Utilisation de caches dans l'interrogation de flux de capteurs

Loïc Petit
sous la direction de
Claudia Roncancio et Cyril Labbé
au LIG dans l'équipe HADAS

Résumé

Au coeur de l'informatique ubiquitaire, les gestions de données de capteurs font l'objet de maintes recherches. Les efforts sont conséquents pour se rapprocher le plus possible des systèmes "haut-niveau" par l'intermédiaire de requêtes déclaratives. Les problèmes sont liés à l'hétérogénéité des données ou encore au traitement plus spécifique de ces flux en tant que base de données très proche du relationnel. Cet article explore les possibilités de gestion de cache sur ces systèmes pour éviter des redondances de traitements. Nous traiterons un moyen de supprimer les paquets redondants qui entraînent des surcharges réseaux et de calculs. De plus nous arriverons à un moyen de traiter plusieurs requêtes tout en partageant les ressources communes. Toutefois l'ensemble de ces améliorations soulève quelques problèmes sous-jacents, nous verrons comment les gérer.

1 Introduction

Les capteurs sont désormais à un prix très faibles et sont de plus en plus sans fils. Ainsi face à la grandeur et l'hétérogénéité des données qui doivent être traités dans un contexte de grande échelle, des phénomènes "d'engorgements" apparaissent. Dans des systèmes classiques, pour pallier à ce problème, on utilise différentes techniques de cache.

Toutefois, l'application des caches repose sur un contexte qui ne s'applique pas forcément ici. En effet, la nature des données est "non persistante", ainsi nous avons une incertitude sur les possibilités d'adaptation, chose qui mérite réflexion.

Le but de cette étude est d'explorer les pistes d'application de cache dans ce contexte.

Un exemple simple est facilement trouvable pour voir comment les techniques de caches peuvent agir. Imaginons un système qui peut interroger deux capteurs toutes les deux minutes afin de connaître le maximum de température durant les 5 dernières minutes. Ce genre d'application est très utile dans le cas d'une chaîne de production dont la température ne doit pas dépasser une certaine valeur T_0. Des améliorations de traitements peuvent se faire sur le nombre de données inutiles qui transitent, car si la température ne varie pas durant deux minutes et qu'un paquet est envoyé chaque seconde, alors nous avons 119 paquets tout à fait inutiles. De plus nous voyons que si nous voulons exécuter un traitement utilisant les données de la première requête, nous devons normalement tout recalculer (à partir du niveau capteur).

Ce document décrit une solution par l'introduction de deux nouveaux opérateurs pour éviter les problèmes évoqués.

Dans ce document la section 2 nous présentera le contexte des flux de données et de capteurs. Ensuite dans la section 3, nous aborderons l'applications des techniques de caches dans l'évaluation des réponses de capteurs, soit les premières idées théoriques sur la question, notamment la perte de sémantique. Par la suite dans la partie 4, nous évoquerons les problèmes soulevés par les principes engagés et comment les résoudre par l'envoi de paquets supplémentaires. Nous continuerons dans la section 5 sur l'introduction des deux opérateurs et leur mise en place dans le plan de requête. Enfin dans la partie 6 nous aborderons deux exemples d'application pour voir l'efficacité de notre solution. Et nous terminerons en section 7 par un résumé de nos avancés et par une vision future du travail pouvant être appliqué.

2 Flux de données et capteurs

Avant de décrire nos travaux, voyons le contexte sur lequel nous sommes. Nous commencerons par décrire ce qu'est un système de gestion de flux de données.

Nous verrons ensuite son application directe sur les capteurs. Nous continuerons par la présentation d'une architecture pour le passage à grande échelle du nombre de capteurs. Par la suite nous aborderons un langage de requêtes sur ces systèmes en mettant en avant les problèmes de sémantiques. Et enfin nous verrons les techniques de caches génériques en regardant ce qui peut nous être utile.

2.1 Systèmes de Gestion de Flux de Données (SGFD)

Les SGFD permettent aux applications d'interroger des données générées continuellement dans un flux [GO03]. Par rapport aux SGBD classiques nous nous intéressons aux différences concernant la gestion des données et des requêtes.

On introduit la notion de *persistance* et de *transition*. On dit d'une donnée qu'elle est persistante si elle est stockée à un endroit et elle sera accessible à n'importe quel moment par la suite. On dit d'une requête qu'elle est transitoire car une fois qu'elle est exécutée, on n'y a plus accès car elle n'existe plus. Ce fonctionnement est ce que l'on retrouve dans les Systèmes de Gestion de Bases de Données (SGBD) classiques. Dans les SGFD nous traitons exactement à l'inverse soit : **données transitoires et requêtes persistantes**.

Ainsi, dans un SGFD, une donnée qui arrive par le flux, se verra traitée (on parle de *donnée consommée*) et une fois le traitement par les requêtes fini, elle n'existera plus. A contrario, les requêtes sont dites persistantes (ou continues) car elle restent actives sur une durée de temps qui peut être illimitée, par exemple une requête de surveillance. Les requêtes doivent être constamment en évaluation afin de prendre en compte les nouvelles valeurs qui viennent s'ajouter au flux.

De manière générale, les données arrivent sans périodicité et le système ne maîtrise pas cette donnée (et n'a pas à le faire). Malgré tout, on peut introduire une périodicité par l'intermédiaire des fenêtres. En effet, le fait de traiter des flux qui peuvent être infinis n'est a priori pas un problème de traitement si celui-ci se réduit à un simple filtre, mais il devient impossible si on utilise des opérateurs dits *bloquant* [BBD+02]. Un opérateur est de ce type s'il ne fournit le résultat que s'il possède l'ensemble des données. Comme exemple d'opérateur *bloquant* on peut citer la jointure, celle-ci a besoin de l'ensemble des données pour fournir le résultat final. Il faut donc réduire le flux à un ensemble fini. Le fenêtrage est un procédé permettant de découper le flux en paquets de données qui pourrons être traités par des opérateurs de tout type (notamment agrégation).

Une dernière précision sur ces systèmes. Une requête, une fois compilée, est un graphe d'opérateurs dont la répartition dépend du réseau (voir section suivante). Un opérateur consomme un ou plusieurs flux en entrée, exécute des opérations et produit un flux continue en sortie. Les opérateurs sont variés : sélection, agrégation, projection, jointure, mais aussi création de fenêtres et jointures de fenêtres.

Maintenant que nous avons vu le contexte global des SGFD, nous pouvons nous pencher sur la question des capteurs.

2.2 Flux provenant de capteurs

Comme nous l'avons dit précédemment, les capteurs sont de plus en plus nombreux dû à leur faible coût et leur installation simple (car sans fils), ce qui explique une bonne partie le développement de l'informatique ubiquitaire.

Le problème qui en résulte est la grande quantité de données (certains capteurs envoient des données périodiquement de l'ordre de la milli-seconde) et surtout l'hétérogénéité de celles-ci. Un capteur peut avoir des types de données très particuliers liés à une seule application qui a ses propres capteurs avec ses propres traitements. Pour pouvoir exploiter ces données de manière non-dépendante d'une application, il faut arriver à introduire une représentation génériques des données pour avoir une vue commune des données hétérogènes.

L'utilisation des adaptateurs (*wrappers*) est la solution la plus répandue dans des systèmes tels que HiFi [CEF+04], GSN [AHS06] ou Borealis [AAB+05] pour résoudre ces problèmes. Le principe repose sur l'utilisation d'une couche logiciel pour interroger par un format générique des commandes que le capteur peut comprendre, et inversement, les réponses sont traduites pour quelles soient comprises par le système.

2.3 Passage à grande échelle

Le fait d'agrandir un parc de capteur, comme par exemple pour une entreprise contrôlant les flux routiers d'une région entière afin de les optimiser, pose de réels problèmes de traitement. En effet, un petit nombre de capteur, il suffit d'avoir un ordinateur reliés aux quelques capteurs, le traitement est simple dans ce cas. Mais à un niveau régional, imaginons 5 000 capteurs qui envoient 20 données toutes les secondes. Cela fait 100 000 données par seconde qu'il faut traiter. Dans le cas de traitements simples tels que le filtrage, il n'y a pas de problème bien que la charge réseau soit massive. Mais dans le cas de jointure entre fenêtres avec des agrégations, le traitement est bien plus complexe et le quasi-temps réel ne sera plus supportable. Or une application peut avoir plusieurs requêtes en même temps. C'est pour cela que nous voulons pouvoir partager l'infrastructure mise

en place entre plusieurs applications qui ont les mêmes sous-requêtes.

Afin de répartir la charge des solutions se forment grâce à l'architecture du réseau utilisé. Par la suite, nous nous baserons sur le formalisme du système proposé par Levent Gürgen[Gür07] qui permet une approche à grande échelle de la gestion de données de capteurs hétérogènes. L'architecture adoptée est une architecture hiérarchique [GLOR05] de services : Site de contrôle (SQS), Passerelles (GQS), Proxy (PQS) et adaptateurs (pour l'hétérogénéité).

La Fig. 1 illustre la structure qui est utilisé. On remarquera la présence d'un *Lookup Service* appelé LS, qui permet de localiser les passerelles et les *wrappers* des capteurs au sein de la hiérarchie.

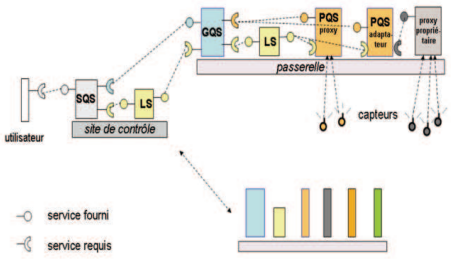


FIGURE 1 – Structure hiérarchique d'un système réparti de capteur

Lorsqu'une requête arrive au site, un programme va compiler cette requête pour en fournir un plan d'exécution. Ces plans d'exécutions représentent les graphes des opérateurs qui seront répartis sur les différents noeuds de la hiérarchie. Ainsi une donnée passera dans les opérateurs (qui sont sur différentes machines) par le chemin que fournit le plan pour arriver à la fin au site de contrôle.

2.4 Un langage de requête pour de données

Les efforts pour fournir un langage déclaratif n'ont pas été vains. Deux systèmes sortent du lot pour fournir un langage "SQL-like" pour les requêtes : STREAM [ABB+03] et TelegraphCQ [Cha03]. L'approche adoptée est d'utiliser le modèle relationnel pour s'approcher le plus de ce qui est déjà utilisé. Malheureusement les langages ne sont pas entièrement formalisés. L'avancée la plus importante se trouve du coté de STREAM avec son langage CQL (Continuous Query Langage) qui permet de traiter une requête continue pour un SGFD générique.

Une requête en CQL est très similaire à une requête SQL2. Toutefois, des modifications ont étés introduites pour permettre la gestion de fenêtres. En effet, on peut utiliser les expressions suivantes :

1. [ROWS n] : Crée une fenêtre de n tuples.
2. [RANGE t seconds] : Crée une fenêtre dont la largeur est de t secondes.
3. [... SLIDE t seconds] : L'ajout de SLIDE permet d'introduire la périodicité de la création de fenêtre.

Ainsi, si on veut obtenir de manière continue la moyenne de la température des mesures de capteurs des 30 dernières secondes toutes les 10 secondes, on aura :

SELECT AVG(temperature) FROM sensors
[RANGE 30 SLIDE 10]

Cependant les implantations n'ont pas étés faites en entier (même dans le projet STREAM). Par exemple, le mot clé SLIDE n'est pas pris en compte dans les prototypes actuels.

D'autres modélisations existent tels que l'approche par XML avec NiagaraCQ [CDTW00] ou une construction plus procédurale par Aurora [ACC+03].

3 Applications des techniques de cache dans l'évaluation des réponses sur les capteurs

Dans cette section nous aborderons tout d'abord une première approche du cache dans les SGFD en remarquant une première amélioration sur les localisations des machines et capteurs. Ensuite nous évoquerons les problèmes majeurs qui ressortent des traitements des flux de capteurs. Nous expliquerons le principe des opérateurs de caches et enfin nous verrons enfin les avantages de nos propositions.

3.1 Amélioration de la localisation

Une première approche de notre étude était l'exploitation des caches duaux pour améliorer les requêtes des Lookup Services (localisation des noeuds). Pourquoi cela ? Car la partie localisation est une base de donnée bien plus classique que les SGFD. En effet, les données sont persistantes dans ce cas et les requêtes ("Quelle est l'adresse IP de ce Proxy ?") sont transitoires.

Une première idée serait donc de pouvoir mettre dans un cache les résultats des requêtes faites aux LS. Ainsi lorsqu'une autre demande la localisation du même noeud, on aura le résultat instantanément sans évaluation.

Nous n'irons pas plus loin dans l'analyse, car nous avons préféré nous concentrer sur l'application des techniques de caches au traitement des flux de données de capteurs réellement. Toutefois, c'est un point qui mérite une étude plus approfondie.

3.2 Encombrement et indépendance des requêtes

Voyons tout d'abord les deux inconvénients que nous pouvons rencontrer dans un contexte de traitement de SGFD de capteur à grande échelle.

Un paquet transmit est un paquet traité

Dans les travaux que nous avons pu analyser, le principe du traitement d'une donnée de "bout en bout" est constant. Lorsqu'une donnée part d'un capteur, elle passera dans chaque opérateur jusqu'au site de contrôle, et chaque opérateur aura exécuté une certaine quantité d'instructions.

Toutefois, si l'état de l'environnement surveillé avec capteurs ne change pas (par exemple : la température ou l'humidité n'a pas changée), alors on aura aussi une charge réseau et une charge processeur qui n'était pas nécessaire. Cela arrive fréquemment dans le cas de capteurs environnementaux qui envoient des données très souvent (de l'ordre de la milli-seconde).

En poussant le raisonnement à un niveau plus haut, si nous avons un traitement qui mesure le débit d'un capteur RFID (considérons un lecteur de code-barre scanne 2 objets par secondes) et que cette donnée est utilisée pour d'autres traitements. Si ce débit reste constant, encore une fois, nous n'avons pas besoin de transmettre les tuples qui engendreront des redondances.

Dans le cas général du contexte de grande échelle, en remplaçant le noeud de capteur par une hiérarchie encore plus grande de noeuds, on doit pouvoir encore supprimer les redondances de traitement.

Une requête mise en place est une requête fermée

Voici le deuxième inconvénient que nous avons relevé. Lorsqu'une requête est préparée et mise en place. Alors elle va créer les opérateurs dont elle a besoin sont déployés sur chaque noeud et ainsi le chemin sera tout tracé pour les données qui arriveront. Le fait est qu'il n'y aura pas d'autres chemins pour les données que celui qu'a tracé le plan.

Deux requêtes concurrentes vont avoir des arbres indépendants même si elles sont similaires et utilisent des "sous-arbres". On peut imaginer un système permettant des partager ces ressources et ainsi ne faire qu'un traitement pour les parties communes.

Un exemple classique pour ce genre de traitement est le contrôle de flux routiers. Des capteurs fournissent des données peu-exploitable. Des requêtes permettent le calcul du débit, de la vitesse etc... et celles-ci sont utilisés par toutes les autres requêtes à plus haut niveau [1]. Normalement les SGFD traitent ceci comme des problèmes de type "Scheduling", mais on peut imaginer que ces flux sont partagés par un cache de requête et donc on peut obtenir un gain très grand de performance.

1. voir l'application Linear Road Benchmark de STREAM

3.3 Un paquet transmit est un paquet utile

Nous venons de voir deux problèmes conséquents. Le but de cette étude est de pallier à ces problèmes.

Nous considérons qu'un paquet d'information est redondant s'il contient la même donnée que le paquet précédent et sa prise en compte dans le calcul ne change pas le résultat. Le traitement de ces paquets entraîne une charge des processeurs donc on doit pouvoir s'affranchir de ces opérations.

Pour reconnaître ces états, nous pouvons supprimer les doublons et utiliser des caches pour vérifier cet état. Nous disposons d'outils permettant d'associer une requête à ses résultats [dJLR07]. Ainsi dans chaque noeud, on peut imaginer un cache partagé qui contient l'ensemble des états des requêtes filles.

Que peut-on gagner comme avantage grâce à ceci ?

Ici, le fait de supprimer des doublons amène au premier principe que nous voulions : *Un paquet transmit est un paquet utile*. Nous pouvons désormais garantir l'invariant suivant : la valeur de l'évaluation des requêtes filles correspond à celles disponibles dans les caches en tout moment. Si on ne reçoit pas de données même pendant suffisamment de temps (grâce à la suppression de doublons), alors la valeur est celle du cache. Ceci peut être mis en place par un opérateur simple qui a un flux d'entrée sort un autre sans doublons, nous verrons ceci dans la section 5.

Le fait de mettre les données dans un cache partagé permet de sortir du "tunnel" établit par le plan de requête (évoqué dans la section précédente). Ainsi, on peut voir l'état des requêtes filles très rapidement et de façon partagée. Ainsi on peut envisager qu'une ressource (une évaluation de requête fille) n'est utilisée qu'une fois.

4 Problèmes soulevés

Notre contribution semble être efficace, toutefois, un certain nombre de points doivent être étudiés. Nous allons d'abord voir la perte éventuelle de sémantique lors de la suppression de doublons. Ensuite nous ferons une étude de cas sur les différents opérateurs pour voir ceux qu'il faut anticiper. Enfin nous donnerons plusieurs solutions envisageables et enfin celle qui nous semble la plus adaptée.

4.1 Perte de sémantique

Le fait de supprimer des paquets constituent en soit une perte d'information. En effet, nous n'avons plus accès à deux informations qui peuvent être importantes pour certaines applications.

- Le nombre de paquets
- La présence du flux dans les fils

Voyons déjà le premier problème. Imaginons une requête comme celle-ci :

SELECT COUNT(*) FROM sensors [RANGE 5]

Elle permet de compter le nombre de données qu'envoient les capteurs toutes les 5 secondes. Or en appliquant notre premier principe, si la température reste constante, ne serait-ce que pendant deux périodes, alors le résultat n'a plus de sens.

Pour le deuxième problème, il est plus subtil. En effet, considérons la requête suivante :

SELECT MAX(value) FROM sensors [RANGE 2]
WHERE value >= 15

Et supposons un flux de données (value, timestamp) comme celui-ci :

(14,1), (15,2), (15,3), (15,4), (16,5), (16,6), (14,7), (14,8)

Nous aurons en théorie le flux de résultat suivant :

15, 15, 16, *null*)

Or rappelons la suppression des doublons nous amène à penser : S'il n'y a pas de valeurs arrivant, on suppose que cette valeur n'a pas changée. Voyons l'état du flux (après suppression des doublons) après l'opérateur de sélection :

(15,2), (15,3), (16,5)

L'opérateur de création de fenêtre va ensuite découper suivant les intervalles $[1, 2]$, $[3, 4]$, $[5, 6]$ et $[7, 8]$. En prenant en compte le principe que nous venons d'évoquer, on a les fenêtres suivantes :

(15,2), (15,3), (16,5), (16,7)

Pourquoi cette valeur (16,7) ? Car selon le flux précédent, le créateur de fenêtre suppose qu'au temps $t = 7$, le flux avait la valeur incorrecte 16. Ainsi, on a un maximum pour cette fenêtre valant 16 et non *null* comme prévu. Donc nous avons une perte d'information dans ce contexte. Toutefois ce problème n'est pas spécialement dépendant des opérateurs et une solution plus globale doit être envisager.

Avant d'aller plus loin, voyons quels opérateurs sont par la perte du nombre exact d'informations.

4.2 Etude de cas

Le problème vient de la nature et de l'implantation des opérateurs. En effet, certains opérateurs ont besoin de tous les paquets alors que d'autres non. Après une étude de cas nous relevons ces deux types :

- Les opérations mathématiques d'accumulations. Aussi bien arithmétique telles que *SUM* ou *PROD*, que logique *BIT_AND* ou *BIT_OR*, ou encore d'ordre textuelle de type *CONCAT*. Si on n'a pas toute les données l'accumulation ne sera pas forcément comme l'utilisateur le demande.
- Le comptage : ceci fait appel au nombre de données, or s'il en manque, ce sera faux.

Nous avons toutefois une liste d'opérateur qui selon leur implantations pourrons être affecté par la suppression de doublons :

- AVG : La moyenne peut être calculée de deux façon, soit en prenant en compte le nombre de tuple, et dans ce cas la formule sera :

  $$\mathrm{AVG}(x) = \frac{1}{n}\sum_{i=0}^{n} x_i$$

  Mais il est toutefois possible de calculer ceci grâce aux *timestamps* qui accompagne notre tuple comme dans une base de donnée temporelle. Ainsi, on aura :

  $$\mathrm{AVG}(x,t) = \frac{1}{t_n - t_0}\sum_{i=0}^{n-1} x_i * (t_{i+1} - t_i)$$

  Et on peut voir rapidement que dans ce cas, la suppression de doublons nous amènerai au même résultat.
- Autres opérations de statistiques telles que les variances ou déviations standard.

4.3 Solutions

Pour pallier au problème de perte d'information, nous proposons deux solutions tout à fait valables mais leurs approches sont totalement différentes.

La première consiste à garder toutes les informations possibles dans le flux tout en gardant notre idée de cache. En effet, notre idée est d'envoyer des paquets très simples qui indiquent un message équivalent à : *Le contenu du flux n'a pas changé*. Celui-ci étant daté (comme tout tuples dans ce genre de flux), on a la même configurations qu'initialement, donc le même traitement.

Ainsi toutes les opérations de comptages sont conservés, pas de traitement supplémentaire est à prendre en compte, et le principe de non-redondance des paquets et des opérations est toujours présent. Toutefois ceci requiert l'envoi d'un paquet, plus léger certes, mais nous n'avons pas pu réduire le nombre d'envois.

La deuxième solution est plus proche des idées que nous avions au départ. Elle consiste a garder le principe de suppression de doublons, mais aussi d'envoyer un paquet dans le cas de perte de présence du flux. Par exemple dans le cas de la sélection vue précédemment, lorsque l'on passe en dessous de la valeur 15 (avec le tuple (14, 7) par exemple), l'opérateur de sélection enverra un paquet contenant un message équivalent à : *Remettre a zéro le cache car le flux est vide*.

Comme nous avons gardé le principe de suppression des doublons, nous devons tout de même distinguer les cas d'utilisation cités dans la section précédente. Tous les opérateurs fils des opérateurs répertoriés comme non compatible avec la suppression de doublons ne devront pas faire cette suppression. Nous choisirons cette mise en oeuvre pour la suite car elle permet un envoi plus réduitt de paquet, sans causer une grande charge de traitement.

Maintenant voyons concrètement comment cette proposition va s'appliquer.

5 Approche de mise en oeuvre

Nous avons vu pour l'instant une idéologie autour de la gestion de flux de données de capteurs. Nous allons maintenant étudier une application concrète sur les travaux de Levent Gürgen pour implanter nos principes. Nous introduirons deux nouveaux opérateurs : un dédié à la gestion de cache et l'autre qui supprime des doublons. Enfin nous verrons la mise en place de ces nouveaux opérateurs sur le plan d'exécution d'une requête.

5.1 Opérateur de cache

Cet opérateur est très simple sur son fonctionnement. Nous renseignons à ce cache le (sous-)plan de requête sur lequel il s'applique. Ensuite grâce à ce plan, nous pouvons affecter une case mémoire dans le cache central de l'ordinateur sur lequel nous sommes. Par la suite une fois un tuple reçu, celui-ci sera mis à l'emplacement alloué afin de pouvoir garder en mémoire sa valeur.

Définition 1 :

On définit l'opérateur CACHE comme ceci :

1. *Un accès au cache du noeud résident C.*
2. **Pour** *chaque donnée arrivant t du plan de requête q.*
   Faire
   C.setCacheValue(q, t)
   Ecrire en sortie t
   Fin pour

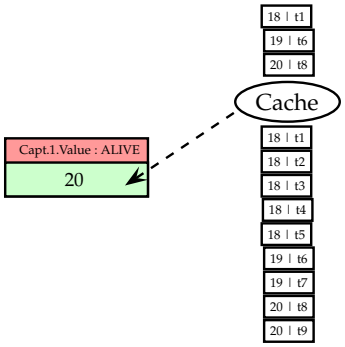


FIGURE 2 – Opérateur de cache sur la valeur d'un capteur

On remarquera que dans le cache se trouve un booléen ALIVE/DEAD qui permet de savoir si le cache est vivant. Ce qui veut dire plus précisément qu'une requête en cours d'exécution utilise ce cache. Cet état permet surtout pour une requête concurrente de savoir si elle peut récupérer cette valeur.

5.2 Suppresseur de doublons

Pour bien voir comment est détecté un doublon, nous introduisons une définition d'équivalence pour les tuples temporels :

Définition 2 :

Soit T_1 et T_2 deux tuples temporels.
On dit que T_1 et T_2 sont équivalent si et seulement si T_1 et T_2
- *ont même taille*
- *ont, pour toutes les colonnes hormis celle allouée au timestamp, des valeurs sont identiques.*

Ceci permet définir la relation suivante : Si T_1 et T_2 sont consécutifs dans le flux et que T_1 est équivalent à T_2 alors celui dont le timestamp est le plus récent est un doublon.

Ainsi on peut créer l'opérateur qui va appliquer cette règle :

Définition 3 :

On définit l'opérateur DOUBLE comme ceci :

1. *Un accès à un tuple de sauvegarde t_0*
2. **Si** *le tuple arrivant t n'est pas équivalent à t_0*
   Alors
   $t_0 = t$
   Ecrire en sortie t
   Fin si

5.3 Répartitions sur le plan de requête

Avec ces deux opérateurs nous voyons que nous pouvons appliquer la théorie que nous avons explicité jusque

là. En effet, pour chaque noeud, nous devons connaître l'état du flux des fils, pour pouvoir à la fois avoir les résultats très rapidement mais aussi pour avoir cette information afin de spéculer sur sa valeur alors que nous ne recevons pas de paquet dû à la suppression (par exemple la création de fenêtre sans données transmises). Ainsi sur chaque branche fille d'un noeud, on doit implanter un opérateur CACHE.

Nous n'avons pas réellement besoin de celui sur les sorties, car cette information n'est pas en soit nécessaire puisque qu'elle est accessible un noeud plus haut, sans communication réseau. De plus les opérateurs n'ont pas besoin de cette information à cet endroit ci.

Les suppresseurs de doublons eux doivent être placés à l'inverse, soit, en sortie. En effet, c'est au moment du transfert qu'il faut éviter la surcharge réseau. Attention toutefois, il ne faut pas placer ces opérateurs dans les cas que nous avons mentionnés dans le paragraphe 4.2, sous peine de voir disparaître la sémantique du système.

Ainsi, on peut en déduire l'algorithme qui applique les transformations de cache à un plan de requête qp quelconque :

```
1  void addCacheToQueryPlan(QueryPlan qp,
       boolean doublons = false) {
2    boolean doublonsFils = doublons;
3    if(!doublonFils) {
4      [[ Ajouter au noeud un suppresseur
         de doublons; ]]
5      for(int i = 0 ; i < qp.nbOperateur
         ; i++) {
6        if(qp.op[i] est dans la catégorie
           des opérateurs refusant des
           doublons) {
7          doublonsFils = true;
8          break;
9        }
10     }
11   }
12   for(int i = 0 ; i < qp.nbFils ; i++)
       {
13     [[ Ajouter au noeud un opérateur de
         cache; ]]
14     addCacheToQueryPlan(qp.fils[i],
         doublonsFils);
15   }
16 }
```

6 Exemples d'applications

Après avoir définit de façon formelle notre proposition de cache. Voyons les effets qui seront appliqués sur deux exemples pratiques. Tout d'abord nous verrons l'état des flux et des données avec ce nouveau traitement. Et enfin nous verrons un exemple de répartition des nouveaux opérateurs sur le plan d'exécution de requête.

6.1 Les principes illustrés

Dans ce premier exemple, nous allons voir comment nos premiers principes de suppressions de doublons fonctionnent et leurs conséquences.

Nous avons cette requête : reporter toutes les trois secondes l'*identifiant* de la pièce dont la température et l'humidité sont trop importantes (resp. 20ºC et 15%). En *CQL* cette requête s'écrit :

SELECT t.id
FROM St AS t [Range 3], Sh AS h [Range 3]
WHERE t.id = h.id AND
t.value >= 20ºC AND h.value >= 15%

Voyons le résultat et le comportement de cette requête suivant les différents principes que nous avons évoqués dans la Fig. 3.

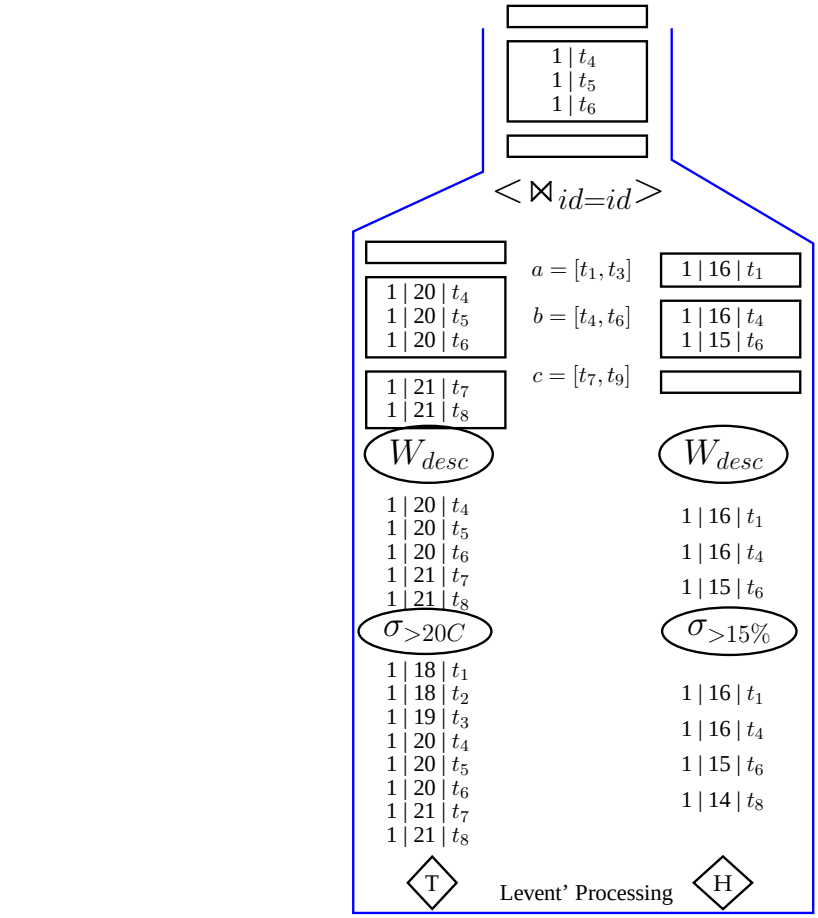
Nous supposons la présence des capteurs T et H respectivement de température et d'humidité. Le schéma ci-dessus décrit la répartition des opérateurs (de haut en bas, la jointure sur fenêtre, le créateur de fenêtre et la sélection. Afin d'illustrer le fonctionnement, ce schéma montre les files de tuples (ou de fenêtres) s'échangeant entre les opérateurs. Pour des raisons de visibilités nous n'avons pas détaillé sur quels noeuds sont répartis les opérateurs.

En *(a)*, nous avons le procédé imaginé par la plupart des solutions (ici celle de Levent Gürgen). Ensuite en *(b)*, nous avons le procédé premièrement imaginé dans la section 3, avec l'emplacement du paquet de fin de flux. Nous remarquons le deuxième problème (technique) soulevé par la suppression de doublons. Dans la troisième fenêtre du coté du capteur d'humidité, on remarque que comme l'indice passe en dessous de 15, la fenêtre n'a plus aucun sens. Sans les gestions de doublons, on sait qu'il y aura au moins un tuple qui va arriver. Alors que dans notre cas, il peut ne pas en avoir car nous les avons retirer, d'où la lecture dans le cache. On notera d'ailleurs la création du tuple (1, 16, t_4) dans la deuxième fenêtre qui a été crée à partir du cache.

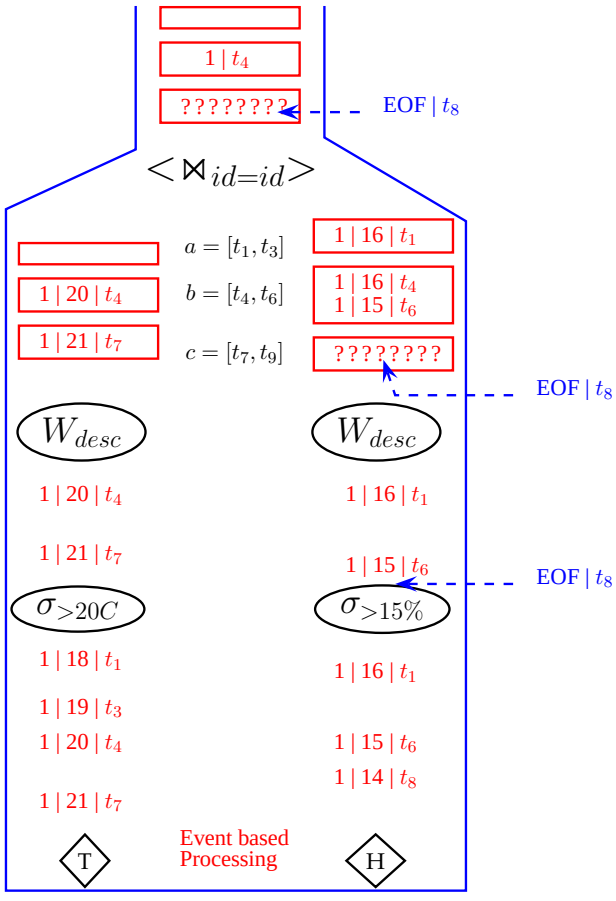
Enfin en *(c)*, nous avons la première solution avec l'ajout de paquets vides avec simplement le timestamp pour indiquer que la valeur n'a pas changé. Ici on peut voir une cohérence de la solution finale.

Et enfin, on peut voir la dernière solution avec les trois paquets EOF indiquant la fin de flux qui se rajoutent sur la première solution. En faisant cette manipulation, nous pouvons rendre valide cette première méthode comme nous l'avons expliqué.

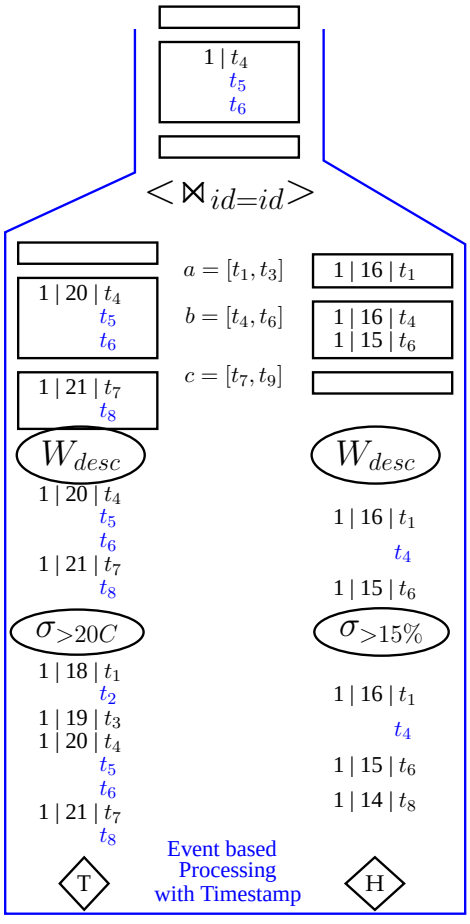
Nous voyons donc que nous avons une bonne amélioration puisque l'opérateurs de jointure se retrouve avec des largeurs de fenêtres allégés, donc on a clairement une bonne amélioration de performance dans le cas de lourds procédés.



(a) Système classique de Levent Gürgen



(b) Solution avec indicateurs de fin de flux



(c) Solution avec timestamps

FIGURE 3 – Les différents procédés de traitements des paquets sur une requête

6.2 Répartition des opérateurs

Prenons l'exemple d'une entreprise qui contrôle deux usines. Dans chaque usine il existe plusieurs capteurs de température. Certains sont là pour mesurer la température ambiante. Ces usines fabriquent des matières sensibles à la chaleur, donc la température ambiante, ne doit pas dépasser un certain seuil.

Le responsable aura donc une requête tournant sur le site de contrôle qui récupérera toutes les 2 minutes la température maximale de tous les capteurs d'ambiance durant les 4 dernières minutes.

Nous prendrons l'exemple d'un maximum sur une fenêtre glissante de 3 secondes :

SELECT MAX(t.value) FROM temp_sensors t
[RANGE 3 SLIDE 2]

Les capteurs étants sur deux passerelles différentes, le traitement final se fera sur le site de contrôle. Les passerelles seront simplement passantes, mais les proxys s'occuperont de la création de fenêtre et du maximum sur celle-ci.

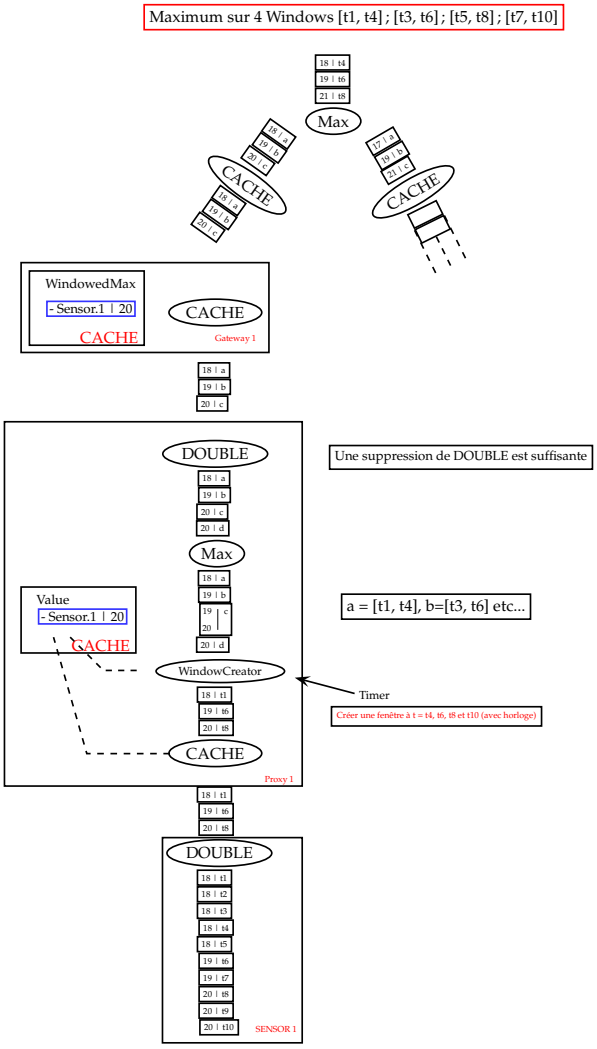


FIGURE 4 – Répartition et traitement des opérateurs de cache et de doublons

La Fig. 4 montre l'ensemble des tuples qui seront échangés ainsi que la répartition des opérateurs de cache.

Comme on peut le voir, au niveau du capteur, nous pouvons y insérer une suppression de doublons. Des solutions de systèmes d'exploitation pour capteur comme TinyOS [Tin] permettent des traitements de ce type directement à cet endroit. Ainsi seuls 3 paquets sont passés au lieu de 10. Lorsque la transmission de paquet est coûteuse en énergie (réseau sans fils), il est important de pouvoir conserver cette énergie pour plus de durabilité.

Dans le proxy, on remarque que le premier cache n'a pas de doublon à supprimer donc il enregistre dans le cache du proxy la valeur courante. Le créateur de fenêtre est lui armé d'un Timer qui lui dira quand créer ses fenêtres. Nous noterons qu'il est obligé de piocher dans le cache pour créer la fenêtre $c = [t_5, t_8]$ par exemple. Après le calcul du maximum, un opérateur de cache est encore appliqué pour pouvoir fournir des données atomiques (la fenêtre d est donc ignorée).

La gateway fait son travail de passer tout en enregistrant dans le cache (sans suppression de doublons). Car si un site voulait se brancher à cette gateway, alors les résultats pourraient se récupérer directement de la gateway sans aller chercher le capteur.

Le site de contrôle traite l'information des deux gateways et on obtient le résultat final sans avoir de charge superflue.

Un énorme avantage qui est notable dans ce genre de requête. Imaginons que la température limite soit $20C$. Donc le capteur 2 a reçu une température trop forte. Toutefois, l'opérateur voudra savoir de quel capteur il s'agit. Ici il n'aura qu'à regarder le détail de la requête supérieur qui est : *Maximum sur une fenêtre du Capteur 1* et *Maximum sur une fenêtre du Capteur 2*. Or ces requêtes seront instantanés car le cache inférieur du site de contrôle comprend ces informations.

7 Conclusions et perspectives

Notre but principal était d'étudier l'opportunité d'application des techniques de caches de requêtes dans le cas d'interrogation de flux de capteurs. En analysant de plus près, nous avons pu trouver que des paquets étaient potentiellement redondants ce qui entraînait des redondances de traitements et donc, une charge processeur supplémentaire.

En analysant, nous avons pu trouver une solution de cache introduisant deux opérateurs : l'un assigné au cache, et l'autre à la suppression. Le tout permettant la suppression de données si celle est réellement possible, ce qui entraîne, dans le cas de lourds traitements tels que les jointures ou agrégation, des bénéfices de performances parfois importants.

De plus, notre solution permet de pouvoir traiter plusieurs requêtes en même temps en partageant les résultats obtenus.

Toutefois, deux points de vues n'ont pas étés suffisamment abordés et peuvent être le sujet d'une étude plus approndie. Nous ne nous sommes pas concentré en détail sur la gestion de requêtes multiples. En effet, notre système permet l'accès aux ressources, chose que nous n'avions pas avant. Malgré tout, il n'y a pas eu d'approfondissement sur la manière en pratique de gérer ceci. L'autre point étant la première optimisation sur la localisation des noeuds. Nous avons dit que des améliorations peuvent être apportés mais nous n'avons pas vu en détail lesquelles et comment les amener.

Remerciements

Je tenais absolument à remercier mes deux tuteurs Claudia Roncancio et Cyril Labbé pour m'avoir si bien accueilli dans le laboratoire. Ils ont su comment m'aiguiller à travers cette première expérience en recherche et c'était vraiment un plaisir de travailler avec eux. Je remercie encore Claudia pour m'avoir donné un vrai sujet de recherche dont les pistes étaient encore inexplorées, ce qui constituait un vrai challenge pour moi. Je comptais partir en recherche, j'en suis maintenant convaincu.

Références

[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 B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, Asilomar, CA, USA, 2005.

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. Stream : The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1) :19–26, 2003.

[ACC+03] D. Abadi, D. Camey, U. Cetintemel, M. Chemiack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora : A new model and architecture for data stream management, 2003.

[AHS06] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Global Sensor Networks. Technical report, 2006. Submitted to IEEE Communications Magazine.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer 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.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq : A scalable continuous query system for internet databases. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 379–390. ACM, 2000.

[CEF+04] Owen Cooper, Anil Edakkunni, Michael J. Franklin, Wei Hong, Shawn R. Jeffery, Sailesh Krishnamurthy, Frederick Reiss, Shariq Rizvi, and Eugene Wu 0002. Hifi : A unified architecture for high fan-in systems. In *VLDB*, pages 1357–1360, 2004.

[Cha03] S. Chandrasekaran. Telegraphcq : Continuous dataflow processing for an uncertain world, 2003.

[dJLR07] Laurent d'Orazio, Fabrice Jouanot, Cyril Labbé, and Claudia Roncancio. Caches sémantiques coopératifs pour la gestion de données sur grilles. In *Actes des 23e Journées Bases de Données Avancées (BDA'2007)*, Marseille, France, October 2007.

[GLOR05] Levent Gürgen, Cyril Labbé, Vincent Olive, and Claudia Roncancio. Une architecture hybride pour i'interrogation et i'administration des capteurs. In *UbiMob '05 : Proceedings of the 2nd French-speaking conference on Mobility and uibquity computing*, pages 37–44, New York, NY, USA, 2005. ACM.

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

[Gür07] Levent Gürgen. *Gestion à grande échelle de données de capteurs hétérogènes.* PhD thesis, septembre 2007.

[Tin] TinyOS : An open-source OS for the networked sensor regime ; tep 114 : Sids : Source and sink independent drivers. http ://www.tinyos.net/tinyos-2.x/doc/html/tep114.html.