. Les paramètres de configuration de *dbjoin* sont présentés dans la table 7.3.

Paramètre	Description
query	Requête *SQL* à joindre sur le SGBD
on	Expression de la condition de jointure (optionnel)

TABLE 7.3 – Paramètres du composant *dbjoin*

L'implémentation de *dbjoin* peut différer selon les capacités du SGBD. De façon générale, il est nécessaire d'utiliser une table temporaire `Tmp` (en mémoire). Au moment de l'exécution de *dbjoin*, l'opérateur insère les données de la séquence fournie dans `Tmp`. Ensuite, il applique la requête *SQL* suivante :

```
SELECT * FROM Tmp AS l NATURAL JOIN (*query*) AS r ON (*on*)
```

Le résultat est transformé en séquence de n-uplet qui est envoyée à la relation temporelle de sortie. Les données temporaires sont supprimées ensuite. Par mesure de performances, les requêtes sont préparées à l'avance.

Certains SGBD implémentent des optimisations particulièrement adaptées à ce genre d'exécution. Sur *H2* par exemple, il est possible d'utiliser la syntaxe `TABLE(A1 T1=?,...,An Tn=?)` pour représenter une table temporaire d'attributs A_i et de types T_i dont les données sont passées comme paramètres lors de l'exécution de la requête. Il est aussi possible d'accélérer l'opération de suppression via l'opération `TRUNCATE` maintenant répandue parmi les SGBD populaires.

7.2.3 Puits de persistance

Enfin, nous présentons l'ensemble des composants puits de couplage. Ces composants ont pour but d'effectuer des opérations d'écritures dans la base de données. Deux catégories de composants sont décrites : les archives de flux ainsi que l'entretien du catalogue décrit par le schéma descriptif.

Archives de flux

La première utilisation de la persistance pour la gestion de flux de données reste avant tout l'historisation. Ces données sont importantes pour effectuer des analyses a posteriori.

Le composant le plus direct dans cette application est *streampersistence*. Celui-ci a pour but de faire persister un flux donné en entrée. Son fonctionnement est simple, pour chaque n-uplet, effectuer une action `INSERT` dans le SGBD sur la table créée au préalable. Son seul paramètre de configuration est le nom de la table qui est utilisée comme archive : *table*.

Cependant, nous avons présenté en section 7.1.2 que les données temps réel sont considérés comme des *volatiles* dans le schéma physique. Ainsi, lors de leur mise en archive, il est nécessaire de créer leurs données d'enregistrement. Le composant *volatilepersistence* a pour but de faire persister un flux donné tout en faisant cette mise à jour. Ses paramètres de configuration sont listés dans la table 7.4.

paramètre	description
table	Nom de la table cible
volatiles	Liste des noms des *volatiles* à enregistrer
attributes	Liste des attributs du flux correspondant à *volatiles*
monitorableId	Attribut du flux correspondant à un *monitorableId*

TABLE 7.4 – Paramètres du composant *volatilepersistence*

Il est important de noter le paramètre *monitorableId*. En effet, pour enregistrer comme persistante une donnée *volatile*, celle-ci doit être une propriété attachée à un des objets du système grâce à son identifiant interne *monitorableId*[5]. Cette opération peut se faire via une jointure avec le schéma descriptif.

Entretien du catalogue

L'entretien du schéma descriptif en fonction des observations que nous pouvons faire grâce aux flux remontés est une opération complexe. En effet, la modélisation que l'utilisateur fait de son système est une vision qui ne correspond souvent pas à l'ensemble des données fournies. L'exemple ci-dessous indique un des cas observés dans la pratique indiquant la difficulté de ce processus de mise à jour.

Exemple 7.4 :
La modélisation du réseau local domestique illustré sur la figure 7.2 montre les relations Devices et Interfaces. Cependant, l'utilisateur ne possède qu'un flux Network(applicationName,ipAddress) lui indiquant l'arrivée de l'application applicationName sur le réseau avec l'IP ip. Il est nécessaire pour chaque n-uplet de créer l'application (en supposant que le nom est un critère d'identification) et l'Interface (associée à l'adresse IP) si besoin. Or, la liaison entre Applications et Interfaces se fait par le concept de Devices. Pour satisfaire les contraintes, il devient nécessaire de créer ou mettre à jour un Device associé.

Le problème de mise à jour de relations à partir de données extraites est toutefois connu dans le domaine de la recherche fondamentale en base de données. En effet, cela est très similaire à la traduction de mises à jour sur une vue [Keller 1985]. Ce problème correspond à la traduction d'une mise à jour d'une vue matérialisée par un ensemble d'opérations. Cependant, à l'écriture du manuscrit, aucun composant générique n'a été développé comme extension d'Astronef pour permettre cette opération, chaque cas de mise à jour est résolu de façon ad hoc.

7.3 Réécriture du plan d'exécution

Notre objectif est de pouvoir interroger tout flux ou relation de façon unifiée grâce au langage Astral. Nous avons décrit dans la section précédente les composants *dbsource* et *dbjoin* permettant d'intégrer les relations persistantes dans Astronef. Nous

5. Ces identifiants sont générés à la création d'un objet *monitorable* par une séquence.

devons indiquer à Astronef la manière de réécrire une requête Astral pour sélectionner un plan de requête efficace pouvant mêler relations persistantes et requêtes continues. Ainsi, l'objectif est de distribuer l'exécution de la requête entre Astronef et le SGBD.

7.3.1 Approche de la distribution de l'exécution

Notre heuristique de distribution que nous choisissons est que pour une requête relationnelle le SGBD sait calculer de façon plus efficace une requête.

Hypothèse 7.1 **Heuristique de l'utilisation prioritaire du SGBD**

> Plus les opérations sont déléguées au SGBD plus le plan de requête est efficace dans le cas général.

Ce choix est argumenté par l'efficacité du SGBD grâce au précalcul des index et autres algorithmes optimisés pour ce support. Ce qui fait que les opérations de sélection ou de jointures sont très efficaces par rapport à un calcul à la volée. Cette heuristique s'inspire de l'hypothèse 6.1 permettant de pousser les projections et sélections au plus proche des sources.

Afin d'appliquer cette transformation, nous adoptons une approche récursive. Nous possédons le composant *dbsource* capable de représenter toute expression Q relationnelle (exprimée en *SQL*) en relation temporelle. Ainsi, si nous avons la requête $S[B] \bowtie \sigma_{\dots} R$ avec S un flux et R une relation persistante. Nous réécrivons premièrement R en composant *dbsource*. Puis, nous transformons la configuration de *dbsource* pour que son expression *SQL* intègre l'opérateur $\sigma_{\dots}$ et ainsi de suite jusqu'à ne plus pouvoir réécrire.

Initialisation de la distribution

Pour initialiser la réécriture avec les composants Asteroid, nous utilisons un sucre syntaxique permettant de traduire l'appel d'une relation *Name* en composant *dbsource* avec ses paramètres de configurations reflétant une mise à jour par *trigger*. Après l'application de cette règle, nous avons un arbre dont toutes les interactions avec le SGBD se font via ces nœuds *dbsource*.

```
1  sugar(
2      [entity,{name: Name},[]],
3      [dbsource, {
4          'attributes': Attributes,
5          'query': Query,
6          'mode': 'trigger',
7          'tables': [Name],
8      },[]]
9  ):-
10     dbattributes(Name,Attributes), % Recupere les attributs de cette relation
11     concat(["SELECT * FROM ", Name], Query). % Forme la requete SQL
```

7.3.2 Règles de macroblocs pour le placement d'opérateurs

Nous souhaitons transformer un bloc `[A,B,[[dbsource,Config,[]],...]]` avec `[A,B,_]` opérateur supporté par *dbsource* en un seul `[dbsource,NewConfig,[]]`. Ceci correspond à la spécification de macroblocs. Il existe deux types de transformations : celles modifiant la requête, celles modifiant le mode de mise à jour.

Réécriture de la requête

Voyons tout d'abord la modification de requête dans *dbsource*. Nous définissons un prédicat capable de transformer les requêtes des sources en une nouvelle. Naturellement, la règle **macrobloc** doit transmettre les attributs et conserver les anciens paramètres de configuration.

Règle 10 **Modification de la requête de dbsource**

Soit $[A, B, C]$ un nœud dont tous les fils sont de type *dbsource* avec pour requêtes *Queries*,
La transformation du macrobloc $[A, B, C]$ en $[dbsource, Config, []]$ est assuré par le prédicat suivant :

dboperator($[A, B, C]$, *Queries*, *NewQuery*).

Avec l'attribut *query* de *Config* égal à *NewQuery*.

Exemple 7.5 :

Nous souhaitons pousser une sélection sur le SGBD. Cette opération est faite par l'application de la clause `WHERE` *sur la requête SQL actuelle. Par mesure de simplification, nous pouvons considérer la requête d'entrée comme une sous-requête à placer dans FROM. Nous obtenons la règle de transformation suivante :*

```
1 dboperator([sigma,B,_], [Query], NewQuery):- !,
2     map_get(B, 'condition', Cond),
3     conditionsql(Cond, Sql), % Traduit la condition en SQL
4     concat(['SELECT * FROM (', Query, ') v WHERE ', Sql], NewQuery).
```

Si le SGBD possède un optimiseur suffisamment puissant, il est capable de traiter efficacement cette nouvelle requête.

Ainsi, il suffit de spécifier l'ensemble des règles permettant de traduire la sémantique Astral en *SQL*. En l'état, nous avons implémenté les règles pour les opérateurs Π, σ, ρ, ${}_aG_b$, $\bowtie$ et $\cup$. Notons qu'il n'est pas évident de transformer les opérateurs binaires. Notamment, il est difficile de transformer la requête suivante $(\mathcal{D}^{f_1} R_1) \bowtie (\mathcal{D}^{f_2} R_2)$ (avec $f_1 \neq f_2$) en $\mathcal{D}^f R$. Nous nous limitons au cas où les modes appliqués aux composants *dbsource* sont égaux (typiquement *trigger* ou *oneshot*).

Changement du mode de mise à jour

Nous nous intéressons maintenant à l'application de l'opération $\mathcal{D}_c^f$ au composant *dbsource* ce qui correspond à un changement de mode. L'application du prédicat **dbsourcemode** permet de traduire la sémantique de l'opérateur $\mathcal{D}_c^f$ par le mode de *dbsource*.

Règle 11 **Modification du mode de dbsource**

Soit $[timetransform, B, [dbsource, Config, []]]$ un nœud de manipulation temporelle dont la transformation temporelle est de type T et dont les paramètres est $Parameters$
La transformation du macro-bloc $[timetransform, B, [dbsource, Config, []]]$ en $[dbsource, NewConfig, []]$ est assuré par le prédicat suivant :

dbsourcemode$(T, Parameters, Config, NewConfig)$.

Exemple 7.6 :
Nous souhaitons effectuer une requête instantanée sur le SGBD et d'en produire la relation temporelle associée. Nous écrivons la règle suivante :

```
1 dbsourcemode("freeze", {'at': Time}, Config, NewConfig):- !,
2     map_merge(Config, {
3         'at': Time,
4         'mode': "oneshot"
5     }, NewConfig).
```

En écrivant une règle pour chaque sémantique d'exécution spécifiée par la table 7.2, nous pouvons au mieux réécrire le plan pour que les composants **dbsource** se mettent à jour lorsqu'il est nécessaire.

Une application de cet ensemble de règles est la capacité à traduire une requête Astral relationnelle pure en *SQL*. Notamment, si toutes les sources issues du SGBD sont utilisés sous forme de requêtes instantanées (ou si l'opérateur $\mathcal{D}^{(t_0,0)}$ est appliqué en tête de la requête) : nous sommes capables d'exécuter une requête instantanée Astral par le SGBD. Ainsi, même pour faire une analyse a posteriori avec des requêtes instantanées, il reste possible d'écrire ses requêtes uniquement en Astral. Malheureusement, le surcoût induit par la réécriture fait que du point de vue des performances, il n'est pas intéressant de faire ainsi.

7.3.3 Cas de la jointure hybride SGFD-SGBD

Grâce à la flexibilité de *dbsource*, nous avons pu réécrire le plan de requête pour placer au plus les opérateurs relationnels sur le SGBD. Toutefois, nous sommes aussi en possession d'un composant opérateur que nous pouvons exploiter : *dbjoin*. Nous avons défini la sémantique de l'opérateur comme capable de supporter toute requête $R_1 \bowtie R_2$ avec R_1 relation temporelle quelconque et R_2 exprimée en algèbre relationnelle sur les relations de la base de données. Ce composant est un macrobloc que nous

pouvons former grâce à une règle. Afin d'illustrer l'application successive des règles, voici l'ensemble des résultats intermédiaires appliqués lors du raisonnement :

$$
\begin{aligned}
R_1 \underset{c}{\bowtie} R_2 &= \sigma_c \left(R_1 \bowtie \mathcal{D}^{\text{change}}_{R_1} R_2 \right) && \text{(via } \textbf{sugar}\text{)} \\
&= R_1 \bowtie_c \mathcal{D}^{\text{change}}_{R_1} R_2 && \text{(via } \textbf{pushselection}\text{)} \\
&= R_1 \bowtie_c \mathcal{D}^{\text{change}}_{R_1} \textbf{dbsource}^{Q}_{trigger} && \text{(via } \textbf{dboperator}\text{)} \\
&= R_1 \bowtie_c \textbf{dbsource}^{Q}_{notify(R_1)} && \text{(via } \textbf{dbsourcemode}\text{)}
\end{aligned}
$$

Enfin, l'application d'une dernière règle va nous permettre de transformer cette jointure en : $\textbf{dbjoin}^{Q}_{c}(R_1)$. Cette règle est exprimée grâce au prédicat **macrobloc** :

```
macrobloc(
    [join,JoinConfig,[
        [A,LeftConfig,C],
        [dbsource,DBConfig,[]]
    ]],
    [dbjoin,NewConfig,[[A,LeftConfig,C]]]
):- % Si...
    map_get(DBConfig, "mode", "notify"), % Le mode de dbsource est notify
    map_get(DBConfig, "dependentRId", RId), % Et notify pointe sur...
    map_get(LeftConfig, "rid", RId), % ... l'entite de gauche,
    !, % alors...
    map_get(DBConfig, "query", Query), % Recuperation de la requete
    map_get(JoinConfig, "attributes", Attr), % Recupere les attributs
    map_get(JoinConfig, "condition", "1==1", Cond), % Condition (1=1 par defaut)
    conditionsql(Cond, CondSQL), % Transformation en SQL
    map_merge(JoinConfig, { % Ajout des nouveaux parametres de configuration
        'query': Query,
        'condition': CondSQL,
        'attributes': Attr,
    }, NewConfig).
```

Toutefois, l'utilisation de cette règle ne semble pas optimale dans tous les cas. Pour effectuer l'opération de jointure (semi-sensible) entre une relation temporelle et un SGBD, il existe deux plans de requête :

P1 Une jointure relationnelle usuelle (par exemple, jointure hachée) est appliquée entre la relation temporelle et un nœud *dbsource* configuré en mode *notify*.

P2 Un *dbjoin* configuré avec la bonne requête *SQL* est appliqué sur la relation temporelle.

En effet, si les relations de la base de données ne sont pas mises à jours, alors il devient raisonnable de dire que la création d'une vue matérialisée en mémoire du résultat intermédiaire peut améliorer les performances avec le plan *P1*. Cette vue devient un cache local. L'opérateur de jointure peut créer des accès plus optimisés comme des tables de hachages pour minimiser son coût. Toutefois, le coût de construction de cette vue est non négligeable, car *dbjoin* exécute une requête sur le SGBD ce qui peut prendre beaucoup de temps.

A contrario, si nous appliquons un *dbjoin* avec le plan *P2*, la sélection des n-uplets fait que la taille du résultat est minimisée comparé au plan *P1*. Les résultats intermédiaires peuvent être optimisés par le SGBD. Toutefois, cette requête est exécutée

pour chaque n-uplet ce qui peut avoir un coût important. Même avec des stratégies de cache évoluées du côté du SGBD, il est possible d'avoir un état de la relation entrante différent pour chaque requête ce qui le rend difficile d'exploiter des caches.

Notre analyse empirique nous indique que la fréquence des mises à jour est déterminante pour privilégier *P1* (basse fréquence) ou *P2* (haute fréquence). Ce point est validé et affiné par les expérimentations que nous menons dans la section 9.3.

7.4 Conclusion

Dans ce chapitre, nous avons présenté l'intégration d'un support persistant relationnel. Dans le cadre de l'observation de système, nous exploitons le support persistant pour la gestion du catalogue du système ainsi que les historiques de flux. L'analyse des quatre dynamiques de données a mis en avant une méthode pour concevoir le schéma physique de la base de données. Astral permet d'unifier les deux mondes pourtant régis par des dynamiques, des modes d'interrogation et des concepts différents. Ainsi, cette solution gère l'hétérogénéité en terme d'évolution des données que nous nous sommes fixés lors de cette thèse.

La mise en œuvre de l'intégration du support persistant s'appuie fortement sur les connaissances dont nous disposons avec Astral en terme de modélisation des sources de données. Nous avons conçu plusieurs composants capables de refléter différentes sémantiques pour les modes de collecte de données, comme en terme de persistance de celles-ci. La flexibilité du moteur de règles développé dans Astronef a permis de mettre en œuvre des optimisations non triviales du plan de requête.

Toutefois, notre approche est améliorable. En effet, dans le cadre de la persistance des données du catalogue, des composants spécifiques doivent être mis en place. L'absence de gestion déclarative pour ce point rend cette tâche délicate.

Nous nous sommes basés sur des heuristiques et sur l'application itérative de règles pour résoudre l'optimisation du plan de requête. Une généralisation de l'approche pourrait optimiser des plans plus complexes, notamment dus à l'ordre des jointures. Par exemple, si nous souhaitons optimiser $(R_1 \bowtie R_2) \bowtie R_3$ avec R_2 et R_3 issus d'un même SGBD. Même en sachant qu'en Astral la jointure soit associative, il est difficile de spécifier une règle permettant $R_1 \bowtie (R_2 \bowtie R_3)$ avec $R_2 \bowtie R_3$ poussé au niveau SGBD. Le domaine des SGBD a rencontré le même problème ce qui a permis l'introduction de la programmation dynamique dans l'optimisation de requêtes. Nous pourrions améliorer nos performances en utilisant un tel procédé.

Nous avons désormais un système de gestion de données persistantes et temps réel. Nous validons notre approche en déployant ce système pour l'observation du réseau domestique.

Table des règles Astronef

143

Partie IV

Expérimentations

Nous avons désormais un langage capable d'interroger tout type de données et nous avons aussi un intergiciel capable d'exécuter ces requêtes. Nous allons maintenant mettre en pratique ces idées. Tout d'abord, nous consacrons un chapitre à l'observation du réseau domestique. Puis, afin d'évaluer les performances de notre intergiciel, nous présentons une évaluation des performances dans un second chapitre.

« What is this place
Filled with so many wonders?
Casting its spell
That I am now under »

Fluttershy, So Many Wonders

8

DomVision : Application au réseau local domestique

Nous avons présenté dans le chapitre d'introduction le domaine d'application privilégié sur lequel cette thèse s'appuie. Ce chapitre permet de mettre en pratique notre solution d'observation. DomVision est l'instanciation d'Astronef-Asteroid sur le réseau local domestique. Ainsi, nous mettons en œuvre un système d'observation par l'écriture de quelques composants et de plusieurs requêtes.

Dans ce chapitre, nous présentons les différents points de mise en application qui nous permettent de valider notre contribution en tant que système d'observation. En premier lieu, dans la section 8.1, nous présentons l'adaptation au système observé. Ensuite, en section 8.2, nous détaillons les requêtes déployées qui tirent profit de l'expressivité d'Astral.

8.1 Adaptation au système observé

Dans cette section, nous analysons le système observé au sens où nous l'avons décrit jusqu'ici dans Astronef et Asteroid. Nous devons modéliser le système observé pour en extraire son schéma qui permet de représenter le catalogue dans la base d'Asteroid. Puis nous nous intéressons aux données temps-réel. Enfin, nous présentons les flux à notre disposition dans ce réseau.

8.1.1 Le schéma du système

Nous présentons ici la vision que nous avons du système que nous observons. Il est important de voir que cette vision ne doit pas être liée aux sources de données disponibles. Elle a pour but de refléter l'interprétation des experts souhaitant observer le système.

Pour chaque concept, il est nécessaire de pouvoir identifier les objets du système pour pouvoir leur associer nos observations. Ainsi, nous détaillons les attributs permettant de désigner un objet de façon unique. Toutefois, dans la pratique, les données collectées sont partielles et ces attributs peuvent être absents. Ces inconsistances peuvent être minimisées par l'utilisation d'heuristiques. Les conséquences de telles inconsistances seraient la création de plusieurs entrées dans le catalogue qui représentent le même objet du système.

Chaque concept est caractérisé par un ensemble de propriétés *statiques*, relatives à sa description, et *stables*, relatives à son état. Les données volatiles sont présentées dans la section suivante. Le schéma est très similaire à celui présenté précédemment dans la figure 7.2. Nous définissons ici trois concepts principaux : les équipements, les applications et les interfaces réseau.

Équipements

La notion d'équipement est une vision orientée utilisateur. En effet, un *équipement* est un châssis physique. Les équipements virtuels ne sont pas représentés. Ce choix permet de représenter à l'utilisateur les appareils qu'il est capable de voir et de manipuler (débrancher par exemple). Quelques exemples : *Livebox*, *STB* [1], Ordinateurs, Tablettes, Disques durs réseaux.

Identification : un équipement *peut* avoir un *numéro de série* unique. L'utilisation des interfaces ou applications liées sont un moyen alternatif.

Applications

Nous désignons par *application* un programme instancié sur un des équipements. À un *équipement* est associé plusieurs *applications*. Les *applications* permettent d'effectuer des tâches sur leurs équipements respectifs. Une *application* en exécution a un état. Dans cette expérimentation, nous ne considérons que *actif* et *inactif*. Plusieurs autres statuts pourraient être utilisés comme *suspendu* ou *gelé*. Notons que la détermination des statuts est généralement complexe, car le système fournit rarement des données précises à ce sujet puisque cela consiste en un autodiagnostic [2]. Quelques exemples d'applications : Partage de contenu, Agent d'administration, Gestion d'impression. Une *application* a un *nom*, une *description* textuelle et potentiellement un *type*.

Identification : une application *peut* fournir un *nom* unique par l'utilisation d'un *UUID*, par exemple. Alternativement, nous pouvons exploiter l'unicité de son *type*. Par exemple, il n'existe qu'une application de passerelle internet sur le réseau domestique.

1. Set Top Box : Boitier connecté à une télévision pour fournir les services de partage de contenus, de télévisions, VOD...
2. L'utilisation d'inférences de haut niveau comme vu en section 2.2 peut devenir intéressante dans ces cas.

Interfaces réseaux

Les *interfaces réseaux* sont moins sujets à interprétation que les deux concepts précédents. Une *interface réseau* représente un point de connexion que possède un *équipement* vers le réseau domestique. Nous supposons que cette interface est une interface *MAC*[3]. Elle possède un *nom* système unique pour l'équipement, un type (wifi, Ethernet), ainsi qu'une adresse *MAC* considéré unique sur le réseau et d'autres adresses comme *IPv4* ou *v6*.

Identification : seule l'adresse *MAC* garantit l'unicité. Si elle n'est pas renseignée, l'adresse *IP* peut être utilisé comme moyen alternatif.

Concepts supplémentaires possibles

Ayant une modélisation des concepts principaux du réseau local domestique, nous pouvons maintenant ajouter des concepts supplémentaires. Bien que nous ne les utilisons pas dans la suite de ce chapitre, il est intéressant de les garder à l'esprit.

Par exemple, pour rendre compte de la topologie du réseau, les *liens* relient deux interfaces réseaux. Les *chemins* sont composés de *liens* réseaux. Et les *canaux de transport* peuvent utiliser un *chemin* pour aussi relier deux *applications*. Toutefois, la collecte de ces informations est soumise à des contraintes de mise en œuvre plus technique comme l'utilisation de protocoles *LLTD* ou *IEEE P1905.1*.

8.1.2 Les données volatiles intéressantes

Comme défini au chapitre 7, les données *volatiles* sont des données temps réel que nous souhaitons archiver en vue d'une analyse a posteriori. La surveillance de l'état d'un réseau domestique passe par l'observation de métriques de charge. Ces dernières sont par exemple visualisées sur des graphiques afin de détecter des comportements anormaux.

Pour les équipements, nous surveillons la charge processeur utilisé (en %) et mémoire occupée (en ko). Pour les interfaces réseau, le débit est un bon indicateur de son état. Au besoin, d'autres métriques plus spécifiques peuvent être enregistrées comme : la latence ou la gigue d'une interface réseau.

Nous souhaitons observer des changements d'état. L'état est caractérisé par un modèle qui contient des données de différentes dynamiques telles que des données de description (*statique*), de configuration (*stable*) et de fonctionnement (périodique ou imprévisible). Par exemple, nous pouvons surveiller les changements de topologie du réseau. Pour cela, nous observons les changements d'adresse *IP* des interfaces, ou de *statut* des équipements, applications et interfaces.

Désormais, nous avons formalisé notre représentation du réseau domestique. Analysons maintenant les données produites par le réseau domestique et pour pouvoir les intégrer à cette représentation.

3. Permettant des interfaces *IP* et *Zigbee* potentiellement

8.1.3 Les données disponibles

Dans notre mise en œuvre, nous utilisons le protocole *UPnP* largement répandu dans le réseau domestique. Ce protocole permet d'annoncer un *Device*[4]. Ce *Device* exposant des *Services* qui contiennent des *Actions*, il est possible de les consulter à distance. Les *Devices* répondent à des profils standards.

De fait, l'annonce des *Device UPnP* sur le réseau produit un flux de données

$$\mathit{UPnPStatus}(\text{uuid, ip, type, friendlyName, status, } \tau).$$

Ce flux indique que le *device UPnP* annoncé avec l'IP *ip*, ayant pour identifiant unique *uuid*, pour profil *type* et pour description textuelle *friendlyName* a changé son statut au temps τ pour *status*.

Nous avions exploré en section 2.1 que les protocoles et agents d'administrations pouvaient être de bons interlocuteurs pour fournir des données. Dans le monde *UPnP*, il existe le profil *DeviceManagement* capable de fournir des données intéressantes. Le tableau 8.1 indique les données que nous pouvons exploiter. Les flux produits ont, en plus des paramètres, les attributs *uuid* correspondants à l'identifiant *UPnP* de l'agent, ainsi que le *timestamp* d'émission τ.

Nom du flux	Paramètre	Description du paramètre
/UPnP/DM/DeviceInfo/PhysicalDevice/		
Serial	`SerialNumber`	Numéro de série de l'équipement
NetworkInterface	`SystemName`	Nom de l'interface réseau
	`MACAddress`	Adresse MAC de l'interface
	`InterfaceType`	Type de l'interface
/UPnP/DM/Configuration/		
IPInterface	`SystemName`	Nom de l'interface réseau
	`IPv4Address`	Adresse *IPv4* de l'interface
/UPnP/DM/Monitoring/		
OperatingSystem	`CPUUsage`	Charge actuelle du processeur
	`MemoryUsage`	Charge actuelle de la mémoire
IPUsage	`SystemName`	Nom de l'interface réseau
	`TotalBytesSent`	Nombre d'octets envoyés
	`TotalBytesReceived`	Nombre d'octets reçus

TABLE 8.1 – Listes des flux intéressant fournis par le profil UPnP-DM

8.1.4 Les composants sources Astronef

Nous avons conçu un composant spécifique pour la formation du flux *UPnPStatus*. Ce composant n'a pas de paramètre et fournit le service *Source*, comme présenté dans le chapitre 6, avec comme attributs ceux mentionnés dans la section 8.1.3. Son fonctionnement est événementiel.

4. Attention au vocabulaire utilisé par le protocole. Un *device* UPnP est en réalité une *application* déployée sur un *équipement*.

Pour la communication sur *UPnP-DM*, nous souhaitons créer un composant réutilisable. Nous avons conçu un composant générique capable d'interroger tous les nœuds du modèle de données d'*UPnP-DM* et de fonctionner de manière événementielle ou périodique. Or, le protocole possède un mécanisme de notification sur changement de valeur sur certaines variable d'état (propriété *EventOnChange* [UPnP Forum 2012b]). Un mécanisme spécifique a été créé pour qu'un composant externe puisse indiquer à la source de collecter les données. Enfin, le composant doit itérer si nécessaire pour récupérer toutes les instances d'un chemin et envoyer l'ensemble des données dans un seul *batch*, comme prévu dans Astral. L'ensemble des paramètres de cette source est présenté dans la table 8.2.

Paramètre	Description
path	Chemin d'accès aux données
parameters	Liste CSV des paramètres souhaités
period	Période de mise à jour du flux (optionnel)
channel	Canal interne d'événement à souscrire (optionnel)

TABLE 8.2 – Paramètres du composant *DMSource*

Nous construisons périodiquement les flux *OperatingSystem* et *IPUsage* qui contiennent des données *périodiques*. Les autres flux sont des données *imprévisibles* et sont construits à chaque arrivée de *Device* de ce profil. Pour cela, nous formons la requête $\sigma_{status=active \wedge type=dm}\ UPnPStatus$ [5]. Cette requête est injectée dans le puits *SourceNotifier* afin de notifier les *Sources* qui se sont inscrites. Ici, toutes les instances de *DMSource* souhaitant être notifiées le sont par le canal d'événement *DM*. Nous pouvons voir la synergie entre les requêtes déployées et les différent composants et services de notre intergiciel. En effet, nous avons réussi à coordonner les deux mécanismes présentés en section 2.1 via l'utilisation de requêtes.

Nous avons vu la représentation du système, les données intéressantes, et la façon d'obtenir ces données. Nous présentons désormais des exemples de requêtes que nous posons au système d'observation pour démontrer son expressivité.

8.2 Expressivité du système d'observation

Dans cette section, nous détaillons les capacités d'interrogation d'Astral appliquées sur DomVision. Tout d'abord, nous détaillons comment le catalogue est entretenu avec les données provenant des flux. Ensuite, nous archivons des données volatiles. Enfin, nous exploitons cet ensemble pour former des requêtes continues de haut niveau.

8.2.1 Entretien du catalogue

Comme présentés dans la section 8.1.3, nous avons à notre disposition plusieurs flux de données pour remplir le catalogue : *UPnPStatus*, *Serial*, *NetworkInterface* et enfin

5. D'autres requêtes pourraient être utilisées en exploitant les mécanismes de notification de changement de variable d'état.

IPInterface. Le premier flux fournit des renseignements obtenus par l'observation de la couche applicative (ici *UPnP*). Il contient des données moins précises que les autres flux. Toutefois, il permet de renseigner des données pour les équipements qui n'ont pas l'agent *UPnP-DM*.

Par exemple, si nous détectons dans notre réseau domestique, un agent *UPnP-DM* et un profil de partage de contenu. Les deux applications sont exécutés sur deux appareils distincts. L'agent *DM* fournit un ensemble de données précises sur le premier dispositif avec les flux présentés en figure 8.1. Nous n'avons pas d'informations précises sur l'autre équipement, toutefois, nous savons qu'il existe un autre dispositif qui peut partager du contenu.

Afin de mettre à jour le catalogue en fonction des n-uplets de *UPnPStatus*, nous séparons ses attributs par concepts sémantiques :

- *ip* appartient à *Interface*
- *uuid* convient au *nom* unique d'*Application*
- *FriendlyName* correspond à la *description* d'*Application*
- *type* et *status* correspondent directement au schéma d'*Application*.
- aucun attribut n'appartient à *Équipement*

Nous créons un composant puits*StatusSink* capable de mettre à jour le catalogue avec ces informations, la figure 8.1 décrit en détail le processus.

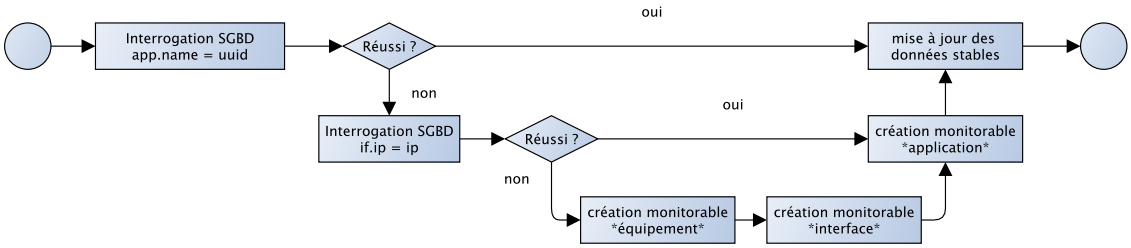


FIGURE 8.1 – Description du processus de mise à jour de *StatusSink*

Concernant les informations venant d'*UPnP-DM*, nous pouvons regrouper tous les flux en un seul flux DM grâce aux clés de jointures *uuid* et *SystemName* [6]. Ainsi, nous obtenons un flux indiquant que l'interface *SystemName* (avec les propriétés données) est sur l'équipement identifié par le *SerialNumber*.

$$DM(uuid, SerialNumber, SystemName, MACAddress, InterfaceType, IPv4Address, \tau)$$
$$DM = \mathcal{I}_{\mathcal{S}}(\text{Serial}[B] \bowtie \text{NetworkInterface}[B] \bowtie \text{IPInterface}[B])$$

Nous pouvons appliquer la même approche que pour *StatusHandler* en créant un composant puits*DMSink* pour mettre à jour le catalogue grâce aux données de DM. La figure 8.2 résume le processus de ce puits.

6. Les sources sont synchronisées par le mécanisme d'événement. Sans cette synchronisation un partitionnement *uuid* serait nécessaire.

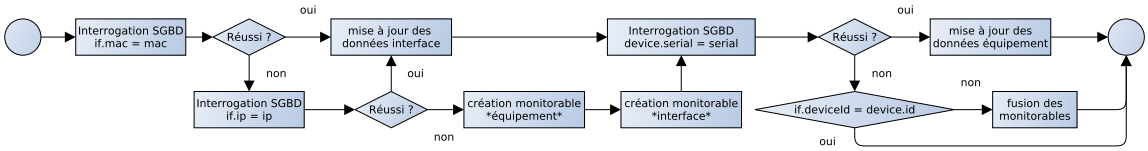


FIGURE 8.2 – Description du processus de mise à jour de *DMSink*

8.2.2 Historisation des données volatiles

Nous souhaitons maintenant archiver les données volatiles. Le premier point important à rappeler est que les flux donnés aux composants *VolatilePersistence* doivent être identifiés, i.e. possèdent un attribut *monitorableId*. Ainsi, le flux *OperatingSystem* doit subir une opération pour transformer son *uuid* en *monitorableId* correspondant à l'*équipement* qui le produit. La requête *CPUMem* fournit le flux avec les attributs $A =$(*CPUUsage*, *MemoryUsage*, *monitorableId*,τ) répondant à ce problème :

$$CPUMem = \Pi_A \mathcal{I}_{\mathcal{S}} \quad (OperatingSystem[B] \bowtie (\rho_{monitorableId/deviceId,uuid/name} Applications \bowtie Monitorable))$$

Ce flux est donné à une instance de *VolatilePersistence* qui archive dans la table *cpumemhistory* les données volatiles : *cpu* et *memory*. Nous pouvons noter que nous avons fait une jointure avec le catalogue sans difficulté.

Le flux *IPUsage* ne nous donne pas les informations que nous souhaitons. En effet, nous voulons obtenir les données de débit, et le flux fournit un compteur d'octet. Toutefois, nous pouvons transformer une suite de comptage (c_n, τ_n) en série de débits (d_n, τ_n) par l'opération $d_n = \frac{c_n - c_{n-1}}{\tau_n - \tau_{n-1}}$ calculée par l'opérateur e^a_f évaluant l'expression f dans l'attribut a. L'opérateur $\mathcal{D}^{(t,i)^-}_{t>t_0}$ est capable de revenir un *batch* en arrière, ainsi la requête suivante[7] forme le flux voulu :

$$BandwidthUsage = \Pi_{\dots} \mathcal{I}_{\mathcal{S}} \left(e^{bwr}_{\frac{tbr-otbr}{\tau-o\tau}} e^{bws}_{\frac{tbs-otbs}{\tau-o\tau}} \left(IPUsage[B] \bowtie \mathcal{D}^{(t,i)^-}_{t>t_0} \rho_{\substack{otbs/tbs \\ otbr/tbr \\ o\tau/\tau}} IPUsage[B] \right) \right)$$

Nous avons maintenant le flux possédant les données que nous souhaitons archiver. Comme pour *CPUMem*, nous devons identifier l'entité sur laquelle nous enregistrons cette donnée. Ceci est faisable par une jointure semi-sensible sur la base de données en utilisant le *uuid* de l'agent et le *SystemName* représentant le *nom* de l'interface. Cela produit un flux *Bandwidth* que nous injectons dans un puits de persistance.

Les autres paramètres volatiles intéressants (changement d'*IP* ou de *status*...) s'enregistrent par le même procédé : extraire la ou les données à historiser ; identifier l'objet de rattachement.

Nous avons désormais un système d'observation capable d'entretenir sa vision du système et qui archive ses données volatiles. Ces données sont disponibles à travers le SGBD ce qui permet de faire diverses interrogations instantanées avec une capacité d'expression équivalente au *SQL*. À l'aide de notre système, dans un projet interne de *Orange Labs*, il a été possible de créer un panneau de contrôle indiquant la topologie du réseau ainsi que son histoire et des graphiques de métriques.

7. TotalBytesSent et TotalBytesReceived ont été notés en tbs et tbr pour plus de facilité.

8.2.3 Formations d'alertes

Nous montrons maintenant quelques exemples de requêtes continues formant des alertes. La première est une simple sélection sur une valeur limite. Par exemple, nous pouvons supposer que les charges processeurs dépassant le seuil de 90% sont à notifier à l'utilisateur. Nous pouvons raffiner ce critère par « *la moyenne sur 1 minute supérieure à 90%* » afin de lisser les pics de valeurs. Nous pouvons exploiter le flux déjà prêt $CPUMem$ pour effectuer cette requête. Le nouveau flux est ensuite injecté à un composant puis de notification.

$$\mathcal{I}_{\mathcal{S}}\, \sigma_{avgcpu>90}({}_{monitorableId}\mathcal{G}^{avgcpu}_{avg(CPUUsage)} CPUMem[T\ 1min\ 1min])$$

Afin de montrer les capacités d'interrogation de notre système, nous pouvons améliorer cette requête en exprimant le critère « *la moyenne sur 1 minute est différente à 10% près de la moyenne historique* ». La valeur historique est obtenue par la relation temporelle produite par $HistAvgCPU = {}_{monitorableId}\mathcal{G}^{avgcpuhist}_{avg(cpu)} cpumemhistory$. Nous obtenons ainsi la requête :

$$\mathcal{I}_{\mathcal{S}}\, \sigma_{|avgcpu-avghist|\geq 10}\, (AvgCPU \bowtie HistAvgCPU)$$

Bien que cette requête mêle données historiques persistantes et données à la volée, nous sommes capables de l'écrire et de l'exécuter dans notre approche. Ceci démontre que notre approche est capable d'intégrer flux et relations persistantes par un langage semi-déclaratif.

8.2.4 Journalisation du catalogue

De plus, si l'utilisateur souhaite connaître les états passés du catalogue, il est possible de garder une trace dans le support persistant de tels changements. Prenons l'exemple de la relation *Interfaces* du catalogue. Nous pouvons créer les flux d'insertion et de suppression de cette relation temporelle grâce aux opérateurs $\mathcal{I}_{\mathcal{S}}$ et $\mathcal{D}_{\mathcal{S}}$. Ainsi, un historique des changements de cette relation peut se traduire par :

$$InterfacesHist = \begin{cases} InterfacesIS = \mathcal{I}_{\mathcal{S}}(Interfaces) \\ InterfacesDS = \mathcal{D}_{\mathcal{S}}(Interfaces) \end{cases}$$

En persistant ces deux flux dans la base de données grâce à un puits de persistance, il est possible de reconstituer la relation *Interfaces* à un temps quelconque τ.

De plus, ces historiques permettent de fournir à l'utilisateur la liste des changements de son catalogue. Par exemple, il est possible de détailler une liste d'événements indiquant « *un nouvel appareil vient de se connecter* » ou « *le profil de partage de contenu a disparu* ». Ce type d'information est très utile dans le cadre du diagnostic afin de tracer au mieux les causes des problèmes.

Alternativement, il est possible d'archiver cette relation par l'utilisation de l'opérateur $\mathcal{R}^{u}_{\mathcal{S}}$. Ceci permettra d'enregistrer l'ensemble de la relation à chaque changement. Nous remarquons que nous retrouvons le principe du calcul incrémental évoqué en section 3.4 en matérialisant une relation par l'ensemble de ses différences ou par l'ensemble de ses états.

8.3 Conclusion

DomVision est le système d'observation produit par Astral-Astronef-Asteroid pour le réseau domestique. Il a montré une large capacité d'adaptation au système observé. L'utilisation de notre solution dans un nouvel environnement passe par plusieurs étapes. Tout d'abord, l'utilisateur doit définir sa représentation du système et les données qu'il souhaite archiver. Ensuite des composants sont créés pour dialoguer avec les fournisseurs de données.

À partir des flux de données disponibles et des capacités d'expressions d'Astral, nous avons pu entretenir le catalogue représentant le système. De même, nous avons archivé les données volatiles que nous souhaitions. Ces données ont été extraites par l'utilisation de requêtes Astral. Par la suite, nous avons créé des alertes impliquant dans le cas le plus complexe des jointures entre les historiques du SGBD et les flux temps-réel. Enfin, nous avons historisé le catalogue descriptif du système.

Nous avons été confrontés aux limites de notre approche lors de l'écriture des composants de mise à jour du schéma descriptif. Il est difficile d'évaluer la pertinence et les limitations des moyens d'identifications alternatifs ce qui peut être source d'inconsistances.

Il est important de noter que toutes les notions et exemples présentés dans cette section ont été implémentés et mis en pratique dans la pratique sur un réseau domestique d'expérimentation comme présenté dans le papier de démonstration [Petit 2011]. De plus, ce projet a été intégré à un intergiciel plus large de gestion de service dans le réseau domestique présenté dans [El Kaed 2011]. DomVision sert ici de fournisseur de données pour permettre de gérer la qualité de service des différentes applications déployées dans ce réseau.

Nous n'avons cependant toujours pas analysé l'aspect performance d'Astronef-Asteroid. Le chapitre suivant présente quelques expérimentations que nous avons pu faire pour mesurer l'efficacité de notre solution.

« There's no need to go struttin' around and showin' off like that. That's my job! »

Rainbow Dash

9 Évaluation de performances

Afin d'effectuer l'observation de systèmes, nous avons présenté dans les chapitres précédents un SGFD extensible capable de se coupler avec un SGBD. Pour permettre le déploiement de cette solution sur tout type de plateforme, nous devons nous assurer que notre système est efficace en terme de performance. Pour cela, nous mesurons différents plans d'exécution afin de montrer les capacités de notre système à optimiser les expressions algébriques.

Tout d'abord, en section 9.1, nous présentons notre cadre expérimental. Ensuite, en section 9.2, nous nous intéressons à l'optimisation de la gestion de flux de données en soit avec le support du *benchmark Linear Road*. Nous traitons l'efficacité du couplage avec le SGBD dans la section 9.3. Enfin, nous concluons en section 9.4.

9.1 Cadre d'expérimentation

L'ensemble de la distribution Astronef-Asteroid est sous forme de *bundles Java-OSGi*. Ainsi, ces prototypes peuvent être déployés sur toute plateforme possédant la technologie *Java*. De plus, l'environnement *OSGi* nous permet d'exploiter un environnement modulaire basé sur les architectures à service. La plateforme *OSGi* doit embarquer le *bundle iPojo* (`http://felix.apache.org/site/apache-felix-ipojo.html`) pour pouvoir utiliser l'architecture à composants orientés service. Nous utilisons dans nos expériences la plateforme *OSGi Apache Felix* en version 3.0.2.

La distribution Astronef est fournie en trois *bundles* obligatoires à déployer pour pouvoir utiliser l'ensemble des fonctionnalités présentés dans cette thèse (api core parser). Ces *bundles* embarquent aussi le moteur *Prova* (`http://prova.ws`, version 3.1.9 minimale) capable d'évaluer l'ensemble des règles présentées. Les extensions à Astronef sont aussi sous forme de *bundles* dont les classes dépendent de l'*api*. Astronef est disponible sous licence Apache 2.0 à l'adresse `http://astral.googlecode.com`.

Asteroid est une extension d'Astronef distribuée en un seul *bundle*. Il embarque le SGBD *H2* (`http://h2database.com`). Ce SGBD est entièrement en *Java* ce qui permet une uniformité en terme de technologies. Le binaire ou le code source d'Asteroid n'est actuellement pas disponible au public.

L'ordinateur utilisé pour les expérimentations possède un processeur *Intel Xeon*, quadricœur de fréquences 2.8Ghz. Il possède 6Go de mémoire vive et un disque dur d'une vitesse de 7200RPM. Le système d'exploitation installé est Linux Ubuntu 11.04. La plupart des expérimentations se sont faites dans un environnement clôt en isolant les processus sur trois cœurs dédiés afin d'éviter les interférences. Enfin, les expérimentations ont été faites dans des conditions les plus stables possible, après que le *JIT* soit passé, après initialisation des caches internes et avec un *garbage collector* (*GC*) le plus stable possible.

Les performances sont mesurées par la latence d'un n-uplet. La latence est mesurée par la différence de timestamp système entre la source et le puits de la requête. Elle permet d'indiquer le temps total nécessaire au traitement d'un n-uplet. Le coût mémoire n'est pas compté, mais comme les trop grands coûts impactent le temps de traitement, notamment en *Java* avec le *GC*, nous supposons que cette métrique est représentative.

9.2 Éléments d'expérimentations sur Linear Road

Dans cette section, nous confrontons Astronef avec le *benchmark Linear-Road* [Arasu 2004b]. L'objectif de cette étude n'est pas de démontrer qu'Astronef est le SGFD le plus rapide. Nous voulons par contre vérifier qu'il est possible d'obtenir des plans de requêtes aussi efficaces en écrivant directement la spécification du *benchmark* en Astral.

9.2.1 Le *Linear Road Benchmark*

Le *Linear Road Benchmark* [Arasu 2004b] (*LRB*) est un *benchmark* de gestion de flux de données maintenant très répandu dans ce domaine. Le cadre d'application est de surveiller des autoroutes pour gérer les alertes d'accidents ainsi que les notifications de facturation au péage fluctuante. Une voiture ne paie le péage que si toutes ces conditions sont vérifiées : (1) il n'y avait pas d'accident dans ce segment, (2) la moyenne de vitesse des voitures du segment était inférieure à 40*mph* et enfin (3) le nombre de voitures sur le segment était supérieur à 50. Nous avons le flux principal $P(xway, dir, seg, vid, pos, speed, \tau)$ indiquant la vitesse la position (ainsi que son segment), la direction et le numéro de l'autoroute d'une voiture *vid*.

Pour pouvoir calculer les notifications en quasi-temps réel[1], il est nécessaire de calculer aussi les vitesses moyennes des véhicules ainsi que les densités des segments. Le *benchmark* permet de voir si le SGFD peut supporter la charge de L autoroutes pendant trois heures. Le score du benchmark est le L maximal tout en respectant les critères.

Dans cette section, nous nous intéressons avant tout à la requête de péage $Toll(vid, avgspeed, toll, \tau)$. Son expression est complexé et est présentée dans l'annexe B. Nous analysons ici des fragments de cette requête qui sont problématiques en terme de performances. Les autres requêtes sont des requêtes faisant intervenir des historiques que nous explorons en détail avec notre cas d'application dans la section suivante.

9.2.2 L'exploitation des commutativités et associativités

L'avantage concret de l'optimisation logique à base de commutation prend tout son sens lorsqu'un langage déclaratif est utilisé. En effet, en *SQL*, il est possible d'exprimer une requête Sélection-Projection-Jointure. Chaque partie est clairement défini dans les clauses du `SELECT ... FROM ... WHERE`. Sa traduction en algèbre relationnelle nécessite plusieurs transformations pour obtenir une expression efficace.

A contrario, si l'expert écrit lui même sa requête en algèbre, celle-ci est déjà efficace. C'est pour cela que l'algèbre est considérée comme un langage impératif. Ainsi, nous n'avons pas pu observer de grandes optimisations avec ou sans l'optimisation logique. Toutefois, lorsqu'un langage déclaratif sera défini, cet aspect prendra de l'importance.

9.2.3 Changement de segment : jointure massive et manipulation temporelle

Dans cette partie, nous analysons la requête permettant de sélectionner les n-uplets de P indiquant un changement de segment du véhicule. Cette requête peut s'écrire de cette façon :

$$\mathcal{R}^{u}_{\mathcal{S}}(P[B] \bowtie_{seg \neq seg2} \rho_{seg2/seg}\Pi_{xway,vid,seg}\mathcal{D}^{(t,i)^-}_{\tau>t_0} P[xway, vid/L]) \tag{Q1}$$

Plusieurs questions de performances sont à mettre en en avant :

– Quel algorithme de jointure ? La taille de $P[xway, vid/L]$ risque d'être importante. Sur l'ensemble du *LRB*, il existe environ 125000 valeurs différentes pour *vid* pour $L = 1$. La jointure par boucle imbriquée n'est pas envisageable. Le calcul à l'aide d'une table de hachage est nécessaire. Nos expériences nous ont indiqué qu'à part pour des cardinalités faibles (de l'ordre de la dizaine), les jointures avec index calculé à la volée sont à privilégier même si cela implique la reconstruction de l'index à chaque évaluation.

– Le mode incrémental est-il utilisable ? L'opérateur $\mathcal{D}$ permet certes de faire passer l'incrémental jusqu'à la jointure. Toutefois, la jointure n'est pas calculable facilement en incrémental, car les δ^- ne sont pas nuls. De plus, $P[B]$ change entièrement entre deux *batchs*. La sélection d'opérateurs statique est préférable.

1. Contrainte de délai de 5 secondes maximum

– Dans ce cas $\mathcal{I}_{\mathcal{S}}$ et $\mathcal{R}^{u}_{\mathcal{S}}$ ont un comportement similaire, car l'intersection des états de la relation est nulle. Est-ce que l'un est privilégié par rapport à l'autre ? Comme les relations ne sont pas incrémentales, $\mathcal{R}^{u}_{\mathcal{S}}$ a un coût linéaire tandis que $\mathcal{I}_{\mathcal{S}}$ a le coût d'une autojointure.

Voici maintenant deux optimisations logiques non triviales (mais démontrable) à cette requête.

$$\mathcal{R}^{u}_{\mathcal{S}}(P[B] \bowtie_{seg \neq seg2} \rho_{seg2/seg} \Pi_{xway,vid,seg} \mathcal{D}_{\mathcal{S}}(P[xway, vid/L])[B] \quad \text{(Q2)}$$

Cette optimisation permet d'exploiter que le coût de $\mathcal{D}_{\mathcal{S}}(P[vid/L])$ est nul si l'opérateur de fenêtre partitionnée fournir une relation temporelle incrémentale (ce qui est le cas dans Astronef). Ainsi, nous obtenons deux jointures sur deux *batchs* dont les cardinalités sont aux alentours de 1000 [Jain 2006]. Toutefois, le coût mémoire de $P[xway, vid/L]$ est important, car il faut tracer les 125000 identifiants. Nous notons que cette optimisation est formalisable dans Astronef.

Nous exploitons désormais qu'un véhicule émet une donnée toutes les 30s. Nous exploitons aussi que notre description de fenêtre est capable de correctement traiter l'inclusion ou l'exclusion des bornes inférieures et supérieures.

$$\mathcal{R}^{u}_{\mathcal{S}}(P[B] \bowtie_{seg \neq seg2} \rho_{seg2/seg} \Pi_{xway,vid,seg} P[T\ 30s\ 30s[) \quad \text{(Q3)}$$

Nous obtenons une jointure avec une cardinalité de 30 *batchs* à droite ($\sim$30000 n-uplets pour $L = 1$) mais dont la table de hachage n'est calculée qu'une fois toutes les 30 secondes. Cependant, pour avoir une optimisation automatique, il faut avoir une connaissance sur les propriétés du flux P. Nous remarquons encore la pertinence de l'utilisation de statistiques pour caractériser les flux. Nous notons B la taille d'un *batch* et V le nombre de *vid* différents pour une autoroute.

Stratégie	Mémoire	Calcul
Q1	$\mathcal{O}(LV)$	$\mathcal{O}(LV + LB)$
Q2	$\mathcal{O}(LV)$	$\mathcal{O}(LB)$
Q3	$\mathcal{O}(LB)$	$\mathcal{O}(LB)$

L = Nombre d'autoroutes
V = Nombre de voitures
B = Taille d'un *batch*

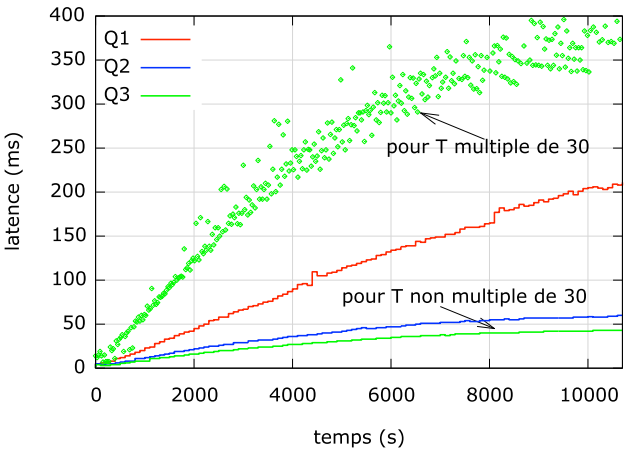


FIGURE 9.1 – Complexités et tracé de latence des différentes stratégies pour la requête de détection des changements de segment

La figure 9.1 trace les médianes (sur 100s) des mesures de latences (ordonnées) que nous obtenons au cours du temps (abscisses). La stratégie Q2 évolue en fonction de la charge du benchmark alors que Q1 évolue selon le nombre de voitures passé (linéaire dans notre cas). Au final, nous obtenons une latence égale à 60ms pour $\mathcal{D}_{\mathcal{S}}$ et

210ms pour Q1. Enfin, la stratégie utilisant une fenêtre temporelle (Q3) a une meilleure performance la plupart du temps, toutefois pour les moments où la fenêtre s'évalue ($\tau \in 30\mathbb{N}$) alors la latence est quasiment décuplée pour passer à $350 - 400ms$. Néanmoins, la charge mémoire dépend uniquement de la taille des *batchs*.

9.2.4 Calcul de vitesse d'un segment : fenêtre et agrégat

Dans cette partie, nous analysons la requête permettant de calculer la vitesse moyenne pour un triplet *(xway,dir,seg)*. Le calcul de l'agrégation telle que la spécification le présente est en trois parties comme la requête suivante le démontre :

$$\underset{seg,dir}{xway,}\mathcal{G}^{avgspeed}_{avg(tmp)}\left(\underset{seg,dir}{m,xway,}\mathcal{G}^{tmp}_{avg(tmp)}\left(\underset{seg,dir}{vid,m,xway,}\mathcal{G}^{tmp}_{avg(speed)}\left(e^{m}_{\lfloor T/60\rfloor}\right)\right)[T\ 5min\ 1min]\right)$$

En supposant que nous appliquons à la lettre ce plan de requête (plan $G1$), nous avons chaque minute : un agrégat d'environ 300000 n-uplets à calculer, suivi de deux agrégats d'une centaine de milliers puis d'une vingtaine de milliers. De plus, ce coût n'est pas amorti, car l'opérateur de fenêtre est bloquant.

Pour améliorer la performance de cette requête, nous avons implémenté l'opérateur Pane [Li 2005] capable de faire des agrégations efficaces sur des fenêtres glissantes (son fonctionnement a été présenté dans la section 3.4.2). Dans ce nouveau plan ($G2$), le coût d'une fenêtre plus agrégation devient amorti puisque le déclenchement d'une nouvelle fenêtre produit quasiment instantanément le résultat. Toutefois, les deux autres agrégations restent à calculer avec de grandes cardinalités.

Nous avons ainsi implémenté un agrégat complexe capable de composer deux agrégats (plan $G3$). Par exemple, l'agrégat $avg(vid/avg(speed))$ calcule la moyenne des moyennes pour chaque véhicule. La complexité en terme de mémoire et de calcul reste le même. Toutefois, il ne nécessite qu'un opérateur et lors de l'évaluation de la fenêtre, les sous-agrégats sont prêt à être calculés ce qui rend l'opération bien plus rapide. Il est de plus possible d'automatiser la réécriture de ce type d'opérations.

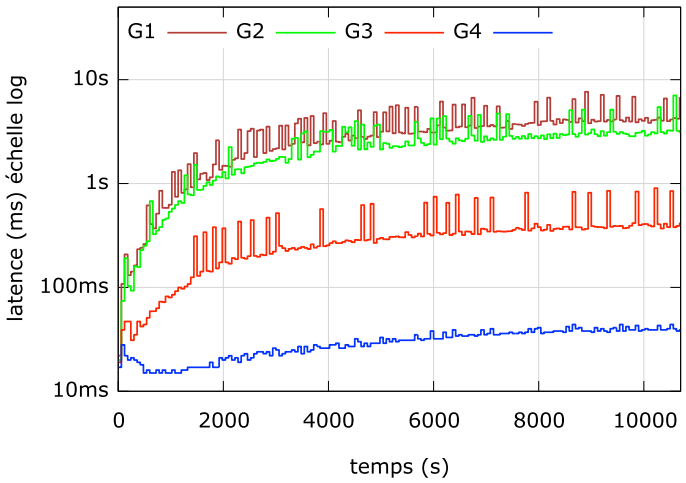


FIGURE 9.2 – Tracé de la latence des stratégies pour le calcul de vitesse

La figure 9.2 montre la latence observée sur les différentes stratégies. Nous voyons effectivement que $G2$ a des performances similaires à la fenêtre simple (gain 25%) car peu de données sont présentes par groupes dans la première agrégation. Cependant, le plan $G3$ suit la même tendance, mais avec un gain de performance de 10. Enfin, nous pouvons voir que le calcul d'un Pane simple (plan $G4$) est très efficace même sur de grandes quantités de données. De plus, il est plus stable au niveau de la mémoire, car nous remarquons qu'il n'y a pas de pic de latences (dues au travail du *garbage collector*).

9.2.5 Allègement des structures internes

Enfin, nous pouvons voir que nous pouvons établir des règles sur les comportements des opérateurs nous permettant de faire des économies de mémoire et de calcul en modifiant les structures internes. En effet, il existe plusieurs façons d'implémenter les relations temporelles, les flux et les séquences de n-uplets. Par exemple, il existe des structures pour les relations temporelles supportant un état, ou deux, ou N. De même, il existe des relations temporelles incrémentales ou non. Nous pouvons appliquer des règles permettant de décider de la meilleure structure en fonction des opérateurs.

Par exemple, dans le cadre de $\mathcal{R}^{u}_{\mathcal{S}}(S)$. Nous pouvons indiquer : un seul état est nécessaire, l'état courant ; si la relation temporelle est incrémentale, alors il faut prévoir un entretien de l'état courant, les δ n'ont pas besoins d'être conservés. À l'inverse, $\mathcal{I}_{\mathcal{S}}$ ne nécessite que des δ ce qui évite d'entretenir l'état courant, coûteux en mémoire et en calcul.

De la même façon, nous pouvons prévoir de supprimer des vérifications de contraintes (le schéma de la séquence est respecté, etc.), si l'implémentation les garantit. Sans ces optimisations, nous observons une augmentation de la latence importante (le double dans le cadre de la requête de changement de segment).

9.3 Choix du plan de jointure dans Asteroid

En section 7.3.3, nous avons présenté un opérateur capable de faire une jointure $\bowtie$ entre une relation temporelle d'Astronef et une relation issue d'un SGBD. Deux plans ont été présentés :

P1 applique la jointure (hachée) dans Astronef en utilisant la relation temporelle issue de *dbsource* comme cache local.

P2 quant à lui exécute l'opération de jointure à l'intérieur du SGBD avec *dbjoin* pour exploiter ses capacités.

Alors que l'optimisation poussant les opérateurs au plus proche du SGBD semble efficace dans la plupart des cas, nous voyons dans cette section que ce n'est pas toujours le cas pour cette opération.

9.3.1 Jointure sur une relation statique

Nous exécutons ici la requête $CPUMem$ présenté en section 8.2.2. Pour rappel, cette requête implique une jointure avec $Application \bowtie Monitorable$ pour pouvoir identifier

le flux de métriques processeurs et mémoires. Voici les paramètres d'expérimentations que nous fixons :

– Les relations temporelles *Monitorable* et *Applications* sont considérées statiques

– La cardinalité d'*Applications* est N

– La cardinalité de *Devices* est $0.2N$ et celle de *Monitorable* est de $1.2N$

– Il existe un index *Application(monitorableId)*, *Monitorable(monitorableId)* et sur *Monitorable(name)*

– Il y a toujours suffisamment de mémoire pour exécuter les requêtes *SQL*.

La requête *SQL* automatiquement générée et donnée aux composants *dbjoin* et *dbsource* est la suivante :

```
1 SELECT deviceId as monitorableId, name
2 FROM Application NATURAL JOIN Monitorable
```

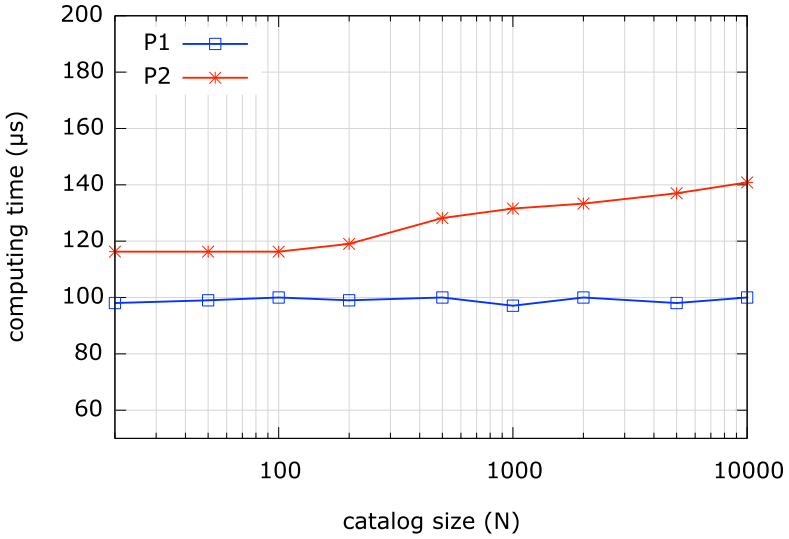


FIGURE 9.3 – Performance d'une jointure sur catalogue statique

Résultats : La figure 9.3 montre la latence des deux plans de requête **P1** et **P2**. Les expérimentations ont montré que le plan *P1* est meilleur et plus stable avec N augmentant. Bien entendu, cela ne pourra pas être le cas pour des valeurs très larges de N. Dans notre cadre expérimental, le *hash-join* ne semble pas être le goulot d'étranglement, car cela consiste à interroger une table de hachage constituée une seule fois. Le coût du *scheduling* et d'autres tâches de l'intergiciel deviennent des facteurs limitants.

Le plan **P2** est plus coûteux, car pour chaque n-uplet, le composant doit se connecter au SGBD et exécuter la requête. Bien que la requête soit préparée, cela introduit un coût supplémentaire. Ce plan utilise l'index stocké sur disque dur, mais ses performances sont moindres comparé à un accès direct à une table de hachage en mémoire.

Si les relations *Monitorable* ou *Applications* sont mis à jour, cela introduit un surcout pour recalculer la table de hachage. Toutefois, le coût est absorbé au fur et à mesure du temps si les relations sont mises à jour rarement. Ainsi, le choix de **P1** est le meilleur pour la jointure avec des relations stables.

9.3.2 Jointure sur un agrégat historique

Dans cette seconde expérimentation, nous exécutons la requête *HistAvgCPU* vu dans la section 8.2.3. Pour rappel, cette requête joint une relation temporelle avec un agrégat effectué sur historique. Voici les paramètres de notre expérience :

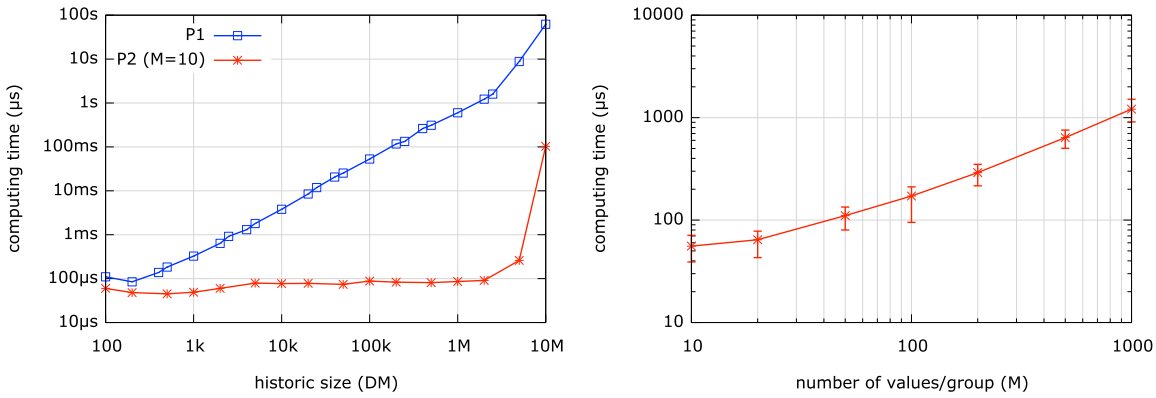
– L'historique est mis à jour dès que possible par un processus extérieur (une requête dans Astronef par exemple).

– Il existe D identifiants d'équipement et M valeurs historiques par identifiants.

– Il y a un index sur *HistCPU(monitorableId)*.

La requête *SQL* générée et donnée en paramètre à *dbjoin* et *dbsource* est la suivante :

```
1 SELECT monitorableId, AVG(cpu) as avg FROM HistCPU GROUP BY monitorableId
```



(a) Performance globale

(b) Performance de l'agrégat avec sélection sur identifiant

FIGURE 9.4 – Performance de la jointure avec agrégat en utilisant la sélection ou non

Résultats : Dans cette expérience, nous avons pu remarquer que les performances de la requête *SQL* de *dbsource* ne dépend que de la taille de l'historique $(M.D)$. Toutefois, ce n'est pas le cas pour un agrégat utilisant une sélection sur l'identifiant avec *dbjoin*, qui ne dépend principalement que de M. La figure 9.4(a) représente l'évolution de la latence de la requête d'agrégat avec et sans la jointure faite par *dbjoin* [2]. La figure 9.4(b) représente l'évolution de la latence en variant M pour l'agrégat avec sélection.

La performance dépend des caractéristiques de l'historique. Plus nous avons un M petit, plus nous avons une petite latence pour le plan **P2**. La latence de **P2** est bornée entre les deux courbes de la figure 9.4(a) comme le cas $M = 10$ représente le minimum de latence et $D = 1$ (équivalent à aucune sélection) est le pire cas. Le choix du plan est clairement en faveur du plan **P2** car son pire cas a des performances similaires à **P1**. Dans l'expérience précédente, nous avions supposé pouvoir absorber le coût de mise à jour. Ce n'est pas le cas ici, car l'historique est régulièrement mis à jour.

2. La perte de performance massive observée après $4M$ n-uplets est due au *buffering* qui ne peut être fait en mémoire et le disque est utilisé. Ce fait a été confirmé par les développeurs d'*H2*.

Stratégies alternatives

Si le résultat de la requête peut être dégradé, il est possible d'utiliser *dbsource* en mode périodique. Dans ce cas, nous pouvons évaluer la condition limite pour passer d'un plan à l'autre en regardant les coûts mesurés dans la figure 9.4.

Nous pouvons toutefois trouver une autre réécriture fournissant un résultat exact. Lorsque la requête est déployée, il y a une distinction entre le passé ($CPUMemHistory^{t_0}$) et le futur représenté par le flux $CPUMem$. En supposant que l'agrégat est séparable par les moyens décrits dans la section 3.4.2, alors nous pouvons séparer les calculs sur le passé et le futur. L'agrégat passé est un calcul instantané effectué par *dbsource* est mode *one-shot*. L'agrégat futur est calculé par les opérateurs ${}_{id}\mathcal{G}_{...}$ et $[\infty]$. Or, ces opérateurs peuvent être regroupés en un seul macrobloc capable de calculer l'agrégat avec un coût mémoire borné.

9.4 Conclusion

La performance est un élément clé pour la gestion des flux de données. En effet, gérer des données temps réel implique les traiter le plus rapidement possible pour ne pas risquer des blocages. Nous avons vu dans ce chapitre que notre approche nous permet de générer des plans de requêtes efficaces autant pour la gestion de flux de données que pour la jointure avec un SGBD.

Nous avons tout de même perçu la limite de notre approche. En effet, nous avons opté pour un système de règle. Cela nous permet d'avoir des résultats rapides et une intégration très efficace. Toutefois, pour la spécification de l'algorithme *Pane* (macrobloc fenêtre-agrégat) par exemple, nous avons dû prévoir le cas où une projection se serait glissée entre la fenêtre et l'agrégat lors de l'optimisation logique. Ce cas-là n'est pas grave, car la sémantique reste identique avec ou sans. Mais dans le cas d'une sélection, ou éventuellement un autre opérateur, nous ne pouvons peut-être plus former notre macrobloc, nous privant d'un gain certain de performance.

Nous avons présenté nos contributions et validé leurs résultats. Nous avons ainsi un système d'observation efficace capable d'interroger tout type de données. Le chapitre suivant présente une extension de ces travaux pour personnaliser les résultats en fonction de l'utilisateur.

Partie V

Personnalisation des résultats

Notre approche permet d'observer un système hétérogène et dynamique. Nous avons montré que notre intergiciel est adaptable aux différentes situations. Lors de nos expérimentations, il nous a paru intéressant de pouvoir adapter aussi les données en fonction des utilisateurs. Ce prochain chapitre présente une extension de notre intergiciel pour effectuer une telle personnalisation des données.

« The shape's fine, just make the whole thing... you know, cooler.
It needs to be about 20% cooler. »

Rainbow Dash

10 Gestion des préférences utilisateurs

Grâce à la popularisation des technologies et des capacités de traitement de données, les systèmes sont de plus en plus complexes et fournissent de plus en plus de données. Le spectre des utilisateurs étant plus large, il est important de pouvoir adapter la masse d'informations récoltée à chaque personne. De nombreux efforts ont été consacrés ces dernières années à la personnalisation des réponses lors de l'accès aux bases de données. Dans ce chapitre, nous explorons une extension à notre approche pour ajouter des opérateurs adaptant les résultats d'interrogations aux préférences utilisateurs. Ainsi, l'utilisateur n'obtient que les données qui l'intéressent.

Dans la section 10.1, nous détaillons les fondements théoriques à la gestion de préférences contextuelles. Une fois les opérateurs décrits dans l'algèbre Astral, nous pouvons présenter les algorithmes utilisés pour les mettre en œuvre dans la section 10.2. Enfin, dans la section 10.3 nous présentons l'intégration faite dans Astronef pour que l'utilisateur puisse personnaliser ses résultats d'interrogation. Cette intégration est accompagnée d'une évaluation de performances pour sélectionner le meilleur algorithme de calcul.

10.1 Modélisation algébrique

Dans cette section, nous introduisons dans le cadre d'Astral-Astronef les opérateurs de *CPref-SQL* [de Amo 2009]. Tout d'abord, nous présentons les notions théo-

riques nécessaires pour appréhender les préférences contextuelles. Puis, nous présentons les deux opérateurs permettant de sélectionner les n-uplets préférés sur une relation temporelle. Ensuite, nous utilisons ceci à notre cadre applicatif. Enfin, nous présentons, comment l'intégration dans Astronef est réalisée.

10.1.1 Préférences contextuelles

Dans cette section nous présentons les principaux concepts concernant le formalisme logique que nous employons pour spécifier et raisonner avec des préférences. Tout d'abord, présentons le concept de préférence contextuelle. Une *préférence contextuelle* est une manière d'exprimer la phrase suivante : « *Lorsque u est vrai, je préfère* Q_1 *à* Q_2 *à attributs W égaux* ». Par exemple, « *pour des films de Woody Allen, je préfère les comédies aux drames à décennie égale* ». La définition 10.1.1 définit formellement cette notion.

Définition 10.1.1 **Préférence Contextuelle**

Soit A un ensemble d'attribut,
Une *règle de préférence contextuelle* (ou *cp-règle*) sur A est une formule φ de la forme :

$$\varphi : u \rightarrow Q_1(X) \succ Q_2(X)\ [W]$$

- X est un attribut non temporel de R tel que $W \subseteq A$, $X \notin W$,
- $Q_i(X)$ est une condition exprimée sur X,
- $\forall x$, $Q_1(x)$ et $Q_2(x)$ ne peuvent être satisfaites simultanément,
- u est une condition ne portant ni sur X ni sur les attributs de W.

La formule u du côté gauche d'une cp-règle φ est appelée *contexte*. L'expression $Q_1(X) \succ Q_2(X)$ du côté droit est appelée *expression de préférence* et les attributs dans W sont appelés des attributs *ceteris paribus* (cf. ci-dessous). Un n-uplet s est dit *compatible* avec une cp-règle φ si s satisfait son contexte.

Une *théorie de préférence contextuelle* (*cp-théorie*) sur A est un ensemble fini Γ de cp-règles sur R. Nous notons par Attr(Γ) l'ensemble d'attributs dans les cp-règles de Γ.

Ainsi, une cp-règle φ sur A induit une *relation binaire* (notée $\succ_\varphi$) sur une séquence de n-uplet d'attributs A, à savoir l'ensemble des paires (s_1, s_2) telles que s_1 est préféré à s_2 par rapport à φ. Cette relation binaire n'est pas nécessairement transitive, ce qui fait que ce n'est pas une *relation d'ordre*. Nous définissons la notion de *Relation de Préférence* inférée par une cp-théorie Γ.

Définition 10.1.2 **Relation de Préférence**

Soit Γ cp-théorie sur A,
La *Relation de Préférence* associée à Γ (notée $\succ_\Gamma$) est définie comme :

$$\succ_\Gamma = (\cup_{\varphi \in \Gamma} \succ_\varphi)^*, \text{ où } * \text{ dénote la } \textit{fermeture transitive}.$$

Cette relation de préférence est dite *consistante* si et seulement si elle est irréflexive. Si tel est le cas, elle devient ainsi une relation d'ordre partielle stricte. Des travaux existent pour vérifier si une cp-théorie est consistante [Wilson 2004]. Nous supposons pour la suite du document que nos relations le sont.

10.1.2 Opérateurs de Préférences

Les opérateurs de préférences calculent les données préférées par rapport à une cp-théorie de référence. Chaque utilisateur donne au système ses préférences sous forme d'une cp-théorie Γ qui constitue ainsi une sorte de *profil utilisateur* qui est évolutif. Concrètement, cette solution va permettre le support de requêtes permettant de retourner les données « les plus préférées » et « les top-k » par l'intégration de deux opérateurs: **Best** et **KBest**.

L'opérateur **Best** sélectionne, dans une relation temporelle ou non temporelle, les n-uplets qui correspondent le mieux à l'ensemble des préférences représentées dans Γ. D'un point de vue mathématique, il s'agit des n-uplets qui ne sont pas dominés dans la hiérarchie des préférences.

Définition 10.1.3 **Best**

Soit R une relation temporelle et Γ une cp-théorie sur le schéma de R.
$\mathbf{Best}(R) : b \mapsto \{u \in R(b) \mid \nexists v \in r(t) \text{ tel que } v \succ_\Gamma u\}$

L'opérateur **KBest** sélectionne les k n-uplets qui correspondent au mieux à la hiérarchie des préférences énoncées dans Γ. Intuitivement $\mathbf{KBest}_k(R)(b)$ retourne l'ensemble des k n-uplets de $R(b)$ qui sont le moins dominés par d'autres n-uplets selon la hiérarchie de préférence. Pour définir sa sémantique, nous introduisons la notion de *niveau* d'un n-uplet (noté $l(u)$) par rapport à une cp-théorie Γ dans la définition 10.1.4. Le niveau reflète l'écart entre ce n-uplet et ceux qui répondent complètement aux préférences de l'utilisateur.

Définition 10.1.4 **Niveau**

Soit R une relation temporelle et Γ une cp-théorie sur le schéma de R.
Le *niveau* de u, $l(u)$, *par rapport à* Γ au *batch* b est défini de manière inductive comme suit:

- Si $\nexists u' \in R(b)$ tel que $u' \succ_\Gamma u$, alors $l(u) = 0$.
- sinon $l(u) = 1 + max\{l(u') \mid u' \succ_\Gamma u\}$

Il est possible de montrer que si $u \succ u'$ alors $l(u) < l(u')$. La réciproque n'est pas vraie. La sémantique de l'opérateur $\mathbf{KBest}_k$ est présentée dans la définition 10.1.5.

10.1.3 Exemple de requête

Supposons l'existence d'un flux V agrégeant l'ensemble des informations *volatiles* d'un système observé. Son schéma est le suivant : V(id, concept, type, volatile, value,

Définition 10.1.5 **KBest**

Soit R une relation temporelle et Γ une cp-théorie sur le schéma de R. Pour un batch donné b, $\mathbf{KBest}_k(R)(b)$ est l'ensemble des k n-uplets $\in R(b)$ ayant le plus petit niveau. L'ordre positionnel est utilisé pour départager des n-uplets de même niveau.

τ). La description de ce schéma est l'expression suivante : l'entité de nature *concept*, d'identifiant *id* et de catégorie *type* a émis la donnée de type *volatile* et de valeur *value* au temps τ. La formation de ce flux est triviale selon les principes développés en section 8.2 et n'est pas détaillé.

Nous souhaitons utiliser ce flux en tant que flux d'information à afficher à l'utilisateur. Afin de rapidement analyser l'activité du réseau, il souhaite obtenir une sélection des meilleures informations le concernant.

Soit Γ la cp-théorie comportant les préférences contextuelles suivantes :

– Pour les données *cpu*, je préfère les valeurs ≥ 90 à identifiants égaux.

$$\varphi_1 : \text{volatile} = \text{cpu} \mapsto (\text{value} \geq 90) \succ (\text{value} < 90)\ [\text{id}]$$

– Je préfère les données de *status* plutôt que de *cpu* ou *mémoire* à *timestamps* égaux.

$$\varphi_2 : (\text{volatile} = \text{status}) \succ (\text{volatile} = \text{cpu} \vee \text{volatile} = \text{mem})\ [\tau]$$

– Pour les interfaces réseaux, je préfère les types *wifi* plutôt qu'*ethernet*.

$$\varphi_3 : \text{concept} = \text{interface} \mapsto (\text{type} = \text{wifi}) \succ (\text{type} = \text{ethernet})$$

La figure 10.1 représente un ensemble de données pour V. Si nous appliquons la relation d'ordre sur cet ensemble de n-uplet, nous pouvons remarquer que par la règle φ_1, nous obtenons que $s_1 > s_4$. Par φ_2 nous obtenons que $s_2 > s_1$. Et enfin, nous avons $s_3 > s_5$ par φ_3. La relation d'ordre $\succ_\Gamma$ est transitive, ainsi $s_2 > s_4$. Le graphe de la figure représente ces relations d'ordres.

V	id	concept	type	volatile	value	τ
s_1	1	equipement	gateway	cpu	95	1
s_2	2	equipement	stb	status	1	1
s_3	3	interface	wifi	bws	150	3
s_4	1	equipement	gateway	cpu	20	4
s_5	5	interface	ethernet	bwr	1200	7

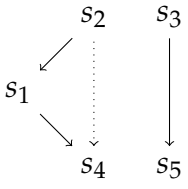


FIGURE 10.1 – Un jeu de données pour V et son graphe de préférence correspondant

En supposant que nous appliquons une fenêtre sur V et que nous obtenons les n-uplets présentés au batch b. Nous obtenons les résultats suivants :

– $\mathbf{Best}(R)(b) = \{s_2, s_3\}$, car ces deux n-uplets ne sont pas dominés.

– $\mathbf{KBest}_3(R)(b) = \{s_1, s_2, s_3\}$ par sélection du niveau 0 et 1.

– $\mathbf{KBest}_4(R)(b) = \{s_1, s_2, s_3, s_4\}$, car s_5 est arrivé après s_4 et l'ordre positionnel domine à niveau équivalent.

Dans la pratique nous pourrions appliquer cet opérateur sur tout type de fenêtre. Pour revenir à notre application, en supposant que V est correctement formé, nous pouvons déployer une requête continue qui peut alimenter l'interface de l'utilisateur avec : les 25 événements les plus importants du dernier jour évaluée toutes les minutes. Cette requête a l'expression $\mathbf{KBest}_{25}(V[T\ 1j\ 1min])$.

Nous avons désormais présenté les deux opérateurs nous permettant la personnalisation de résultat. Dans la section suivante, nous détaillons leur implémentation.

10.2 Mise en œuvre

Cette section présente les algorithmes que nous avons développés pour les opérateurs **Best** et **KBest**. Ces opérateurs nécessitent de connaître la hiérarchie des n-uplets déduite d'un ensemble de préférences. Cette hiérarchie est déduite du *graphe de préférence GP*. Nous présentons ci-après deux algorithmes pour la gestion du graphe : l'un est calculé sur l'état courant de la relation temporelle d'entrée, l'autre travaille sur les différences de la relation temporelle de manière incrémentale.

10.2.1 Le graphe de préférence

L'ordre de préférence entre deux n-uplets t_1 et t_2 selon une règle φ est construit à partir d'une fonction $Compare(t_1, t_2, \varphi_k)$ qui retourne $\emptyset$ si t_1 et t_2 sont incomparables, 1 si t_1 est préféré à t_2 et -1 sinon. L'algorithme 2 étend la comparaison à une théorie Γ. Cette dernière utilise un ensemble de règles pour calculer l'ordre de préférence, mais ne calcule pas la clôture transitive.

Graphe de préférence : Les opérateurs **Best** et **KBest** sont appliqués sur un ensemble de n-uplets TS et ont besoin du graphe de préférence relatif à TS. Pour son implémentation, nous avons adopté la structure $Graph(Next, Prec, Src)$ défini comme suit: $Next$ associe chaque n-uplet à la liste des n-uplets **qu'il domine**. $Prec$ associe chaque n-uplet à la liste des n-uplets **qui le dominent**. Src est l'ensemble des n-uplets non dominés, représentant la source du graphe. Afin de fournir de bonnes performances, les implémentations de ces structures sont des *hash-sets* ou *hash-maps*.

La construction et la mise à jour du graphe utilisent des méthodes *Graph.Insert* et *Graph.Delete* qui fonctionnent comme suit. Pour insérer un n-uplet dans un graphe, la méthode *Graph.Insert* itère sur les nœuds du graphe pour mettre à jour *Next*, *Prec* et l'ensemble *Src*. Comme le coût de l'insertion et de suppression dans une structure hachée peut être considéré comme $\mathcal{O}(1)$, le coût global de l'insertion d'un n-uplet dans le graphe est de $\mathcal{O}(|G|)$. Pour la suppression d'un n-uplet s du graphe, la méthode *Graph.Delete* itère sur les nœuds connectés à s. Le coût est de $\mathcal{O}(\deg(s))$.

Pour une théorie de préférences Γ donnée et un ensemble de n-uplets TS, la construction du graphe de préférence complet est réalisée par l'algorithme 3.

10.2.2 Calcul de Best et KBest

Par définition, le graphe inclut l'ensemble *Src* qui contient les n-uplets les plus préférés. L'ensemble *Src* est le résultat de l'opérateur *Best*. Toutefois, l'algorithme peut-être optimisé en évitant de construire complètement *GP*. La méthode *Graphe.Insert*

peut être optimisée pour tenir compte de cela. La complexité est alors réduite à $\mathcal{O}(|Src|)$. La complexité de $Best(R)(b)$ devient $\mathcal{O}(NS)$ où $N = |R(b)|$ et $S = |Best(R)(t)|$.

L'algorithme principal utilisé pour calculer *KBest* à partir du *GP* est un tri topologique de Kahn limité aux k premiers résultats. Voir l'algorithme 1.

10.2.3 Évaluation incrémentale du GP

Cette section présente un algorithme pour le calcul incrémental de *GP*. Le fait que les requêtes avec préférences sur les flux s'expriment sur des séquences de fenêtres motive cette méthode. Il est nécessaire de construire le *GP* pour l'ensemble des n-uplets de la fenêtre *courante*. Comme deux fenêtres consécutives peuvent se superposer, le nouveau *GP* peut être construit par mise à jour incrémentale du graphe *courant*.

Supposons que δ_R^- contienne les n-uplets *sortants* de la fenêtre et δ_R^+ contienne ceux qui *rentrent* dans la nouvelle fenêtre. Il n'y a pas d'intersection entre ces deux ensembles. Ces ensembles sont utilisés par l'algorithme 4 pour construire le *GP* de la nouvelle fenêtre à partir du *GP* de la précédente.

Il est important de noter que l'approche incrémentale est intéressante si la différence entre deux graphes successifs est faible comparée à la taille totale du graphe. Dans un tel cas, une grande proportion du *GP* est réutilisée. Dans le cas contraire, la création du *GP de zéro* est meilleure. Le tableau 10.1 résume l'ensemble des complexités algorithmiques.

10.3 Intégration et Expérimentations

Dans cette section, nous présentons l'intégration de l'implémentation des opérateurs **Best/KBest** dans notre système Astronef-Asteroid. Tout d'abord, nous détaillons les règles qu'il est nécessaire de fournir à Astronef pour prendre en compte ces opérateurs. Puis, nous expérimentons les versions statiques et incrémentales de nos algorithmes afin de déterminer une *heuristique* permettant de sélectionner le meilleur.

10.3.1 Intégration dans Astronef

Nous avons désormais des définitions Astral correctement posées. Nous pouvons maintenant intégrer ces opérateurs à Astronef. En premier lieu, nous avons conçu un composant opérateur capable de calculer les sémantiques de **Best** et **KBest** de manière statique en calculant à partir de l'état de $R(b)$. Ce composant prend évidemment en paramètre k qui peut être égal à -1 pour indiquer une sémantique de **Best**.

L'intégration du composant se fait en deux parties. Tout d'abord, il est nécessaire d'enregistrer le composant en tant qu'opérateur [1]. Ensuite, il faut renseigner ses définitions Astral dans le moteur de règle. Dans notre cas, quatre règles sont suffisantes :

```
1 sugar([best,Config,C],[kbest,NewConfig,C]):-
2     map_put(Config,['k',-1],NewConfig), !. % best = kbest avec k=-1
3
```

1. Cette opération est faite au moment de l'installation de l'extension dans Astronef.

Algorithme 1 : Calcule KBest$(R)(t)$

Data : La structure GP, k le nombre de n-uplet demandé
Res ← **new TreeSet**() `// Ensemble ordonné`
if $k < |Src|$ **then**
  `// Src contient plus de` k `n-uplets`
  $N \leftarrow |Src| - k$
  `// L'ordre position de Src est utilisé`
  foreach $s \in Src$ **do**
    `// pour garder les` k `n-uplets les plus récents`
    if $N = 0$ **then** Res.add(s)
    else $N \leftarrow N - 1$
  return *Res*
NextLvl ← *Src*; *id* ← 0
PrecCount ← **new HashMap**()
while *id* $< k$ ***and*** *id* $< |Src| +$*Prec.count()* **do**
  `// Buffer contient les n-uplets du même niveau`
  if *Buffer* = ∅ **then**
    foreach $t \in$*NextLvl* **do**
      Buffer.push(t)
    NextLvl.clear()
  $t \leftarrow$ Buffer.pop()
  foreach $s \in$ *Next.get*(t) **do**
    `// Pour tout n-uplet dominé par` t
    $n \leftarrow$PrecCount.get(s)
    if $n =$***null*** **then**
      $n =$Prec.get(s).size()
    if $n = 1$ **then**
      NextLvl.add(s)
      `//` s `fait parti du prochain niveau`
    else PrecCount.put$(s, n-1)$
    `// Il n'y a plus de nœud dans cet ensemble`
  Res.add(t)
return *Res*

Algorithme 2 : ComparT(t_1, t_2, Γ)

Data : $\Gamma = \{\varphi_1, ..., \varphi_k\}$ une théorie
Result : $\{1, -1, \emptyset\}$,
  resp. $\{t_1 >_\Gamma t_2, t_1 <_\Gamma t_2,$ inc.$\}$
foreach $\varphi_k \in \Gamma$ **do**
  $r \leftarrow Compare(t_1, t_2, \varphi_k)$
  if $r \neq \emptyset$ **then return** r
return $\emptyset$

Algorithme 3 : Créer GP

Input : TS un ensemble de n-uplets
Data : La structure de GP
foreach $s \in TS$ **do** *Graph*.insert(s)

Algorithme 4 : GP incrémental

Input : $\delta_R^-(t,i)$ and $\delta_R^+(t,i)$
Data : La structure du GP
foreach $s \in \delta_R^-(t,i)$ **do**
Graph.Delete(s)
foreach $s \in \delta_R^+(t,i)$ **do**
Graph.Insert(s)

	Créer GP	GP Incrémental
Best	$\mathcal{O}(N.S)$	$\mathcal{O}(\Delta.N)$
KBest	$\mathcal{O}(N^2)$	$\mathcal{O}((\Delta + k)N)$

TABLE 10.1 – Complexité de Best/K-Best

```
4 typerules([kbest,_,_,[T]],T):- relation(T), !.
5 attribrules([kbest,_,_,[A]], A):- !.
6
7 implrules([kbest,_,_,_], "KBest"):- !.
```

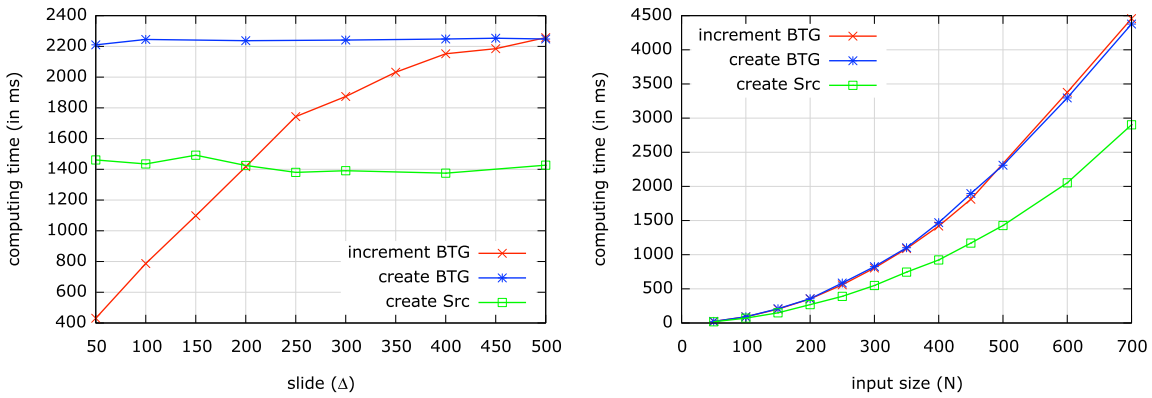
Une fois ce code renseigné dans le service *KnowledgeBase* d'Astronef (présenté en 6.5), nous pouvons librement utiliser les opérateurs **Best** et **KBest** sur toutes relations temporelles. Il est intéressant de voir que si le SGBD supporte le langage *CPref-SQL*, nous pouvons pousser les opérateurs **Best** et **KBest** sur le SGBD via une règle Asteroid.

10.3.2 Expérimentations pour la sélection du plan de requête

Nous avons à notre disposition deux composants capables de calculer de manière incrémentale ou non les opérateurs de préférences contextuelles. Dans cette section, nous expérimentons pour déterminer les cas où il est préférable de sélectionner l'un ou l'autre choix.

L'expérience a été réalisée dans un cadre applicatif financier. Ainsi, 30,000 n-uplets ont été collectés depuis des flux de cours d'actions [2]. Les préférences utilisées sont de la même complexité que celles présentées en section 10.2, les détails sont présents dans notre papier [Petit 2012a]. Soit S un flux, la requête que nous analysons est la suivante :

$$\mathbf{KBest}_k(S[N \text{ slide } \Delta])$$



(a) En variant Δ sur KBest (pour $N = 500$)

(b) En variant N sur KBest (pour $\Delta = N$)

FIGURE 10.2 – Temps de calcul de *Créer GP*, *GP Incrémental* et de la maintenance de *Src*

Résultats : Les expérimentations montrent que le temps d'évaluation de **KBest** est dominé par la construction-mise à jour du *GP*. Nous avons aussi observé que la structure du *GP* d'une fenêtre à l'autre pouvait changer. La profondeur maximale du graphe varie de 2 à 6 et le nombre de n-uplets non-dominés varie de 1 à N. Les grands changements de structures correspondent aux pires cas de l'algorithme incrémental.

2. Fournit par le service Dukascopy's Data Export. Disponible à http://www.dukascopy.com/swiss/english/data_feed/csv_data_export

La figure 10.2(a) montre le temps de calcul des deux algorithmes du *GP* : *créer* et *incrémental*. Elle indique aussi le temps nécessaire au calcul réduit de la maintenance de *Src*, qui est directement utilisée pour l'opérateur **Best**.

Durant les expérimentations, nous avons utilisé plusieurs tailles de fenêtres N et de taux Δ. Nous remarquons que les changements de taux de mise à jour n'impactent pas le temps de création du *GP*, tandis que l'algorithme incrémental obtient un gain de 6 pour un ration $N/\Delta = 10$. Étonnement, les deux algorithmes du *GP* se comportent de la même façon lorsque $\Delta \sim N$, ce qui correspond au cas où il n'y aurait pas ou peu d'intersections entre les fenêtres successives. Cela indique que dans la version incrémentale, les suppressions dans le *BTG* prennent peu de temps comparé aux insertions. Ceci est notamment dû au fait que l'insertion nécessite l'évaluation des comparaisons (coûteuses dans notre cadre expérimental). Le coût de ces suppressions peut aussi être augmenté dans le cas où le *GP* est un graphe fortement connexe, ce qui est difficile à trouver dans la pratique.

La variation de la taille de la fenêtre (figure 10.2(b) avec $\Delta = N$) montre que le comportement n'est pas impacté par le nombre de n-uplets impliqués. Comme prévu, l'évolution semble quadratique selon N et l'approche incrémentale suit strictement la performance de l'algorithme de création.

Nous pouvons désormais intégrer notre composant incrémental à l'intérieur d'Astronef par la création de la règle suivante :

```
1 implrules([kbest,B,[C]], "DynamicKBest"):-
2     map_get(B, "incremental", "true", "true"), % Mode incremental par defaut
3     dynamicoperator(C), !. % Si le noeud fils est aussi incremental
```

Pour plus de complétude, nous pourrions établir une règle disant que si nous sommes dans le cas $k = -1$ (i.e. **Best**) et que le ratio N/Δ de la description de fenêtre, s'il y en a une, est inférieur à 2 : alors nous devons utiliser l'opérateur **Best** non-incrémental.

10.4 Conclusion

Pour démontrer la flexibilité de l'architecture, et pour répondre au besoin d'introduire les points de vues utilisateurs dans l'observation, nous avons intégré un nouvel opérateur de préférences dans notre solution. Cet opérateur nous a permis de sélectionner les données les plus intéressantes selon le profil de l'utilisateur. Cette intégration s'est fait via la spécification de cinq courtes règles ce qui valide que l'ajout de composants soit simple.

Dans les quelques approches qui existent dans la littérature [Kontaki 2012, Morse 2007, Mouratidis 2006], les préférences sont appliquées uniquement sur des fenêtres glissantes. Or, dans notre cas, grâce à l'expressivité de notre système d'interrogation, nous pouvons appliquer cette opération sur tout type de données, qu'elles proviennent d'une base de données relationnelle ou de flux.

L'évaluation de performances nous a démontré que le choix incrémental est en général meilleur que sa version statique. Toutefois, pour le cas de l'opérateur **Best**, il peut être préférable de choisir la version statique si les variabilités de la relation temporelles sont trop fortes. De travaux futurs pourraient s'intéresser à la découverte de motifs plus précis pour affiner ce choix.

Ce travail a été présenté dans le papier [Petit 2012a], ainsi que dans les conférences nationales avec les papiers [Roncancio 2012] et [Petit 2012b].

« Dear Princess Celestia,
Sometimes you can feel like what you have to offer is too little to make a difference, but today, I learned that everypony's contribution is important, no matter how small. »

Fluttershy

11 Conclusion et perspectives

La popularisation de la technologie a permis d'implanter des dispositifs et des applications de plus en plus développés à la portée d'utilisateurs non experts. Comme les dispositifs se diversifient, des systèmes complexes, distribués et ubiquitaires se forment. Or, en général, ces utilisateurs n'ont pas les compétences pour maîtriser ces systèmes en cas de dysfonctionnements. Nous nous intéressons à pouvoir observer les données de ces systèmes à distance pour aider à les comprendre et les diagnostiquer.

Les objectifs plus détaillés de la thèse sont rappelés en section 11.1. La section 11.2 présente l'ensemble des contributions que nous avons établies. Enfin, la section 11.3 présente les perspectives de recherches.

11.1 Rappel des objectifs

Actuellement, il existe plusieurs approches capables d'établir un tel système d'observation. Chaque approche possède un avantage propre : la collecte de données par les systèmes d'administration, la gestion de la sémantique des données par l'informatique contextuelle, l'analyse de grands volumes par les entrepôts de données, et le traitement des données en temps réel par la gestion de flux de données. Notre orientation s'est faite vers ce dernier domaine, car elle est la seule à gérer de façon déclarative le traitement de données en requête continue. D'une manière plus globale, cette thèse a pour but d'enrichir nos connaissances sur la gestion de données issues de systèmes dynamiques hétérogènes.

Nous souhaitons concevoir une solution générique d'observation de système. Nous souhaitons résoudre deux points importants : gérer l'hétérogénéité des données issues du système observé ; et permettre le système d'observation à s'adapter facilement.

Système observé : hétérogénéité des données

Nous souhaitons pouvoir observer différents types de systèmes. Cela implique que la description des systèmes en terme de schéma conceptuel de données est hétérogène. Les données qui émanent du système sont de tous types et représentent différents fragments du système. Le schéma de ces données diffère en fonction des sources de données. Ainsi, il est nécessaire d'avoir une intégration des données et une capacité de traitement puissante.

Le système en observation est dynamique et ses données évoluent au fur et à mesure du temps. Il existe deux catégories de données : les données persistantes et les données temps réel. Ces dernières peuvent être stockées pour analyse a posteriori, traitées en temps réel, croisées avec des données passées ou consolidées. Ainsi, il est important de permettre d'interroger les différentes données.

Système d'observation : adaptabilité aux besoins

Le système d'observation doit être capable de s'adapter efficacement à son environnement. Afin de fournir une solution flexible, il est préférable d'avoir un langage déclaratif. Ce langage doit toutefois avoir un pouvoir d'expression permettant de gérer l'hétérogénéité du dynamisme des données.

De plus, chaque utilisateur possède son interprétation du système, et nous devons pouvoir faciliter cette personnalisation. Il est également important que le système d'observation soit extensible. Par exemple, le support de l'ajout de fonctions tierces pour les utilisateurs experts permet de fournir des capacités d'interrogation plus spécialisées.

Dans cette thèse, nous avons principalement mis en avant la gestion de l'évolution des données et l'adaptabilité grâce aux travaux sur les flux de données. La gestion de l'hétérogénéité des schémas conceptuels est assurée par la capacité à interroger aussi bien les données temps réel que persistantes.

11.2 Contributions de cette thèse

La contribution de cette thèse sur l'observation de systèmes se découpe en trois parties. Premièrement, nous avons proposé **une algèbre** de gestion des requêtes continues et instantanées sur flux et relations. Deuxièmement, nous avons proposé un **intergiciel extensible** capable d'évaluer des requêtes exprimées grâce à l'algèbre. Enfin, nous avons présenté l'extension de cet intergiciel pour intégrer les supports persistants dans l'expressivité des requêtes.

11.2.1 Langage de requête formel pour une interrogation généralisée

Dans la littérature, les langages de requêtes existants dans le cadre de la gestion de flux de données ont montré des lacunes en terme de clarté sémantique. Deux requêtes

exécutées sur deux systèmes peuvent donner des interprétations différentes. Notre approche a été de redéfinir une algèbre de gestion des flux de données avec comme objectif d'être indépendant du système d'implémentation pour avoir des expressions de requêtes claires.

Le langage algébrique *Astral* présenté dans le chapitre 4 est un dérivé de l'algèbre relationnelle pour les flux de données. Ainsi, les connaissances concernant la manipulation, mais aussi les équivalences de requêtes, du modèle relationnel peuvent être réutilisées. Comme beaucoup d'autres langages de l'état de l'art, cette algèbre sépare les notions de flux et de relation temporelles. Cette approche permet de clarifier les requêtes en interdisant par exemple les opérations flux→flux sans passer par le domaine relationnel via un opérateur de fenêtre.

Astral possède trois avantages : des fondations solides, une expressivité accrue et une intégration de requêtes instantanées et continues. En effet, les définitions fondamentales d'Astral formalisent les notions d'ordre et d'équivalence de requêtes. Ces fondations ont permis la spécification d'opérateurs non ambigus ainsi que la preuve de résultats non triviaux comme l'asymétrie de la jointure, ou l'équivalence de requêtes à temps de départs différents (la transposabilité).

L'expressivité d'Astral permet de rassembler les propositions actuelles en une algèbre intégrée. Par exemple, une grand quantité de sémantiques d'exécution de l'opérateur de séquence de fenêtres sont possibles dans notre modèle. De plus, la formalisation de l'opérateur de manipulation temporelle permettant de sélectionner un état passé d'une relation temporelle permet l'intégration des requêtes continues et instantanées. Cette intégration est notre pierre angulaire pour pouvoir gérer l'hétérogénéité de dynamisme des données du système. La validation de ces points forts fait l'objet du chapitre 5.

11.2.2 Intergiciel d'évaluation de requête extensible

Astral permet d'écrire toutes requêtes sur des flux ou relations. L'intergiciel *Astronef*, présenté en chapitre 6 met en œuvre une telle évaluation. Sa structure interne se base sur les architectures à composants orientés services. Cette approche permet d'ajouter, enlever, reconfigurer des composants à tout moment. Cela rend l'approche extensible en terme d'architecture.

Astral est un langage indépendant du système d'implémentation et Astronef définit ses composants en terme algébrique. En effet, chaque composant opérateur doit définir son équivalent algébrique en fonction de sa configuration. Ainsi, un moteur de règle permet de transformer une expression algébrique en plan d'exécution.

Cette transformation se fait en deux étapes, comme dans les SGBD. Tout d'abord, l'expression est réécrite afin de réduire la taille des résultats intermédiaires. Cette réécriture utilise les résultats d'équivalences de requêtes données par les preuves faites avec Astral. À partir de la nouvelle expression, un ensemble de règles permet de sélectionner un plan de requête efficace par l'utilisation d'heuristiques (détection de motifs). Le chapitre 9 présente des expérimentations qui valident cette construction de plan de requête.

11.2.3 Intégration d'un support persistant à l'intergiciel

Le langage Astral permet d'intégrer les requêtes continues et requêtes instantanées. Asteroid est une extension à Astronef capable de coupler un SGBD aux traitements des requêtes continues.

L'intégration des données persistantes dans le cadre de l'observation de système est importante. En effet, nous avons remarqué que les données persistantes et les données temps réel ont des motifs d'évolutions différents (statique, stable, périodique et imprévisible). Ces dynamiques permettent de conditionner la place de la donnée dans le schéma de la base de données ainsi que les traitements associés. Ainsi, le schéma est séparé en deux sections, les données représentant le système (schéma descriptif, contenant des données stables et statiques), et les archives de flux (schéma historique, contenants des données périodiques et imprévisibles).

Nous avons intégré ces notions dans Astronef par le développement de plusieurs composants. Les puits de persistances insèrent des nouvelles données dans le schéma descriptif (composants dédiés) ou dans les historiques (composants génériques). Nous avons décrit le comportement d'une source capable de représenter toute requête relationnelle sous forme de relation temporelle Astronef. Cette relation temporelle subit des mises à jour régulières suivant un mode de rafraichissement spécifique (périodique, événementiel, *trigger*). Cette source est configurable et grâce à la formalisation Astral ainsi qu'aux règles Astronef, nous pouvons effectuer le placement du plus grand nombre d'opérateurs sur le SGBD pour en exploiter ses capacités. Enfin, un dernier composant permet d'effectuer par un SGBD une jointure entre une relation temporelle et une relation classique. Dans le cadre de requêtes hybrides, le choix du meilleur plan d'exécution a été présenté dans la section 9.3.

L'ensemble du système d'observation a été mis en œuvre et expérimenté sur le réseau local domestique afin d'en explorer son expressivité. L'instance de ce système, baptisé *DomVision*, a été détaillée dans le chapitre 8. Nous démontrons notamment que nous pouvons effectuer des intégrations de données hétérogènes en termes de schéma comme en terme de dynamique. Nous arrivons de plus à former des requêtes continues capables d'utiliser des données temps réel ainsi que des historiques.

11.2.4 Gestion des préférences

Face au large panel d'utilisateurs, le système va être surveillé par des personnes dont les intérêts peuvent diverger. Afin de pouvoir gérer les points de vue de chacun, nous introduisons un moyen de personnaliser les résultats d'une requête dans le chapitre 10. Nous intégrons ainsi deux nouveaux opérateurs **Best** et **KBest** capables d'effectuer cette tâche.

Chaque utilisateur exprime ses préférences contextuelles dans un profil et ces opérateurs adapteront les résultats à ce profil. Comme notre solution d'observation interroge des données venant de flux ou de relations persistantes, alors nous sommes capables de gérer les préférences sur ces deux supports de manière intégrée. Jusqu'à présent les travaux permettant la personnalisation sur les flux étaient limités à des opérateurs particuliers (fenêtres glissantes). Notre approche permet une généralisation de la gestion de préférences sur les données issues d'un système hétérogène et

dynamique.

Deux implémentations permettent de calculer ces opérateurs en exploitant ou non l'évaluation incrémentale des données. Enfin, une évaluation de performances permet de montrer les conditions où l'évaluation incrémentale est plus efficace. Cet ajout de fonctionnalité démontre de plus l'extensibilité d'Astronef à intégrer un nouvel opérateur. Nous avons non seulement pu intégrer une nouvelle fonctionnalité, mais aussi nous avons pu donner des règles à l'optimiseur pour construire un plan de requête efficace.

11.3 Perspectives de recherches

Avec l'avènement des mouvements tels que le *BigData* ou l'*Internet des objets*, la gestion des données a subi un renouveau depuis quelques années. Nous avons pu apporter notre contribution toutefois, il reste de nombreux défis scientifiques à relever en continuité de notre travail. Nous avons identifié quatre pans de recherches à explorer sur le court et le long terme.

11.3.1 De la représentation des données d'un système

Dans notre analyse de l'état de l'art, nous avons vu que peu d'approches permettaient de *modéliser* un ensemble de flux de données. Dans le chapitre 7, nous avons présenté le schéma physique d'Asteroid. Ce schéma est découpé en deux sections. Tout d'abord, le schéma descriptif représentant par un schéma normalisé le modèle du système. Ensuite, le schéma historique permet d'archiver une liste d'historiques. Cette approche est avant tout dirigée par l'implémentation.

En effet, cette modélisation nous permet d'ajouter des historiques de flux à volonté, toutefois, le lien entre les flux et les concepts est établi par un identifiant unique. De façon similaire, nous avons considéré que les flux de données étaient des flux indépendants, sans liens entre eux a priori.

Or, dans la conception d'une base de données, le développeur identifie ses données, établit des dépendances fonctionnelles, et construit un schéma respectant ces dépendances (entité relation ou modèle *UML*). Dans notre cas, nous n'avons pu opérer de la sorte pour deux raisons : (1) la notion de dépendance fonctionnelle n'a pas été définie pour les flux (2) souvent, le schéma des flux ne peut pas être choisi, il est préexistant et nous subissons ce que le système nous fournit.

Dans Asteroid, nous avons séparé la gestion des données volatiles de la description du système. Cette séparation s'est faite à cause des dynamiques différentes. Est-il toutefois possible de créer une modélisation pour gérer ces classes de dynamiques ? Il serait ainsi possible d'intégrer des sources ayant des classes de dynamiques conflictuelles. Par exemple, une donnée de lieu peut être statique sur certains objets et volatile sur d'autres. Cet exemple produirait un historique de flux et une entrée dans le schéma descriptif du système qu'il faudrait gérer à la main lors de l'interrogation du système.

En dehors des approches de base de données classiques, le domaine de la gestion de contexte, comme nous l'avons montré, fournit plusieurs outils pour effectuer des représentations de données hétérogènes. Être capable de formaliser un contexte uni-

forme permettrait d'utiliser notre approche dans des applications plus larges (*context-aware softwares*, *service level checking*,...).

11.3.2 De l'expressivité des langages de requêtes

Astral est un langage algébrique capable d'interroger, de manière instantanée ou continue, des flux ou des relations (temporelles). Ce langage est très expressif et permet de manipuler de manière claire nos entités comme nous l'avons démontré dans le chapitre 5.

Néanmoins, nous n'avons en l'état que très peu d'outils pour analyser l'expressivité d'un langage de requête dans le cadre des flux en dehors de la comparaison avec les outils actuels. Pour l'algèbre relationnelle, nous savons sa limitation grâce à la logique du premier ordre (en excluant la récursion). Dans le cas d'Astral, nous n'avons pas de moyens en l'état pour quantifier grâce à des opérateurs logiques la classe des résultats possibles.

Une fois cette expressivité connue, de la même façon que le *SQL* a été défini pour l'algèbre relationnelle, il devient possible d'établir un langage déclaratif aussi expressif qu'Astral. L'expression en termes déclaratifs permet aux experts métiers du système, non-spécialiste de la gestion de flux de données, de manipuler aisément les données. Sachant qu'une mise en œuvre efficace est garantie à partir de l'expression algébrique d'une requête, il nous faut désormais faciliter l'accès à ces capacités.

11.3.3 De la performance de l'observation

Nous avons présenté avec Astronef, dans le chapitre 6, une méthode générique, automatique et extensible pour évaluer de manière efficace les requêtes continues. Nous avons montré son efficacité dans le chapitre 9. Toutefois, les limites de cette approche peuvent être rapidement atteintes. En effet, lorsque deux règles sont applicables, alors l'une est privilégiée à l'autre.

De même, lors de la jointure de plusieurs relations, l'optimisation de requête classique de SGBD a montré que l'approche par règle n'est pas suffisante. La recherche de solutions optimales par algorithmes de programmation dynamique permet en général la résolution de ce problème. Toutefois, il est nécessaire de prédire les tailles des relations. Ces approches sont difficiles et deviennent importantes lorsque les expressions de requêtes sont générées grâce à un langage déclaratif.

Afin d'envisager l'observation de plusieurs millions d'entités, il est nécessaire d'établir des solutions de stratégies globales. Nous avons établi dans cette thèse des règles capables d'optimiser une requête isolée. Dans le cadre où des centaines de requêtes sont établies sur plusieurs nœuds distribués, une optimisation locale ne suffit plus. Ainsi, il est nécessaire de concevoir des algorithmes de partage de requête et de calcul à l'exécution de nouveaux plans de requêtes distribués. Notre formalisation de l'équivalence de requête transposée nous permet d'envisager le partage automatique de plan. Toutefois, de nouvelles approches sont à envisager pour pouvoir effectuer un passage à l'échelle des infrastructures de gestions de flux de données.

11.3.4 De l'analyse et la compréhension d'un système

Dans cette thèse, nous avons conçu une méthodologie générique pour gérer les données d'un système d'observation. Cette gestion avait pour but de mieux comprendre le système et éventuellement en cas de problème, de pouvoir le diagnostiquer. Nous avons pu mettre en pratique cette solution sur un problème en production chez *Orange France*.

Il a été observé sur quelques centaines de *Livebox* en France un problème récurrent de coupure de service VoIP pour raisons inconnues. Nous avons été contactés pour aider à la résolution de ce problème. Les experts métiers nous ont donné accès à l'ensemble des données de configuration de l'appareil (un accès *pull* et un accès événementiel), soit environ 10 000 paramètres. Nous avons installé notre prototype sur le réseau d'une *Livebox* présentant le problème pour tracer les événements ainsi que les changements des données de configuration, collectées toutes les deux minutes (minimum supportable).

Nous avons pu effectuer plusieurs analyses a posteriori grâce au stockage des données sur Asteroid. Nous n'avons toutefois pas pu mettre en évidence le problème. Les conséquences du problème ont pu être observées en remarquant que la VoIP devenait inactive. Toutefois, il n'a pas été possible de **comprendre** le problème, ni en trouver la **cause**. De cette expérience, trois conclusions sont possibles : l'observation du système ne permet pas d'identifier le problème, les requêtes choisies ne mettent pas en évidence le problème, ou les données qui auraient permis de conclure n'étaient pas accessibles. À l'heure actuelle, il n'est pas possible de trancher. Toutefois, nous avons exploité les capacités que pouvaient nous fournir nos sources de données, ce qui a permis l'analyse d'autres données concernant la qualité de la ligne ADSL au cours du temps par exemple.

La compréhension d'un système est un domaine à part entière. Ce domaine permet de fournir des méthodes et des outils, issus des analyses statistiques et des intelligences artificielles multiagents, pour mieux appréhender la complexité d'un système. Nous avons fourni un outil pour observer les données d'un système. Cet outil peut être une base sur laquelle peuvent s'appuyer d'autres approches.

Bibliographie

[Abadi 2003] Daniel Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul et Stan Zdonik. *Aurora: a new model and architecture for data stream management*. The VLDB Journal - The International Journal on Very Large Data Bases, vol. 12, no. 2, pages 120–139, 2003. *5 citations pages 45, 51, 53, 55, et 94*

[Abadi 2005] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska et Uğur Çetintemel. *The Design of the Borealis Stream Processing Engine*. Conference on Innovative Data Systems Research (CIDR), Janvier 2005. *cité page 53*

[Abdullah 2009] Ahsan Abdullah. *Analysis of mealybug incidence on the cotton crop using ADSS-OLAP (Online Analytical Processing) tool*. Computers and Electronics in Agriculture, vol. 69, no. 1, pages 59–72, 2009. *cité page 36*

[Aberer 2007] Karl Aberer, Manfred Hauswirth et Ali Salehi. *Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks*. Dans International Conference on Mobile Data Management (MDM), pages 198–205, 2007. *cité page 53*

[Akyildiz 2002] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam et E. Cayirci. *Wireless sensor networks: a survey*. Computer Networks, vol. 38, no. 4, pages 393–422, Mars 2002. *cité page 11*

[Angles 2008] Renzo Angles et Claudio Gutierrez. *The expressive power of SPARQL*. Dans International Semantic Web Conference (ISWC), *LNCS 5318*, pages 114–129. Springer, 2008. *cité page 31*

[Arasu 2002] Arvind Arasu, Shivnath Babu et Jennifer Widom. *An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations*. Rapport technique 2002-57, Stanford University, 2002. *2 citations pages 40 et 45*

[Arasu 2003] Arvind Arasu et Gurmeet Manku. *Approximate Counts and Quantiles over Sliding Windows*. Rapport technique 2003-72, 2003. *cité page 60*

[Arasu 2004a] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz et Mayur Datar. *STREAM: The Stanford Data Stream Management System*. pages 1–21, Janvier 2004. *5 citations pages 16, 45, 67, 94, et 96*

[Arasu 2004b] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier et Anurag Maskey. *Linear road: a stream data management benchmark*. Proceedings of the International conference on Very Large Data Bases (VLDB), vol. 30, pages 480–491, Janvier 2004. *cité page 158*

[Arasu 2004c] Arvind Arasu et Jennifer Widom. *Resource Sharing in Continuous Sliding-Window Aggregates*. Rapport technique 2004-15, 2004. *cité page 58*

[Arasu 2006a] Arvind Arasu. *Continuous Queries Over Data Streams*. PhD Thesis, Stanford University, 2006. *2 citations pages 46 et 49*

[Arasu 2006b] Arvind Arasu, Shivnath Babu et Jennifer Widom. *The CQL continuous query language: semantic foundations and query execution*. The VLDB Journal - The International Journal on Very Large Data Bases, vol. 15, no. 2, pages 121–142, 2006. *2 citations pages 46 et 47*

[Babcock 2002] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani et Jennifer Widom. *Models and Issues in Data Stream Systems*. Rapport technique, Stanford University, Mars 2002. *2 citations pages 44 et 51*

[Babcock 2003] Brian Babcock, Shivnath Babu, Mayur Datar et Rajeev Motwani. *Chain: Operator Scheduling for Memory Minimization in Data Stream Systems*. Dans Proceedings of the ACM international conference on Management of Data (SIGMOD), pages 253–264. ACM, 2003. *cité page 59*

[Babu 2001] Shivnath Babu et Jennifer Widom. *Continuous queries over data streams*. SIGMOD Record, vol. 30, no. 3, pages 109–120, Septembre 2001. *cité page 45*

[Balazinska 2007] Magdalena Balazinska, Yongchul Kwon, Nathan Kuchta et Dennis Lee. *Moirae: History-enhanced monitoring*. Dans Conference on Innovative Data Systems Research (CIDR), pages 375–386, 2007. *2 citations pages 16 et 57*

[Ben Mokhtar 2007] Sonia Ben Mokhtar, Dany Preuveneers, Nikolaos Georgantas, Valérie Issarny et Yolande Berbers. *EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support*. Journal of Systems and Software, vol. 81, pages 785–808, Août 2007. *cité page 32*

[Bonino 2008] Dario Bonino et Fulvio Corno. *DogOnt - Ontology Modeling for Intelligent Domotic Environments*. Dans Amit Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy Finin et Krishnaprasad Thirunarayan, editeurs, International Semantic Web Conference (ISWC), *LNCS 5318*, pages 790–803, Berlin, Heidelberg, 2008. Springer. *cité page 32*

[Botan 2007] Irina Botan, D Kossmann et Peter M Fischer. *Extending XQuery with window functions*. Dans Proceedings of the International conference on Very Large Data Bases (VLDB), pages 75–86. VLDB Endowment, 2007. *cité page 52*

[Botan 2010] Irina Botan, Roozbeh Derakhshan et Nihal Dindar. *SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems*. Proceedings of the VLDB Endowment, vol. 3, no. 1-2, pages 232–243, 2010. *3 citations pages 16, 49, et 99*

[Botan 2012] Irina Botan, Peter M Fischer, Donald Kossmann et Nesime Tatbul. *Transactional Stream Processing*. Dans Proceedings of the International Conference on Extending Database Technology (EDBT), page TBP. ACM, 2012. *cité page 56*

[Brenna 2007] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte et Walker White. *Cayuga: a high-performance event processing engine*. Proceedings of the ACM international conference on Management of Data (SIGMOD), pages 1100–1102, 2007. *cité page 39*

[BroadBand Forum 2011a] BroadBand Forum. *CPE WAN Management Protocol Amendment 4*. Rapport technique 069, BBF, 2011. *cité page 22*

[BroadBand Forum 2011b] BroadBand Forum. *Data Model Template for TR-069-Enabled Devices*. Rapport technique 106, BBF, 2011. *cité page 23*

[Bruneton 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Qu et Jean-bernard Stefani. *The FRACTAL component model and its support in Java*. Software Pratice And Experience, vol. 36, pages 1257–1284, 2006. *cité page 115*

[Buchholz 2003] T Buchholz, A. Küpper et M Schiffers. *Quality of context: What it is and why we need it*. Dans Proceedings of the Workshop of the OpenView University Association (OVUA), pages 1–14, 2003. *cité page 31*

[Carney 2002] Don Carney, Uğur Çetintemel, Mitch Cherniack et Christian Convey. *Monitoring streams: a new class of data management applications*. Proceedings of the International conference on Very Large Data Bases (VLDB), vol. 28, pages 215–296, Janvier 2002. *cité page 45*

[Carney 2003] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack et Michael Stonebraker. *Operator scheduling in a data stream manager*. Proceedings of the International conference on Very Large Data Bases (VLDB), vol. 29, pages 838–849, Septembre 2003. *2 citations pages 52 et 116*

[Case 1990] J Case, M Fedor et M Schoffstall. *RFC 1157: A simple network management protocol (SNMP)*, 1990. *2 citations pages 12 et 22*

[CCITT 1992] CCITT. *Management Framework for Open Systems Interconnection (OSI) for CCITT Applications*. Data Communication Networks, vol. X.700, pages 1–15, 1992. *cité page 22*

[Cervantes 2004] Humberto Cervantes et Richard S. Hall. *Autonomous adaptation to dynamic availability using a service-oriented component model*. Dans International Conference on Software Engineering (ICSE), pages 614–623. IEEE, 2004. *cité page 114*

[Chan 2008] Marie Chan, Daniel Estève, Christophe Escriba et Eric Campo. *A review of smart homes- present state and future challenges*. Computer methods and programs in biomedicine, vol. 91, no. 1, pages 55–81, Juillet 2008. *cité page 12*

[Chandrasekaran 2003] Sirish Chandrasekaran, Owen Cooper et Amol Deshpande. *TelegraphCQ: Continuous dataflow processing for an uncertain world*. Conference on Innovative Data Systems Research (CIDR), Janvier 2003. *2 citations pages 45 et 98*

[Chen 2000] Jianjun Chen, David DeWitt, Feng Tian et Yuan Wang. *NiagaraCQ: a scalable continuous query system for Internet databases*. SIGMOD Record, vol. 29, no. 2, pages 379–390, 2000. *2 citations pages 39 et 44*

[Codd 1970] Edgar Frank Codd. *A relational model of data for large shared data banks*. Communications of the ACM, vol. 13, no. 6, 1970. *cité page 44*

[Codd 1972] Edgar Frank Codd. *Relational Completeness of Data Base Sublanguages*. Computer Sciences, 1972. *cité page 108*

[Codd 1993] Edgar Frank Codd, S. B. Codd et C. T. Salley. *Providing OLAP to User-Analysts: An IT Mandate*. Rapport technique, E.F. Codd Associates, 1993. *cité page 37*

[Coyle 2006] Lorcan Coyle, Steve Neely, G Rey, Graeme Stevenson et M. *Sensor fusion-based middleware for assisted living*. Dans International Conference on Smart Homes and Health Telematics (ICSHT), pages 1–8. IOS Press, 2006. *cité page 12*

[Cranor 2003] Chuck Cranor, Theodore Johnson, Oliver Spataschek et Vladislav Shkapenyuk. *Gigascope: a stream database for network applications*. Proceedings of the ACM international conference on Management of Data (SIGMOD), pages 647–651, 2003. *2 citations pages 39 et 44*

[Datar 2002] Mayur Datar, Aristides Gionis, Piotr Indyk et Rajeev Motwani. *Maintaining Stream Statistics over Sliding Windows*. Dans Proceedings of the annual ACM-SIAM Symposium on Discrete Algorithms (SODA). Society for Industrial and Applied Mathematics, 2002. *cité page 60*

[de Amo 2009] Sandra de Amo et Marcos Roberto Ribeiro. *CPref-SQL : A Query Language Supporting Conditional Preferences Categories and Subject Descriptors*. Dans Proceedings of the ACM Symposium on Applied Computing (SAC), pages 1573–1577. ACM, 2009. *cité page 169*

[Dey 1999] Anind K Dey et Gregory D Abowd. *Towards a Better Understanding of Context and Context-Awareness*. Rapport technique, Georgia Institute of Technology, College of Computing, 1999. *cité page 28*

[DMTF 2007] DMTF. *CIM Query Language Specification*, 2007. *cité page 26*

[DMTF 2010] DMTF. *Web Services for Management (WS-Management) Specification*, 2010. *cité page 23*

[DMTF 2012a] DMTF. *Common Information Model*. `http://dmtf.org/standards/cim`, 2012. *cité page 23*

[DMTF 2012b] DMTF. *Web-Based Entreprise Management*. `http://dmtf.org/standards/wbem`, 2012. *cité page 26*

[Docherty 2009] Liam Stephen Docherty. *An Ontology Based Approach Towards A Universal Description Framework for Home Networks*. Phd, University of Stirling, Scotland, UK, 2009. *cité page 33*

[Duller 2011] Michael Duller, JS Rellermeyer et Gustavo Alonso. *Virtualizing Stream Processing*. Dans Proceedings of the ACM/IFIP/USENIX Middleware, pages 269–288. Springer, 2011. *cité page 54*

[Eisenberg 2004] Andrew Eisenberg, Jim Melton, Krishna Kulkarni et JE Michels. *SQL: 2003 has been published*. SIGMOD Record, vol. 33, no. 1, pages 119–126, 2004. *cité page 44*

[El Kaed 2011] Charbel El Kaed, Loïc Petit, Louvel Maxime, Antonin Chazalet, Yves Denneulin et François-Gaël Ottogalli. *INSIGHT: interoperability and service management for the digital home*. Dans Proceedings of the ACM/IFIP/USENIX Middleware Industry Track Workshop, pages 3:1–3:6. ACM, 2011. *cité page 155*

[El Kaed 2012] Charbel El Kaed. *Home Devices Mediation using ontology alignment and code*. Phd, Université de Grenoble, 2012. *2 citations pages 25 et 30*

[Escoffier 2007] Clement Escoffier, Richard S. Hall et Philippe Lalanda. *iPOJO: an Extensible Service-Oriented Component Framework*. Dans IEEE International Conference on Services Computing (SCC), pages 474–481. IEEE, 2007. *cité page 115*

[Eugster 2003] Patrick Th. Eugster, Pascal a. Felber, Rachid Guerraoui et Anne-Marie Kermarrec. *The many faces of publish/subscribe*. ACM Computing Surveys, vol. 35, no. 2, pages 114–131, Juin 2003. *cité page 52*

[Franklin 2005] Michael J Franklin, Shawn R Jeffery, Sailesh Krishnamurthy, Frederick Reiss, Shariq Rizvi, Eugene Wu, Owen Cooper, Anil Edakkunni et Wei Hong. *Design considerations for high fan-in systems: The HiFi approach*. Conference on Innovative Data Systems Research (CIDR), Janvier 2005. *cité page 54*

[Gajski 1984] Daniel Gajski, Won Kim et Shinya Fushimi. *A Parallel Pipelined Relational Query Processor: An Architectural Overview*. Dans Proceedings of the annual international symposium on Computer architecture (ISCA), pages 134–141. ACM, 1984. *cité page 60*

[Galpin 2009] Ixent Galpin, Christian Y A Brenninkmeijer, Farhana Jabeen, Alvaro A A Fernandes et Norman W Paton. *Comprehensive Optimization of Declarative Sensor Network Queries*. Scientific and Statistical Database Management, vol. 5566, pages 339–360, 2009. *2 citations pages 59 et 61*

[Geihs 2001] Kurt Geihs. *Middleware challenges ahead*. Computer, vol. 34, no. 6, pages 24–31, 2001. *cité page 12*

[Gilbert 2001] A C Gilbert, Y Kotidis, S Muthukrishnan et M J Strauss. *QuickSAND : Quick Summary and Analysis of Network Data by*. Rapport technique November, AT&T Labs, 2001. *cité page 36*

[Golab 2003] Lukasz Golab et M. Tamer Özsu. *Issues in data stream management*. SIGMOD Record, vol. 32, no. 2, 2003. *cité page 39*

[Gray 1997] Jim Gray, Adam Bosworth, Andrew Layman, Don Reichart et Hamid Pirahesh. *Data Cube : A Relational Aggregation Operator*. Data Mining and Knowledge Discovery, vol. 53, no. 1, pages 29–53, 1997. *cité page 37*

[Gripay 2010] Yann Gripay, Frédérique Laforest et Jean-Marc Petit. *A simple (yet powerful) algebra for pervasive environments*. Dans Proceedings of the International Conference on Extending Database Technology (EDBT), page 359. ACM, 2010. *cité page 50*

[Gu 2005] T Gu, H Pung et D Zhang. *A service-oriented middleware for building context-aware services*. Journal of Network and Computer Applications, vol. 28, no. 1, pages 1–18, 2005. *cité page 33*

[Gürgen 2006] Levent Gürgen, Claudia Lucia Roncancio, Cyril Labbé et Vincent Olive. *Transactional issues in sensor data management*. Proceedings of the workshop on Data management for sensor networks (VLDB-DMSN), pages 27–32, Septembre 2006. *cité page 56*

[Gürgen 2007] Levent Gürgen. *Gestion à grande échelle de données de capteurs hétérogènes*. PhD Thesis, INP Grenoble, Janvier 2007. *4 citations pages 39, 44, 54, et 98*

[Han 2007] Dong-Hong Han, Guo-Ren Wang, Chuan Xiao et Rui Zhou. *Load Shedding for Window Joins over Streams*. Journal of Computer Science and Technology, vol. 22, no. 2, pages 182–189, Avril 2007. *2 citations pages 53 et 59*

[Han 2008] Li Han, Salomaa Jyri, Jian Ma et Kuifei Yu. *Research on Context-Aware Mobile Computing*. Dans International Conference on Advanced Information

Networking and Applications Workshops (WAINA), pages 24–30. IEEE, 2008. *cité page 28*

[Harper 2003] Richard Harper. Inside the Smart Home. Springer, 2003. *cité page 12*

[Hellerstein 2010] Joseph M. Hellerstein. *The declarative imperative*. SIGMOD Record, vol. 39, no. 1, page 5, Septembre 2010. *cité page 35*

[Hong 2009] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke et Alan Demers. *Rule-based multi-query optimization*. Dans Proceedings of the International Conference on Extending Database Technology (EDBT), page 120. ACM, 2009. *cité page 58*

[Hwang 2005] Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Uğur Çetintemel, Michael Stonebraker et Stan Zdonik. *High Availability Algorithms for Distributed Stream Processing*. International Conference on Data Engineering (ICDE), pages 779–790, 2005. *cité page 53*

[Inmon 2002] WH Inmon. Building the data warehouse (3rd Edition). Wiley, 2002. *cité page 36*

[Jagadish 1995] H Jagadish, Inderpal Mumick et Abraham Silberschatz. *View maintenance issues for the chronicle data model (extended abstract)*. Proceedings of the ACM Symposium on Principles of Database systems (PODS), pages 113–124, 1995. *2 citations pages 44 et 46*

[Jain 2006] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo et Chitra Venkatramani. *Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core*. Dans Proceedings of the ACM international conference on Management of Data (SIGMOD), pages 431–442. ACM, 2006. *cité page 160*

[Jain 2008] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts et Stan Zdonik. *Towards a streaming SQL standard*. Proceedings of the VLDB Endowment, vol. 1, no. 2, pages 1379–1390, 2008. *6 citations pages 16, 48, 88, 96, 97, et 100*

[Jiang 2004] Qingchun Jiang et Sharma Chakravarthy. *Scheduling strategies for processing continuous queries over streams*. Key Technologies for Data Management, vol. 3112, pages 16–30, 2004. *cité page 59*

[Jurdak 2008] Raja Jurdak, Abdelhamid Nafaa et Alessio Barbirato. *Large Scale Environmental Monitoring through Integration of Sensor and Mesh Networks*. MDPI Sensors: Molecular Diversity Preservation International Sensors, Novembre 2008. *2 citations pages 11 et 98*

[Kalfoglou 2001] Yannis Kalfoglou. *Exploring ontologies*. Dans The Handbook of Software Engineering and Knowledge Engineering, pages 863–887. World Scientific Publishing, 2001. *cité page 29*

[Kang 2005] Dong-oh Kang, Kyuchang Kang, Sunggi Choi et Jeunwoo Lee. *UPnP AV architectural multimedia system with a home gateway powered by the OSGi platform*. International Conference on Consumer Electronics (ICCE), vol. 51, no. 1, pages 87–93, 2005. *cité page 12*

[Keller 1985] Arthur M Keller. *Algorithms for translating view updates to database updates for views involving selections, projections, and joins*. Proceedings of the ACM Symposium on Principles of Database systems (PODS), pages 154–163, 1985. *cité page 137*

[Kessis 2010] Mehdid Kessis. *Gestion intégrée et multi-échelles des systèmes répartis : Architecture et canevas intergiciel orientés composants*. Phd, Université de Grenoble, 2010. *cité page 25*

[Kontaki 2012] M. Kontaki, a. N. Papadopoulos et Y. Manolopoulos. *Continuous Top-k Dominating Queries*. IEEE Transactions on Knowledge and Data Engineering, vol. 24, no. 5, pages 840–853, Mai 2012. *cité page 177*

[Korhonen 2003] Ilkka Korhonen, Juha Pärkkä et Mark Van Gils. *Health monitoring in the home of the future*. IEEE Engineering in Medicine and Biology Magazine, vol. 22, no. 3, pages 66–73, 2003. *cité page 12*

[Kozlenkov 2004] Alexander Kozlenkov et Michael Schroeder. *PROVA: Rule-based Java-scripting for a bioinformatics semantic web*. Dans Data Integration in the Life Sciences, LNCS, pages 17–30. Springer, 2004. *cité page 118*

[Krämer 2009] Jürgen Krämer et Bernhard Seeger. *Semantics and implementation of continuous sliding window queries over data streams*. Transactions on Database Systems (TODS), vol. 34, no. 1, pages 1–49, Avril 2009. *3 citations pages 50, 51, et 61*

[Law 2007] Yan-nei Law et Carlo Zaniolo. *Load Shedding for Window Joins on Multiple Data Streams*. Dans International Conference on Data Engineering Workshop (ICDEW), pages 674–683. IEEE, Avril 2007. *cité page 59*

[Levene 2003] Mark Levene et George Loizou. *Why is the snowflake schema a good data warehouse design?* Information Systems, vol. 28, no. 3, pages 225–240, Mai 2003. *cité page 37*

[Li 2005] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos et Peter A Tucker. *No Pane , No Gain : Efficient Evaluation of Sliding-Window Aggregates over Data Streams*. SIGMOD Record, vol. 34, no. 1, pages 39–44, 2005. *2 citations pages 60 et 161*

[Liu 1999] Ling Liu, Calton Pu et Wei Tang. *Continual queries for Internet scale event-driven information delivery*. IEEE Transactions on Knowledge and Data Engineering, vol. 11, no. 4, pages 610–628, 1999. *cité page 44*

[Liu 2010] Mengmeng Liu, S.R. Mihaylov, Zhuowei Bao, Marie Jacob, Z.G. Ives, B.T. Loo et Sudipto Guha. *SmartCIS: Integrating digital and physical environments*. SIGMOD Record, vol. 39, no. 1, pages 48–53, 2010. *cité page 56*

[López De Vergara 2001] Jorge E. López De Vergara, Julio Guijarro et Patrick Goldsack. *Modelling and developing the information to manage an Internet Data Centre*. Dans Proceedings of the HP Openview University Association Eighth Plenary Workshop (HP-OVUA), pages 1–11, 2001. *cité page 24*

[Madden 2002] Samuel R Madden, Michael J Franklin, Joseph M. Hellerstein et Wei Hong. *TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks*. ACM SIGOPS Operating Systems Review, vol. 36, no. SI, pages 131–146, 2002. *2 citations pages 39 et 44*

[Madden 2005] Samuel R Madden, Michael J Franklin, Joseph M. Hellerstein et Wei Hong. *TinyDB: an acquisitional query processing system for sensor networks*. Transactions on Database Systems (TODS), vol. 30, no. 1, pages 122–173, Mars 2005. *cité page 59*

[Maier 2005] David Maier, Jin Li, Peter Tucker, Kristin Tufte et Vassilis Papadimos. *Semantics of Data Streams and Operators*. Dans International Conference on Database Theory (ICDT), pages 37–52. Springer, 2005. *2 citations pages 48 et 51*

[Melton 2002] Jim Melton. Advanced SQL:1999 – Understanding Object-Relational and Other Advanced Features. Morgan Kaufmann, 2002. *cité page 44*

[Minsky 1974] Marvin Minsky. *A framework for representing knowledge*. The Psychology of Computer Vision, 1974. *cité page 29*

[Morse 2007] M. Morse, J.M. Patel et W.I. Grosky. *Efficient continuous skyline computation*. Information Sciences, vol. 177, no. 17, pages 3411–3437, Septembre 2007. *cité page 177*

[Motorola Mobility, Inc. 2012] Motorola Mobility, Inc. *EDGE™ Manager*. http://www.motorola.com/Video-Solutions/US-EN/Products-and-Services/Software/EDGE-Service-Assurance-Software-Suite/EDGE-Manager_US-EN, 2012. *2 citations pages 25 et 26*

[Mouratidis 2006] Kyriakos Mouratidis, Spiridon Bakiras et Dimitris Papadias. *Continuous monitoring of top-k queries over sliding windows*. Dans Proceedings of the ACM international conference on Management of Data (SIGMOD), page 635. ACM, 2006. *cité page 177*

[Niang 2011] Cheikh Niang, Béatrice Bouchou, Moussa Lo et Yacine Sam. *Automatic Building of an Appropriate Global Ontology*. Advances in Databases and Information Systems (ADBIS), pages 429–443, 2011. *cité page 33*

[Niang 2012] Cheikh Niang, Béatrice Bouchou, Moussa Lo et Yacine Sam. *Querying a Semi-automated Data Integration System*. Proceedings of the international conference on Database and Expert Systems Applications (DEXA), pages 305–313, 2012. *cité page 33*

[Oracle 2010a] Oracle. *Best Practices for Real-time Data Warehousing*. Rapport technique May, Oracle, 2010. *cité page 37*

[Oracle 2010b] Oracle. *Oracle Data Integrator*. http://www.oracle.com/technetwork/middleware/data-integrator/overview/index.html, 2010. *cité page 37*

[Padovitz 2008] Amir Padovitz et SW Loke. *Multiple-agent perspectives in reasoning about situations for context-aware pervasive computing systems*. IEEE Transactions on Systems, Man and Cybernetics, vol. 38, no. 4, pages 729–742, 2008. *2 citations pages 31 et 32*

[Palma 2009] Wenceslao Palma, Reza Akbarinia, Esther Pacitti et Patrick Valduriez. *P2P Join Query Processing over Data Streams*. Dans Journées Bases de Données Avancées (BDA), Namur, Belgium, 2009. *cité page 59*

[Patroumpas 2006] Kostas Patroumpas et Timos K Sellis. *Window specification over data streams*. International Conference on Semantics of a Networked World (ICSNW), vol. 4254, pages 445–464, 2006. *2 citations pages 48 et 99*

[Patroumpas 2011] Kostas Patroumpas et Timos Sellis. *Subsuming Multiple Sliding Windows for Shared Stream Computation*. Dans Proceedings of the international conference on Advances in Databases and Information Systems (ADBIS), pages 56–69. Springer, 2011. *cité page 48*

[Petit 2009] Loïc Petit, Abdelhamid Nafaa et Raja Jurdak. *Historical data storage for large scale sensor networks*. Dans Journées francophones Mobilité et Ubiquité (Ubimob), page 45. ACM, 2009. *cité page 37*

[Petit 2010] Loïc Petit, Cyril Labbé et Claudia Lucia Roncancio. *An algebric window model for data stream management*. Dans Proceedings of the International ACM Workshop on Data Engineering for Wireless and Mobile Access (MOBIDE), pages 17–24. ACM, 2010. *cité page 108*

[Petit 2011] Loïc Petit, Claudia Lucia Roncancio, Cyril Labbé et François-Gaël Ottogalli. *DomVision : Intergiciel de gestion de données pour l'environnement domestique*. Dans Journées Bases de Données Avancées (BDA), 2011. *cité page 155*

[Petit 2012a] Loïc Petit, Sandra De Amo, Claudia Lucia Roncancio et Cyril Labbé. *Top-k Context-Aware Queries on Streams*. Dans Proceedings of the international conference on Database and Expert Systems Applications (DEXA), LNCS, pages 397–411. Springer, 2012. *2 citations pages 176 et 178*

[Petit 2012b] Loïc Petit, Sandra De Amo, Claudia Lucia Roncancio et Cyril Labbé. *Top-k Context-Aware Queries on Streams*. Dans Journées Bases de Données Avancées (BDA), 2012. *cité page 178*

[Petit 2012c] Loïc Petit, Cyril Labbé et Claudia Lucia Roncancio. *Revisiting Formal Ordering in Data Stream Querying*. Dans Proceedings of the ACM Symposium on Applied Computing (SAC), pages 813–818, Riva del Garda, Italy, 2012. ACM. *cité page 108*

[Reiss 2007] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani et Joseph M. Hellerstein. *Enabling Real-Time Querying of Live and Historical Stream Data*. Dans International Conference on Scientific and Statistical Database Management (SSDM), pages 28–37. IEEE, 2007. *2 citations pages 16 et 56*

[Roncancio 2012] Claudia Lucia Roncancio, Loïc Petit, Sandra de Amo et Cyril Labbé. *Préférences utilisateurs dans la gestion de données ubiquitaires*. Dans Journées francophones Mobilité et Ubiquité (Ubimob). Cépaduès, 2012. *cité page 178*

[Rose 1990] MT Rose. *RFC 1155: Structure and Identification of Management Information for TCP/IP-based internets*, 1990. *cité page 22*

[Sellis 1988] Timos K Sellis. *Multiple-query optimization*. Transactions on Database Systems (TODS), vol. 13, no. 1, pages 23–52, 1988. *cité page 58*

[Seshadri 1995] Praveen Seshadri, Miron Livny et Raghu Ramakrishnan. *SEQ: A model for sequence databases*. Transactions on Knowledge and Data Engineering, Janvier 1995. *2 citations pages 44 et 95*

[Sloman 1994] Morris Sloman. *Network and Distributed Systems Management*. page 666, 1994. *cité page 22*

[Soylu 2009] Ahmet Soylu, Patrick De Causmaecker et Piet Desmet. *Context and Adaptivity in Pervasive Computing Environments: Links with Software Engineering and*

Ontological Engineering. Journal of Software, vol. 4, no. 9, pages 992–1013, 2009. *cité page 28*

[SQLstream 2010] Inc. SQLstream. *SQLstream*. http://www.sqlstream.com, 2010. *cité page 46*

[Srivastava 2004] Utkarsh Srivastava et Jennifer Widom. *Memory-Limited Execution of Windowed Stream Joins*. Dans Proceedings of the International conference on Very Large Data Bases (VLDB), volume 30, pages 324–335. VLDB Endowment, 2004. *2 citations pages 53 et 59*

[Sullivan 1996] Mark Sullivan. *Tribeca: A Stream Database Manager for Network Traffic Analysis*. Proceedings of the International conference on Very Large Data Bases (VLDB), page 594, Septembre 1996. *cité page 44*

[Sullivan 1998] Mark Sullivan et Andrew Heybey. *Tribeca: A System for Managing Large Databases of Network Traffic*. Dans Proceedings of the USENIX Annual Technical Conference, pages 13–24. USENIX Association, USENIX Association, 1998. *cité page 44*

[Sun Microsystems 2002] Sun Microsystems. *Java™ Management Extensions Instrumentation and Agent Specification, v1.2*, 2002. *cité page 24*

[Sybase 2010] Inc. Sybase. *Aleri Streaming Platform*. http://www.sybase.com/products/financialservicessolutions/aleristreamingplatform, 2010. *cité page 16*

[Systar 2012] Systar. *Business Activity Monitoring*. http://www.systar.com/, 2012. *cité page 12*

[Szewczyk 2004] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Alan Hamilton, Michael Mainwaring et D Estrin. *Habitat Monitoring with Sensor Networks*. Communications of the ACM, vol. 47, no. 6, pages 34–40, 2004. *cité page 11*

[Tatbul 2003] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik et Mitch Cherniack. *Load Shedding on Data Streams*. Proceedings of the ACM Workshop on Management and Processing of Data Streams (MPDS), Janvier 2003. *cité page 53*

[Tatbul 2006] Nesime Tatbul et Stan Zdonik. *Window-aware load shedding for aggregation queries over data streams*. Proceedings of the International conference on Very Large Data Bases (VLDB), vol. 32, pages 799–810, Janvier 2006. *cité page 53*

[Terry 1992] Douglas Terry, David Goldberg, David Nichols et Brian Oki. *Continuous queries over append-only databases*. SIGMOD Record, vol. 21, no. 2, pages 321–330, Juin 1992. *3 citations pages 40, 44, et 58*

[Thomsen 2008] Christian Thomsen, Torben Bach Pedersen et Wolfgang Lehner. *RiTE: Providing on-demand data for right-time data warehousing*. Dans International Conference on Data Engineering (ICDE), pages 456–465. IEEE, 2008. *cité page 37*

[Tucker 2003] Peter A. Tucker, David Maier et David Sheard. *Applying punctuation schemes to queries over continuous data streams*. IEEE Data Engineering Bulletin, vol. 26, no. 1, pages 33–40, 2003. *cité page 53*

[UPnP Forum 2012a] UPnP Forum. *Device Management: ConfigurationManagement:2*, 2012. *cité page 23*

[UPnP Forum 2012b] UPnP Forum. *Device Management:2*. http://upnp.org/specs/dm/dm2/, Février 2012. *2 citations pages 22 et 151*

[Vassiliadis 2009] Panos Vassiliadis, Alkis Simitsis et Eftychia Baikousi. *A taxonomy of ETL activities*. Dans Proceedings of the ACM international workshop on Data warehousing and OLAP (DOLAP), pages 25–32, Hong Kong, China, 2009. ACM. *cité page 37*

[Vilei 2006] Antonio Vilei, Gabriella Convertino et Fabrizio Crudo. *A new UPnP architecture for distributed video voice over IP*. Dans Proceedings of the international conference on Mobile and ubiquitous multimedia (MUM), pages 2:1–2:7. ACM Press, 2006. *cité page 12*

[W3C 2004] W3C. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. http://www.w3.org/TR/rdf-concepts/, Février 2004. *cité page 29*

[Weikum 2010] Gerhard Weikum et Martin Theobald. *From information to knowledge: harvesting entities and relationships from web sources*. Dans Proceedings of the ACM Symposium on Principles of Database systems (PODS), pages 65–76. ACM, 2010. *cité page 35*

[Wilschut 1991] Annita N Wilschut et P M G Apers. *Dataflow query execution in a parallel main-memory environment*. International conference on Parallel and distributed information systems, pages 68–77, Janvier 1991. *cité page 60*

[Wilson 2004] Nic Wilson. *Extending CP-Nets with Stronger Conditional Preference Statements*. Dans American Association for Artificial Intelligence (AAAI), pages 735–741. AAAI Press, 2004. *cité page 171*

[Witkowski 2007] Andrew Witkowski, Srikanth Bellamkonda, Hua-Gang Li et Vince Liang. *Continuous queries in oracle*. Proceedings of the International conference on Very Large Data Bases (VLDB), vol. 33, pages 1173–1184, Janvier 2007. *2 citations pages 46 et 57*

[Yao 2002] Yong Yao et Johannes Gehrke. *The cougar approach to in-network query processing in sensor networks*. SIGMOD Record, vol. 31, no. 3, pages 9–18, 2002. *2 citations pages 39 et 44*

[Zabbix 2012] Zabbix. *IT Infrastructure Monitoring*. http://www.zabbix.com/, 2012. *cité page 12*

[Zhou 2006] Yongluan Zhou, Ying Yan, Feng Yu et Aoying Zhou. *PMJoin: Optimizing Distributed Multi-way Stream Joins by Stream Partitioning*. Dans Database Systems for Advanced Applications (DASFAA), *LNCS 3882*, pages 325–341, Berlin, Heidelberg, 2006. Springer. *cité page 59*

[Zoho Corp. 2012] Zoho Corp. *Manage Engine: Monitor Server & Application Performance in Physical, Virtual & Cloud Environments*. http://www.manageengine.com/products/applications_manager/, 2012. *cité page 12*

A
Démonstrations

Ce chapitre présente l'ensemble des démonstrations des propriétés et théorèmes non triviaux décrits dans ce manuscrit.

A.1 Proposition 4.2.1 : Inclusion de la sélection

Soit b un *batch* quelconque, alors $(\sigma_c(R))(b) = \Sigma_c(R(b))$. Par définition, Σ_c filtre les n-uplets de l'ensemble fournit. Alors, nous obtenons naturellement $\Sigma_c(R(b)) \subset R(b)$. Comme l'inclusion ensembliste est trivialement un cas particulier de l'inclusion de séquence de n-uplets présenté en définition 4.1.12, alors, $(\sigma_c(R))(b) \subseteqq R(b)$. D'où le résultat.

■

A.2 Théorème 5.1 : Transmission temporelle

A.2.1 Lemme des propriétés de τ_S

Avant de détailler cette démonstration, nous devons prouver le lemme A.1.

Lemme A.1 **Propriétés de τ_S**

$$\begin{aligned} \forall n \in \mathbb{N}, \quad \tau_S(n) &\geq (t_0, 0) \\ \forall b \in \mathbb{T} \times \mathbb{N}, \quad \tau_S(\tau_S^{-1}(b)) &\leq (b) \\ \forall n \in \mathbb{N}, \quad \tau_S^{-1}(\tau_S(n)) &\geq n \end{aligned}$$

De plus, si $\exists s \in S, \mathcal{B}_S(s) = b$, alors $\tau_S \circ \tau_S^{-1}(b) = b$.

Comme tout *timestamp* d'un flux initialisé est au moins plus grand que t_0, alors, la première position a un timestamp plus grand que t_0. Sachant l'hypothèse 4.1, τ_S est croissante, alors $\forall n \in \mathbb{N}, \tau_S(n) \geq (t_0, 0)$.

Soit $b \in \mathbb{T} \times \mathbb{N} \geq (t_0, 0)$. Sachant la définition formelle de τ_S^{-1}.

- Si $|S| < +\infty \wedge b \geq \tau_S(|S|)$, alors nous pouvons écrire $\tau_S(\tau_S^{-1}(b)) = \tau_S(|S|) \leq b$.
- Sinon : $\tau_S^{-1}(b) = \sum_{n=0}^{|S|-1} n \, \mathbf{1}_{[\tau_S(n), \tau_S(n+1)[}(b)$

Soit j étant l'unique position telle que $b \in [\tau_S(j), \tau_S(j+1)[$. Alors, $\tau_S(\tau_S^{-1}(b)) = \tau_S(j) \leq b$. Alors $\exists s \in S$ tel que $\mathcal{B}_S(s) = b$, alors $\exists j \in \mathbb{N}$ tel que $\tau_S(j) = b$.

Enfin, soit $n \in \mathbb{N}$, comme τ_S est croissante non stricte, il peut y avoir des *batchs* égaux. Soit $[x, y]$ le plus grand intervalle tel que $\forall j \in [\![x, y]\!]$, $\tau_S(j) = \tau_S(n)$, par définition $n \in [x, y]$.

$$\begin{aligned} \tau_S^{-1}(\tau_S(j)) &= \sum_{n=0}^{|S|-1} n \, \mathbf{1}_{[\tau_S(n), \tau_S(n+1)[}(\tau_S(j)) \\ &= \sum_{n=x}^{y} n \, \mathbf{1}_{[\tau_S(n), \tau_S(n+1)[}(\tau_S(j)) \\ &= y \mathbb{1}_{[\tau_S(b), \tau_S(b+1)[}(\tau_S(j)) \\ &= y \end{aligned}$$

Ainsi, $[\tau_S(n), \tau_S(n+1)[= \emptyset$ pour $n \in [x, y[$. Comme $b \geq n$, nous obtenons le résultat.

A.2.2 La démonstration

La fonction d'attente pour une DSF positionnelle est la suivante :

$$\gamma(b) = \left\lfloor \frac{\tau_S^{-1}(b) - \beta(0)}{r} \right\rfloor$$

Avec la DSF fournie $[\alpha, j+k, 1]$, $\gamma(b) = \tau_S^{-1}(b) - k$.
Soit $s \in E_b \Leftrightarrow s \in S$ et

$$\begin{array}{rl} & \tau_S \circ \alpha \circ \gamma(b) \leq \mathcal{B}_S(s) \leq \tau_S(\beta(\gamma(b))) \\ \Leftrightarrow & \tau_S \circ \alpha \circ \gamma(b) \leq \mathcal{B}_S(s) \leq \tau_S(\gamma(b)) \\ \Leftrightarrow & \tau_S \circ \alpha \circ \gamma(b) \leq \mathcal{B}_S(s) \leq \tau_S(\tau_S^{-1}(b)) \end{array}$$

La dernière inégalité est équivalente à $\tau_S \circ \alpha \circ \gamma(b) \leq \mathcal{B}_S(s) \leq b$. L'implication est simple dû à la propriété A.1. La réciproque doit être détaillée. Si $\mathcal{B}_S(s) \leq b$, il évident que $\forall s \in S, \mathcal{B}_S(s) \notin]\tau_S(\tau_S^{-1}(b)), b[$, car s'il existait un tel n-uplet $\tau_S(\tau_S^{-1}(b))$ serait égal à b ce qui est absurde. Ainsi, $\mathcal{B}_S(s) \leq \tau_S(\tau_S^{-1}(b))$.

Alors, $S[\alpha, j, 1](b) = F_b = s \in S, \begin{cases} \tau_S \circ \alpha \circ \gamma(b) \leq \mathcal{B}_S(s) \leq b \\ \text{rpos}_{E_{t,i}}(s) \leq \tau_S^{-1}(b) - \alpha \circ \gamma(b) \end{cases}$

$$\begin{array}{l} \Psi_b(s,t) \in S' \wedge \mathcal{B}_{S'}(\Psi_b(s,t)) = b \\ \quad \Leftrightarrow s \in F_b \wedge s \notin F_{b^-} \\ \quad \Leftrightarrow s \in S \wedge [...]_1 \leq \mathcal{B}_S(s) \leq b \wedge \text{rpos}_{E_b}(s) \leq [...]_2 \wedge \\ \qquad ([...]_3 > \mathcal{B}_S(s) \vee \mathcal{B}_S(s) > b^- \vee \text{rpos}_{E_{b^-}}(s) > [...]_4) \end{array}$$

Comme $\tau_S \circ \alpha \circ \gamma(b) \leq \mathcal{B}_S(s) \wedge \tau_S \circ \alpha \circ \gamma(b^-) > \mathcal{B}_S(s)$ est impossible car $\tau_S \circ \alpha \circ \gamma$ est croissante. De plus, $\text{rpos}_{E_b}(s) \leq \tau_S^{-1}(b) - \alpha \circ \gamma(b) \wedge \text{rpos}_{E_{b^-}}(s) > \tau_S^{-1}(b^-) - \alpha \circ \gamma(b^-)$ est aussi impossible car le nombre de n-uplet avec une position plus grande que s augmente en passant de b^- à b. Nous obtenons finalement le résultat suivant :
$\Leftrightarrow s \in F_b \wedge \mathcal{B}_S(s) > b^-$
Ainsi, $\mathcal{B}_S(s) = b$.

■

A.3 Corollaire 5.1 : Équivalence de la composition fenêtre-*streamer*

Sachant $S' = \mathcal{I}_S(S[\alpha, i+k, 1])$,
En développant les expressions, nous voyons trivialement que le théorème de transmission temporelle nous démontre en substance que, $S' \subseteq S$. Or, si la fenêtre contient au moins le dernier *batch*, alors, pour un *batch* donné (t,i), si $s \in S$ et $\mathcal{B}_S(s) = (t,i)$ alors $s \in S[W](b)$ et $s \notin S[W](b^-)$. Ainsi, $\Psi_{(t,i)}(s,t) \in S'$ et $\mathcal{B}_{S'}(\Psi_{(t,i)}(s,t)) = (t,i)$. Donc, nous avons équivalence.
Les autres expressions du corollaires sont des conséquences triviales de cette DSF.

■

A.4 Table 4.2 : Équivalences de DSF

A.4.1 Fenêtre cumulative

En analysant la démonstration de la transmission temporelle, nous pouvons en extraire la propriété suivante :

$$s \in E_{t,i} \Leftrightarrow s \in S \wedge (t_0, 0) \leq \mathcal{B}_S(s) \leq (t, i)$$

Nous devons désormais prouver que $E_{t,i} = S[0, j, 1](t, i)$. Comme, $\beta(\gamma(t, i)) - \alpha(\gamma(t, i)) = \tau_S^{-1}(t, i)$.

Soit $s \in E_{t,i}$. Alors $\tau_S^{-1}(t, i)$ correspond à la position du n-uplet ayant le plus grand φ dans $E_{t,i}$.

$$\underset{E_{t,i}}{\text{rpos}}(s) = \tau_S^{-1}(t, i) - \underset{E_{t,i}}{\text{pos}}(s)$$

Ainsi $\tau_S^{-1}(t, i) - \text{pos}_{E_{t,i}}(s) \leq \tau_S^{-1}(t, i)$ ce qui est trivialement vrai.

Étant donné que le raisonnement est réciproque, alors : $S[0, j, 1](t, i) = \{s \in S, \mathcal{B}_S(s) \leq (t, i)\}$

■

A.4.2 Dernier batch

Ce résultat est démontré similairement à la démonstration précédente.

Premièrement, nous remarquons grâce au lemme A.1 que $\tau_S(\gamma(t, i)) = \tau_S(\tau_S^{-1}(t, i)) \leq (t, i)$. Par définition, $\exists s \in S$, $\mathcal{B}_S(s) = \tau_S(\tau_S^{-1}(t, i)) = (t', i')$. Alors, $\tau_S^{-1}((t', i')^-)$ est la position maximale de tout n-uplet ayant un *batch* strictement inférieur à (t, i). D'où le résultat : $[B] =]\tau_S^{-1}(\tau_S(i)^-), i, 1]$.

■

A.5 Table 5.1 et 5.2 de commutativité des projections et sélection

La plupart des équivalences présentée dans ce tableau sont issues des connaissances de l'algèbre relationnelle. Concernant la projection, il est important de remarquer que la commutativité ne perturbe pas l'aspect temporel des relations notamment parce que les n-uplets sont *identifiés* par φ. Ainsi, soit s et s' deux n-uplets et A un ensemble d'attributs, alors nous avons $s = s' \Leftrightarrow s[A] \Leftrightarrow s'[A]$. Donc, la projection n'a pas beaucoup d'influence et peut commuter facilement.

Concernant la sélection, c'est un peu plus délicat car la sélection perturbe naturellement la *position* d'un n-uplet. C'est pour cela qu'il n'est pas possible de permuter avec l'opérateur de séquence de fenêtres. Toutefois, les démonstrations sont triviales

en suivant les définitions. Par exemple, pour un opérateur de séquence de fenêtres temporel :

$$\begin{aligned} s \in (\sigma_c S)[\alpha,\beta,\gamma](t,i) &\Leftrightarrow s \in (\sigma_c S) \wedge (\alpha(\gamma(t,i)),0) \leq \mathcal{B}_{(\sigma_c S)}(s) \leq (\beta(\gamma(t,i)),i) \\ &\Leftrightarrow s \in S \wedge c(s) \wedge (\alpha(\gamma(t,i)),0) \leq \mathcal{B}_S(s) \leq (\beta(\gamma(t,i)),i) \\ &\Leftrightarrow c(s) \wedge s \in S[\alpha,\beta,\gamma](t,i) \\ &\Leftrightarrow s \in \sigma_c(S[\alpha,\beta,\gamma](t,i)) \end{aligned}$$

■

A.6 Propriété 5.3.2 : Associativité des jointures et unions

Ces opérateurs sont associatifs dans l'algèbre relationnelle. Leurs expressions dans Astral est similaire à la notion d'identifiant physique près. Analysons l'effet de l'associativité sur $\Phi^{\times}$ et $\Phi^{\cup}$.

Soit $a,b,c,d,e,f \in \mathbb{I}^6$,

$$\begin{aligned} \Phi^{\times}(a,\Phi^{\times}(b,c)) \leq \Phi^{\times}(d,\Phi^{\times}(e,f)) &\Leftrightarrow (a,(b,c)) \leq (d,(e,f)) \\ &\Leftrightarrow a < d \vee a = d \wedge (b,c) \leq (e,f) \\ &\Leftrightarrow a < d \vee a = d \wedge (b < e \vee b = e \wedge c \leq f) \end{aligned}$$

De même, nous obtenons le résultat similaire :

$$\begin{aligned} \Phi^{\times}(\Phi^{\times}(a,b),c) \leq \Phi^{\times}(\Phi^{\times}(d,e),f) &\Leftrightarrow ((a,b),c) \leq ((d,e),f) \\ &\Leftrightarrow (a,b) < (d,e) \vee (a,b) = (d,e) \wedge c \leq f \\ &\Leftrightarrow (a < d \vee a = d \wedge b < e) \vee \\ &\qquad a = d \wedge b = e \wedge (b < e \vee b = e \wedge c \leq f) \\ &\Leftrightarrow a < d \vee a = d \wedge (b < e \vee b = e \wedge c \leq f) \end{aligned}$$

Donc les deux ordres sont équivalents et nous obtenons l'associativité.
La démonstration est similaire pour $\Phi^{\cup}$.

A.7 Théorème 5.2 : Transposabilité générale des DSF

Tout d'abord il est évident de voir que les taux d'évaluations r doivent être nécessairement identiques. Soient γ et γ' les fonctions d'attentes des descriptions.

A.7.1 Le cas temporel

Soit $(t,i) \in \mathbb{T} \times \mathbb{N}$ tel que $\gamma'(t,i) \geq 0$,
La condition sur le flux de l'opérateur de séquence de fenêtre dans la première requête est :

$$(\alpha(\gamma(t,i)),0) \leq \mathcal{B}_S(s) \leq (\beta(\gamma(t,i)),i)$$

Comme le flux d'entrée S est naturellement transposable, nous avons $(\sigma_{t\geq t_1}S, t_0) = (S, t_1)$. Nous restreignons la précédente condition avec $t \geq t_1$

$$(\max(\alpha(\gamma(t,i)), t_1), 0) \leq \mathcal{B}_S(s) \leq (\beta(\gamma(t,i)), i)$$

Nous souhaitons que la condition impliquée par $[\alpha', \beta', r]$ soit équivalente. Nous pouvons déjà voir que

$$\begin{aligned}
\gamma'(t,i) &= \left\lfloor \frac{t-\beta'(0)}{r} \right\rfloor \\
&= \left\lfloor \frac{t-\beta(0)+\beta(0)-\beta'(0)}{r} \right\rfloor \\
&= \left\lfloor \frac{t-\beta(0)}{r} - K \right\rfloor \\
&= \gamma(t,i) - K \qquad \text{comme } K \in \mathbb{N}
\end{aligned}$$

Maintenant, nous avons dans la seconde requête les conditions suivantes :

$$\begin{aligned}
(\alpha'(\gamma'(t,i)), 0) &\leq \mathcal{B}_S(s) \leq (\beta'(\gamma'(t,i)), i) \\
(\alpha'(\gamma(t,i) - K), 0) &\leq \mathcal{B}_S(s) \leq (\beta'(\gamma(t,i) - K), i)
\end{aligned}$$

Sachant les conditions sur α et β, les conditions sur le premier flux et le second sont exactement équivalentes. D'où le résultat.

■

A.7.2 Lemme de transposabilité de τ

Avant de démontrer le cas positionnel, il nous faut vérifier les effets de la transposabilité sur τ.

Lemme A.2 **Lemme de transposabilité de τ**

Soit S un flux naturellement transposable de t_0 à t_1.
Soit D la constante égale au nombre d'n-uplets présents dans le flux avant le *batch* $(t_1, 0)$. Formellement $D = \tau^{-1}_{(S,t_0)}((t_1,0)^-)$.
Alors, les égalités suivantes sont vraies :

$$\begin{aligned}
\forall n \in \mathbb{N}, \quad \tau_{(S,t_1)}(n) &= \tau_{(S,t_0)}(n+D) \\
\forall t,i \in T \times \mathbb{N} \geq (t_1,0), \quad \tau^{-1}_{(S,t_1)}(t,i) &= \tau^{-1}_{(S,t_0)}(t,i) - D
\end{aligned}$$

Nous avons naturellement, $(S, t_1) = (\sigma_{t\geq t_1}S, t_0)$.
Puis il devient simple d'impliquer que $\tau_{\sigma_{t\geq t_1}S}(n) = \tau_S(n+D)$ de par la nature de D.

$$
\begin{aligned}
\tau^{-1}_{(S,t_1)}(t,i) &= \textstyle\sum_{n=0}^{+\infty} n\mathbf{1}_{[\tau_{(S,t_1)}(n),\tau_{(S,t_1)}(n+1)[}(t,i) \\
&= \sum_{n=0}^{+\infty} n\mathbf{1}_{[\tau_{(S,t_0)}(n+D),\tau_{(S,t_0)}(n+1+D)[}(t,i) \\
&= \sum_{n=D}^{+\infty} (n-D)\mathbf{1}_{[\tau_{(S,t_0)}(n),\tau_{(S,t_0)}(n+1)[}(t,i) \\
&= \tau^{-1}_{(S,t_0)}(t,i) - D - \sum_{n=0}^{D-1} (n-D)\mathbf{1}_{[\tau_{(S,t_0)}(n),\tau_{(S,t_0)}(n+1)[}(t,i)
\end{aligned}
$$

Comme nous prenons un *batch* $(t,i) \geq (t_1,0)$, et $\tau_{(S,t_0)}(D-1) \leq (t_1,0)$, la dernière somme est nulle, d'où le résultat.

■

A.7.3 Le cas positionnel

Soit $j \in \mathbb{N}$, tel que $j \geq K$,

$$
\begin{aligned}
\alpha'(j) &= \max(0, \alpha(j+K) - D) \\
\alpha'(j) + D &= \max(D, \alpha(j+K)) \\
\alpha'(j-K) + D &= \max(D, \alpha(j)) \\
\tau_{(S,t_0)}(\alpha'(j-K) + D) &= \max(\tau_{(S,t_0)}(D), \tau_{(S,t_0)}(\alpha(j)))
\end{aligned}
$$

Cette dernière égalité est vraie grâce à la propriété de croissance de τ. L'invocation des propriétés de la transposabilité de τ nous donnent,

$$
\tau_{(S,t_1)}(\alpha'(j-K)) = \max((t_1,0), \tau_{(S,t_0)}(\alpha(j)))
$$

.

Nous devons maintenant vérifier la même condition sur γ',

$$
\begin{aligned}
\gamma'(t,i) &= \left\lfloor \frac{\tau_{(S,t_1)}(t) - \beta'(0)}{r} \right\rfloor \\
&= \left\lfloor \frac{\tau_{(S,t_0)}(t) - D - \beta'(0)}{r} \right\rfloor \\
&= \left\lfloor \frac{\tau_{(S,t_0)}(t) - D - \beta(0) + \beta(0) - \beta'(0)}{r} \right\rfloor \\
&= \left\lfloor \frac{\tau_{(S,t_0)}(t) - \beta(0)}{r} - K \right\rfloor \\
&= \left\lfloor \frac{\tau_{(S,t_0)}(t) - \beta(0)}{r} \right\rfloor - K \qquad \text{as } K \in \mathbb{N} \\
&= \gamma(t,i) - K
\end{aligned}
$$

Nous pouvons désormais conclure en disant que :

$$\forall t, i \geq \gamma'(t,i) \geq 0, \tau_{(S,t_1)}(\alpha'(\gamma'(t,i)) = \max(\tau_{(S,t_0)}(\alpha(\gamma(t,i))), (t_1, 0))$$

La démonstration est similaire pour β'.

■

A.8 Proposition 5.3.5 : Transposabilité des DSF linéaires

Comme $K = \frac{\beta'(0)-\beta(0)+D}{r} = \frac{d'-d+D}{r}$, la première condition est naturelle,
Comme $\beta'(i) = \beta(i+K) - D \Leftrightarrow c'i + d' = ci + cK + d - D$. Cette condition est vraie si et seulement si $c' = c$. Ainsi, nous avons, $cK = d' - d + D$, si $K \neq 0$, alors $c = r$ (sinon, aucune condition n'est nécessaire).
Comme $\alpha'(i) = \max(B_{t_1}, \alpha(i+K) - D) \Leftrightarrow \max(a'i + b', B_{t_1}) = \max(B_{t_1}, \max(B_{t_0}, ai + aK + b - D)) = \max(B_{t_1}, ai + aK + b - D)$. Sachant que $a \neq 0$, et i suffisamment au dessus de B_{t_1}, nous pouvons déduire que $a = a'$ (cette égalité est aussi trivialement vraie pour $a = 0$). Nous déduisons aussi simplement que $b' = aK + b - D$. Dans le cas $a = 0$, pas d'autres simplifications ne peuvent être faites.

■

A.9 Corollaire 5.3 : Transposabilité pseudo-naturelle des DSF linéaires

Cette démonstration utilise directement les résultats de la propriété 5.3.5.
Nous avons $d + B_{t_1} - d - B_{t_0} + D \in r\mathbb{N} \Leftrightarrow B_{t_1} - B_{t_0} + D \in r\mathbb{N}$. Dans le cas temporel, $t_1 - t_0 \in r\mathbb{N}$ et dans le cas positionnel $\tau^{-1}_{(S,t_0)}(t_1) \in r\mathbb{N}$ (sachant les définitions de B et de D).
Nous avons $K \neq 0$ comme $t_1 \neq t_0$ par définition (de même en positionnel). Alors $c = r$.
Considérons $a \neq 0$, alors $B_{t_1} = B_{t_0} + aK - D \Leftrightarrow aK = B_{t_1} - B_{t_0} + D = rK$. Ainsi, $a = r$.
Considérons $a = 0$, alors $\max(B_{t_1}, b + B_{t_1}) = \max(B_{t_1}, b + B_{t_0} - D)$.

– Si $b > 0$, alors $b + B_{t_1} = \max(B_{t_1}, b + B_{t_0} - D) \Leftrightarrow b = \max(0, b - rK)$, donc $b = 0$ ce qui est absurde.

– Si $b \geq 0$, alors $B_{t_1} = \max(B_{t_1}, b + B_{t_0} - D) \Leftrightarrow 0 = \max(0, b - rK)$ ce qui est vrai pour tout $b \geq 0$

■

B

Linear Road : expression de *Toll* en Astral

Dans ce chapitre, nous indiquons l'expression Astral de la requête *Toll* du *Linear Road Benchmark* (LRB). Les expressions présentées dans ce chapitre sont celles déployées dans Astronef à la traduction en *XML* près.

B.1 Le flux d'entrée

Nous supposons que S est le flux fournit par le *LRB*. Ce flux contient les données de tous les flux disponibles dans le *LRB*. Il est nécessaire de filtrer sur son type ainsi que ne garder que les bons attributs correspondant au flux de positions.

PositionReports	
Flux	vid, speed, xway, lane, dir, seg, pos, τ
$\Pi_{vid,speed,xway,lane,dir,seg,pos,\tau}\sigma_{type=0}S$	
Flux entrant des positions venant des voitures	

B.2 Détection d'accident

Tout d'abord, nous formons la relation permettant de détecter les voitures qui sont arrêtées (la position n'a pas bougé pendant au moins 4 relevés). Puis, nous calculons l'accident qui résulte de l'arrêt de deux voitures au même endroit. Nous utilisons l'agrégat *countd* équivalent au |COUNT DISTINCT| de SQL pour compter le nombre de positions différentes pour une voiture.

StoppedCar	
Relation	vid, pos, dir, xway
$\Pi_{vid,pos,dir,xway}\sigma_{cpos\geq 4 \wedge npos=1 \wedge ndir=1}$ $_{vid,pos,dir,xway}G^{npos,cpos,ndir}_{countd(pos),count(pos),countd(dir)}\sigma_{lane\in[1,3]}PositionReports]T\ 120s\ 60s]$	
Indique les identifiants de voitures qui sont arrêtées	

AccidentInSeg	
Relation	seg, xway
$\Pi_{seg,xway}e^{seg}_{\lfloor \frac{pos}{5280} \rfloor}(StoppedCar \bowtie \rho_{vid2/vid}StoppedCar)$	
Liste les segments qui ont actuellement un accident	

B.3 Calcul de statistiques des segments

Le calcul des statistiques sont des agrégats calculés à la volée concernant le nombre total et la vitesse moyenne des voitures sur un segment. Cela a été analysé dans la section 9.2.4.

CarDensity	
Relation	xway, seg, dir, cars
$_{xway,seg,dir}G^{cars}_{countd(vid)}PositionReports]T\ 60s\ 60s]$	
Indique le nombre de voitures dans chaque segment et pour chaque direction	

AverageVelocity	
Relation	xway, seg, dir, avgspeed
$_{xway,seg,dir}\mathcal{G}^{avgspeed}_{avg(m/avg(vid/avg(speed)))}(e^{m}_{\lfloor \tau/60 \rfloor}PositionReports]T\ 5min\ 1min])$	
Indique la vitesse moyenne des voitures pour chaque segment et chaque direction	

B.4 Calcul de *Toll*

Tout d'abord, nous calculons les changements de segments tel que nous l'avons analysé dans la section 9.2.3.

SegChange	
Relation	vid, speed, xway, lane, dir, seg, pos, τ
$\Pi_{\setminus seg2} PositionReports[B] \underset{seg \neq seg2}{\bowtie} \rho_{seg2/seg} \Pi_{xway,vid,seg} PositionReports[T\ 30s\ 30s[$	
Donne les derniers n-uplets qui indiquent un changement de segment	

Maintenant, nous pouvons procéder au calcul de *Toll*. L'expression étant complexe, nous l'avons séparé en 4. D'abord, nous ajoutons les informations statistiques dans T_a, ensuite nous ajoutons les informations des accidents dans T_b, nous calculons le prix du péage avec les informations que nous avons et enfin, nous formons le flux *Toll*. Nous utilisons l'opérateur $\overset{v}{\bowtie\!\!\!\subset}$ étant équivalent au *LEFT OUTER JOIN*, la valeur nulle n'existant pas dans notre formalisme, la valeur v est utilisé à cet effet. Nous utilisons aussi la syntaxe de l'opérateur ternaire *condition ? vrai : faux* pour calculer le coût du péage.

Résultat intermédiaire T_a	
Relation	vid, speed, xway, lane, dir, seg, pos, avgspeed, cars, τ
$(\sigma_{lane \neq 4} SegChange) \overset{0}{\bowtie\!\!\!\subset} ((\sigma_{avgspeed \leq 40} AverageVelocity) \bowtie (\sigma_{cars \geq 50} CarDensity))$	
Ajout des données statistiques à *SegChange* nécessaires au calcul du péage	

Résultat intermédiaire T_b	
Relation	vid, speed, xway, lane, dir, seg, pos, avgspeed, cars, seg2, τ
$T_a \underset{\substack{dir=0 \wedge seg2-seg \in [0,4] \vee \\ dir=1 \wedge seg-seg2 \in [0,4]}}{\overset{-1}{\bowtie\!\!\!\subset}} (\rho_{seg2/seg} AccidentInSeg)$	
Ajoute l'information d'accident dans les segments environnant	

Résultat intermédiaire T_c	
Relation	dir, seg, xway, toll, vid
$\Pi_{dir,seg,xway,toll,vid} e^{toll}_{(seg2=-1 \wedge avgspeed \leq 40 \wedge cars \geq 50)\ ?\ 2(cars-50)^2\ :\ 0}\ T_b$	
Calcul du prix du péage selon les spécifications de LinearRoad	

Toll	
Flux	vid, toll, avgspeed
$\mathcal{R}^{u}_{\mathcal{S}}(\Pi_{avgspeed,toll,vid}\ T_c \overset{0}{\bowtie} AverageVelocity)$	
Flux indiquant à la voiture *vid* roulant à *avgspeed* doit payer *toll*	


CLOSED
due to lack of datum

Gestion de flux de données pour l'observation de systèmes

Résumé

La popularisation de la technologie a permis d'implanter des dispositifs et des applications de plus en plus développés à la portée d'utilisateurs non experts. Ces systèmes produisent des flux ainsi que des données persistantes dont les schémas et les dynamiques sont hétérogènes. Cette thèse s'intéresse à pouvoir observer les données de ces systèmes pour aider à les comprendre et à les diagnostiquer.

Nous proposons tout d'abord un modèle algébrique Astral capable de traiter sans ambiguïtés sémantiques des données provenant de flux ou relations. Le moteur d'exécution Astronef a été développé sur l'architecture à composants orientés services pour permettre une grande adaptabilité. Il est doté d'un constructeur de requête permettant de choisir un plan d'exécution efficace. Son extension Asteroid permet de s'interfacer avec un SGBD pour gérer des données persistantes de manière intégrée.

Nos contributions sont confrontées à la pratique par la mise en œuvre d'un système d'observation du réseau domestique ainsi que par l'étude des performances. Enfin, nous nous sommes intéressés à la mise en place de la personnalisation des résultats dans notre système par l'introduction d'un modèle de préférences top-k.

Mots-clés : flux de données, observation, algèbre, optimisation de requête, équivalence de requêtes, base de données, données dynamiques

Data Stream Management for Systems Monitoring

Abstract

Due to the popularization of technology, non-expert people can now use more and more advanced devices and applications. Such systems produce data streams as well as persistent data with heterogeneous schemas and dynamics. This thesis is focused on monitoring data coming from those systems to help users to understand and to perform diagnosis on them.

We propose an algebraic model Astral able to treat data coming from streams or relations without semantic ambiguity. The engine Astronef has been developed on top of a service-oriented component framework to enable a large adaptability. It embeds a query builder which can select a composition of components to provide an efficient query plan. Its extension Asteroid interfaces with a DBMS in order to manage persistent data in an integrated manner.

Our contributions have been confronted to practice with the deployment of a monitoring system for the digital home and with a performance study. Finally, we extend our approach with an operator to personalize the results by introducing a top-k preference model.

Keywords : data stream, monitoring, algebra, query optimization, query equivalence, databases, dynamic data