Note sur la gestion de flux de données

Sylvain Ferrandiz

2007D016

Novembre 2007

Département Informatique et Réseaux
Groupe IC2 : Interaction, Cognition et Complexité

Gestion de flux de données

Sylvain Ferrandiz

9 novembre 2007

Table des matières

Introduction

Afin de mener une analyse scientifique des phénomènes qui se produisent dans notre environnement, nous procédons à toutes sortes de mesures. Notre connaissance progresse à mesure que les données récoltées valident certaines hypothèses au détriment d'autres. Les outils et procédés de mesure, de gestion et d'analyse de données forment donc la base à partir de laquelle se développe la compréhension de notre environnement.

Les données sont récoltées, gérées, analysées, suivant un processus souvent standardisé. L'élaboration des outils nécessaires à la mise en œuvre d'un processus de traitement de données a reposé jusqu'à il y a peu sur un modèle "data pull" :

il faut aller chercher les données.

Mais il faut avouer que de nombreux challenges actuels nécessitent de passer à un modèle "data push" :

les données se présentent d'elles-même.

La métaphore du flux se présente alors naturellement pour représenter le mouvement des données. Notons que le modèle "data pull" n'empêche pas la considération de flux de données. C'est plutôt qu'avec ce modèle les données vont rencontrer des barrages sur leur route puisqu'elles devront attendre, à un moment ou un autre, que quelqu'un vienne les chercher.

Dans ce document, nous portons notre attention sur les situations dans lesquelles la rencontre de barrages dans le processus n'est pas souhaitable. Plus particulièrement, nous nous intéressons aux conséquences sur la gestion de données d'un passage à un modèle "data push". Autrement dit, comment faire pour diminuer les résistances au passage de données qui se présentent d'elles-mêmes ?

Le chapitre 1 présente la notion de système de gestion de flux de données (SGFD). Le chapitre 2 est consacré à la description d'un système commercial : StreamBase. Si on s'intéresse plus particulièrement à ce dernier c'est uniquement dans un but didactique. Le chapitre 3 contient une présentation rapide de plusieurs autres systèmes commerciaux : TruViso, Coral8 et Aleri. En annexe, le lecteur trouvera également quelques notes sur l'utilisation d'un système libre : TelegraphCQ.

Chapitre 1

Les systèmes de gestion de flux de données

Les avancées de l'électronique et de l'informatique ont enrichi la pratique de la récolte et de la gestion des données. La constante est l'accroissement des capacités de traitement, tant au niveau de l'acquisition que du stockage ou de l'accès aux données. Mais lorsque l'information doit être extraite instantanément de données récoltées continuellement, le passage par un SGBD traditionnel peut constituer un goulot d'étranglement. Afin de pallier à cette éventualité, de nouveau systèmes ont récemment fait leur apparition : les systèmes de gestion de flux de données ([BBD+02], [GO03]).

1.1 Généralités sur la gestion de flux de données

1.1.1 La gestion de bases de données

Dans les entreprises, on manipule souvent des données ayant la même structure. Prenons l'exemple de la liste des membres du personnel : pour chaque personne, on enregistre le nom, le prénom, le sexe, la date de naissance, l'adresse, la fonction dans l'entreprise, etc. Si elles sont gérées par des moyens informatiques, on dit qu'elles constituent une base de données.

Les Systèmes de Gestion de Base de Données (SGBD ou DBMS) permettent une gestion informatique efficace des bases de données en exploitant la structure des données afin d'éliminer toute redondance. Un langage de haut niveau permet de formuler des requêtes pour interroger le système et manipuler les données. Un SGBD est donc un système autorisant un stockage et un accès optimisés.

La philosophie sous-jacente à l'utilisation d'un SGBD est la suivante :

- les données sont structurées
- les données peuvent être stockées
- le traitement peut être différé
- les requêtes sont éphémères.

1.1.2 Vers la gestion de flux de données

Un flux de données est une suite de tuples ayant tous la même structure. Cette structure est représentée par un *schéma*, comprenant le nom des champs du tuple et leur type. La différence entre un flux et une table est le caractère ordonné des tuples. L'ordre est souvent déterminé par un champ (typiquement la date, mais pas nécessairement) et on parle à son sujet de *champ d'ordre* ou *champ d'agencement* (ordering field). Un exemple de tuples d'un flux est donné à la figure 1.1.

Timestamp	Source	Destination	Bytes	Protocol	Duration
...	...	...	...	...	...
9878	10.211.113.18	137.214.16.29	12	http	2
9879	213.112.34.56	1.0.0.3	36	ftp	8
9880	56.78.54.89	10.9.9.134	128	https	36
...	...	...	...	...	...

FIG. 1.1 – Portion d'un flux.

Gestion de flux par un SGBD. – En présence de flux, si le choix est fait d'alimenter et de maintenir un entrepôt de données, la stratégie employée est la suivante [CCH07] :

- pour chaque flux de données, un ensemble d'agrégats est prédéfini,
- pour chaque flux de données, ces agrégats sont calculés et stockés temporairement et localement,
- un ETL (Extract, Transform, Load) extrait et transforme l'information contenue dans ces agrégats, puis alimente périodiquement l'entrepôt de données.

La figure 1.2 illustre cette stratégie de gestion.

SGFD. – Dans un nombre non négligeable de situations faisant intervenir des flux de données, l'utilisation d'un SGBD ne convient pas ; on entre dans le domaine de la gestion de flux dès lors que

- les données du flux n'ont pas vocation à être stockées,
- les données nécessitent un traitement immédiat,
- les requêtes sont exécutées continuellement.

En lieu et place d'un SGBD, un Système de Gestion des Flux de Données (SGFD) doit être considéré. En plus des fonctionnalités classiques d'un SGBD, celui-ci autorise une gestion et une analyse à la volée et en temps réel de données se présentant continuellement (c.f. figure 1.3). On distingue dans la suite les SGFD *spécifiques*, qui exploite la structure des données, des SGFD *génériques*, qui exploitent le fait que les données sont structurées. Les premiers ont vocation à traiter une problématique et des données particulières (traffic IP, réseau de capteurs). La structure des données est connue a priori et le SGFD est développé

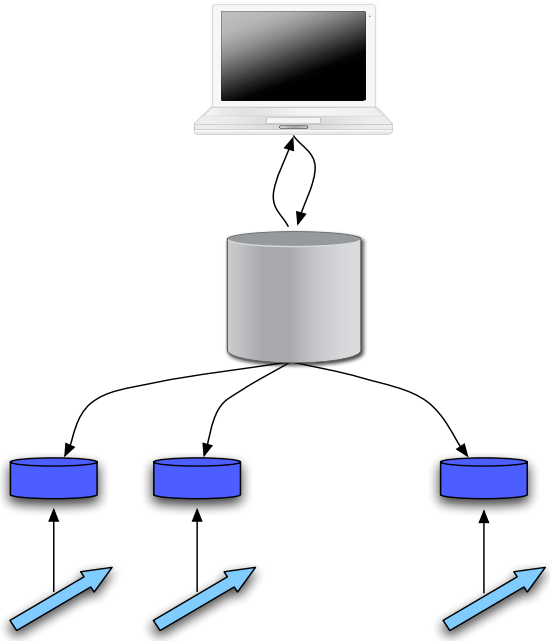

FIG. 1.2 – Sans système de gestion des flux de données. 1. Calcul d'agrégats prédéfinis. 2. Alimentation périodique de l'entrepôt. 3. Exploitation des données de l'entrepôt.

spécifiquement pour traiter ce type de données. Les seconds sont développés pour traiter des données structurées quelconques.

Requêtes continues. – La gestion de flux de données inclut l'expression d'un nouveau type de requêtes : les requêtes *continues*. Celles-ci, au contraire des requêtes *ponctuelles*, ont vocation à suivre l'évolution des flux et à être réexécutées automatiquement. C'est par exemple le cas d'une requête visant à calculer toutes les heures la consommation téléphonique agrégée sur la dernière heure.

Dans un SGFD, les données sont volatiles et les requêtes sont statiques (par opposition avec un SGBD, où les données sont statiques et les requêtes sont éphémères). Ainsi, un SGFD se caractérise par la possibilité d'exécuter une requête de manière continuelle sur des données dont le traitement ne peut être différé.

Réduction de la charge. – Dans un modèle "data push", les données se présentent d'elles-mêmes, à leur propre rythme. Le système de gestion ne connaissant pas à l'avance ce rythme et ses fluctuations, des mécanismes d'adaptation sont à mettre en place pour faire face à toutes les situations. Notamment, lorsque le rythme s'accélère au point que le système ne traite plus instantanément les données, un mécanisme de réduction de la charge (load shedding) procède à un échantillonage afin de réduire le nombre de tuples présents ou entrants dans le système. Le système entre alors dans un mode "dégradé" et le mécanisme doit réaliser un compromis entre niveau de réduction de la charge et certains critères de qualité.

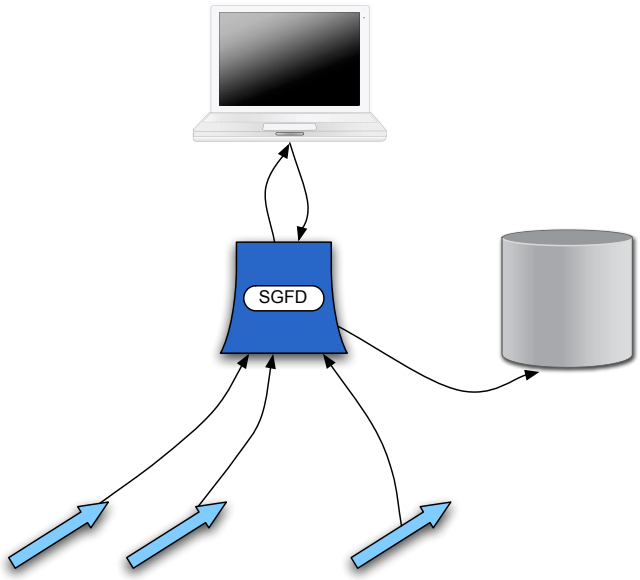

FIG. 1.3 – Avec un système de gestion des flux de données, l'utilisateur interroge directement, instantanément et continuellement les données des flux.

Dans de nombreux domaines, la gestion de flux de données répond à de nouveaux besoins. En voici quelques exemples :

- trading
- gestion de trafic dans un réseau IP
- vérification des SLA (accords de niveau de service) entre opérateurs
- exploitation d'un réseau de capteurs
- analyse de transactions
- analyse de clics
- ...

1.1.3 A quoi ressemble un SGFD générique ?

Architecture d'un SGFD générique. – La figure 1.4 montre l'architecture d'un SGFD générique. Le moteur constitue le cœur du système : il organise et optimise le traitement des données afin de satisfaire efficacement au plan de requête. Les requêtes sont stockées dans un dépôt et une mémoire de travail est allouée. Des adaptateurs font entrer et sortir les données du système (de et vers les applications clientes). Enfin, différentes interface de communication permettent d'interagir avec le moteur (instructions en ligne de commande, par exemple) ou de développer ses propres fonctionnalités.

Le moteur du système. – La qualité d'un moteur repose à la fois sur des caractéristiques fonctionnelles et des caractéristiques techniques. Parmi les premières, on a par exemple la possibilité (ou non) de faire évoluer le plan de requête au cours du temps. On distingue dans ce dernier cas les requêtes *prédéfinies* des requêtes *ad hoc* : les premières

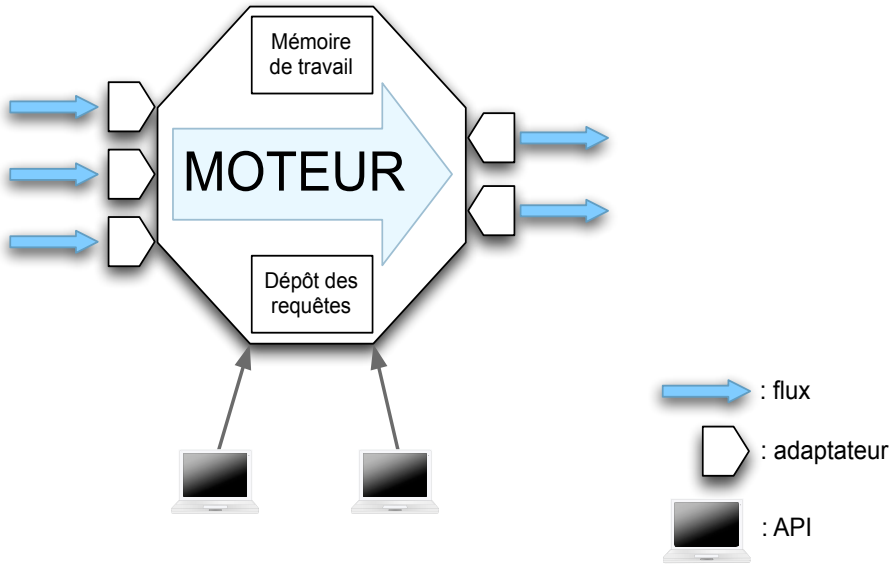

FIG. 1.4 – Architecture référence d'un système de gestion de flux de données.

sont définies avant l'arrivée des flux et les secondes sont ajoutées au fur et à mesure. L'accès à des données partagées, une base de données externe par exemple, constitue une autre fonctionnalité.

Parmi les fonctionnalités possibles, nous choisissons de considérer les suivantes :
- évolution du plan de requête
- accès à des données partagées
- comportement en cas de chute du serveur.

Pour ce qui est de l'analyse technique d'un moteur, elle repose conjointement sur
- le nombre de requêtes du plan
- le débit des données en entrée
- la mémoire de travail consommée
- le temps de traitement des tuples
- la qualité des résultats retournés.

Etant donnée la variété des domaines d'application, un comparatif technique des moteurs semble difficile à réaliser. Il semble plus judicieux d'adapter un point de vue pragmatique et de comparer les SGFD dans un cadre au plus proche de l'application visée. Notons qu'un benchmark existe : Linear Road [ACG+04].

Le fenêtrage. – La dynamicité des flux fait apparaître la notion d'opération bloquante. Ainsi, il est impossible de réaliser une jointure de deux flux. Une manière de résoudre ce problème consiste à considérer des portions finies du flux : les fenêtres. On réalisera une jointure sur les 20 derniers tuples ou sur les dix dernières minutes. La figure 1.5 donne des exemples de fenêtrage d'un flux.

Couramment, le découpage d'un flux en partie finie repose sur la spécification d'une taille de fenêtre et d'un décalage entre deux fenêtres successives. C'est ce qui permet de

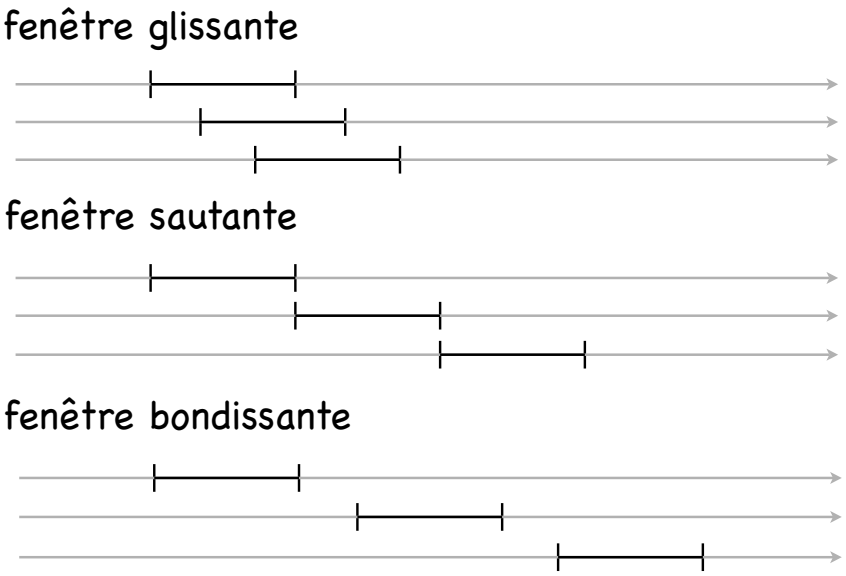

FIG. 1.5 – Exemples de fenêtrage d'un flux.

définir des fenêtres *glissantes* (le pas de décalage est inférieur à la taille de la fenêtre), des fenêtres *sautantes* (le pas de décalage est égal à la taille de la fenêtre) et des fenêtres *bondissantes* (le pas de décalage est supérieur à la taille de la fenêtre). Mais il existe d'autres types de fenêtres, notamment celles à extrêmité fixe.

Le fenêtrage repose sur un découpage du domaine d'agencement des tuples. Ce découpage peut être *logique* et reposer sur la spécification d'un nombre de tuples à considérer dans une fenêtre, ou *physique* et reposer sur la spécification d'un intervalle. Dans la mesure où le domaine d'agencement est généralement temporel, on distingue la datation *explicite* des tuples (le marquage temporel fait partie du schéma du flux) de la datation *implicite* (le marquage temporel est le temps d'arrivée du tuple dans le système).

Le langage d'expression des requêtes. – Comme pour les SGBD, les SGFD utilisent un langage de haut niveau pour l'expression des requêtes. Le langage de définition introduit la notion de flux, analogue à la notion de table dans le cas des SGBD. Le langage de manipulation intègre la notion de fenêtre afin de rendre applicables les opérations d'agrégation et de jointure.

En plus des notions de flux et de fenêtre, le langage est d'autant plus pertinent qu'il permet de réaliser certaines opérations particulières au traitement de flux. De plus, dans le cas où le système autorise la gestion conjointe de tables et de flux, il est intéressant de disposer d'opérateurs faisant le lien entre ces deux types de structures (mise à jour d'une table conditionnellement à l'arrivée d'un tuple dans un flux, par exemple).

On trouve dans [GO03] une liste d'opérations fondamentales pour le développement d'application traitant les flux ("fondamentales" étant à prendre dans le sens "communes à de nombreux besoins d'analyse de flux") :

– sélection : la plupart des applications requièrent des filtrages complexes

- agrégations mixtes : les tendances sont capturées par des agrégats, qu'on souhaite parfois comparer (un minimum avec une moyenne par exemple)
- multiplexage/démultiplexage : les flux sont décomposés pour réaliser des traitements ad hoc séparés puis recomposés
- jointure : de plusieurs flux ou de flux et de données statiques
- fenêtrage
- recherche d'items fréquents : requêtes top-k ou basées sur un seuil
- fouille de flux : reconnaissance de pattern, recherche de similarités, prédiction

Notons qu'on peut étendre les deux dernières opérations et parler de modélisation, ou encore d'extraction d'information au sens large.

Les adaptateurs. – Les adaptateurs constituent la clé de la généricité des systèmes de gestion de flux. Un *adaptateur d'entrée* lit les données provenant d'une source externe et les place dans un flux d'entrée d'une application. Les données peuvent provenir d'un fichier, d'un socket, d'une base de données, etc. Un *adaptateur de sortie* lit les données provenant d'un flux de sortie d'une application, convertit les données dans un format approprié et les envoie à destination.

Un adaptateur fonctionne soit en *interne* soit en *externe*. Dans le premier cas, il fait partie de l'application : il naît et meurt avec la requête. Dans le second cas, il est en dehors du processus. L'avantage de travailler en interne est un gain de performance. Mais si l'adaptateur tombe, il risque d'emporter l'application et le serveur avec lui. En externe, l'adaptateur doit être lancé manuellement, les communications entre lui et le serveur sont un peu plus lentes, mais on gagne en flexibilité : l'adaptateur peut être lancé sur un ordinateur distant et n'occasionne aucun dommage colatéral s'il dysfonctionne.

Les interfaces. – On dispose d'un moteur et d'un langage de spécification de tâches. Un SGFD fournit également un ensemble d'interfaces de communication à travers lesquelles programmer le moteur. Par exemple, le système propose un ensemble d'instructions en ligne de commandes afin de mener différentes tâches d'administration, de lancer les applications, etc. Ou encore, le système peut être muni d'une interface graphique facilitant le développement des applications. Enfin, le système fournit parfois des facilités d'intégration.

Les services de développement. – Un SGFD est d'autant plus à même de répondre à un grand nombre de besoins qu'il peut être enrichi par l'utilisateur. Ce dernier peut souhaiter développer :

- ses fonctions propres (fonctions statistiques, par exemple)
- ses opérateurs propres (module d'analyse de données, par exemple)
- ses adaptateurs (pour se connecter à ses propres sources, par exemple).

Il est donc intéressant de disposer d'aides au développement de tels objets, et même d'aides au développement d'applications clientes. Il convient donc de prendre en compte dans l'évaluation d'un SGFD les possibilités d'extension ainsi que les kits de développement fournis.

1.1.4 Les caractéristiques attendues d'un SGFD (point de vue de [SCZ05])

Dans [SCZ05], les auteurs proposent un lot de 8 conditions que doit satisfaire un SGFD.

Règle 1. – Le système à travers lequel passe le flux doit minimiser la résistance au flux. Ainsi, tout traitement doit pouvoir se passer d'un stockage impératif des données. De plus, le système ne doit pas attendre que certaines conditions extérieures soient réunies pour fonctionner ; il doit être actif.

Règle 2. – Le système exploite une extension du langage SQL pour la gestion des données. La sémantique du SQL standard est étendue en introduisant la notion de fenêtre (portion finie d'un flux) et des opérateurs spécifiques au traitement de flux. De plus, on peut attendre d'un système qu'il permette à l'utilisateur de définir ses propres opérateurs.

Règle 3. – Le système fait face aux imperfections. Avec la notion de flux apparaît de nouveaux types d'exceptions : délai de réception entre deux tuples, arrivée des tuples dans le désordre, retard, manque, accumulation. Dans un système de gestion de flux, il est par exemple impossible d'attendre indéfiniment l'arrivée d'un tuple. Un délai maximal d'exécution doit être adjoint à toute opération potentiellement bloquante.

Règle 4. – Le système assure la reproductibilité des résultats. C'est notamment important dès lors que les tuples arrivent dans le désordre : le résultat doit être le même que s'ils arrivaient dans l'ordre.

Règle 5. – Le système mêle les données statiques et les données dynamiques. Il doit autoriser le stockage, l'accès et la modification de données statiques et permettre leur combinaison avec des données dynamiques. De plus, le langage utilisé doit être uniforme.

Règle 6. – Le système garantit la disponibilité et l'intégrité des données. Ceci est requis par les applications tournant continuellement et devant potentiellement faire face à une chute du système.

Règle 7. – Le système fonctionne de manière distribuée. Les applications sont alors réparties sur plusieurs machines afin de passer à l'échelle. Le système doit également être capable de répartir le traitement sur plusieurs processeurs et de supporter le multi-threading, tout cela de manière automatique, transparente et efficace.

Règle 8. – Le système répond instantanément. Il fournit ainsi des réponses en temps réel en faisant face à un gros volume de données, ce qui nécessite un moteur hautement optimisé.

Commentaires. – Les règles 2, 5 et 7 ont trait aux fonctionnalités des systèmes. La plupart des systèmes respectent (ou annoncent respecter) ces règles. Les autres règles portent sur la performance des systèmes. Ces règles sont binaires (le système répond instantanément ou non) et nous serons plus larges quant à leur interprétation. Les performances

sont en effet fortement dépendantes de nombreux paramètres (nombre de flux, débit des flux, opérations effectuées) et il est certainement préférable d'adopter le point de vue suivant pour chacune des règles : à partir de quel moment les performances du système se dégradent-elles ?

1.1.5 Les caractéristiques attendues d'un SGFD (point de vue personnel)

L'évaluation d'un SGFD repose sur l'évaluation des entités suivantes :
- le moteur de gestion des données et des requêtes
- le langage d'expression des requêtes
- les interfaces de communication avec les clients :
  - entrée/sortie des données
  - modes d'interaction avec le moteur

Evaluation du moteur. – L'évaluation du moteur de gestion se fait suivant deux directions : fonctionnelle (fonctionnalités proposées) et technique (performance). L'évaluation fonctionnelle répertorie les services proposés par le moteur, en réponse à des questions comme :
- le plan de requête peut-il évoluer dynamiquement ?
- une application peut-elle accéder à des données partagées ?
- quelle gestion des pics d'activité ?
- que se passe-t-il si le serveur tombe ?
- ...

Evaluation du langage. – Au niveau de l'évaluation du langage de requête, il est important de répondre à des questions comme :
- quel type de fenêtres peut-on construire ?
- quelles opérations particulières sur les flux le langage permet-il d'effectuer ?
- ...

Evaluation des interfaces de communication. – En ce qui concerne les adaptateurs, la question est simplement : quels sont les adaptateurs disponibles ? Une autre question porte sur les différents modes d'interaction avec le moteur : quels sont les modes disponibles ?

Evaluation des aides au développement. – Des axes secondaires d'évaluation sont également à prendre en considération, comme l'évaluation des services d'aide au développement :
- peut-on développer ses propres fonctions, opérateurs, adaptateurs ?
- dispose-t-on de kits de développement ?
- quelles sont les facilités de développement d'applications clientes ?
- ...

1.2 Tour d'horizon des SGFD

Nous parcourons sommairement ici l'écosystème des SGFD, en présentant des systèmes spécifiques et des systèmes génériques. Il ne s'agit aucunement ici de nous lancer dans une évaluation comparative de tous les systèmes.

1.2.1 Systèmes de gestion de flux de données spécifiques

Nous présentons ici une liste non exhaustive de SGFD spécifiques à un domaine d'application :

- COUGAR (Cornell)
- GigaScope (AT&T)
- Hancock (AT&T)
- NiagaraCQ (OGI/Wisconsin)
- OpenCQ (Georgia Tech)
- StatStream
- Streaminer (UIUC)
- Tapestry (Xerox)
- TinyDB (Berkeley)
- Tradebot
- Tribeca (Bellcore)

COUGAR [DGR+03]. – Le projet Cougar, mené à l'Université de Cornell, vise à programmer les capteurs d'un réseau à l'aide de requêtes exprimées dans un langage déclaratif de haut niveau (variante de SQL). Le réseau est constitué de capteurs non mobiles connectés sans fil. Les contraintes portent sur la capacité de communication (bande passante), la consommation d'énergie et la capacité de calcul. Chaque capteur constitue une source de données structurées. Le réseau contient des nœuds de *visualisation* : les données des capteurs sont poussées vers ces nœuds dans lesquelles elles sont stockées jusqu'à ce qu'elles soient consommées par l'application.

La problématique est de fournir une interface expressive de programmation à l'aide de requêtes tout en assurant une optimisation de la gestion des ressources. L'utilisateur est alors en mesure de modifier dynamiquement le comportement du réseau tout en le préservant au maximum en minimisant l'utilisation des ressources.

GigaScope [CJSS03]. – GigaScope est un système propriétaire (AT&T) orienté gestion de réseaux : analyse de trafic, détection d'intrusion, analyse de configuration des routeurs, etc. Une particularité de GigaScope est l'abandon du modèle fenêtré. L'évaluation d'opérateurs bloquants repose sur une analyse des champs ordonnés des flux et des propriétés de la requête.

Explicitons le modèle adopté. Il repose sur les constatations suivantes :

- les flux possèdent un ou plusieurs champs d'agencement (date de début et date de fin d'un événement, par exemple)
- ces champs vérifient une propriété d'agencement, comme
  - (stricte) croissance/décroissance

  - croissance dans un groupe G (par exemple, la date de début d'une agrégation de paquets IP est croissante dans le groupe { sourceIP, destIP, sourcePort, destPort, protocol})

L'utilisation des champs et propriétés d'agencement permet de rendre non bloquants les opérateurs bloquants. Par exemple, sous la propriété de croissance des champs d'agencement :

- jointure : le prédicat de jointure doit contenir une contrainte sur un champ d'agencement afin de définir une fenêtre jointe
- group-by et agrégation : un champ d'agencement doit apparaître parmi les champs de regroupement. Lorsqu'un tuple arrive pour agrégation dont le champ d'agencement est supérieur à n'importe lequel de ceux présents dans un groupe courant, on peut affirmer que les groupes courants sont fermés et ne vont pas recevoir de mise-à-jour dans l'avenir. Tous les groupes fermés sont envoyés en sortie.

Le langage proposé est GSQL, une restriction du langage SQL augmenté par des opérations spécifiques au traitement de flux. Ainsi, GSQL permet la spécification de sélections, de jointures, d'agrégations et de fusions de flux. Une jointure concerne deux flux uniquement et doit inclure une contrainte sur un attribut d'agencement. Pour une agrégation, l'un des champs de regroupement doit être un attribut d'agencement. L'opérateur de fusion assemble plusieurs flux en un en conservant la propriété d'agencement d'un champ. L'utilisateur peut développer ses propres fonctions.

Au-delà des aspects fonctionnels de GigaScope, les auteurs de [CJSS03] décrivent leur retour d'expérience suite à des tests d'évaluation de performance. Nous les reportons ici car ils nous paraissent avoir une portée générale :

- réduire la charge d'une application est critique pour les performances et le plus en amont est le mieux
- passer par le disque peut porter un coup fatal aux performances
- un plan de requêtes suffisamment complexe nécessitera approximation des résultats et échantillonnage, mais c'est une solution de dernier recours.

Ils ajoutent également, rejoignant [CcC+02], que la mesure de performance d'un SGFD n'est pas la vitesse à laquelle il produit le résultat mais le niveau de charge qu'il peut soutenir avant de faire sauter des tuples. Notons également que, si l'on doit perdre des tuples, il est pertinent de chercher à éliminer les tuples les moins intéressants. Cette fois, les auteurs proposent un autre point de vue que celui donné dans [CcC+02], plus simple à analyser et à implémenter : les tuples intéressants sont ceux produits le plus en aval.

Hancock [CFPR00], [CFP+04]. – Hancock est un langage spécifique développé par AT&T afin de calculer des agrégats (ou : signatures) à partir de flux de données transactionnelles. Basé sur du C, ce langage permet d'écrire et de lire des programmes de manière plus efficace (en temps et en mémoire).

NiagaraCQ [CDTW00]. – NiagaraCQ est un système de gestion de plans de requêtes continues sur des bases de données de l'internet. L'objectif est de gérer un grand nombre de requêtes, de manière dynamique (les requêtes sont ajoutées, éliminées continuellement). L'efficacité de la gestion est assurée par le groupement des requêtes similaires, pour un traitement non redondant.

Le système est distribué et permet de construire des requêtes dans un langage proche de XML-QL sur des bases XML distribuées. Les requêtes sont écrites en XML-QL et se voient adjoindre une information temporelle : date de première exécution de la requête, date d'expiration de la requête et délai séparant deux exécutions de la requête.

OpenCQ. – OpenCQ permet la définition et le calcul de requêtes sur des bases de données persistantes structurées distribuées sur un large réseau (données bibliographiques). Les requêtes ont par exemple pour but de notifier un changement dans la base. WebCQ est une adaptation d'OpenCQ qui permet le suivi de pages web (données "non structurées"). Le langage de requête est un langage orienté objet.

WebCQ ([LPT00]) est un système de détection et de notification des changements de pages web arbitraires. C'est une application fonctionnant en tant que serveur. Elle est composée de plusieurs parties, dont les trois principales sont les suivantes :

- un robot détectant les changements dans les pages web
- un outil de présentation personnalisée des changements
- un service de notification.

L'utilisateur enregistre sa requête (une *sentinelle*) à l'aide d'un formulaire HTML et spécifie le type d'information qu'il souhaite voir surveillée. La sentinelle est soumise au moteur dès lors que l'utilisateur a spécifié une adresse mél pour les notifications. Le moteur lance le robot périodiquement. Lorsque des changements intéressants sont détectés, un courriel de notification est envoyé.

WebCQ résulte d'un travail effectué à Georgia Tech, terminé au début des années 2000. Le soft est disponible gratuitement.

StatStream. – StatStream est un outil de calcul en temps réel d'indicateurs statistiques (corrélation) sur un ensemble de séries temporelles. StatStream prend en entrée un ensemble de flux et détermine les paires de flux dont la corrélation dépasse un seuil fixé par l'utilisateur. La périodicité de calcul de l'indicateur est fixée par l'utilisateur et la taille de la fenêtre sur laquelle est calculé l'indicateur est un paramètre utilisateur. L'intérêt de StatStream repose sur sa capacité à traiter un grand nombre de flux.

Tapestry [TGNO92]. – Le système Tapestry convertit des requêtes SQL statiques en requêtes continues afin de répondre efficacement à la requête originelle lorsque de nouvelles informations sont ajoutées à la base. Par exemple, une requête statique comme "sélectionner les courriels envoyés par telle personne" possède également un intérêt dans le cas où la base de données est dynamique.

Ce système n'a aujourd'hui plus qu'un intérêt historique.

TinyDB. – TinyDB est un système d'extraction d'informations dans un réseau de capteurs. Il fournit une interface de type SQL afin de spécifier les requêtes d'extraction d'information, ainsi que des paramètres spécifiques comme le débit. Pour une requête particulière, TinyDB collecte les données issues des capteurs, les filtre, les agrège et les envoie à destination d'un PC, tout en assurant une exploitation efficace des ressources du réseau.

TinyDB résulte d'un travail effectué à l'Université de Berkeley, terminé en 2003. Le soft est disponible gratuitement.

Tribeca [Sul96]. – L'objectif de Tribeca est analogue à celui de GigaScope : l'analyse du trafic réseau. Le langage est un langage procédural (la requête est structurée comme un graphe acyclique orienté) définissant trois types d'opérations simples :

- qualification : filtre les tuples d'un flux
- projection : recombinaison des champs d'un flux
- agrégation : application d'une fonction à tous les tuples d'un flux retournant une valeur unique,

et différents types d'opérations combinant plusieurs flux :

- multiplexage/démultiplexage : partition et recombinaison d'un flux, afin d'appliquer un traitement à chacune des parties
- fenêtrage : fenêtres glissantes ou sautantes
- filtre sur fenêtres : comparaison de tuples proches dans le temps.

1.2.2 Systèmes de gestion de flux de données généralistes

Nous présentons ici une liste non exhaustive de SGFD généralistes :

- Aleri
- TruViso (aka Aminsight)
- Aurora, Medusa, Borealis (Brandeis, Brown, MIT)
- Coral8
- Nile (Purdue)
- STREAM (Stanford)
- StreamBase
- TelegraphCQ (Berkeley)

STREAM. – STREAM est un SGFD implémenté en C++ et doté d'une interface graphique. Il se base sur le langage de requête CQL, extension de SQL, afin de formuler des requêtes continues sur des flux de données. STREAM manipule des flux, des relations et leurs interactions. Ainsi, CQL établit :

- un langage de requête relationnel, pour opérer sur les relations,
- un langage de spécification de fenêtre pour convertir des flux en relations,
- un groupe de trois opérateurs pour transformer les relations en flux.

STREAM est basé sur un temps logique et, lorsque la datation n'est pas explicite, le système passe automatiquement en datation implicite. STREAM permet l'usage de fenêtres glissantes uniquement. De plus, il est impossible de définir des requêtes ad hoc.

TelegraphCQ/TruViso. – Après une première tentative infructueuse, qui consistait à implémenter un système de gestion en Java [SFMH01], l'équipe du projet TelegraphCQ s'est orienté vers le développement d'une extension du SGBD open source postgreSQL. Les résultats du travail initial sur le requêtage continu ont été réemployés, notamment les *eddies* (pour le routage des tuples et la planification des opérateurs) et les *fjords* (pour les communications inter-opérateurs).

Le système TelegraphCQ est décrit plus en détail dans une prochaine section. L'annexe A contient les notes relatives à son utilisation. TruViso (initialement Aminsight) est la version commerciale de TelegraphCQ. On la présente au chapitre 3.

Aurora/StreamBase. – Le système Aurora, et ses congénères Medusa et Borealis, est décrit plus précisément dans une prochaine section. Sa caractéristique principale est de proposer une interface de développement d'application à l'aide de diagrammes (boîtes et flèches). StreamBase est la version commerciale d'Aurora. On la présente au chapitre 3.

Aleri. – La plate-forme de gestion de flux d'Aleri est une solution commerciale pour la gestion de flux de flux de données. Elle est assez proche de la solution StreamBase et se démarque par l'existence d'un composant optionnel : Aleri OLAP Server. Celui-ci permet une analyse de données multidimensionnelles issues des données collectées dans des flux. Nous présentons Aleri plus en détail dans le chapitre 3.

Coral8. – Coral8 est une solution commerciale, disponible depuis 2003. Une version d'évaluation gratuite est disponible à l'adresse http ://www.coral8.com. Ce moteur exploite le langage CCL (Continuous Computation Language) qui est une extension de SQL. Une spécificité du langage est la possibilité de détection de séquences ou d'ensembles d'événements.

La solution Coral8 supporte une gestion distribuée sur un ensemble de clusters tout en assurant une optimisation des performances des applications distribuées.

1.2.3 Système inductif de gestion de flux de données

Le projet Stream Mill est à l'intersection de la gestion de flux et des systèmes inductifs de gestion de données. Il est porté par le Web Information Systems Lab du Computer Science Department de UCLA. Le système Stream Mill (v2.0) est disponible librement depuis octobre 2005.

Stream Mill utilise le langage ESL (Expressive Stream Language) [LTWZ05], [WZ03]. Ce langage est basé sur SQL. Il permet de définir des requêtes continues sur flux de données et des requêtes ad hoc sur des relations. Le caractère inductif résulte de la possibilité de poser des requêtes de type "fouille de données", i.e. mettant en œuvre des algorithmes de fouille de données.

1.2.4 La gestion d'événements complexes

Evénement complexe. – Un *objet événement* (*c.f.* http ://complexevents.com) est un objet qui représente un événement de manière concrète sur un support de stockage, généralement dans le but d'être traité informatiquement. Par exemple, un objet événement peut être : un ordre d'achat, un courriel de confirmation d'une réservation, un message émis par un capteur RFID.

Un événement *complexe* ou *composite* est une abstraction composée de plusieurs événements, par exemple : le crack boursier de 1929 (une abstraction basée sur plusieurs milliers d'événements, comme les échanges d'action individuelles), le tsunami en Indonésie de 2004 (une agrégation de plusieurs événements naturels), une instruction CPU.

Les applications de suivi d'événement, comme la gestion de la chaîne d'approvisionnement basée sur les puces RFID, la supervision de réseaux de capteurs, diffèrent des

applications traitant des flux de données en ce que la classe des requêtes envisagées est plus riche. Par exemple [DGP+07], l'utilisateur devrait pouvoir établir si un produit frais a passé plus d'une heure au-dessus de 25°C entre le moment où il est sorti de la ferme (événement initial) et le moment où il est arrivé en magasin (événement final). Ou encore, les applications de supervision basées sur l'usage de puces RFID nécessitent la manipulation d'événements composites [GC+07].

Les langages de gestion de flux de données ne sont pas toujours adaptés pour exprimer de telles requêtes. La définition de nouveaux langages est donc envisagée afin de conduire la détection et la gestion d'événements complexes sur des flux. Les projets SASE [GC+07], Cayuga [DGP+07] et SNOOP [AC06] ont vocation à développer ces nouveaux systèmes (citons également ODE [GJS92], plus ancien). Les systèmes Cayuga et SASE sont des systèmes génériques de gestion d'événements complexes. Ils exploitent des langages à la sémantique formelle bien définie.

Le langage Cayuga. – Le langage CEL (Cayuga Event Language) est basé sur une algèbre d'opérateurs projetée sur une syntaxe de type SQL. Toute requête Cayuga possède la forme suivante :

```
SELECT < attributes >
FROM < stream expression >
PUBLISH < output stream >
```

Une expression de type flux est composée en utilisant un constructeur unaire `FILTER` et deux constructeurs binaires `NEXT` et `FOLD`. Le premier sélectionne les événements du flux entrant qui satisfont un prédicat. Appliqué à deux flux S_1 et S_2, `NEXT` combine chaque événement de S_1 avec l'événement suivant S_2 satisfaisant un predicat. Le constructeur `FOLD` a trois paramètres : la condition de choix de l'événement entrant à l'itération suivante, la condition d'arrêt de l'itération et l'agrégat à calculer à chaque étape. Intuitivement, `FOLD` est une version itérée de `NEXT` qui traite des événements composées d'un nombre quelconque d'événements.

Le langage SASE. – Le langage SASE possède également une structure de haut-niveau proche de SQL. La structure globale d'une requête dans ce langage est :

```
[FROM < stream name >]
EVENT < event pattern >
[WHERE < qualification >]
[WITHIN < window >]
[RETURN < return event pattern >]
```

Par exemple, la requête suivante détecte le vol d'un produit : elle rapporte l'élément qui a été enlevé de son étagère et sorti du magasin sans être passé par la caisse :

```
EVENT SEQ(SHELF_READING x, !(COUNTER_READING y), EXIT_READING z)
WHERE x.TagId=y.TagId ∧ x.TagId=z.TagId
WITHIN 12 hours
```

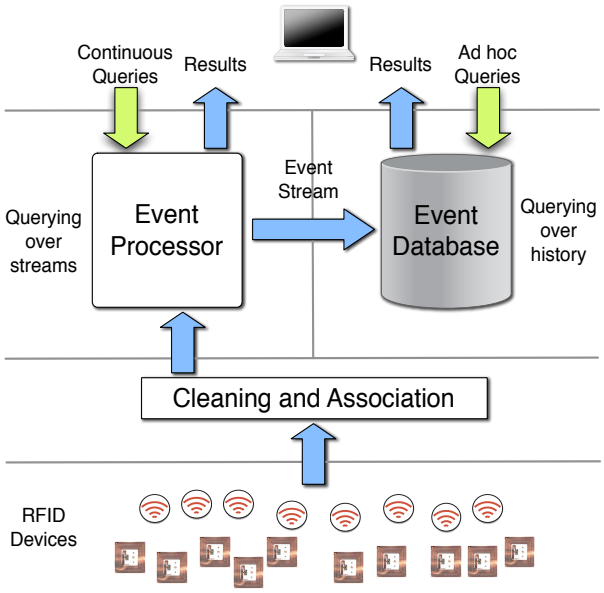

FIG. 1.6 – SASE System Architecture.

```
RETURN x.TagId, x.ProductName, z.AreaId
```

La clause `EVENT` emploie le constructeur `SEQ` qui spécifie la séquence d'intérêt : occurence d'un événement `SHELF_READING` suivi par la non occurrence d'un événement `COUNTER_READING` suivi par l'occurrence d'un événement `EXIT_READING`. La clause `WITHIN` specifie une fenêtre glissante sur les 12 dernières heures. Le langage SASE+ étend le langage SASE [DIG07].

Avantages. – D'après [DGH+06], pour le traitement de données sous forme de flux, les langages Cayuga et SASE+ diffèrent des langages basés sur SQL suivant deux aspects. Tout d'abord, les langages comme CQL peuvent être limités quant au nombre de requêtes qui peuvent être traitées en simultané. La sémantique formelle sous-jacente à Cayuga ou SASE+ autorise une optimisation multi-requêtes au niveau algébrique et pas seulement au niveau informatique, ce qui conduit à de meilleures performances. Ensuite, le paradigme procédural utilisé par Aurora et Borealis ne possède pas de spécification formelle, ce qui limite une fois encore les opportunités d'optimisation.

Notons que le système SASE autorise la gestion de données historisées, ce qui est reflété par son architecture (*c.f.* Fig.1.6).

1.3 TelegraphCQ

Le SGFD TelegraphCQ a été développé dans le cadre d'un projet mené par l'Université de Berkeley. Nous le présentons ici plus à titre d'illustration des concepts introduits dans ce chapitre. Le parti pris par les développeurs de TelegraphCQ est d'étendre le SGBD PostgreSQL afin de poser des requêtes continues sur des flux de données. Le système

hérite donc les caractéristiques de PostgreSQL. On présente plus en détail l'architecture et le fonctionnement de TelegraphCQ, en nous basant sur [KCC+03].

1.3.1 La gestion de flux par TelegraphCQ

TelegraphCQ supporte un langage de définition et un langage de manipulation incluant la notion de fenêtre glissante. Les interactions possibles avec TCQ, en dehors de celles natives à postgreSQL, sont :

- création de flux (archivés ou non),
- création de sources,
- création d'adaptateurs,
- définition de requêtes continues.

Les déclarations `CREATE STREAM` et `DROP STREAM` créent et éliminent une structure de flux de données dans la base. La définition d'un flux requiert l'existence d'un champ de type temporel, à spécifier avec la contrainte `TIMESTAMPCOLUMN`. Voici un exemple de création de flux :

```
CREATE STREAM measurements (
  tcqtime TIMESTAMP TIMESTAMPCOLUMN,
  stationid INTEGER,
  speed REAL
) TYPE ARCHIVED
```

TelegraphCQ attend d'une source qu'elle initie une connexion et lui fournisse le nom du flux. Plusieurs sources peuvent simultanément contacter TelegraphCQ et pousser des données. Aucune autre restriction n'est posée sur le format des données, dans la mesure où c'est l'adaptateur associé, défini par l'utilisateur, qui assurent la gestion des opérations.

Chaque adaptateur consiste en la définition de trois fonctions `init`, `next` et `done`. Ces fonctions sont appelées par TelegraphCQ afin de traiter les données externes. Elles sont enregistrées en utilisant la déclaration standard PostgreSQL `CREATE FUNCTION`.

Les requêtes sont considérées comme continues si elles opèrent sur au moins un flux. De telles requêtes ne peuvent être stoppées que par l'utilisateur. Ce sont des requêtes SQL (sans sous-requêtes) avec une clause fenêtrée : cela rend finie la quantité de données qui entrent dans le champ de traitement de la requête. L'intervalle spécifié doit pouvoir être converti automatiquement dans le type intervalle reconnu par PostgreSQL. Un exemple de requête est le suivant :

```
SELECT    ms.stationid, s.name, s.highway, s.mile, AVG(ms.speed)
FROM      measurements ms, station s
WHERE     ms.stationid = s.stationid
GROUP BY  ms.stationid, s.name, s.highway, s.mile
WINDOW    ms ['10 minutes']
```

Le système TelegraphCQ étend donc le SGBD PostgreSQL en ajoutant à la structure de table la structure de flux, en proposant une gestion fenêtrée des flux, en étendant le langage SQL. Le moteur autorise quant à lui la définition de requêtes continues ad hoc.

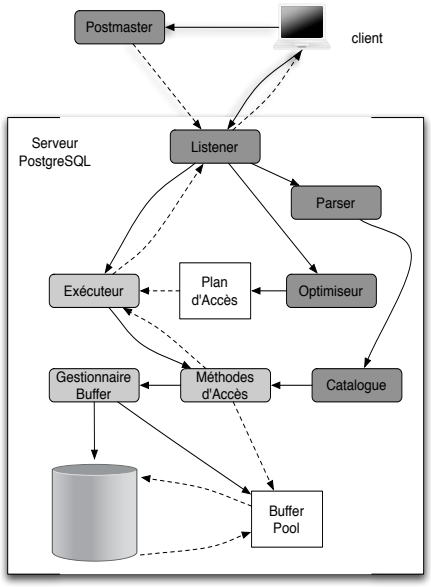

FIG. 1.7 – Architecture PostgreSQL.

Les services de développement d'applications, d'extension du moteur et du langage sont ceux de PostgreSQL.

1.3.2 Le moteur du système

Le système TelegraphCQ est basé sur le SGBD PostgreSQL. L'architecture de ce SGBD est reportée sur la Fig.1.7. Le système repose sur un modèle un processus/une connexion et le *Postmaster* gère les processus serveurs en réponse aux nouvelles connexions. Le *Listener* traite les requêtes des clients et retourne les résultats. Lorsqu'une requête est déposée, elle est lue, optimisée et compilée dans le Plan d'Accès puis traitée par l'Exécuteur.

Les composants légèrement modifiés dans TelegraphCQ sont en gris foncé sur la figure : le *Postmaster*, le *Listener*, le Catalogue Système, le *Parser* de Requêtes et l'Optimiseur. Ceux ayant nécessité de profondes modifications sont en gris clair : l'Exécuteur, le Gestionnaire de *Buffer* et les Méthodes d'Accès.

L'architecture originelle de PostgreSQL a été revue. Le traitement des requêtes continues s'effectue en effet en back-end dans TelegraphCQ. La Fig.1.8 montre l'architecture de TelegraphCQ.

Comme pour PostgreSQL, le Listener traite chaque connexion qu'il reçoit et accepte des requêtes. S'il s'agit d'instructions de définition des données ou de requête sur des tables, l'exécution s'effectue en front-end. S'il s'agit de requêtes continues, elles sont préplanifiées et mises en attente dans une file pour un traitement back-end. L'Exécuteur back-end traite continuellement les éléments de la file et les placent dynamiquement dans l'ensemble des requêtes évaluées. Les résultats des requêtes sont placés dans des files spécifiques à chaque client. A chaque fois que le front-end envoie un traitement en back-end,

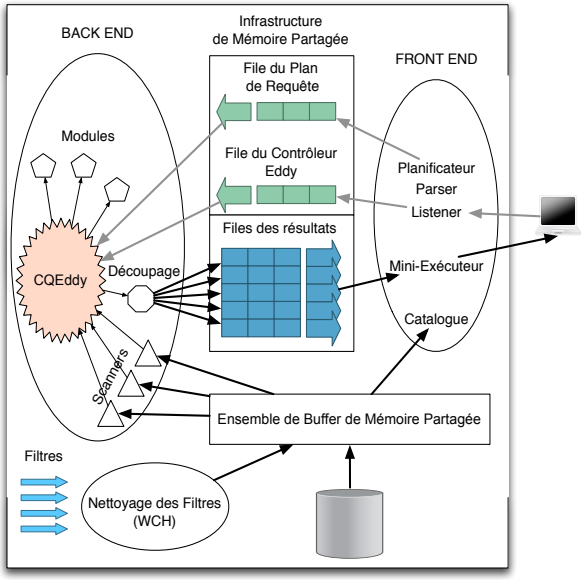

FIG. 1.8 – Architecture de TelegraphCQ.

un plan minimal local est créé et exécuté par le Mini-Exécuteur afin de continuellement traiter les résultats de la file et les transmettre au client.

1.3.3 De la version 0.2 à la version 2.1

La description précédente porte sur la version beta 0.2 de TelegraphCQ. Une version 2.1 a depuis été réalisée. Première différence, la version 0.2 (17 juillet 2003) tourne sous Linux x86 (RedHat, Debian) et MacOS 10.2 alors que la version 2.1 (v2.0 le 1er juillet 2005) tourne sous Linux Fedora Core 1. Les nouveautés annoncées sont les suivantes :

- correction de bogues, meilleure stabilité
- optimisation
- nouvelle sémantique de fenêtrage : fenêtres glissantes, sautantes, bondissantes à l'aide de la clause `RANGE BY '' SLIDE BY '' START AT ''`]
- syntaxe de type SQL99 pour l'opérateur `WITH` et requêtes récursives
- intégration de tuples de ponctuation ("heartbeats")
- les tuples éliminés pour cause de surcharge peuvent donner lieu à un résumé
- Flux.

En dehors de l'optimisation de la performance du code, les apports de la version 2.1 portent donc sur le langage de requêtes, le load shedding, la gestion de tuples de ponctuation et l'intégration de Flux.

Quelles fenêtres ? – Le fenêtrage est opéré par l'ajout d'une clause de la forme `[RANGE BY '' SLIDE BY '' START AT '']` après chaque flux apparaissant dans une requête. Ensemble, ces trois paramètres permettent de spécifier une fenêtre glissante (sliding window), sautante (jumping window) ou bondissante (hopping window).

Load shedding – TelegraphCQ, dans sa dernière version, possède un mécanisme de réduction de la charge. Plusieurs modes de gestion sont possibles et spécifiés lors de la définition du flux, par l'intermédiaire de la clause `ON OVERLOAD`. Par exemple :

```
CREATE STREAM [stream name] TYPE UNARCHIVED ON OVERLOAD [BLOCK/DROP/KEEP]
```

- `BLOCK` : arrêt de la lecture si le traitement ne suit plus
- `DROP` : oubli des tuples
- `KEEP` : conservation d'une synthèse

L'argument `KEEP` doit être complété de la manière suivante :

- `KEEP COUNTS` : conservation du nombre de tuples
- `KEEP REGHIST` : conservation d'un histogramme multidimensionnel régulier
- `KEEP MYHIST` : conservation d'un histogramme multidimensionnel (certainement non régulier)
- `KEEP WAVELET` : conservation d'un histogramme basé sur les coefficients d'une décomposition en ondelettes
- `KEEP SAMPLE` : conservation d'un échantillon réservoir.

Pour les données archivées, TelegraphCQ ne procède à aucun triage et se `BLOCK` simplement.

1.3.4 Version commerciale

Le projet TelegraphCQ est terminé. La dernière version est la 2.1 et date de 2005. Plusieurs participants au projet ont essaimé et fondé Amalgamated Insight en 2005, aujourd'hui Truviso (http ://truviso.com). Le travail effectuée sur TelegraphCQ a été exploité afin de développer un moteur de gestion de flux et de requêtes continues et de proposer des services exploitant ce moteur.

1.4 Le système Medusa-Aurora-Borealis

Les projets Medusa, Aurora et Borealis résultent d'une collaboration entre les universités de Brandeis et de Brown et le MIT. Medusa est un système distribué dont Aurora est le moteur de traitement des requêtes sur chaque site. Borealis étend le couple Medusa-Aurora afin d'optimiser la gestion distribuée.

1.4.1 Aurora

Le SGFD Aurora dispose d'une interface graphique pour la spécification du plan de requête, basée sur le paradigme procédural qui consiste à enchainer boîtes et flèches : les données traversent un graphe orienté sans cycle dont les noeuds sont constitués par les opérateurs. Ce plan résulte de la combinaison de plusieurs opérateurs de base. L'optimisation de ce plan de requête est effectuée dynamiquement. Aurora autorise la définition de requêtes ad hoc et leur optimisation. Le langage utilisé est StreamSQL. Aurora peut également stocker un historique. Enfin, Aurora permet l'abandon de données (load shedding),

effectue une planification en temps réel et propose une introspection (analyse statique et dynamique de la gestion des ressources).

Opérateurs [ACe+03]. – Aurora propose huit opérations primitives afin d'exprimer les requêtes sur des flux. L'opérateur *Windowed* agit sur un ensemble fini de tuples consécutifs d'un flux. Il applique une fonction sur la fenêtre puis avance la fenêtre. *Slide* avance la fenêtre en la décalant dans le sens du flux d'un certain nombre de tuples. *Tumble* est analogue à Slide si ce n'est que deux fenêtres consécutives n'ont aucun tuple en commun. *Latch* est identique à Tumble mais maintient un état interne entre deux calculs sur deux fenêtres distinctes. Enfin, *Resample* autorise une interpolation entre deux tuples du flux entrant.

A côté des opérations de fenêtrage, on trouve les opérateurs agissant sur un seul tuple à la fois. L'opérateur *Filter* laisse passer les tuples d'un flux vérifiant une certaine condition. Un cas spécial d'utilisation est *Drop*, qui élimine des tuples aléatoirement et uniformément, le taux étant un paramètre de l'opérateur. *Map* applique une fonction à chaque tuple d'un flux. *GroupBy* partitionne les tuples provenant de plusieurs flux en les groupant dans un flux suivant qu'ils partagent les mêmes valeurs sur un même ensemble de champs. Enfin, *Join* apparie les tuples provenant de plusieurs flux, à condition que la différence des marqueurs temporels soit bornée.

Optimisation des requêtes continues. – Les opérations sur les flux traitent les tuples à leur arrivée. Une application Aurora est susceptible d'être composée d'un grand nombre de boîtes, de faire face à un débit rapide et d'être modifiée dynamiquement. L'optimisation est effectuée hors-ligne, dynamiquement et emploie les stratégies suivantes :

- insertion ou déplacement des opérations map en amont de l'application, autant que faire se peut, afin de réduire la taille des tuples traités subséquemment,
- combinaison des boîtes, dans la mesure où un examen deux à deux des opérateurs montre que les opérations map et filter peuvent être combinés avec la plupart des opérateurs, au contraire des opérateurs binaires ou exploitant un fenêtrage.
- réagencement des boîtes lorsque deux opérateurs sont commutatifs, selon un modèle de performance. Chaque boîte b a un coût $c(b)$ (défini comme le temps d'exécution espéré pour le traitement d'un tuple) et une sélectivité $s(b)$ (définie comme le nombre espéré de tuples en sortie pour un tuple en entrée). Pour deux boîtes b_i et b_j, avec b_j suivant b_i, le coût de traitement est $c(b_i) + s(b_i)c(b_j)$. Echanger les opérateurs donne une formule analogue et la condition permettant de décider l'ordre des opérateurs est $s(b_i)/c(b_i) > s(b_j)/c(b_j)$.

Un diagramme Aurora se découpe naturellement en sous-diagrammes, connectés en des points de connexion. Chacun des sous-diagrammes peut être optimisé individuellement. L'optimiseur traverse périodiquement les sous-diagrammes et tourne en toile de fond.

Architecture. – La figure 1.9 décrit l'architecture du système Aurora. Les données sources et les données sortant des boîtes sont transmises au routeur, qui se charge de les diriger soit vers les applications clientes soit vers le gestionnaire de stockage. Ce dernier a pour fonction de maintenir la file associée à la boîte et de gérer le buffer. Le répartiteur sélectionne une boîte pour exécution, s'assure du traitement à effectuer et passe la main

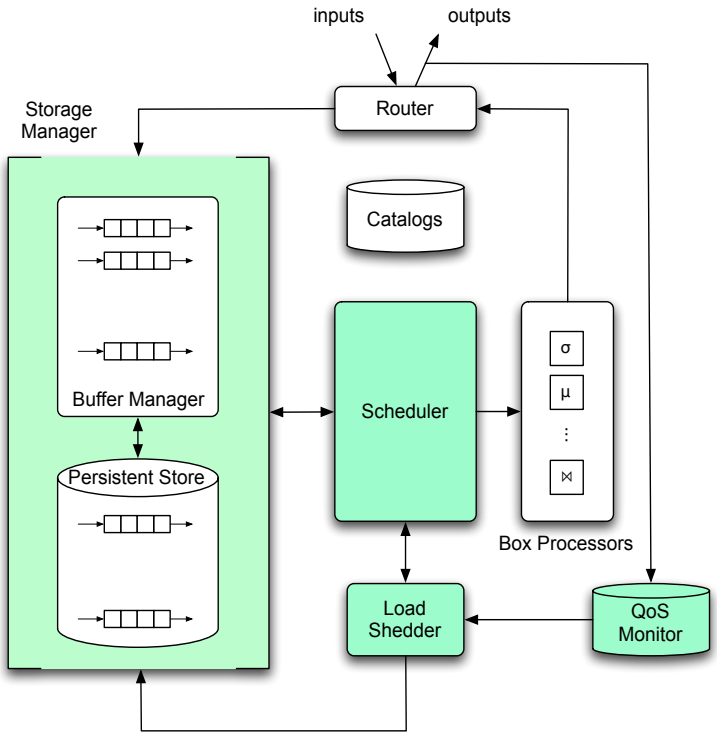

FIG. 1.9 – Architecture du système Aurora.

au processeur. Celui-ci exécute l'opération appropriée et renvoie les tuples de sortie vers le routeur. Le répartiteur s'assure de la prochaine tâche à effectuer et le cycle se répète. Le moniteur de QoS upervise continuellement les performances du système et active le load shedding si nécessaire. Le catalogue contient les informations en rapport avec la topologie du réseau, les entrées, les sorties, la QoS et les statistiques. Il est utilisé par tous les composants.

Qualité de service. – Aurora tente de maximiser la QoS perçue pour les données qu'il produit en sortie. La QoS est une fonction de plusieurs attributs, comme :

- le temps de réponse : la QoS se dégrade à mesure que les délais s'allongent
- l'élimination de tuples : si les tuples sont soumis au load shedding, la QoS des sorties affectées se détériore
- les valeurs produites : la QoS dépend de l'importance des valeurs produites ou non.

La spécification d'une fonction de QoS multidimensionnelle étant impraticable, Aurora adopte la tactique suivante : pour chaque flux de sortie, l'application attend de l'administrateur qu'il fournisse une courbe de QoS pour chaque attribut.

Gestionnaire de stockage – Le travail du gestionnaire de stockage consiste à stocker tous les tuples requis par une application : ceux qui passent à travers l'application et ceux utilisés aux points de connexion. Chaque opérateur fenêtré nécessite le stockage d'un historique des tuples. Le gestionnaire traite un ensemble de files de longueurs variables.

Une file est associée à chaque sortie d'une boîte et partagée par tous les successeurs de cette boîte.

Répartition. – Les décisions de répartition ont pour but de maximiser la QoS globale mais de réduire le coût de traitement des tuples. Aurora applique différentes heuristiques afin de répondre simultanément aux contraintes temps-réel et à la réduction des coûts. Des priorités sont assignées afin de sélectionner les sorties et d'explorer les opportunités de traitement de wagons de tuples.

Train scheduling est un ensemble d'heuristiques qui tentent de générer des files aussi grandes que possibles avant de procéder à un traitement, de traiter des wagons complets de tuples à chaque fois et de passer les tuples aux boîtes suivantes sans avoir à passer sur le disque.

Le temps de latence de chaque tuple en sortie est la somme du temps de traitement et du temps d'attente. Le premier est une fonction du débit d'arrivée des tuples et des coûts liés à chaque boîte, alors que le second est fonction de la répartition. Le but d'Aurora est d'assigner des priorités aux sorties afin d'atteindre un temps d'attente qui maximise la QoS globale.

Introspection. – Aurora effectue une introspection statique et dynamique afin de détecter les situations de surcharge. L'analyse statique détermine si la partie hardware supportant l'application est correctement dimensionnée. Si le taux de production espéré des tuples est $r(d)$ pour chaque source d, l'analyse suivante peut être menée.

Si la boîte b suit directement une source d, alors la quantité de données passant dans b devrait être au moins aussi grande que le débit de la source : $c(b)r(d) \leq 1$. Le calcul du débit de sortie de b donne $\min(1/c(b), r(d)) \times s(b)$. En procédant itérativement, on peut calculer le débit de sortie et les ressources en calcul pour chaque boîte de l'application. On peut alors calculer les ressources agrégées minimales requises par unité de temps $MinCap$. Le système Aurora de capacité C ne peut pas faire tenir la charge si C est inférieur à $MinCap$.

L'analyse dynamique utilise la QoS relative au délai d'obtention des tuples de sortie. Lorsque Aurora fournit un tuple à une application cliente, la courbe de QoS correspondant au délai de service est utilisée pour vérifier que le délai effectif pour ce tuple est à un niveau acceptable. Cette analyse permet au système de détecter la surcharge.

Load shedding. – Lorsqu'un état de surcharge est détecté, Aurora tente de réduire le volume de tuples à traiter. Le compromis réalisé par Aurora repose sur l'information de QoS : la réduction de charge est équilibrée par la QoS.

1.4.2 Medusa

Medusa est un système distribué dont Aurora est le moteur de traitement des requêtes sur chaque site. Medusa s'occupe de la gestion globale des requêtes et des ressources à travers le réseau.

1.4.3 Borealis

Borealis étend le couple Medusa-Aurora et hérite donc les fonctionnalités d'Aurora pour la gestion de flux et les fonctionnalités de Medusa pour l'aspect distribué [AAB+05]. Borealis répond aux attentes suivantes :

- révision dynamique des résultats des requêtes (données manquantes au moment de l'évaluation, données révisées a posteriori),
- modification dynamique des requêtes (paramètres ou structure),
- optimisation distribuée,
- tolérance aux fautes.

1.4.4 Version commerciale : Stream Base Engine

La dernière version d'Aurora est la 1.0.6 et date du 17 avril 2003. Les technologies développées ont servi la commercialisation de la solution Stream Base Engine par StreamBase, société fondée en 2003 entre autres par Mike Stonebraker (http ://www.streambase.com/). Le projet Borealis semblait toujours actif en fin 2006, mais on peut penser qu'il arrive en fin de vie et que les résultats des travaux seront (sont déjà ?) intégrés dans StreamBase. Le chapitre suivant à pour objet la présentation de la solution StreamBase

1.5 Conclusion

Les flux sont de plus en plus présents en pratique et la demande d'outils de gestion de ces flux se fait croissante. Il n'est donc pas étonnant de voir les solutions fleurir. Nous avons essayé dans ce chapitre de donner un aperçu des solutions existantes tout en dégageant les caractéristiques de la gestion de flux qui nous sont apparues importantes.

Nous résumons l'histoire des SGFD en trois temps, non successifs dans l'absolu :

- développement par des entités privées de SGFD spécifiques en réponse à un fort besoin opérationnel (AT&T avec GigaScope et Hancock par exemple) et développement par des entités universitaires de solutions répondant à un besoin spécifique qu'on classe retrospectivement dans la catégorie des SGFD (StatStream, WebCQ par exemple)
- dès l'apparition/la création d'une niche pour les SGFD généralistes
  - montage de projets universitaires afin de défricher le domaine de la gestion de flux (Stream, TelegraphCQ, Aurora, entre autres)
  - développement de solutions commerciales, à partir de ces projets (Aminsight, StreamBase) ou non (Aleri, Coral8)

Si le passage en phase commerciale peut laisser à penser que le domaine de la gestion de flux est arrivé à maturité, l'exploration de domaines comme la gestion de flux d'événements complexes fait apparaître de belles perspectives, avec des applications toujours plus nombreuses et des développements formels toujours plus nécessaires.

Dans la mesure on l'on souhaite évaluer et comparer des solutions génériques, nous nous concentrons dans la suite de ce document sur l'aspect fonctionnel et non la question des performances. Nous regroupons pour cela les fonctionnalités des SGFD génériques dans quatre grandes catégories :

1. connection avec le monde extérieur (entrée/sortie des données)
2. interaction avec le moteur (langage de requête, modes d'interaction)
3. services de tuning (moteur, adaptateur, langage)
4. aide à l'intégration dans les applications, services de développement d'applications.

Chapitre 2

StreamBase

Ce chapitre contient une présentation et une description des fonctionnalités de StreamBase. De plus, il contient un ensemble d'exemples d'applications illustratives. Nous nous intéressons plus en détail à ce système à titre didactique.

2.1 Fonctionnalités

Le système StreamBase est un système générique proposant une extension du langage SQL incluant la structure de flux, la définition de fenêtres et la manipulation de données statiques : StreamSQL. Comme on peut le voir sur la figure 2.1, la solution StreamBase est composée :

- du StreamBase Engine, le moteur de traitement des données et des requêtes,
- du StreamBase Studio, un environnement de développement intégré,
- d'adaptateurs, faisant l'interface entre les données externes et les données internes,
- d'applications clientes,
- d'interfaces de programmation, pour le développement d'applications clientes, d'adaptateurs, d'opérateurs,
- d'un jeu de commandes, pour exécution dans un shell
- d'une interface pour interaction avec des données externes.

En plus du mode "textuel" classique, qui consiste à écrire les requêtes dans le langage StreamSQL, le studio offre la possibilité de définir graphiquement les requêtes (ou : *applications*), en combinant un ensemble de boîtes (les *opérateurs graphiques*) sous forme d'un graphe orienté acyclique par simple drag and drop de ces boîtes. Une fois stocké, un tel graphe d'application n'est autre qu'un fichier XML. Notons que le système permet de dériver le code StreamSQL à partir de la forme graphique si besoin est. Mais ce n'est pas pour autant que le lancement d'une application sous forme graphique nécessite une traduction préliminaire en StreamSQL. Dans les deux cas, graphique et textuel, les applications sont directement traduites en langage de bas niveau.

Fenêtrage. – Dans StreamBase, le mode de gestion des fenêtres est à la fois souple et riche. Ainsi :

- chaque opérateur potentiellement bloquant (agrégation, jointure) dispose de son propre mode de définition d'une fenêtre

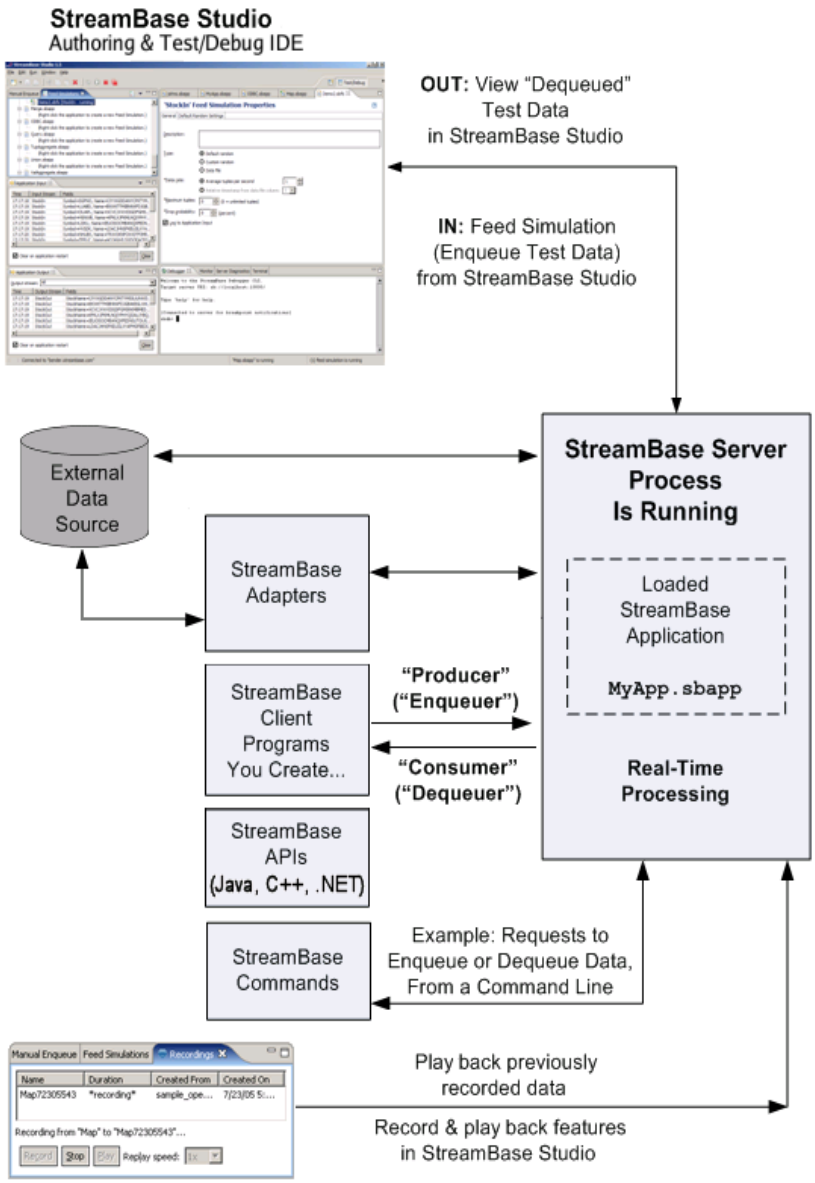

FIG. 2.1 – La solution StreamBase

- une fenêtre peut être définie indépendamment de tout opérateur et tout flux, afin d'être réutilisée
- l'utilisateur peut mettre en place une fenêtre glissante sur un flux indépendamment de tout opérateur.

De plus, dans la plupart des cas, les fenêtres peuvent être logiques, basées sur le temps système ou basées sur un champ d'agencement (usuellement un timestamp mais pas nécessairement).

Lignes de commande. – StreamBase dispose d'un ensemble d'instructions exécutables en ligne de commande. Les principales sont :

- sbd : lancement/arrêt et paramétrage du processus serveur
- sbadmin : lancement/arrêt des applications
- sbc : entrée/sortie des données

D'autres fonctionnalités sont disponibles en ligne de commande, comme le debugger, le simulateur qu'on retrouve dans le Studio ou encore un générateur de client Java, un gestionnaire de fonctionnement distribué.

Container. – Le moteur accepte l'ajout ou la suppression de requêtes (on parlera ici d'applications) au cours du temps. La notion de container permet de spécifier et séparer les ressources nécessaires au fonctionnement de chacune d'entre elles. La correspondance est nécessairement univoque : un container $\Longleftrightarrow$ une application. Un exemple de fonctionnement en ligne de commande est le suivant :

```
> sbadmin addContainer c1 FirstApp.sbapp
> sbadmin addContainer c2 SecondApp.sbapp c2.InputStream = c1.OutputStream
```

Comme l'exemple le suggère, l'utilisation de containers permet de connecter des applications entre elles, un flux de sortie de l'une alimentant un flux d'entrée de l'autre.

Mode HA (High-Availability). – La commande sbadmin peut être paramétrée pour qu'une application continue à tourner sur une machine de secours lorsque la machine de base tombe. Ce mode de fonctionnement n'est disponible que dans la version Entreprise du soft.

Il consiste à diriger tous les tuples vers les deux machines à la fois, la seconde possédant la copie exacte de l'application tournant sur la machine principale. La machine secondaire ne traite pas les données entrantes et se contente de recevoir des messages envoyés par la machine principale. Ces messages lui permettent de ne conserver que les tuples n'ayant pas encore été complètement traités sur la première machine.

La commande sbd dispose d'une option de relance du serveur ayant chuté. Cela permet de repartir à partir du dernier message sauvegardé.

2.1.1 Le langage StreamSQL

StreamSQL est une extension du langage SQL pour la définition et la manipulation de flux. En plus du mode textuel, nous avons vu que StreamBase propose également un mode graphique, qui repose sur la combinaison d'opérateurs graphiques. La correspondance entre les opérateurs StreamSQL et les opérateurs graphiques n'est pas bijective : les opérateurs graphiques sont parfois la matérialisation graphique d'un opérateur StreamSQL mais peuvent aussi correspondre à une combinaison d'opérateurs SQL, comme nous allons le voir. Néanmoins, nous n'effectuerons plus la distinction dans la suite et parlerons simplement d'opérateurs, dans tous les cas.

Le langage de définition. – StreamSQL augmente le langage SQL en fournissant des instructions de définition de flux et de fenêtres :

- `CREATE INPUT STREAM` : définition des champs d'un flux entrant
- `CREATE OUTPUT STREAM` : définition d'un flux sortant (plusieurs syntaxes possibles)
- `CREATE STREAM` : définition d'un flux interne à l'application

- `CREATE TABLE` : définition d'une table
- `CREATE INDEX` : création d'un index
- `CREATE WINDOW` : définition d'une fenêtre
- `CREATE MATERIALIZED WINDOW` : définition d'une fenêtre avec stockage des tuples (en mémoire ou sur disque)
- `CREATE METRONOME` : émission périodique de tuples
- `DECLARE` : définition d'une variable dynamique.

Le modèle de création de fenêtre est le suivant :

```
CREATE WINDOW Window_Id (Window_Specification) ;
```

La spécification de la fenêtre se fait au format suivant :

```
SIZE size ADVANCE increment
TIME | TUPLES | ON identifiant_champ_1
```

`SIZE` spécifie la taille de la fenêtre en nombre de tuples, en secondes où selon les valeurs d'un champ. Si `TIME`, c'est le temps du système qui est considéré ; si `TUPLE`, c'est le nombre de tuples qui caractérise une fenêtre ; enfin, `ON identifiant` permet de se baser sur un champ, `size` et `increment` étant interprétés comme des valeurs du champ.

Le langage de manipulation. – StreamSQL augmente les fonctionnalités des opérateurs de manipulation SQL et propose des opérateurs spécifiques aux flux :

- `DELETE` : suppression d'une ligne d'une table conditionnellement à l'évolution d'un flux
- `INSERT` : insertion d'une ligne dans une table existante par des données d'un tuple d'un flux
- `INSERT/UPDATE` : insertion/ mise à jour sélective d'une ligne d'une table pré-existante par des données d'un tuple d'un flux
- `REPLACE` : suppression et remplacement d'une ligne d'une table pré-existante par des données d'un tuple d'un flux
- `TRUNCATE` : vide une table conditionnellement à l'arrivée d'un tuple dans un flux
- `UPDATE` : mise à jour sélective d'une ligne d'une table pré-existante par des données d'un tuple d'un flux
- `APPLY` : inclusion d'un diagramme StreamBase, d'un module StreamSQL, d'un opérateur Java, d'un adaptateur embarqué
- `BSORT` : agencement en plusieurs passes (paramètre utilisateur) sur un buffer
- `GATHER` : combinaison de plusieurs flux avec test d'identité
- `HEARTBEAT` : ajout de tuples "bouche-trous" dans un flux existant
- `MERGE` : fusion de deux flux, avec garantie sur l'ordre de sortie
- `SELECT` : extension de l'opération SQL
- `UNION` : fusion de plusieurs flux, sans garantie sur la sortie.

Les opérateurs graphiques. – StreamBase encapsule certains traitements récurrents dans des opérateurs accessibles dans le Studio. Dans certains cas, il s'agit simplement de la version graphique d'un opérateur StreamSQL. La palette des opérateurs graphiques est donnée à la figure 2.2 :

- `Filtre` (1/1) : sélection des tuples entrants par application de tests

FIG. 2.2 – Palette des opérateurs graphiques.

- `Application` (1/1) : application d'une fonction à chacun des tuples d'un flux entrant et/ou ajout, suppression, modification d'un champ
- `Join` (2/1) : jointure fenêtrée de deux flux
- `Aggregate` (1/1) : calcul de fonctions agrégatives sur une fenêtre
- `Split` : distribution des tuples d'un flux vers différents fils d'exécution, avec contrôle sur l'ordre des fils
- `Metronome` (0/1) : émission périodique d'un tuple, basée sur le temps système
- `Heartbeat` (1/1) : émission périodique d'un tuple (soit un tuple entrant, soit un tuple "bouche-trou")
- `BSort` (1/1) : tri approximatif
- `Gather` (n/1) : assemblage des tuples partageant la même clé
- `Merge` (2/1) : fusion de deux flux de même schéma, avec spécification d'un champ d'agencement
- `Union` (n/1) : fusion de plusieurs flux
- `Java Operator` : application d'un opérateur développé en Java

Nous présentons plus en détail certains opérateurs.

Jointure. – La jointure consiste à apparier les tuples de exactement deux flux satisfaisant une condition. Pour que cette opération ne soit pas bloquante, il est nécessaire de travailler sur des parties finies des deux flux. Dans StreamBase, l'utilisation d'une jointure repose sur l'hypothèse que les tuples arrivent dans l'ordre.

StreamBase propose deux types de jointure :

- par tuple
- par valeur.

Dans une jointure par tuple, une fenêtre logique est spécifiée pour chacun des deux flux. La taille des buffers est ainsi contrôlée par l'utilisateur. Dans une jointure par valeur,

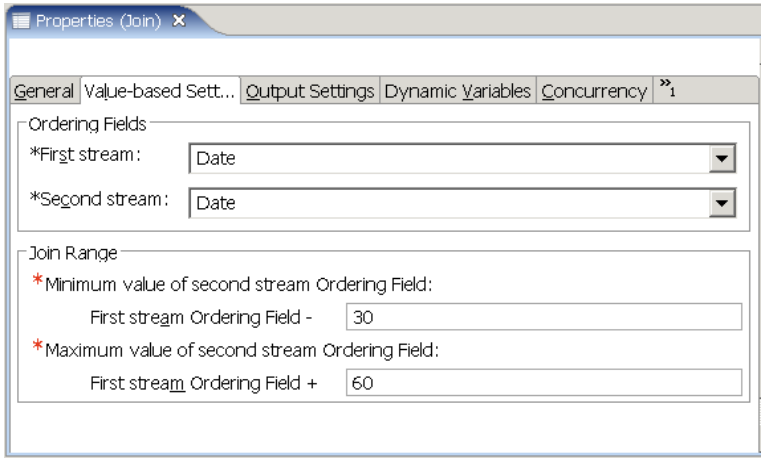

FIG. 2.3 – Paramétrage de l'opérateur de jointure basée sur les valeurs. Les schémas des deux flux possèdent ici chacun un champ Date, qu'on utilise comme champs d'agencement pour la jointure. Une différence minimale (30) et une différence maximale (60) entre ces deux champs sont spécifiées. Dans les deux cas, les différences sont calculées relativement au premier flux : un tuple du second flux ne peut être joint avec un tuple du premier flux que s'il le précède de 30s au plus ou s'il le dépasse de 60s au plus.

un champ d'agencement est spécifié pour chacun des deux flux. Une contrainte est ensuite fixée sur l'écart maximal entre les valeurs de ce champ pour deux tuples à apparier (les champs d'agencement doivent être de type comparable). Le paramétrage d'une jointure par valeur est illustré à la figure 2.3.

Gather. – Cet opérateur reçoit au moins deux flux et concatène les tuples possédant une clé commune. Il est souvent employé lorsque les tuples d'un flux ont été séparés pour subir des traitements différents et doivent être réassemblés. Un exemple de fonctionnement est donné à la figure 2.4. Les tuples entrants sont bufferisés pour chaque valeur du champ clé et un tuple est émis dès que la valeur du champ clé a été vue sur chaque entrée. Dans le cas où il n'est pas certain qu'une valeur de clé apparaisse sur chaque entrée, il est possible de spécifier un champ d'agencement et un time-out sur ce champ afin d'éviter une explosion de la taille du buffer.

Merge. – Cet opérateur, dont un exemple de fonctionnement est donné à la figure 2.5, fusionne les tuples de (exactement) deux flux entrants partageant le même schéma. Les tuples de chaque flux entrant sont bufferisés et un tuple est émis quand un nouveau tuple de l'autre flux entre avec une valeur du champ d'agencement plus élevée. Contrairement à l'opérateur Union, le flux de sortie est ainsi ordonné.

BSort. – Cet opérateur rétablit un ordre sur l'un des champs du flux entrant. Les tuples entrants sont bufferisés et, lorsque le buffer est plein, les tuples dont la valeur du champ à ordonner est minimale sont émis. La fenêtre peut être logique ou basée sur les valeurs d'un champ spécifié (physique). Un exemple de fonctionnement est donné à la figure 2.6,

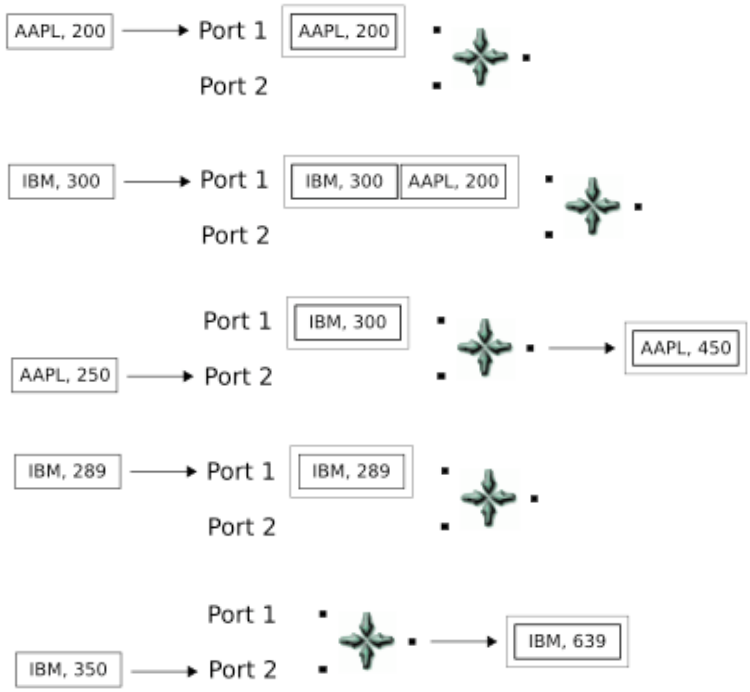

FIG. 2.4 – Exemple de passage de tuples dans l'opérateur Gather.

2.1.2 Les données persistantes

StreamBase permet de travailler avec des données persistantes. Celles-ci prennent différentes formes :

- les tables externes (JDBC table).
- les tables internes (query table)
- fenêtres matérialisées.

L'accès à ces données se fait par l'intermédiaire de l'opérateur graphique `Query`. Le mode de fonctionnement est le suivant : à chaque fois qu'un tuple entre dans l'opérateur, une requête est déclenchée, effectuant une lecture, une écriture, une suppression, etc. Plusieurs de ces opérateurs peuvent être associés à une même structure mais un opérateur ne peut être associé qu'à une seule requête.

Tables externes. – Une table externe est une structure autorisant l'accès à une source de données persistente, externe, à travers une interface JDBC. La source doit être déclarée dans le fichier de configuration du projet. Dès lors, pour utiliser une table externe dans une application, il suffit de choisir la source de données parmi la liste des sources pré-spécifiées. L'opérateur `Query` permet alors de poser une requête SQL sur cette table (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc). Le langage utilisé est le langage natif de la source de données, pas celui de StreamBase.

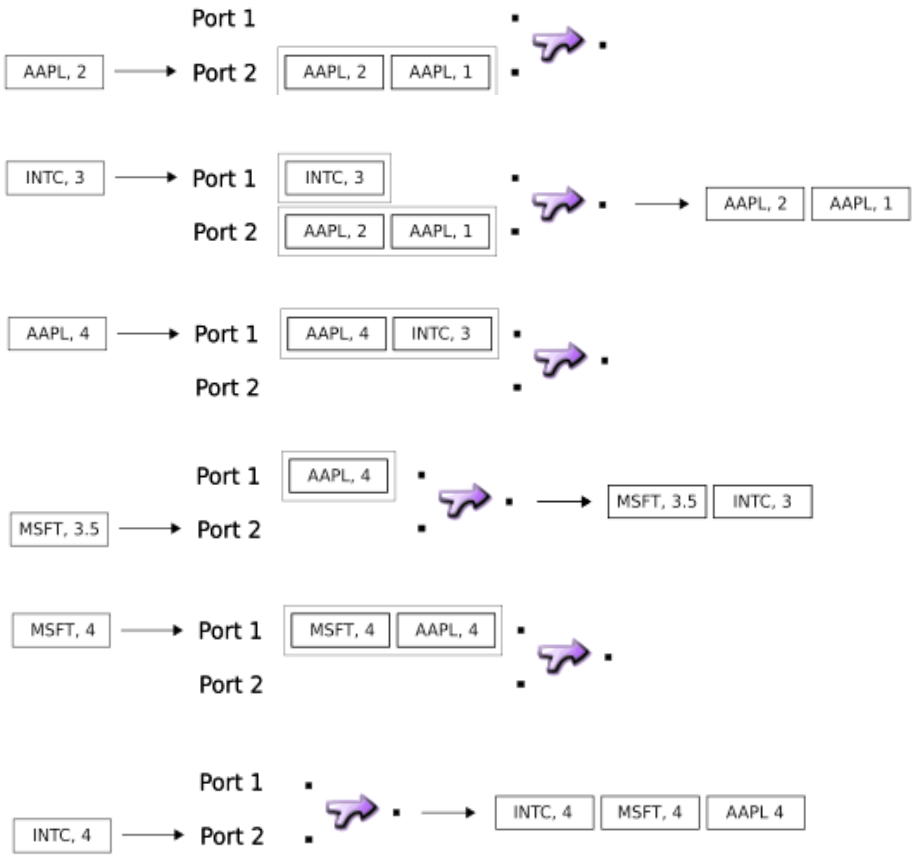

FIG. 2.5 – Exemple de passage de tuples dans l'opérateur Merge.

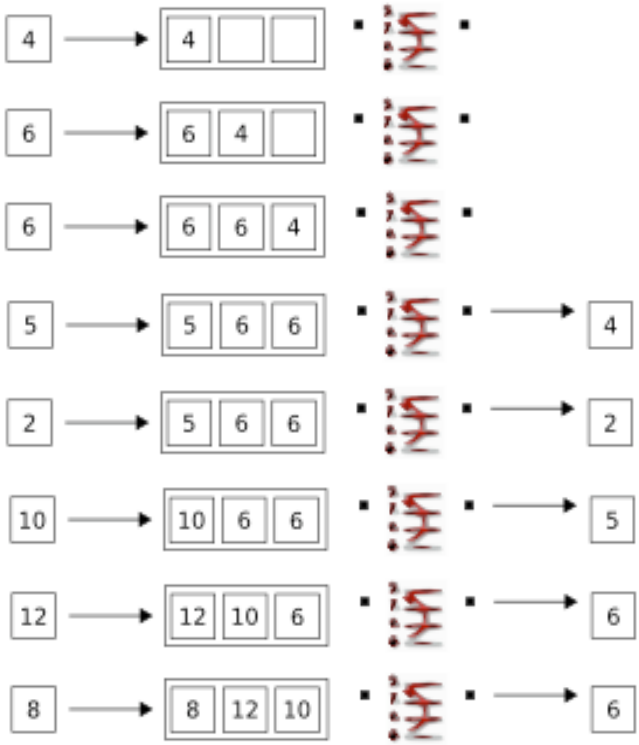

FIG. 2.6 – Exemple de passage de tuples dans l'opérateur BSort.

Tables internes. – Une table interne stocke des tuples en dehors de tout flux, ce qui permet de maintenir un état à l'intérieur d'une application StreamBase. Une telle table peut résider sur disque ou en mémoire. L'opérateur `Query` permet la lecture, l'écriture et la mise à jour de la table.

Fenêtres matérialisées. – Les fenêtres matérialisées permettent à l'utilisateur de manipuler une parte finie d'un flux en dehors de tout opérateur. Une telle fenêtre est définie soit par un nombre de tuples, soit par un intervalle de temps, soit par la valeur d'un champ. Dans le premier cas, le nombre de tuples ne peut dépasser une valeur fixe. Dans le second cas, la fenêtre contient les tuples arrivés durant les dernières s secondes. Autrement dit, les tuples sont ordonnées suivant le temps d'arrivée dans le système. Enfin, le dernier cas est analogue au deuxième, si ce n'est que l'ordre considéré est celui d'un champ d'agencement spécifié par l'utilisateur.

Une telle fenêtre est alimentée par un flux et l'opérateur `Query` permet un accès en lecture uniquement.

2.1.3 Les adaptateurs

Rappelons que si un adaptateur fait partie de l'application, il est qualifié d'interne, par opposition à externe. Dans le premier cas, il naît et meurt avec la requête. Dans le second cas, il est en dehors du processus.

Dans StreamBase, les adaptateurs d'entrée embarqués fournis sont les suivants :

- lecteur de fichier CSV
- lecteur de données CSV sur un socket
- lecteur de fichiers d'expression régulière
- lecteur d'expression régulière sur un socket
- adaptateur StreamBase-To-StreamBase
- adaptateur de souscription aux services Reuters (financier)
- adaptateur EMS/JMS.

Les adaptateurs de sortie embarqués fournis sont les suivants :

- émetteur de courriels
- écriture de fichier CSV
- écriture de fichier XML
- émetteur de tuples vers un serveur http.

Les adaptateurs externes fournis sont les suivants :

- pour Excel (entrée/sortie)
- pour JDBC (entrée/sortie)
- pour EMS/JMS (entrée/sortie)
- pour Reuters RFA (sortie) (financier)
- pour Reuters SFC (entrée) (financier)
- pour REDIPlus (service de courtage, Goldmann/Sachs)
- pour Rendezvous (entrée/sortie, logiciel de gestion de messages).

2.1.4 Les API

StreamBase fournit différentes API pour le développement d'applications ou pour l'extension des fonctionnalités :
- API Java : développement d'applications clientes, d'opérateurs propres, de fonctions propres, d'outils de surveillance
- API C++ : développement d'applications clientes, de fonctions propres
- API .NET (sous Windows uniquement) : développement d'applications clientes.

2.2 Un problème de consommation électrique

On prend pour cet exemple des données de consommation électrique. Un compteur électrique délivre un index de consommation croissant avec le temps. La consommation électrique sur une période de temps s'obtient par différence de deux index. Dans l'application ici envisagée, nous allons :
- lire les données à travers un socket
- agréger les données afin d'obtenir une consommation horaire
- filtrer les consommations dépassant un certain seuil
- envoyer un courriel lorsqu'il y a dépassement et stocker le tuple associé dans un fichier.

Les données. – On dispose de données relatives à un compteur électrique. Le schéma est composé des cinq champs :
- identifiant du compteur
- type de compteur
- type du contrat souscrit par le client
- index relevé
- date du relevé.

Lire sur un socket. – Nous utilisons l'adaptateur de lecture de données CSV sur un socket fourni avec StreamBase. Ses paramètres sont l'adresse IP du serveur de données et le port sur lequel se connecter, le caractère séparateur de champs, le délimiteur de chaînes de caractères et le format du marquage temporel.

Le serveur de données est un programme Java qui émet sur un port spécifié et envoie les tuples contenus dans un fichier un par un, à une vitesse donnée. Les paramètres du programme sont l'adresse du fichier, le port, et la durée entre deux émissions de tuples. Si ce programme s'appelle SimpleServer et se trouve (SimpleServer.class et SimpleServer.jar) dans le dossier CustomLibraries de l'espace de travail, la commande

```
java SimpleServer ..\Resources\Data.txt 9000 2
```

met l'application en émission sur le port 9000 sur la machine résidente pour servir les données du fichier Data.txt à la vitesse d'une ligne toutes les 2 ms.

Composition de l'application. – L'adaptateur de lecture de données CSV sur un socket est paramétré de manière à travailler sur le port 9000. Il est important de noter

que les données doivent être au format adéquat et notamment que les champs doivent apparaître dans l'ordre spécifié au moment du paramétrage de l'adaptateur.

Les données sont ensuite agrégées : sur une fenêtre de taille une heure sautante d'une heure, la différence entre les valeurs relevées initiale et finale est calculée. On obtient ainsi la consommation horaire pour chaque heure. On pose alors un filtre chargé de détecter si la consommation dépasse un certain seuil (ici 69, afin de limiter le nombre de courriels émis par la suite). Les tuples passant le filtre sont stockés dans un fichier à l'aide de l'adaptateur d'écriture dans un fichier CSV. Cet adaptateur est paramétré par le nom du fichier destination.

Les tuples filtrés se voient adjoindre deux champs supplémentaires : un champ "Objet" et un champ "Corps". Le premier contient le titre, le second le corps, du courriel qui va être émis par l'adaptateur d'émission. Celui-ci est paramétré par le nom du serveur sortant, l'adresse mél de l'émetteur, les adresses mél des receveurs. Il est tout aussi possible de définir le mél "en dur" et d'éviter l'étape précédente d'ajout des champs "Objet" et "Corps", bien qu'on perde ainsi en souplesse.

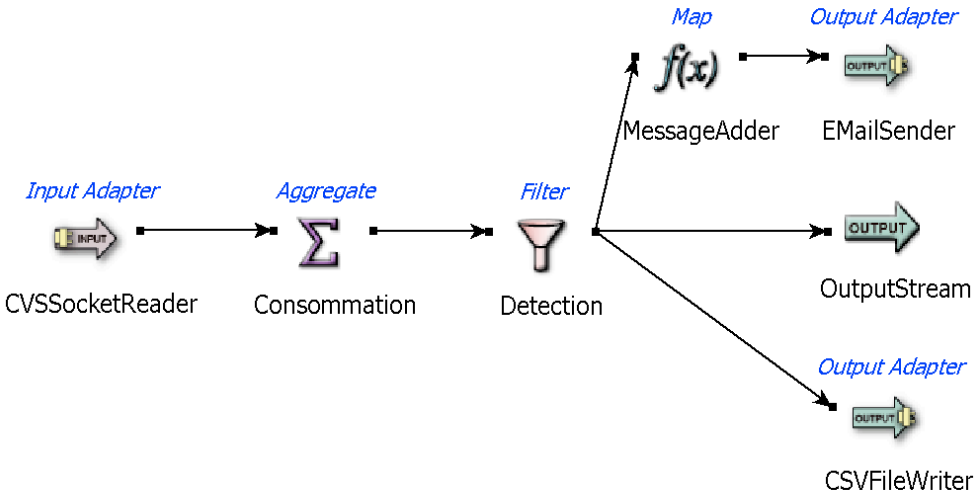

FIG. 2.7 – Diagramme de l'application relative à la détection de dépassement de consommation électrique.

2.3 Un problème de prédiction de consommation téléphonique

On prend pour cet exemple des données relatives à un centre d'appels. Elle correspondent à un problème de prédiction du nombre d'appels entrants. Le nombre d'appels est cumulé sur chaque demi-heure d'une journée ouvrée, de 8h30 à 17h30. Par exemple, la quantité associée à 8h30 est le nombre d'appels entre 8h30 et 9h. On dispose de données allant de début septembre 2006 à fin février 2007. On dispose ainsi de 2242 tuples.

Les données. – Le schéma des données est composé des champs :

- identifiant du flux d'appels entrants
- jour de la semaine (lundi à vendredi)
- date
- heure
- position de la semaine dans le mois
- timestamp
- nombre d'appels sur la demi-heure
- prédiction du modèle.

Un modèle a été estimé sur les 1558 premiers (début septembre à fin décembre) et validé sur les 684 restants (début janvier à fin février). C'est un modèle linéaire basé sur un ensemble de 144 facteurs.
Dans l'application ici envisagée, nous cherchons à détecter la dérive de la prédiction, ce qui se décompose en les tâches suivantes :

- lire les données à travers un socket
- lire les coefficients du modèle à travers un socket
- joindre les données avec le modèle de prédiction
- calculer la prédiction et filtrer
- envoyer un courriel d'alerte signalant un écart important entre la prédiction et la réalité.

Composition de l'application. – Le diagramme de l'application est donné à la figure 2.8. Nous avons déjà vu, dans l'exemple précédent, comment récupérer des données à travers un socket et comment envoyer un courriel à l'aide des adaptateurs adéquats.

D'une part, les coefficients du modèle sont stockés dans une fenêtre matérialisée, de taille 1. Ainsi, à chaque fois qu'un nouveau modèle est estimé par ailleurs, ses coefficients viennent remplacer les coefficients précédemment stockés. D'autre part, le schéma du flux de données est augmenté afin de contenir les facteurs intervenant dans le modèle.

Une requête sur données partagées est ensuite posée afin de joindre les données du flux et les coefficients du modèle. Un filtre effectue ensuite le calcul de prédiction, qui ne contient que des multiplications de champs et des additions (modèle linéaire), pour le comparer à la valeur réelle et détecter un écart trop élevé (supérieur à 5% par exemple). Un courriel est envoyé en cas de dépassement.

2.4 Conclusion

StreamBase est un SGFD disposant d'un mode de définition des requêtes basé sur des diagrammes. C'est donc un très bon système pour s'initier à la gestion de flux et mieux en comprendre les spécificités et chausse-trappes, d'autant plus qu'il est facile à prendre en main. De plus, c'est un système disposant de fonctionnalités riches, ouvrant de nombreuses possibilités d'utilisation du système et en faisant un vrai système générique.

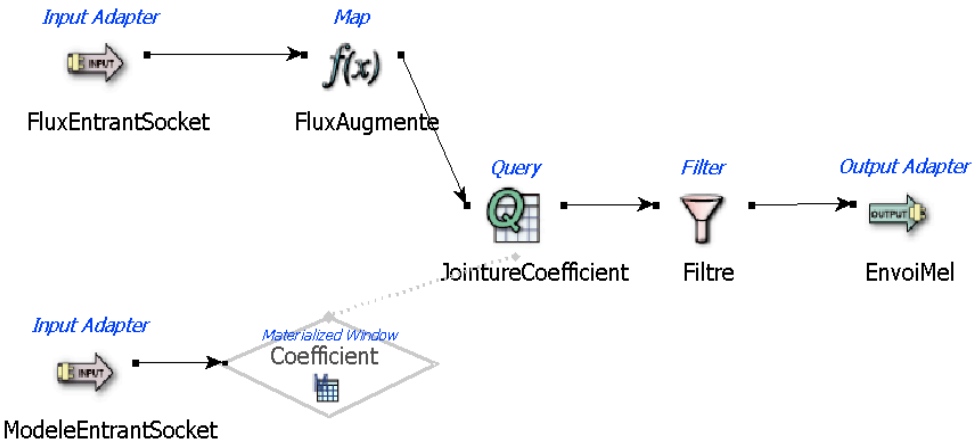

FIG. 2.8 – Diagramme de l'application relative à la prédiction de consommation téléphonique d'un centre d'appels.

	Développeur	Entreprise
OS supporté	Red Hat Enterprise Linux 3 AS (32-bit) Windows XP Pro Windows Server 2003	Red Hat Enterprise Linux 3 AS (32-bit or 64-bit) Red Hat Enterprise Linux 4 AS (64-bit) Novell SUSE Linux Enterprise Server 9 (64-bit) Windows XP Pro Windows Server 2003 Sun Solaris 10 and Solaris 8 (both 64-bit ; command-line only)
License	Essai	Souscription ou perpétuelle
Prix	Gratuit	En fonction de la taille du déploiement
StreamBase Studio IDE (création/test applications)	Oui	Oui (N/A sur Solaris)
StreamSQL	Oui	Oui
Java, C++, .NET APIs	Oui	Oui
Architecture Multi-threaded	Oui	Oui
Stockage des résultats dans des tables en mémoire	Oui	Oui
Stockage des résultats dans des tables sur disque	Non	Oui
Adaptateurs	Oui	Oui
Requêtes sur source JDBC externe	Non	Oui
Connection serveur UNIX distant	Non	Oui
Démarrage du serveur SB par ligne de commande	Non	Oui

TAB. 2.1 – Différences entre la version développeur et la version entreprise de StreamBase.

Chapitre 3

D'autres systèmes de gestion de flux

Dans ce chapitre, nous présentons les fonctionnalités de trois systèmes commerciaux de gestion de flux : Coral8, Aleri et TruViso. Par rapport à la description du système StreamBase au chapitre précédent, nous entrons ici moins dans les détails des fonctionnalités. En effet, la plupart sont communes à celles de StreamBase. C'est pourquoi nous insisterons plus sur les fonctionnalités particulières à chaque système.

3.1 TruViso

A l'instar de TelegraphCQ, dont il est la version commerciale, TruViso (aka Aminsight) augmente le SGBDR PostgreSQL et fournit un moteur de traitement de requêtes continues sur des flux.

La solution Aminsight est composée :

- du moteur de traitement des requêtes continues
- d'un framework d'intégration
- d'un visualiseur temps-réel.

Le framework d'intégration gère les composants faisant entrer et sortir les données dans le moteur :

- requêtes
- adaptateurs
- applications clientes.

Le visualisateur est composé d'un serveur de visualisation et d'un outil de composition graphique afin de pousser les données vers un client web et de générer dynamiquement des graphiques affichant les résultats de plusieurs requêtes continues. Il exploite la technologie Flex Data Services d'Adobe.

3.1.1 Le langage

Le langage de définition. – Le langage de définition autorise la manipulation de tables, de flux et de relations actives. Ces dernières sont des relations liées à une requête continue et possède deux modes : `append` ou `replace`. Dans le premier cas, le contenu de la fenêtre

associée à la requête est ajouté au contenu de la relation. Dans le second cas, il vient remplacer le contenu de la relation.

Les flux entrants sont créés par la commande `CREATE STREAM`. Comme pour TelegraphCQ, l'un des champs du schéma doit porter une marque `CQTIME` et doit être du type `TIMESTAMP`. La colonne associée est soit présente dans le flux soit générée par le système. On le spécifie en utilisant soit le marqueur `USER`, soit le marqueur `SYSTEM`. L'ordonnancement des tuples se fait donc de manière explicite ou implicite. Dans le cas explicite, le moteur peut réordonner les tuples entrant dans les limites d'un intervalle de temps spécifié par l'utilisateur.

De plus, les tuples d'un flux peuvent être archivés pour consultation ultérieure, ou non. Il faut pour cela préciser le type, `ARCHIVED` ou `UNARCHIVED`, du flux. Voici un exemple de création d'un flux :

```
create stream MyFirstStream (
   Symbol varchar(10),
   Price float,
   Volume int,
   Time timestamp CQTIME USER
   ) type UNARCHIVED ;
```

Modèle d'exécution d'une requête. – Un flux dérivé est obtenu par exécution d'une requête continue, ce qui revient à employer la commande CREATE STREAM AS. Une requête continue se décompose en un graphe d'exécution d'opérateurs de trois types :

- opérateurs StoR (flux à relation) : s'appliquent à chaque flux associé et produit une séquence de relations (plus informellement, il s'agit d'une "fenêtre")
- opérateurs RtoR (relation à relation) : pour chaque relation produite par un opérateur StoR, l'opérateur RtoR s'applique à cette relation (et éventuellement d'autres apparaissant dans la clause FROM), construisant ainsi une suite de relations
- opérateurs RtoS (relation à flux) : pour chaque relation produite par un opérateur RtoR, l'opérateur produit une nouvelle fenêtre de tuples, généralement en comparant la relation fraichement produite avec les relations précédentes.

Fenêtrage. – Les opérateurs StoR sont les opérateurs de fenêtrage. Aminsight fournit les opérateurs suivants :

- Sliding Window : cet opérateur repose sur la spécification d'un intervalle de déplacement et d'un intervalle de visualisation. Le premier définit la périodicité de construction d'une relation à partir du flux. Le second définit les tuples du flux à extraire sous forme de relation. Les intervalles sont définis de manière logique ou physique.
- Chunking Window : cet opérateur découpe un flux en fenêtres successives, contigües et disjointes. Il possède deux modes d'utilisation. Le premier consiste à créer une fenêtre pour chaque valeur du champ d'agencement. Chaque fenêtre contient des tuples aux valeurs du champ d'agencement identiques. Le second mode autorise plus de latitude. Une suite de durées étant spécifiée, par exemple '2 seconds', '4 seconds', '10 seconds', le découpage se fait suivant ces durées et se répète.

- Landmark Window : cet opérateur repose sur la spécification d'un intervalle de déplacement et d'un intervalle de remise à zéro. Le premier définit la périodicité de construction d'une relation. Le second définit une suite de points de coupure.
- Partitioning Window : chacune des trois fenêtres précédentes peut se voir adjoindre une clause de partitionnement et une date d'expiration. La clause de partitionnement sert à découper le flux en sous-flux dont les tuples ont une valeur identique pour le champ clé. L'opérateur StoR est alors évalué séparément sur chacun des sous-flux.

Exemple de requête. – Comme pour les flux entrants, un flux dérivé doit posséder un champ identifié `CQTIME`. Ce champ est à spécifier explicitement dans la requête à l'aide d'une fonction spéciale, afin de garantir la monotonie de ce champ. Notons que pour un flux dérivé il est inutile, et donc impossible, de catégoriser le champ d'ordonnancement en `USER` et `SYSTEM`. Voici un exemple de requête :

```
create stream MySecondStream (
   Symbol varchar(10),
   AveragePrice float,
   Time timestamp cqtime
) as
(select
   Symbol,
   sum(Price * Volume) / sum(Volume) as AveragePrice,
   advance_agg(Time) as windowtime
from
   MyFirstStream <visible '5 seconds' advance '3 seconds'>
group by Symbol) ;
```

3.1.2 Les adaptateurs

En entrée, Aminsight propose :

- un protocole propriétaire de communication de données au format CSV à travers un socket
- un enrichissement de la commande PostgreSQL `copy` pour qu'elle s'applique à des flux.

En sortie, Aminsight hérite les interfaces de PostgreSQL (JDBC, ODBC).

3.1.3 Les données persistantes

Aminsight étant une extension de PostgreSQL, il autorise la manipulation de flux aussi bien que de tables. Des jointures entre flux et tables sont possibles.

3.1.4 Les API

Aminsight hérite les API de PostgreSQL, ce qui permet de définir des fonctions, agrégats propres comme en le fait avec PostgreSQL. La construction d'adaptateurs propres et d'applications clientes est possible, en Java.

3.1.5 Le framework d'intégration

Le framework d'intégration, réunissant tout ce qui a trait à l'entrée et la sortie des données dans le moteur, est configuré selon un modèle XML. L'élément <config> est la racine de la configuration et contient les éléments <queries> (une liste de définition de requêtes), <connectors> (une liste des adaptateurs et applications clientes) et <engines> (liste de propriétés de connexion JDBC au moteur, une seule connexion pour le moment).

3.1.6 Le visualiseur

En dehors du client SQL classique, un client de visualisation web permet d'écrire et poser des requêtes puis de visualiser en temps réel les résultats de ces requêtes. Il fonctionne avec n'importe quel navigateur supportant le plug-in Flash Player 9 d'Adobe. Il exploite les technologies Flex Data Services d'Adobe pour le push des données et Flex Data Chart pour les graphiques.

Une *visualisation* est spécifiée dans le format XML, et se compose de trois parties :
- la ou les requête(s), dans le langage exploité par Aminsight
- les paramètres
- les composants visuels (graphes, diagrammes, etc).

3.2 Coral8 et Aleri

Les systèmes Coral8 et Aleri sont assez proches du système StreamBase. C'est pourquoi nous n'effectuons pas une présentation détaillée des fonctionnalités.

3.2.1 Coral8

La solution Coral8 est composée de :
- Coral8 server : le moteur
- Coral8 Studio : environnement intégré pour le développement d'applications
- Coral8 SDK : développement d'adaptateurs, de fonctions et d'application clientes (C/C++, Java, Perl, Python, .NET)
- instructions en ligne de commande

Le Coral8 Studio fournit un éditeur de texte afin d'écrire les requêtes dans le langage CCL, permet d'associer un adaptateur à un flux, propose une visualisation de la requête sous forme de diagramme, inclut un environnement de test. Précisons que, contrairement à StreamBase, Coral8 ne dispose pas d'un ensemble d'opérateurs graphiques. Le diagramme est constitué par les flux entrants/sortants, des tables de données et des boîtes contenant les requêtes CCL.

Le langage. – Le langage CCL est de type SQL. Un compilateur est responsable de la transformation de la requête CCL en un ensemble de primitives de plus bas niveau. Le fenêtrage repose sur une datation explicite ou implicite, sur un temps logique ou physique. Le langage permet la définition de fenêtres glissantes et sautantes, sans contrôle utilisateur sur le décalage entre deux fenêtres successives.

Une particularité du langage CCL réside dans la possibilité d'exprimer une recherche de séquences d'événements dans plusieurs flux. La requête suivante, par exemple dans le cadre d'une application de supervision basée sur des puces RFID, vérifie si un tag a été vu par les lecteurs A et B, puis C, mais pas D, dans une fenêtre de 10s :

```
Insert Into StreamAlerts
Select StreamA.Id
From StreamA a, StreamB b, StreamC c, StreamD d
Matching [10 seconds : a && b, c, !d]
On a.Id = b.Id = c.Id = d.Id
```

Les adaptateurs. – Coral8 possède les adaptateurs natifs suivants :
- générateur aléatoire de messages
- lecture/écriture de fichiers binaires, CSV, XML ou d'expressions régulières
- lecture/écriture de données CSV ou d'expressions régulières sur un socket
- Rendezvous (entrée/sortie)
- JDBC (entrée/sortie)
- RSS (entrée)
- DB (entrée/sortie)
- XML over HTTP
- envoi de courriels
- JMS (entrée/sortie)
- WebSphere MQ (entrée/sortie).

Cette liste est quasiment identique à celle des adaptateurs de StreamBase.

Lignes de commande. – Le système propose trois programmes utilisables en ligne de commande : c8_server, c8_compiler, c8_client. La combinaison de ces trois programmes permet de réaliser les tâches nécessaires au déploiement d'une application : lancement et accès au serveur, création et compilation du module CCL, création de l'espace de travail et exécution de l'application.

Les API. – L'utilisateur peut développer les adaptateurs propres à ses besoins à l'aide d'un kit de développement, en C/C++, Java, Perl, Python, .NET pour les adaptateurs externes et en C/C++ pour les adaptateurs internes.

Clustering. – Le système Coral8 est capable de distribuer les traitements afin d'améliorer les performances, résister à la chute d'un serveur, ou encore de fédérer les traitements. L'amélioration des performances par exploitation d'un clustering passe par l'un des trois niveaux suivants, les trois niveaux pouvant être combinés :
- inter flux : différents flux sont envoyés à différentes machines
- intra flux : un flux est découper en sous-flux, chacun étant traité séparément, les résultats étant fusionnés (interviennent ici des techniques d'optimisation de répartition de la charge)
- processus : le traitement d'un flux est découpé en plusieurs étapes indépendantes, chacune étant traitée sur une machine.

Coral8 dispose d'un mode gestion de la perte d'un serveur dans un cluster : les machines du cluster se partagent les états et aucun état n'est perdu lorsqu'un serveur tombe. Enfin, il

est courant de voir une entreprise mettre en place des clusters "régionaux" (indépendants) pour le prétraitement des données et un cluster central pour le traitement "final". Si Coral8 permet d'envisager de tels scenari, la répartition automatique et optimale des requêtes ne sera disponible que dans les versions futures du soft.

3.2.2 Aleri

La plate-forme de gestion de flux d'Aleri est une solution commerciale de gestion de flux de données. Elle se compose des modules suivants :

- Aleri Connect, pour le traitement des connections au et à partir du Stream Processor (le moteur)
- Aleri Service-Authoring, pour le développement d'application (via Aleri Studio)
- Aleri Interrogator, pour le traitement de requêtes ponctuelles sur les flux dérivés (avec des interfaces ODBC et JDBC, et une interface C++ supportant les requêtes SQL)
- Aleri Admin, un outil d'administration et de gestion.

Le système Aleri traite les données entrantes en fonction des applications définies par l'utilisateur. Il supporte la dynamicité du plan de requête, permet d'historiser les données, autorise un accès aux données via l'interface ODBC, JDBC.

Le développement des applications passe soit par l'Aleri Studio, un environnement de développement intégré basé sur Eclipse, soit par l'Aleri SQL, un sous-ensemble du langage SQL agrémenté d'opérateurs spécifiques à la gestion de flux, soit par l'Aleri XML, qui permet de définir les schémas des flux. Le studio permet la définition d'applications sous forme de diagrammes, en utilisant des boîtes pour les opérations et en les reliant par des flèches, à l'instar de StreamBase. Il fournit également un environnement de simulation, de test et de debugging.

Un certain nombre d'adaptateurs sont fournis avec la solution. Il est possible pour l'utilisateur de définir ses propres adaptateurs (C++, Java, .NET) à l'aide d'une API, tout comme des fonctions propres.

Enfin, la solution contient un composant optionnel qui lui est propre : Aleri OLAP Server. Celui-ci possède une capacité d'analyse de données multidimensionnelles issues des données collectées dans des flux. Ce serveur charge continuellement les données à partir de la plate-forme Aleri et procède à une agrégation à la demande.

3.3 Conclusion

Les solutions proposées par Coral8 et Aleri sont proches, en terme de fonctionnalités, de la solution StreamBase : environnement de développement intégré, palette d'adaptateurs, accès aux données persistantes, aide au développement d'applications, etc. Aleri se démarque en proposant un composant d'analyse OLAP de données de flux. Mais on peut noter que, quelle que soit la solution, le catalogue des fonctionnalités proposées s'étoffe à vitesse grand V : le marché est encore jeune et de nouveaux besoins apparaissent fréquemment, aussitôt pris en compte.

Notons que seul Truviso se démarque en fournissant une solution très légère car adossée à PostgreSQL (les fonctionnalités sont celles du SGBD) et en proposant pour mode d'interaction avec le moteur de passer par un client web.

Conclusion

Nombreuses sont les applications nécessitant le traitement continuel et à la volée de données qui se présentent d'elles-mêmes. Dans ce cadre, les systèmes de gestion de base de données classiques sont inadaptés : la priorité n'est pas le stockage mais l'analyse. C'est pourquoi de nouveaux systèmes ont été développés : les systèmes de gestion de flux de données.

Dans ce document, nous avons précisé les tenants et les aboutissants du passage à un modèle "data-push" de la gestion de données et les caractéristiques qui en découlent : requêtes continues, extension du langage SQL, approximation, etc. Nous avons de plus effectué un tour d'horizon des systèmes de gestion de flux et précisé ce qu'un utilisateur peut attendre d'un tel système : adaptateurs, environnement de développement intégré, API, etc.

Plusieurs systèmes de gestion de flux de données sont disponibles. Nous avons passé en revue les fonctionnalités de quatre d'entre eux : StreamBase, Coral8, Aleri et TruViso. En terme de fonctionnalités, les trois premiers sont riches et assez proches (environnement de développement intégré, nombreuses API, nombreux adaptateurs) alors que le dernier est (pour l'instant) plus léger car hérite les fonctionnalités de PostgreSQL. En terme de performance, les applications envisagées pour l'instant n'ont pas nécessité d'entrer dans une phase de test intensif : nos besoins actuels sont largement couverts par n'importe quel SGFD.

La question des performances est un point critique, qui nécessite d'être étudié au plus près de l'application envisagée. Les performances sont susceptibles de varier grandement en fonction des paramètres de l'application (nombre de requêtes, complexité des requêtes, nombre de flux, débit des flux, etc). Mais on peut noter que les SGFD actuels (comme StreamBase, Truviso, Coral8 et Aleri) ont plus vocation à traiter un grand nombre de tuples qu'un grand nombre de requêtes.

Enfin, notons que le monde de la gestion de flux est vaste et que nous en avons explorer dans ce document une petite partie. Nous n'avons par exemple pas présenté l'Event Processing Platform de Progress Apama ou encore discuté le System S d'IBM.

Annexe A

Retour d'utilisation de TelegraphCQ

Ce chapitre contient mes notes relatives à l'utilisation de TelegraphCQ, rien de plus (pas de catalogue des fonctionnalités, pour l'instant), rien de moins.

A.1 Prise en main

Création d'un nouvel ensemble de bases de données : `$ initdb`

`initdb` initialise le répertoire de données, les catalogues partagés et réalise diverses tâches d'administration. L'option `-D` permet de spécifier le répertoire dans lequel stocker le cluster de bases de données. C'est la seule information requise par `initdb` mais on peut tout aussi bien la spécifier par l'intermédiaire de la variable d'environnement `PGDATA`.

Création d'une base de données : `$ createdb`

Un exemple de création d'une base de données `demo` : `$ createdb demo`. Pour créer une base de données, le serveur doit être lancé.

Lancement/arrêt du serveur : `pg_ctl start [-w] [-s] [-D datadir] [-I filename] [-o options] [-p path]`

`pg_ctl stop [-W] [-s] [-D datadir] [-m s[mart] / f[ast] / i[mmediate]] [-o options] [-p path]`

Les options sont les suivantes :

`-D datadir` : emplacement des fichiers de la base de données. Par défaut, la variable d'environnment `PGDATA` est utilisée.

`-l filename` : nom du fichier vers lequel sont dirigés les logs de sortie.

`-m mode` : spécification du mode de fermeture accidentelle.

`-o options` : spécification des options à passer directement à la commande postmaster. Les options sont usuellement entre guillemets pour assurer un passage en groupe.

`-p path` : spécification de la localisation de l'exécutable postmaster. Par défaut, l'exécutable est cherché dans le même répertoire que `pg_ctl`.

`-s` : impression des erreurs seulement, pas des messages d'information.

`-w` : attente de l'ouverture ou du crach pour terminer (défaut pour stop).

`-W` : pas d'attente de l'ouverture ou du crach pour terminer (défaut pour start).

Lancement du serveur PostgreSQL en mode TCQ : `$ pg_ctl start -D${PGDATA} -l LogfileName -o "-t DatabaseName -u UserName -G -i -Q 64 -d 1"`

`-t DatabaseName` : nom de la base de données sur laquelle se connecter.

`-i` : autorise le client à se connecter à l'aide des sockets TCP/IP.

`-G` : démarrage de l'exécuteur TCQ en mode eddy.

`-Q` : spécification du nombre maximum de queues de résultats.

`-d` : spécification du niveau de debug. Plus la valeur est élevée, plus le nombre de warning l'est.

Lancement d'un client : $ psql -C DatabaseName

`-C` : les requêtes `SELECT` sont soumises avec curseur, afin de voir le résultat.

`-I n` : les résultats sont renvoyés par paquets de taille `n`. Par défaut, `n` vaut 1.

`-i m` : élimine la requête après que `m` lignes de résultats ont été renvoyés.

Création d'un flux :

```
CREATE SCHEMA traffic ;
CREATE STREAM traffic.measurement (stationid INT, speed REAL, tcqtime TIMESTAMP
    TIMESTAMPCOLUMN) TYPE ARCHIVED ;
```

`tcqtime` est le champ d'indice temporel obéissant à la contrainte `TIMESTAMPCOLUMN`.

Association d'un wrapper au flux : `ALTER STREAM traffic.measurement ADD WRAPPER csvwrapper ;`

Lancement d'une requête

Envoi des données : $ cat filename | source.pl localhost 5533 csvwrapper,DataFileName

A.2 Exemple d'utilisation

On rapporte maintenant un exemple de mise en œuvre de TelegraphCQ. Les données considérées sont les trajectoires de trois mobiles, reportées sur la fig.A.1. Trois fichiers csv contiennent les données relatives à chaque trajectoire. un exemple de tuple est le suivant :

```
1,0.00628314396555895,0.027515115293407,2007-03-06 0 :0 :1
```

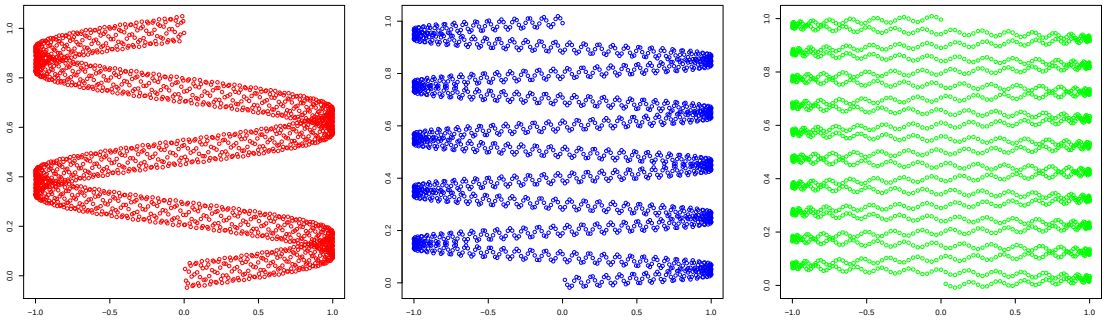

FIG. A.1 – Trajectoires des trois mobiles considérés.

Le premier champ identifie le mobile, le second champ contient l'abscisse de sa position, le troisième champ l'ordonnée, et le quatrième champ contient la marque temporelle.

PostgreSQL et TelegraphCQ v2.1 sont supposés installés. Tout d'abord, le cluster est créé et initialisé :

```
initdb -D /home/postgres/pgCluster
```

Afin de limiter la redondance, il est préférable de spécifier l'adresse du cluster une fois pour toute :

```
PGDATA=/home/postgres/pgCluster
initdb
```

Ensuite, la base est créée, ce qui nécessite le lancement du serveur :

```
pg_ctl start -l logfile.log
createdb HelicoidDatabase
pg_ctl stop -l logfile.log
```

Un client peut alors être lancé afin de travailler sur cette base :

```
psql -C HelicoidDatabase
```

La première chose à faire est de créer les 3 flux et de leur associer un adaptateur :

```
CREATE SCHEMA HelicoidSchema ;
CREATE STREAM HelicoidSchema.Stream1 (Id INT, X FLOAT, Y FLOAT, TcqTime
TIMESTAMP TIMESTAMPCOLUMN) TYPE ARCHIVED ;
ALTER STREAM HelicoidSchema.Stream1 ADD WRAPPER csvwrapper ;
CREATE STREAM HelicoidSchema.Stream2 (Id INT, X FLOAT, Y FLOAT, TcqTime
TIMESTAMP TIMESTAMPCOLUMN) TYPE ARCHIVED ;
ALTER STREAM HelicoidSchema.Stream2 ADD WRAPPER csvwrapper ;
CREATE STREAM HelicoidSchema.Stream3 (Id INT, X FLOAT, Y FLOAT, TcqTime
TIMESTAMP TIMESTAMPCOLUMN) TYPE ARCHIVED ;
```

```
ALTER STREAM HelicoidSchema.Stream3 ADD WRAPPER csvwrapper ;
```

Nous sommes alors en mesure de poser une requête sur un flux. Par exemple, très simplement :

```
SELECT * FROM HelicoidSchema.Stream1 ;
```

Dans un autre shell, on lance le flux corespondant :

```
head -20 Helicoid1.txt
| perl Source.pl localhost 5533 csvwrapper,HelicoidSchema.Stream1
```

et le flux sortant est dirigé vers le serveur TelegraphCQ. D'autres requêtes basiques peuvent être formulées afin de tester le fenêtrage :

```
SELECT
Max(HelicoidSchema.Stream1.Y),
COUNT(HelicoidSchema.Stream1.Y),
wtime(*)
FROM HelicoidSchema.Stream1[RANGE BY '10 seconds' SLIDE BY '10 seconds'
START AT '2007-03-06 0 :0 :1'] ;

SELECT
Max(HelicoidSchema.Stream1.Y),
COUNT(HelicoidSchema.Stream1.Y), wtime(*)
FROM HelicoidSchema.Stream1[RANGE BY '10 seconds' SLIDE BY '5 seconds' START
AT '2007-03-06 0 :0 :1'] ;

SELECT
Max(HelicoidSchema.Stream1.Y),
COUNT(HelicoidSchema.Stream1.Y), wtime(*)
FROM HelicoidSchema.Stream1[RANGE BY '10 seconds' SLIDE BY '20 seconds'
START AT '2007-03-06 0 :0 :1'] ;
```

ou de tester le passage de deux flux :

```
SELECT *
FROM
HelicoidSchema.Stream1 [RANGE BY '10 seconds' SLIDE BY '10 seconds' START
AT '2007-03-06 0 :0 :1'],
HelicoidSchema.Stream2 [RANGE BY '10 seconds' SLIDE BY '10 seconds' START
AT '2007-03-06 0 :0 :1']
WHERE
HelicoidSchema.Stream1.TcqTime = HelicoidSchema.Stream2.TcqTime
AND HelicoidSchema.Stream1.Y < HelicoidSchema.Stream2.Y ;
```

Cherchons maintenant à détecter la présence du mobile dans une zone fixe, par exemple une bande horizontale $\{(x, y);\ 0.2 < y < 0.3\}$. Un exemple de requête correspondante serait le suivant :

```
SELECT
HelicoidSchema.Stream1.X,
HelicoidSchema.Stream1.Y,
HelicoidSchema.Stream1.TcqTime
FROM
HelicoidSchema.Stream1 [RANGE BY '1 seconds' SLIDE BY '1 seconds']
WHERE
HelicoidSchema.Stream1.Y < 0.3 AND HelicoidSchema.Stream1.Y > 0.2 ;
```

Envisageons pour terminer une zone mobile : la bande horizontale est $\{(x, y);\ 0.2 < y < y(t)\}$ avec $y(t) = 0.2 + t/2000$, le temps étant mesuré en secondes. Nous commençons par créer un flux correspondant à la zone mobile :

```
CREATE STREAM HelicoidSchema.Stripe (YMin FLOAT, YMax FLOAT, TcqTime TIMESTAMP
TIMESTAMPCOLUMN) TYPE ARCHIVED ;
ALTER STREAM HelicoidSchema.Stripe ADD WRAPPER csvwrapper ;
```

puis on formule la requête testant l'appartenance du mobile à la zone :

```
SELECT
HelicoidSchema.Stream1.X,
HelicoidSchema.Stream1.Y,
HelicoidSchema.Stripe.YMin,
HelicoidSchema.Stripe.YMax,
HelicoidSchema.Stream1.TcqTime
FROM
HelicoidSchema.Stream1 [RANGE BY '1 seconds' SLIDE BY '1 seconds'],
HelicoidSchema.Stripe [RANGE BY '1 seconds' SLIDE BY '1 seconds']
WHERE
HelicoidSchema.Stream1.TcqTime = HelicoidSchema.Stripe.TcqTime
AND HelicoidSchema.Stream1.Y > HelicoidSchema.Stripe.YMin
AND HelicoidSchema.Stream1.Y < HelicoidSchema.Stripe.YMax ;
```

Bibliographie

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[AC06] Raman Adaikkalavan and Sharma Chakravarthy. Snoopib : interval-based event specification and detection for active databases. *Data Knowl. Eng.*, 59(1) :139–165, 2006.

[ACe+03] Daniel J. Abadi, Don Carney, Ugur &#199 ;etintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora : a new model and architecture for data stream management. *The VLDB Journal*, 12(2) :120–139, 2003.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road : A stream data management benchmark. In *VLDB*, pages 480–491, 2004.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS '02 : Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[CcC+02] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.

[CCH07] F. Clérot, B. Csernel, and G. Hébrail. Tutoriel gestion et fouille de flux de données. Technical report, EGC'07, 2007.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq : a scalable continuous query system for internet databases. In *SIGMOD '00 : Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390, New York, NY, USA, 2000. ACM Press.

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

[CFPR00] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock : a language for extracting signatures from data streams. In *Knowledge Discovery and Data Mining*, pages 9–17, 2000.

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope : a stream database for network applications. In *SIGMOD '03 : Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651, New York, NY, USA, 2003. ACM Press.

[DGH+06] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[DGP+07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga : A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[DGR+03] Alan Demers, Johannes Gehrke, Rajmohan Rajaraman, Niki Trigoni, and Yong Yao. The cougar project : a work-in-progress report. *SIGMOD Rec.*, 32(4) :53–59, 2003.

[DIG07] Y. Diao, N. Immerman, and D. Gyllstrom. Sase+ : An agile language for kleene closure over event streams. Technical report, UMass, march 2007.

[GC+07] Daniel Gyllstrom, Eugene Wu 0002, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. Sase : Complex event processing over streams (demo). In *CIDR*, pages 407–411, 2007.

[GJS92] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite event specification in active databases : Model & implementation. In *VLDB '92 : Proceedings of the 18th International Conference on Very Large Data Bases*, pages 327–338, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[GO03] Lukasz Golab and M. Tamer Ozsu. Issues in data stream management. *SIGMOD Rec.*, 32(2) :5–14, 2003.

[KCC+03] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Frederick Reiss, and Mehul A. Shah. Telegraphcq : An architectural status report. *IEEE Data Eng. Bull.*, 26(1) :11–18, 2003.

[LPT00] Ling Liu, Calton Pu, and Wei Tang. Webcq-detecting and delivering information changes on the web. In *CIKM '00 : Proceedings of the ninth international conference on Information and knowledge management*, pages 512–519, New York, NY, USA, 2000. ACM Press.

[LTWZ05] Chang Luo, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo. A native extension of sql for mining data streams. In *SIGMOD '05 : Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 873–875, New York, NY, USA, 2005. ACM Press.

[SCZ05] Michael Stonebraker, Ugur Cetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4) :42–47, 2005.

[SFMH01] Mehul A. Shah, Michael J. Franklin, Samuel Madden, and Joseph M. Hellerstein. Java support for data-intensive systems : experiences building the telegraph dataflow system. *SIGMOD Rec.*, 30(4) :103–114, 2001.

[Sul96] Mark Sullivan. Tribeca : A stream database manager for network traffic analysis. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, page 594. Morgan Kaufmann, 1996.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *SIGMOD '92 : Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330, New York, NY, USA, 1992. ACM Press.

[WZ03] Haixun Wang and Carlo Zaniolo. Atlas : A native extension of sql for data mining. In *SDM*, 2003.

Dépôt légal : 2007 – 4ème[r] trimestre
Imprimé à l'Ecole Nationale Supérieure des Télécommunications – Paris
ISSN 0751-1345 ENST D (Paris) (France 1983-9999)

Ecole Nationale Supérieure des Télécommunications
Groupe des Ecoles des Télécommunications - membre de ParisTech
46, rue Barrault - 75634 Paris Cedex 13 - Tél. + 33 (0)1 45 81 77 77 - www.enst.fr
Département INFRES