Fraîcheur et validité de données répliquées dans des environnements transactionnels

Cécile Le Pape — Stéphane Gançarski

Laboratoire d'Informatique de Paris 6 - Univ. Pierre etMarie Curie
8, rue du Capitaine Scott F_75015 Paris
{Cecile.Lepape,Stephane.Gancarski}@lip6.fr

RÉSUMÉ. *Nous proposons un canevas permettant de gérer la qualité des données d'un cluster de bases de données répliquées de façon optimiste. Il repose sur un modèle de qualité des données. Qualitativement, nous distinguons les notions de « fraîcheur » et de « validité » des données. Quantitativement, la qualité est exprimée par des mesures de divergence entre la donnée lue et un état « parfait » de la même donnée. L'utilisateur définit un niveau minimal de qualité pour ses requêtes, que le système utilise pour élaborer le plan d'exécution satisfaisant le plus rapide. Notre approche est un intergiciel respectant l'autonomie des applications et des bases de données. Elle a été mise en œuvre dans le prototype* Refresco *dans le cas du contrôle de la fraîcheur pour des configurations répliquées en mode monomaître. Les résultats obtenus montrent que le relâchement de la fraîcheur peut augmenter les performances significativement et que ces bénéfices sont encore plus nets lorsque la fraîcheur est spécifiée au niveau de la relation plutôt qu'au niveau de la base de données.*

ABSTRACT. *We propose a framework for managing the quality of data replicated optimistically on a database cluster. It is based on a model of quality of data. Qualitalively, we make the difference between "freshness" and "validity" of data. Quantitatively, data quality is expressed through divergence measures between the data read and the same data with perfect quality. Users specify a minimum level of quality and the system uses it to elaborate the fastest satisfying execution plan. Our approach is a middleware that preserves application and database autonomy. We provide an experimental validation for mono-master replication configurations and freshness relaxing based on our prototype* Refresco*. The results show that freshness control can help increase query throughput significantly. They also show significant improvement when freshness requirements are specified at the relation level rather than at the database level.*

MOTS-CLÉS : *qualité des données, réplication, cluster, fraîcheur, validité.*

KEYWORDS: *data quality, replication, cluster, freshness, validity.*

RSTI - ISI – 9/2004. Qualité des systèmes d'information, pages 165 à 185

1. Introduction

Les transactions représentent depuis longtemps un des outils fondamentaux permettant d'assurer la qualité d'un système d'information. Les systèmes de gestion de base de données (SGBD) garantissent pour la plupart une bonne exécution des transactions au travers des bien connues propriétés ACID (Atomicité, Cohérence, Isolation et Durabilité). Cependant, les mécanismes internes aux SGBD (ex. verrouillage en deux phases) permettant de garantir ces propriétés ont un coût qui peut ralentir le système et par conséquent augmenter les temps de réponse des transactions. L'une des approches pour améliorer la situation est de considérer séparément les transactions de mise à jour (qui influent sur le contenu des données stockées) et les requêtes en lecture seule (qui ne modifient pas les données). Si les propriétés ACID doivent être absolument respectées pour les premières afin de ne pas introduire d'incohérence dans les données (irrémédiable dans le cas général), les requêtes quant à elles peuvent accepter le relâchement de certaines propriétés pour bénéficier de meilleurs temps de réponse. En effet, l'utilisation des résultats de telles requêtes reste à la discrétion des utilisateurs qui les initient : ils peuvent aussi bien être insérés tels quels dans un document (rapport de statistiques par exemple) que servir uniquement d'indicateurs plus ou moins fiables (dans une recherche par exemple). C'est la raison pour laquelle de nombreuses approches ont proposé de relâcher les propriétés transactionnelles, principalement l'isolation, pour les requêtes en lecture seule. L'idée principale est de permettre à des requêtes de lire des données écrites par des transactions qui n'ont pas encore été validées, données dites « sales » afin de diminuer le temps de réponse.

Si, de plus, les bases de données sont répliquées, garantir les propriétés ACID des transactions a un coût encore plus important car il s'agit également de gérer la cohésion des répliques. L'intérêt de la réplication est qu'en dupliquant une même donnée sur plusieurs sites (ou nœuds), on améliore sa disponibilité et on augmente le débit transactionnel par l'équilibrage de la charge et le parallélisme. Cette solution s'avère de plus en plus économiquement viable grâce à la réduction des prix des machines et à l'augmentation de la vitesse des réseaux. Elle s'applique à de nombreux environnements (internet, réseaux mobiles, clusters de PC) et dans des configurations diverses, selon que les nœuds sont maîtres, *i.e.* acceptent les transactions de mise à jour, ou esclaves, *i.e.* n'acceptent que les requêtes en lecture seule et les propagations depuis les nœuds maîtres. Cependant, la réplication pose deux nouveaux problèmes : l'isolation globale et la fraîcheur. Garantir l'isolation globale signifie que toute transaction touchant une donnée doit mettre à jour l'ensemble des copies de la donnée avant qu'une autre transaction puisse y accéder, de façon à donner l'illusion d'une donnée unique (propriété connue sous le nom de *1-copy serializability*). En d'autre termes, toutes les copies d'une même donnée doivent voir passer les mêmes transactions dans le même ordre (ou dans un ordre compatible). Cette propriété peut être obtenue de plusieurs manières, selon le mode de réplication utilisé. En réplication synchrone, la transaction doit acquérir les verrous de toutes les copies avant de pouvoir la modifier, avant de suivre un processus de validation à deux phases pour assurer le consensus. Cependant, la lourdeur introduite par de tels protocoles de verrouillage et de validation fait que la réplication synchrone ne passe pas l'échelle en nombre de nœuds. En réplication asyn-

chrone, seule la copie locale est mise à jour par la transaction initiale. Les autres copies sont ensuite mises à jour par des transactions ultérieures qui propagent les effets de la transaction aux autres sites. Pour garantir l'isolation globale, des mécanismes supplémentaires doivent être introduits afin de garantir que les effets produits sont équivalents à ceux de la réplication synchrone. Il s'ensuit que, pendant un certain temps, les copies d'une même donnée divergent : certaines ont déjà reçu la propagation des effets d'une transaction alors que d'autres sont en attente. Ce phénomène introduit la notion de fraîcheur d'une donnée : plus une donnée diverge par rapport aux copies ayant déjà reçu toutes les mises à jour (exécutées localement ou bien propagées depuis d'autres sites), moins elle est fraîche. Là encore, l'utilisateur peut accepter qu'une requête en lecture seule lise des données qui ne sont pas parfaitement fraîches si la réponse est plus rapide et si la fraîcheur est contrôlée.

Bien que reposant sur des phénomènes différents, la lecture de valeurs sales et la lecture de valeurs obsolètes sont structurellement très semblables, puisque dans les deux cas, les valeurs lues diffèrent de la valeur idéale (validée dans un cas, fraîche dans l'autre) par les effets d'un certain nombre de transactions (transactions non validées, transactions pas encore propagées). Notre approche est d'exploiter cette similitude afin de traiter simultanément les deux approches au sein du même canevas. Dans ce cadre, les problèmes à résoudre sont les suivants : comment mesurer la divergence entre une valeur lue et la valeur parfaite correspondante ? Comment permettre à l'utilisateur de spécifier ses besoins en termes de qualité requise des valeurs qu'il va lire ? Comment garantir que ces besoins vont être satisfaits ? Comment exploiter le relâchement de la qualité pour augmenter le débit du système en équilibrant au mieux la charge ?

Plusieurs approches ont abordé partiellement le problème, comme par exemple [GRA 76, YU 00a, WU 92, RÖH 02]. Cependant, la plupart d'entre elles ne traitent pas fraîcheur et validité simultanément. La plupart ne fonctionnent que si l'on modifie considérablement le gestionnaire de concurrence existant, ce qui en réduit l'applicabilité car en pratique il est très coûteux de faire de telles modifications, voire impossible lorsque le code source du SGBD sous-jacent n'est pas disponible. Certaines, comme [RÖH 02], n'offrent qu'une seule mesure divergence et ne fonctionnent que sur certaines configurations (réplication par copie primaire). Enfin, aucune n'offre une interface permettant aux utilisateurs de spécifier leurs besoins en termes de qualité requise des données lues de manière ergonomique.

Notre approche propose de répondre au problème de manière globale. Ses principales contributions sont : (1) un modèle de qualité traitant uniformément validité et fraîcheur, intégrant plusieurs mesures de divergence, à plusieurs niveaux de granularité, (2) une approche intergicielle générique utilisable directement entre toute application et SGBD relationnel existant, sans modification ni de l'un ni de l'autre, (3) des algorithmes efficaces de calcul des mesures permettant d'équilibrer la charge transactionnelle en fonction des besoins de qualité des requêtes, et/ou de maintenir les sites à un niveau de qualité de données défini par les utilisateurs, (4) une première expérimentation du canevas proposé dans le cadre d'un service d'hébergement d'applications pharmaceutiques.

La suite de cet article se compose de la manière suivante. La section 2 présente le modèle de qualité, intégrant différentes mesures de divergence et leur évaluation dans un contexte d'intergiciel, et son application à différentes configurations. La section 3 présente l'architecture de notre canevas. Le prototype *Refresco*, qui met en œuvre une partie du canevas, fait l'objet de la section 4. La section 5 présente les travaux connexes. La section 6 conclut cet article.

2. Modèle de qualité basé sur la fraîcheur et la validité

Cette section propose un modèle pour définir la qualité de données relationnelles répliquées de façon optimiste. La notion de qualité recouvre traditionnellement beaucoup de dimensions (complétude, validité, cohérence, pertinence, précision, etc.). Dans cet article, nous nous intéressons uniquement aux dimensions de *fraîcheur* et de *validité* des données. Au cœur de ce modèle, la notion de *qualité d'une donnée* est définie de façon à être adaptée à différentes politiques de réplication. Quatre mesures de qualité sont définies, correspondant à des besoins applicatifs divers. Nous proposons des algorithmes d'évaluation de ces mesures sans intrusion dans les SGBDs locaux. Le modèle de qualité des données permet aussi de spécifier des besoins plus complexes portant sur un ensemble de données, sous la forme d'une *formule de qualité*. Une telle formule de qualité est adaptée aussi bien aux besoins d'une requête qu'à la maintenance de la qualité des données un nœud du cluster.

2.1. *Définitions*

2.1.1. *Modèle des transactions*

L'utilisateur envoie des transactions composées d'opérations d'écritures et de lectures sur les données. Comme illustré par la figure 1, une transaction passe de l'état « acceptée » à l'état « exécutée » [1], puis à l'état « validée » ou « abandonnée » et enfin à l'état terminal « notifiée ».

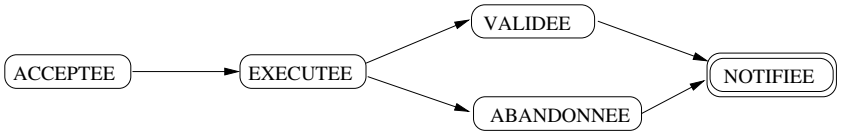

Figure 1. *Etats d'une transaction*

– *ACCEPTEE* : le système garantit à l'utilisateur qu'il sera notifié de la terminaison de la transaction et recevra ses résultats.

– *EXECUTEE* : la transaction a lu ou écrit des données et ses effets sont visibles par d'autres requêtes.

1. Les cas de panne ne sont pas traités dans cet article.

– *VALIDEE* : les effets de la transaction sont durables et ne peuvent plus être défaits. Nous faisons l'hypothèse que les transactions de mise à jour sont validées par le système de réplication sous-jacent selon un ordre global de référence.

– *ABANDONNEE* : les effets de la transaction sont défaits.

– *NOTIFIEE* : l'utilisateur a été notifié de la terminaison de la transaction et a éventuellement reçu ses résultats.

Ces états ont des sens différents selon le système de réplication sous-jacent et la configuration des nœuds. Dans le cas d'une configuration mono-maître (*primary copy* [GRA 96]) seul le nœud maître reçoit les mises à jour et les propage ensuite aux autres nœuds, la validation locale sur le nœud maître suffit à valider globalement la transaction. A l'inverse, dans une configuration multimaître (*primary group*), il faut l'accord de tous les nœuds du groupe pour qu'une mise à jour soit validée après son exécution sur chaque nœud.

2.1.2. *Modèle des données*

On distingue d'une part la *donnée logique* a et d'autre part les *copies physiques* a_i sur les nœuds. La donnée logique a correspond à la vue logique demandée par les utilisateurs, *i.e.* la donnée demandée par le code de l'application. Selon la granularité désirée, a est une partie plus ou moins importante de la base de donnée. A la granularité la plus grossière, la base toute entière peut être considérée comme une unique donnée. A l'autre extrême, la donnée la plus précise d'une base de données relationnelle est l'attribut d'un tuple, appelé *élément*. Les granularités intermédiaires sont la relation, un ensemble de tuples ou l'ensemble des éléments d'une colonne (*Attribut*).

Donnée : := BD | Relation | Tuples | Attribut | Element

Une copie physique a_i est la donnée physiquement stockée sur un nœud i. Une donnée logique peut correspondre à plusieurs copies physiques lorsqu'elle est répliquée.

La donnée logique reflète l'*état de référence*, c'est-à-dire celui dans lequel devraient se trouver toutes les copies si elles reflétaient exactement les transactions de mise à jour validées, dans l'ordre global de validation. Une copie sera dite *de qualité parfaite* si elle est dans l'état de référence.

Propriété 1 *Une copie est de qualité parfaite si et seulement si*

– *elle est* **valide**, i.e. *elle ne reflète que des transactions à jour validées, dans l'ordre global de validation, et si*

– *elle est* **fraîche**, i.e. *elle reflète toutes les transactions validées***.**

Dans le cadre d'une politique de réplication asynchrone, les copies physiques ne sont pas toujours de qualité parfaite. La qualité d'une copie est obtenue en mesurant la divergence entre l'état de la copie et celui de la donnée logique. En fonction du mode de validation et de propagation des mises à jour, la copie peut passer par différents

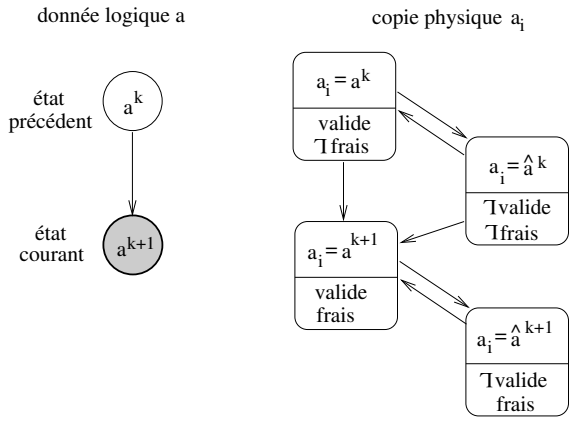

Figure 2. *Etats d'une donnée logique et des copies physiques*

états illustrés par la figure 2. Supposons qu'une donnée logique a passe de l'état a^k à l'état a^{k+1} par la validation d'une nouvelle transaction T_k. L'état a^{k+1} devient donc l'état de référence. Soit a_i une de ses copies physiques initialement dans l'état a^k. Si a_i reste dans cet état, elle n'est plus fraîche car elle ne reflète pas les modifications de T_k. Mais elle est toujours valide. Si a_i est modifiée, elle peut passer par les états suivants :

$a_i : a^k \rightarrow a^{k+1}$ La transaction T_k est directement appliquée sur a_i qui revient à une qualité parfaite.

$a_i : a^k \rightarrow \hat{a}^k$ Une ou plusieurs transactions de mise à jour non validées ont été appliquées sur a_i à partir de l'état a^k. La copie n'est donc plus valide. Elle n'est plus fraîche non plus car elle ne reflète pas encore T_k. A partir de cet état $\hat{a}^k$ soit les transactions non validées sont abandonnées, soit une transaction de réparation permet de ramener la copie dans un état parfait.

$a_i : a^{k+1} \rightarrow \hat{a}^{k+1}$ Une ou plusieurs transactions de mises à jour non validées ont été appliquées sur a_i à partir de l'état a^{k+1}. La copie n'est donc plus valide mais elle est toujours fraîche car elle n'est en retard sur aucune transaction validée.

Une donnée peut être en attente de rafraîchissement ou de validation. On appelle *AttExec(a_i)* l'ensemble des transactions validées sur la donnée (logique) a et en attente d'exécution sur la copie physique a_i. On appelle *AttValid(a_i)* l'ensemble des transactions effectuées sur la copie a_i et en attente de validation. Ces mises à jour peuvent finalement être validées ou bien abandonnées. On appelle enfin *Att(a_i)* l'ensemble des transactions de mises à jour en attente sur a_i : *Att(a_i) = AttExec(a_i) ∪ AttValid(a_i)*

2.2. *Applications à différentes configurations*

La définition de la qualité des données présentée section 2.1.2 s'applique à différentes configurations des nœuds du cluster. Nous considérons ici quatre configura-

tions qui reprennent et étendent la classification donnée par [GRA 96]. La première correspond aussi bien à une configuration non répliquée qu'à un mode de réplication synchrone. Les trois suivantes correspondent à des modes de réplication asynchrone différents. La figure 3 résume le type de qualité des données qui existe potentiellement selon les configurations. Ces configurations peuvent être combinées pour former des configurations hybrides.

configuration	valide et ¬ frais	¬ valide et frais	¬ valide et ¬ frais
un seul nœud / replication synchrone	impossible	scénario 1	impossible
monomaître	scénario 2	scénario 1 sur le nœud maître	impossible
multimaîtres	impossible	scénario 3	impossible
multidélégués	idem scénario 2	idem scénario 1	scénario 4

Figure 3. *Type de qualité selon les configurations*

Par la suite, on note $L_i(a_k)$ (respt. $E_i(a_k)$) la lecture (respt. l'écriture) du granule a_k du nœud k par la transaction T_i. On note également E^* la propagation sur un nœud des effets d'une écriture E initialement effectuée sur un autre nœud.

Afin de couvrir la plupart des modèles de réplication existant, nous considérons les trois types de nœud suivants :

– un *nœud maître* accepte les transactions de mise à jour et les requêtes en lecture seule. Il participe à la validation globale d'une transaction : son accord est nécessaire pour que la transaction soit validée globalement.

– un *nœud délégué* accepte les transactions de mise à jour et les requêtes en lecture seule mais ne participe pas à la validation globale d'une transaction.

– un *nœud esclave* n'accepte que les requêtes en lecture seule. Il est mis à jour uniquement par les propagations des effets des transactions effectuées sur les autres types de nœud.

2.2.1. *Un seul nœud ou réplication synchrone*

Dans ces configurations, ou bien le cluster est réduit à un seul nœud ou bien tous les nœuds fonctionnent selon un mode de réplication synchrone. Ces deux configurations sont présentées simultanément car elles offrent les mêmes types de qualité. Ce sont les configurations les plus simples : toute validation locale entraine la validation globale d'une transaction (directement dans le premier cas, après une validation globale dans le deuxième cas). Néanmoins, le relâchement de l'isolation locale peut entrainer la lecture de données non valides, ainsi que l'illustre le scénario suivant sur un nœud k quelconque :

(scenario 1) nœud k : $L_1(a_k)\ E_1(a_k)\ L_2(a_k)\ C_1$

Au moment de $L_2(a_k)$, a_k n'est pas valide car elle reflète l'écriture $E_1(a_k)$ de la transaction T_1 qui n'est pas encore validée. Ce scénario se produit au niveau « READ UNCOMMITTED » des niveaux d'isolation ANSI, dénoté comme une lecture sale. Par ailleurs, quels que soient les scénarios, les données sont toujours fraîches car si une transaction est validée, la copie locale reflète nécessairement les mises à jour.

2.2.2. *Monomaître*

Cette configuration comporte un nœud maître et plusieurs nœuds esclaves, de façon similaire à la configuration *primary master* de [GRA 96]. Les mises à jour sont envoyées sur le nœud maître. Une transaction est validée dès qu'elle est validée sur le nœud maître. Les autres nœuds, esclaves, attendent simplement que les mises à jour du nœud maître leur soient transmises. Sur le nœud maître, le même phénomène de lectures de données non valides que pour la configuration précédente peut se produire. Sur les nœuds secondaires, les données peuvent ne pas être fraîches mais sont toujours valides comme l'illustre le scénario suivant :

(scenario 2)
nœud maître N_0 : $L_1(a_0)\ E_1(a_0)\ C_1$
nœud secondaire N_i : $L_2(a_i)\ E_1^*(a_i)$

Au moment de la lecture $L_2(a_i)$ de a_i sur le nœud N_i, a_i est valide mais pas fraîche car elle ne reflète encore pas l'écriture de a par T_1 qui a été validée sur le nœuf maître N_0. Ce scénario se produit dans les systèmes qui permettent la lecture de données cachées (documents en cache dans les navigateurs, entrepôts de données qui ne sont pas rafraîchis immédiatement).

2.2.3. *Multimaîtres*

Cette configuration ne comporte que des nœuds maîtres et correspond à la configuration *lazy group* dans [GRA 96]. Les mises à jour peuvent être envoyées sur tous les nœuds du cluster. Il faut l'accord de tous les nœuds pour qu'une transaction de mise à jour soit validée. La lecture d'une donnée fraîche, mais non valide peut se produire, comme illustré par le scénario suivant :

(scénario 3)

nœud i N_i :	$L_1(a_i)\ E_1(a_i)\ L_2(a_i)$	C_1
nœud j N_j :	$E_1^*(a_j)$	C_1

Au moment de la lecture $L_2(a_i)$ sur le nœud N_i, a_i n'est pas valide car la transaction T_1 n'a pas encore été validée globalement par N_i et N_j. Elle est cependant toujours fraîche.

2.2.4. *Multi-délégués*

Cette configuration ne comporte que des nœuds délégués, ainsi qu'un nœud maître N_0. Les mises à jour peuvent être envoyées sur tous les nœuds du cluster. Seul N_0 est autorisé à valider globalement les transactions. Le scénario 1 peut se produire sur ce nœud de validation N_0. Le scénario 2 peut également se produire si une transaction initiée sur N_0 est en attente de propagation sur les autres nœuds. Dans cette configura-

tion certains scénarios peuvent conduire à la lecture de données à la fois non fraîches et non valides, comme illustré ci-dessous :

(scénario 4)

nœud N_0 (maître) :	$E_1(a_0)\ C_1$			$E_2^*(a_0)\ A_2$
nœud N_i (délégué) :		$E_2(a_i)L_3(a_i)$		

Au moment de la lecture $L_3(a_i)$ sur le nœud N_i, a_i n'est pas valide car la transaction T_2 n'a pas encore été validée par N_0. Elle n'est pas fraîche non plus car E_1 qui a été validée n'a pas été encore été propagée.

2.3. *Mesures de divergence*

Diverses mesures de divergence ont été proposées dans [ALO 90, BAR 94, GAL 95, WU 95, YU 00a]. Nous les adaptons à notre définition de la qualité et à notre architecture intergicielle. Nous proposons également une définition adaptée à différentes granularités des données.

Soit a une donnée logique. Soit a_i une copie physique de a et *Attr(*a_i*)* les transactions en attente sur a_i. Nous définissons quatre mesures, *NbOp*, *NbElt*, *Age* et *Num* pour évaluer la qualité de a_i, c'est-à-dire la divergence entre a et a_i.

Mesure_de_Divergence : := NbOp | NbElt | Age | Num

2.3.1. *NbOp*

C'est le nombre de transactions en attente sur la donnée.

NbOp(a_i) = | *Att*(a_i)|

Cette mesure est utile par exemple pour les relations d'historisations, qui reflètent la suite des évènements s'étant produits sur une partie de la base (historique des ventes, etc.).

2.3.2. *NbElt*

C'est le nombre d'éléments non valides ou non frais de la donnée. Si la donnée est un élément (donnée atomique) elt_i,

$$\textit{NbElt}(elt_i) = \begin{cases} 0 & \text{si } \textit{Att}(elt_i) = \oslash \\ 1 & \text{sinon} \end{cases}$$

Pour toute donnée composée de plusieurs éléments (un tuple, un ensemble de tuples, une colonne, une relation ou la base), c'est la somme :

$\textit{NbElt}(a_i) = \sum_{elt_i \in a_i} \textit{NbElt}(elt_i)$

Cette mesure est adaptée aux applications qui font surtout des mises à jour de tuples plutôt que des insertions/suppressions de tuples. Pour un ensemble de tuples constant, elle permet de mesurer le nombre de mises à jour effectuées.

2.3.3. *Age*

C'est l'âge de la plus ancienne transaction en attente sur la donnée.

Age(a_i) = dateCourante() - Min ({dateMiseAJour(t,a_i) , t ∈ AttValid(a_i)} ∪ {dateValidation(t), t ∈ AttExec(a_i)})

où *dateMiseAJour(t,a_i)* est la date [2] de mise à jour de la donnée sur le nœud et *dateValidation(t,a_i)* est la date de validation de t sur le nœud.

Cette mesure est adaptée à des contraintes de nature temporelles. C'est le cas par exemple de toute application qui fait du rafraîchissement périodique (notification de nouveaux mails, de mouvements de stocks, de news ; rafraichissement d'un entrepôt de données).

2.3.4. *Num*

C'est la distance euclidienne entre la valeur de la , soit à l'ensemble des éléments d'une même colonne.

Si la donnée est un élément elt_i (elt étant la valeur courante, parfaite, de la donnée logique),

$$Num(elt_i) = | elt - elt_i |$$

Si la donnée est l'ensemble des éléments d'une colonne,

$$Num(a_i) = Max\{Num(elt_i), elt_i \in a_i\}$$

Cette mesure est bien adaptée à des applications numériques, comme le calcul de statistiques ou la gestion de stocks par exemple.

2.4. *Spécifications de la qualité des données*

Spécifier la qualité requise d'une donnée consiste à borner la divergence autorisée. Pour cela, nous définissons la notion d'*atome de qualité*.

Atome_de_Qualité : := (Donnée, Mesure_de_Divergence, Seuil, Mode)

Spécifier un tel atome consiste tout d'abord à choisir une donnée a, en fonction de la granularité désirée. Ensuite, le *seuil* s est une borne maximale de divergence autorisée pour la *mesure de divergence* m voulue. Il s'agit donc d'une condition de la forme $m(a) \leq s$. Enfin, le choix du mode permet de qualifier la nature de la divergence.

Mode : := FRAICHEUR | VALIDITE

2. Nous faisons l'hypothèse que l'horloge des nœuds est synchronisée. C'est une hypothèse raisonnable dans le contexte d'un cluster de machines.

Avec le mode *fraîcheur*, il s'agit de mesurer la fraîcheur de la donnée qui se base sur les transactions validées en attente de propagation. Avec le mode *validité*, l'évaluation concerne les mises à jour effectuées sur la donnée mais toujours en attente de validation (ou d'annulation).

Nous définissons ensuite la notion de *formule de qualité* comme une combinaison logique de plusieurs atomes de qualité :

Formule_de_Qualité : := Atome_de_Qualité
| Formule_de_Qualité $\vee$ Formule_de_Qualité
| Formule_de_Qualité $\wedge$ Formule_de_Qualité

Combiner dans une formule plusieurs atomes de qualité permet de spécifier la qualité requise d'un ensemble de données distinctes. Elle permet également de spécifier une formule de qualité complexe pour une même donnée. Ainsi, une formule de qualité permet aussi bien de spécifier la qualité des requêtes en associant une formule à une requête, que de spécifier la qualité des nœuds, en associant une formule à l'ensemble des noeuds.

3. Architecture

La figure 4 présente l'architecture cluster de bases de données, qui étend [LEP 04]. Elle se présente sous la forme d'un intergiciel, accessible *via* une interface standard JDBC et qui masque la couche de données à l'application. Globalement, l'application envoie ses demandes de transactions à l'intergiciel (*transaction shipping*) qui choisit le nœud d'exécution adéquat du cluster en fonction de la qualité demandée et qui retourne les résultats à l'application. Une architecture d'intergiciel a l'avantage d'être indépendante des SGBDs locaux, donc par exemple de supporter des bases de données hétérogènes. Elle permet également de porter le système sur des SGBDs propriétaires dont les sources sont indisponibles, qui sont majoritaires sur le marché. Remarquons que les bases de données peuvent être géographiquement distantes si le réseau qui les relie est à haut débit. Dans ce cas, l'intergiciel est répliqué sur tous les nœuds et les structures de données qu'il gère sont placées en mémoire partagée.

3.1. *Métabase*

Tout d'abord, l'utilisateur spécifie la qualité des données qu'il désire. En effet, puisque la qualité est liée à la sémantique de l'application, elle ne peut pas être inférée par une simple analyse du code de la transaction. L'utilisateur a le choix entre deux politiques : une politique orientée requête ou une politique orientée cluster. Dans le premier cas, c'est aux requêtes qu'est associée la qualité désirée. La qualité concerne les données lues par les requêtes. Dans le deuxième cas, ce sont les nœuds de données constituant le cluster qui doivent satisfaire une qualité minimale. Dans ce cas, la même qualité est délivrée pour toutes les requêtes. En contrepartie, la gestion des données est uniforme sur tous les nœuds et pour toutes les requêtes, donc plus simple. Dans une

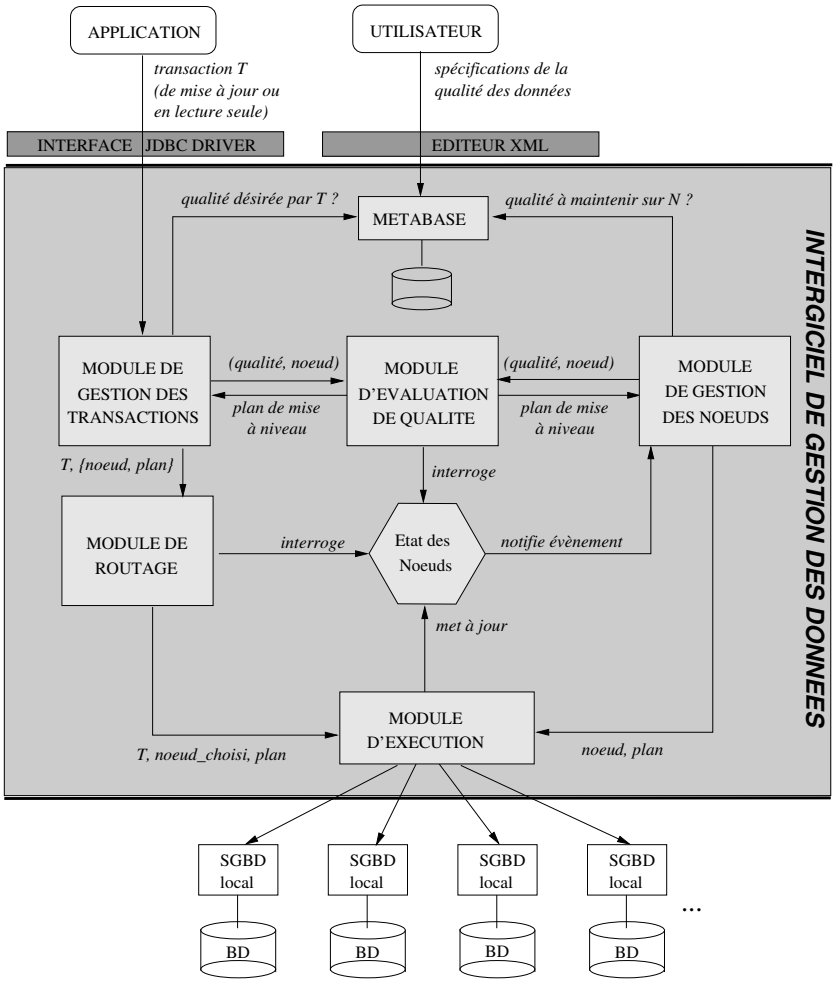

Figure 4. *Architecture*

politique orientée requête, la qualité est spécifiée individuellement mais la gestion des données du cluster est plus complexe car elle doit s'adapter à chaque traitement. Les deux politiques peuvent aussi être combinées : le cluster fournit des données de qualité minimale garantie pour l'ensemble des requêtes mais une requête peut spécifier ses propres besoins si elle désire une meilleure qualité. Dans tous les cas, le système gère la propagation des mises à jour dans les nœuds du cluster de façon à garantir les besoins spécifiés par l'utilisateur. Toutes ces informations sont spécifiées en XML au travers d'une interface adéquate et stockées dans un document appelé *Métabase*. Il n'est donc pas nécessaire de modifier le code de l'application. L'avantage de cette solution est de simplifier le portage des applications sur le système, ce qui est particu-

lièrement pertinent pour les applications de grande taille dont la complexité rendrait toute modification de code longue, coûteuse et source d'erreur.

3.2. *Module d'évaluation de qualité*

Ce module permet d'évaluer la qualité des données d'un nœud. Pour cela, il consulte l'état courant des nœuds du cluster. Dans le cas où les données hébergées par le nœud sont d'une qualité insuffisante, il calcule un *plan de mise à niveau* du nœud. Afin d'atteindre un niveau de fraîcheur satisfaisant, le plan consiste en une séquence de mises à jour à effectuer sur le nœud pour amener ses données à la qualité désirée. Afin d'atteindre un niveau de validité suffisant, le plan consiste à forcer l'ordre de sérialisation sur le nœud si le SGBD sous-jacent le permet. Sinon, il faut faire attendre la requête jusqu'à ce que suffisamment de transactions aient été validées. Il faut également s'assurer que les transactions ultérieures à la requête et pouvant réduire le niveau de validité du nœud soient sérialisées après la requête sur le nœud. A nouveau, ceci est obtenu soit en forçant le SGBD à sérialiser dans l'ordre d'arrivée, s'il le permet, soit en faisant atteindre les transactions jusqu'à ce que la requête soit exécutée.

Dans notre cadre d'une architecture intergicielle, nous évaluons la qualité des données à partir d'une connaissance externe de l'état des nœuds, c'est-à-dire des transactions de mise à jour qui y ont été exécutées. Nous nous appuyons d'une part sur l'analyse syntaxique du code des transactions et d'autre part sur des informations dynamiques fournies *a posteriori* soit par les drivers SQL soit par lecture des journaux des SGBDs locaux[3]. L'algorithme générique d'évaluation de la qualité d'une donnée est présenté dans la figure 5. La qualité d'une donnée a sur un nœud i pour une mesure m est évaluée comme la divergence entre la valeur de a sur i et sa valeur de référence, divergence donnée par la fonction *lireQualité(a,m,i)*. Pour chaque transaction t en attente sur la donnée a sur le nœud i, l'algorithme évalue la quantité de modifications faite par t sur a pour la mesure m avec la fonction *evalModifs(t,a,m)*, puis met à jour la divergence.

```
lireQualite( a, m, i){
  // a donnée, m mesure, i noeud de données
  // Att = ensemble des transactions en attente sur a pour le noeud i
  qualité = 0
  POUR t dans Att FAIRE
     qualité = changeQualite(qualite, m, evalModifs(t,a,m))
  FINPR
  retourne qualité
}
```

Figure 5. *Algorithme générique d'évaluation des mesures de qualité*

3. Par exemple, l'outil Logminer d'Oracle permet d'interroger les journaux

3.2.1. *Evaluation des transactions en attente*

Evaluer *Att(a_i)*, c'est estimer quelles sont parmi les transactions de mise à jour en attente sur le nœud N_i celles qui touchent la donnée a. Bien entendu, si la granularité de la donnée a est celle de la base, toutes les transactions en attente sont concernées. Si a est de granularité plus fine, certaines transactions ne touchent pas nécessairement la donnée. Il faut donc filtrer les transactions. L'analyse syntaxique du code des transactions est une méthode simple car de nombreux outils d'analyse de code SQL sont disponibles et c'est une méthode rapide car elle peut être effectuée à la compilation. Elle est bien adaptée aux cas où les données sont des relations ou des attributs car ces informations sont directement incluses dans le code des transactions. Si les données sont des tuples ou des éléments, cette méthode n'est suffisante que dans des cas très simples (la clause de sélection référence directement les identifiants des tuples touchés). En effet, dans le cas général, il n'est pas possible de connaître l'extension des transactions (les données qu'elles touchent) à partir de leur simple intension (leur code). Dans des cas plus complexes, la lecture des journaux devient nécessaire.

3.2.2. *Mise à jour générique de la qualité*

La qualité est mise à jour par la fonction *changeQualite(q, m, evalModifs(t,a,m))* qui, en fonction de la qualité courante q et des modifications induites sur la donnée a par la transaction t, rééevalue la qualité de a. Selon les mesures, il peut s'agir d'une augmentation de l'ancienne qualité (*NbOp, NbElt, Num*), du calcul d'un minimum entre l'ancienne et la nouvelle qualité (*Age*), etc.

3.2.3. *Evaluation des modifications opérées par une transaction*

L'évaluation *evalModifs(t,a,m)* des modifications d'une transaction t sur une donnée a dépend de la mesure m :

– *EvalModifs(t,a,NbOp)* est constant, égal à 1.

– *EvalModifs(t,a,NbElt) = ntt * nct*,

où *ntt* est le nombre de tuples touchés par la transaction et *nct* est le nombre de colonnes touchées par la transaction. L'information du *ntt* est retournée par le driver après exécution d'une transaction. L'information du *nct* est inférée du code de la transaction par analyse syntaxique.

– *EvalModifs(t,a,Age)* = θ - *dateEnvoi(t,i)*,

où θ est la date d'évaluation de la fonction et *dateEnvoi(t,i)* est la date d'envoi de t sur le nœud i qui constitue une évaluation par excès de la date de mise à jour de a par t sur i.

– *EvalModifs(t,a,Num)* est la mesure la plus délicate à évaluer. Elle s'obtient en combinant l'analyse du code de la transaction et les informations dynamiques sur son exécution (*ntt* ou journaux du SGBDs). Par exemple :

*EvalModifs("update R set attr=attr+10", R.attr, Num) = 10*ntt*

3.3. *Modules de gestion des transactions et de routage*

Le gestionnaire des transactions reçoit les demandes de transactions de l'application. S'il s'agit d'une requête, il interroge la métabase pour savoir s'il existe une demande particulière de qualité pour cette requête. Dans le cas contraire, les données lues seront d'une qualité par défaut (spécifiée dans la métabase). Pour chaque nœud du cluster susceptible d'exécuter les requêtes (cela dépend de la configuration du cluster), il consulte l'évaluateur de qualité pour connaître le plan de mise à niveau du nœud. Dans le meilleur cas, le nœud est de qualité suffisante et le plan est vide.

C'est ensuite le module de routage qui choisit le meilleur nœud d'exécution des transactions t. Pour cela, il évalue pour chaque nœud une fonction de coût *coût* qui tient compte de la charge du nœud i et du coût de l'éventuel plan de mise à niveau :

coût(t,i) = charge(i) + rafraîchissement(i,t) + validation(i,t)

La charge du nœud N_i, *charge(i)*, est évaluée comme la somme de l'estimation des temps d'exécution restants des transactions en cours d'exécution sur le nœud. Le coût du plan de mise à niveau est composé d'une part du coût de rafraîchissement si le plan prévoit un tel rafraîchissement du nœud et d'autre part du coût de validation du nœud si le plan prévoit un telle validation. Le coût de rafraîchissement *rafraîchissement(i,t)* du nœud N_i pour la transaction t est la somme de l'estimation des temps d'exécution des transactions qu'il faut propager sur N_i pour obtenir la fraîcheur désirée. Le coût de validation $validation(t, i)$ du nœud N_i pour la transaction t est la somme de l'estimation des temps d'exécution des transactions non validées sur le nœud N_i.

3.4. *Module de gestion des nœuds*

La maintenance de la qualité des données des nœuds est effectuée par le gestionnaire des nœuds. Il consulte la métabase pour connaître la politique de qualité du cluster. En fonction de la qualité à maintenir sur les nœuds et des événements qui modifient l'état des nœuds, il décide de la propagation des mises à jour d'un nœud sur l'autre et peut déclencher des plans de mise à niveau automatiquement sur certains nœuds du cluster.

3.5. *Module d'exécution*

La communication avec les SGBDs locaux est assurée par le module d'exécution. Il y envoie les demandes de transactions et les plans de mise à niveau, s'assure de leur validation et récupère les résultats demandés par les requêtes. Il valide les transactions selon un ordre global que nous appelons *ordre de référence* et met à jour le module qui reflète l'état des nœuds. Par exemple, dans le cas d'une configuration monomaître (un seul nœud maître et plusieurs nœuds esclaves), l'ordre de référence est l'ordre de sérialisation des transactions sur le nœud maître.

4. *Refresco* : un prototype opérationnel

Afin de valider notre approche, nous avons développé un prototype appelé *R*efresco (*Routing Enhancer through FREShness COntrol*), qui met en œuvre les concepts et algorithmes présentés dans les sections précédentes pour une configuration monomaître. Ce prototype a été développé en Java (jdk 1.4) dans le cadre du projet RNTL Leg@net dont le but est de porter sur le cluster d'un hébergeur (ASP) une application de gestion de pharmacie. La base de données est répliquée entièrement sur quatre nœuds, chacun doté d'un server Oracle 8i sous Linux. L'intergiciel est déployé sur un cinquième nœud. Tous les nœuds (Pentium IV 2Ghz, 512 Mb RAM) sont connectés par un réseau Ethernet haut débit (1 GBit/s). La base de données est conforme au banc d'essai TPC-R avec un facteur d'échelle de 1. Le banc d'essai est composé de six flux de transactions, qui correspondent à la fonction RF1 de TPC-R et de six flux de requêtes qui sont tirées au hasard parmi les requêtes TPC-R. Le temps d'exécution moyen est de 4ms pour une transaction et de deux minutes pour une requête. Chaque expérience a une durée de 20 minutes. Pour chaque expérience, nous mesurons le nombre de requêtes par heure.

Nous présentons par la suite les résultats les plus intéressants obtenus avec Refresco. Pour plus de résultats et de détails, le lecteur est invité à consulter [LEP 04].

4.1. *Impact du seuil de fraîcheur*

Ces expériences se concentrent sur l'influence du seuil de fraîcheur sur les performances des transactions et des requêtes, pour les mesures *Age*, *NbOp* et *NbElt* et la granularité BD. Le seuil varie de 0 à 1200 secondes pour la mesure *Age*, de 0 à 160000 transactions pour la mesure *NbOp* et de 0 à 240000 tuples pour la mesure *NbElt*. Au-delà de ces limites, le seuil devient trop grand par rapport à la durée de l'expérience (20 minutes) et même le nœud le plus obsolète serait assez frais pour satisfaire n'importe quelle requête.

Les premières mesures (non présentées ici pour des raisons d'espace) montrent que le débit des transactions sur le nœud maître n'est pas affecté par l'exécution des requêtes. Ce résultat, bien qu'évident, est important pour les applications où les transactions représentent une activité prioritaire. Par exemple, dans une pharmacie, les ventes au comptoir ne doivent subir aucun retard, par rapport aux requêtes statistiques sur les stocks ou le chiffre d'affaire.

La figure 6 montre que le relâchement de la fraîcheur améliore le débit des requêtes de manière significative. Par exemple, pour un seuil de 300s et pour la mesure *Age*, le système traite deux fois plus de requêtes que pour un seuil à 0, pour atteindre un débit de 70% du débit de référence (colonne de droite), quand aucune transaction n'est envoyée sur le nœud maître. Ce seuil de 300s est tout à fait acceptable par exemple dans les applications de pharmacie où les statistiques peuvent parfaitement être calculées sur des données obsolètes depuis des heures, voire des jours.

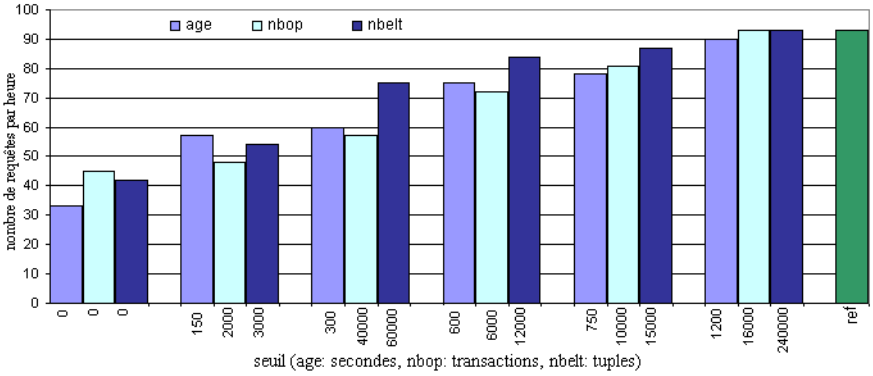

Figure 6. *Influence de la fraîcheur sur le débit des requêtes*

D'autres mesures montrent que la surcharge induite par le système (routage des requêtes et rafraîchissement des nœuds esclaves) reste raisonnable et décroît avec le seuil de fraîcheur. Le temps de routage est toujours négligeable alors que le temps passé à rafraîchir le devient à partir d'un seuil de 600s. Ceci s'explique par le fait que plus le seuil est grand, moins les nœuds nécessitent de rafraîchissement pour satisfaire les requêtes. Bien entendu, comme les nœuds deviennent de moins en moins frais avec le temps, leur rafraîchissement ultérieur sera plus coûteux, mais celui-ci pourra être effectué dans des périodes où les nœuds esclaves sont peu ou pas du tout utilisés (rafraîchissement en arrière-plan).

4.2. *Impact de la granularité*

Afin d'évaluer l'influence de la granularité sur les performances, nous avons défini un taux de conflit qui reflète la proportion de requêtes potentiellement en conflit sur les relations avec les transactions sur le nœud maître. Cette notion n'a de sens que pour les granularités plus fines que la base de données pour laquelle, par définition, toutes les requêtes sont potentiellement conflictuelles. Chaque expérience a été exécutée une fois avec la granularité BD, puis une fois avec la granularité Relation. Avec un taux de conflit assez bas (0.15), les bénéfices d'une granularité plus fine sont particulièrement intéressants puisque le temps moyen d'exécution d'une requête est divisé par 4.

La figure 7 montre le temps d'occupation des trois nœuds esclaves pour les deux granularités, en distinguant les requêtes conflictuelles avec les transactions de celles qui ne le sont pas. Avec une granularité BD, les requêtes étant toutes conflictuelles, elles sont routées de manière uniforme sur les trois nœuds. Avec une granularité Relation, les nœuds apparaissent spécialisés : le nœud 2 reçoit les requêtes non conflictuelles alors que les nœuds 1 et 3 reçoivent les requêtes conflictuelles. Les requêtes non conflictuelles peuvent être exécutées sans rafraîchissement, ce qui explique que

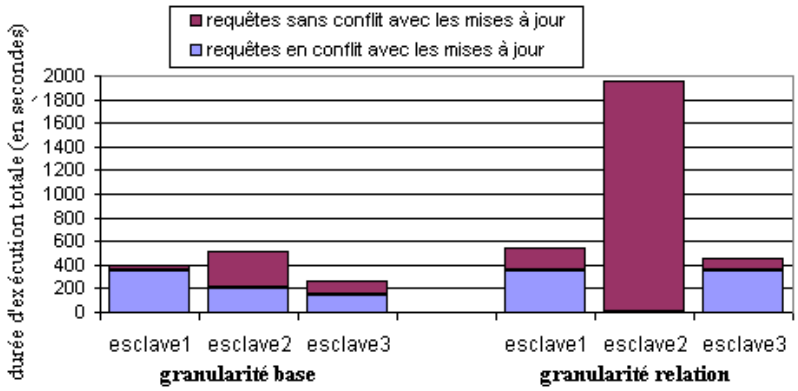

Figure 7. *Equilibrage de la charge des requêtes sur les nœuds esclaves*

le nœud 2 puisse traiter beaucoup plus de requêtes. Ce phénomène d'auto-adaptation vient du comportement de l'algorithme de routage qui inclut le coût de rafraîchissement comme critère de choix du nœud d'exécution d'une requête. Les nœuds qui viennent d'être rafraîchis pour exécuter des requêtes conflictuelles deviennent du coup de meilleurs candidats pour l'exécution des prochaines requêtes conflictuelles, et donc le phénomène se renforce avec le temps.

5. Travaux connexes

Autoriser la lecture de données obsolètes est une approche largement répandue pour améliorer les performances. Dans le cadre d'une architecture de cluster de nœuds, le projet FAS de l'ETH de Zurich [RÖH 02] montre comment le relâchement de la fraîcheur diminue les temps de réponse des requêtes OLAP et compare différentes stratégies de propagation des mises à jour. Cependant leur modèle ne propose qu'une seule mesure de divergence, similaire à notre mesure *Age*, et se limite à la granularité de la base de données. Dans le contexte de *data shipping*, où les données sont cachées chez le client, relâcher la fraîcheur des données cachées permet d'éviter l'interrogation systématique d'un serveur distant en cas de *cache miss*. Dans [ALO 90], les données cachées restent interrogeables tant qu'un seuil de fraîcheur n'est pas dépassé. Les mesures sont variées mais ne tiennent pas compte de la granularité des données. De plus les requêtes ne peuvent pas spécifier leurs besoins individuellement car la qualité est associée aux données du cache, pas aux requêtes. Dans le projet TRAPP de Stanford [OLS 00] les requêtes précisent des « contraintes de précision » que le système satisfait en combinant les données de plusieurs caches et du nœud maître distant. Ce projet est surtout dédié à l'optimisation des performances de requêtes d'agrégation numérique. De plus, il nécessite de reprogrammer les gestionnaires de transactions locaux. Le projet [GUO 04] propose également des garanties temporelles sur la fraîcheur de données cachées, en ajoutant le concept de classes de cohérence : toutes les données d'un même groupe proviennent d'une même image des données (*snapshot*). Leurs

spécifications prennent la forme d'une clause SQL supplémentaire, méthode élégante mais qui ne supporte pas l'exécution de code SQL existant.

Le concept de données invalides a été originellement défini par [GRA 76], inspirant largement par la suite le standard ANSI [ANS 92] et ses degrés d'isolation qu'implémentent plus ou moins la plupart des systèmes actuels. Néanmoins, il s'agit seulement de contrôler des phénomènes que l'utilisateur accepte de voir se produire ou non, pas de mesurer quantitativement la divergence. Les transactions epsilon [PU 91] offrent un framework théorique intéressant pour le contrôle de la divergence, proposant plusieurs métriques de la divergence et incluant la lecture de données invalides. Néanmoins, elles nécessitent la modification des gestionnaires de transactions locaux pour permettre une gestion fine de compteurs d'erreur « importée » ou « exportée » par transaction. Le projet TACT [YU 00a, YU 00b] propose une architecture intergicielle [YU 00a, YU 00b] qui implémente le modèle de cohérence continue autorisant un compromis entre précision et performances. La fraîcheur est spécifiée par des mesures similaires aux nôtres tandis que la mesure *Order Error* autorise la lecture de données ne respectant pas l'ordre global de sérialisation.

6. Conclusion

Dans cet article, nous présentons un canevas permettant de relâcher la qualité des données (isolation et fraîcheur) pour des requêtes en lecture seule afin d'augmenter les performances en temps de réponse. Cette approche est motivée par le fait que de nombreuses requêtes ne nécessitent pas une qualité parfaite et qu'il est donc possible d'exploiter un relâchement de la qualité des données qu'elles lisent. A ces fins, nous présentons un modèle qui traite de manière uniforme validité et fraîcheur. Ce modèle permet de spécifier, au travers d'une interface XML, le niveau de qualité requise par une requête à différents niveaux de granularité et pour plusieurs mesures de divergence. Pour chacune de ces mesures, une méthode d'évaluation est fournie. Ces méthodes peuvent servir à maintenir un nœud à un certain niveau de qualité et sont aussi intégrées à l'algorithme de routage des requêtes.

Notre approche est originale au sens où elle intègre validité et fraîcheur, plusieurs mesures et niveaux de granularité et se base sur une architecture intergicielle qui la rend exploitable sur de nombreuses configurations sans avoir à modifier ni le code des applications, ni celui des SGBD sous-jacents. A notre connaissance, aucune autre approche ne présente l'ensemble de ces propriétés.

Afin de valider notre canevas, nous avons développé le protype *Refresco* sur le cluster du LIP6, sous Linux et Oracle 8i. Les expérimentations menées sur ce prototype à l'aide du banc d'essai TPC-R montrent que des gains substantiels peuvent être obtenus en relâchant la fraîcheur avec un seuil raisonnable, quelle que soit la mesure utilisée et même avec un nombre de nœuds assez faible, ce qui est économiquement important. Ils montrent aussi que la surcharge induite par le routage et le rafraîchissement des nœuds reste acceptable et que la spécification de la qualité à un niveau de granularité fin permet d'augmenter encore les performances.

Les travaux actuels et futurs concernant notre framework sont nombreux et de nature variée. Tout d'abord, nous souhaitons compléter *Refresco* afin d'inclure de nouvelles mesures et de nouveaux niveaux de granularité. Ensuite, bien que le but *Refresco* soit de montrer qu'un faible nombre de nœuds peut suffire à obtenir des gains de performances satisfaisants, nos allons poursuivre les expérimentations sur des clusters de plus grandes tailles (ex. 64 nœuds) afin de vérifier que notre approche passe à l'échelle en nombre de nœuds. Enfin, nous allons mettre en œuvre le relâchement de la validité sur le nœud maître, en permettant aux requêtes de s'exécuter sur le nœud maître sur des états non validés.

Une autre direction est d'adapter les algorithmes à une configuration multimaître, afin d'éviter un phénomène d'engorgement sur le nœud maître. Pour cela, nous allons dans un premier temps nous focaliser sur la fraîcheur et utiliser une méthode asynchrone préventive entre les nœuds maîtres. Cette solution, introduite dans [GAN 02] puis développée dans [PAC 03, COU 04], exploite la vitesse du réseau haut débit pour offrir le même niveau d'isolation que la réplication synchrone sans la surcharge liée à cette dernière. A plus long terme, nous inclurons le relâchement de la validité en utilisant une méthode de réplication asynchrone optimiste (*i.e.* basée sur la réconciliation des transactions entre nœuds maîtres), introduite dans [GAN 02] et en cours de développement au LIP6.

7. Bibliographie

[ALO 90] ALONSO R., BARBARÁ D., GARCIA-MOLINA H., « Data Caching Issues in an Information Retrieval System », *ACM TODS*, vol. 15, n° 3, 1990, p. 359-384.

[ANS 92] American National Standard for Information Systems - Database language - SQL, « ANSI X3-135-1992 », November 1992.

[BAR 94] BARBARÁ D., GARCIA-MOLINA H., « The Demarcation Protocol : A Technique for Maintaining Constraints in Distributed Database Systems », *VLDB Journal*, vol. 3, n°3, 1994, p. 325-353.

[COU 04] COULON C., PACITTI E., VALDURIEZ P., « Scaling up the Preventive Replication of Autonomous Databases in Cluster Systems », *Int. Conf. VecPar (to appear)*, 2004.

[GAL 95] GALLERSDÖRFER R., NICOLA M., « Improving Performance in Replicated Databases through Relaxed Coherency », *Int. Conf. on VLDB*, 1995.

[GAN 02] GANÇARSKI S., NAACKE H., PACITTI E., VALDURIEZ P., « Parallel Processing with Autonomous Databases in a Cluster System », *Int. Conf. On Cooperative Information Systems (CoopIS)*, 2002.

[GRA 76] GRAY J., LORIE R. A., PUTZOLU G. R., TRAIGER I. L., « Granularity of Locks and Degrees of Consistency in a Shared Data Base », *IFIP Working Conference on Modelling in Data Base Management Systems*, 1976, p. 365-394.

[GRA 96] GRAY J., HELLAND P., O'NEIL P. E., SHASHA D., « The Dangers of Replication and a Solution », JAGADISH H. V., MUMICK I. S., Eds., *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, ACM Press, 1996, p. 173-182.

[GUO 04] GUO H., LARSON P.-A., GOLDSTEIN J., « Relaxed Currency and Consistency : How to Say "Good Enough" in SQL », GERHARD WEIKUM ARND CHRISTIAN KONIG S. D., Ed., *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM Press, 2004, p. 815-826.

[LEP 04] LE PAPE C., GANÇARSKI S., VALDURIEZ P., « REFRESCO : Improving Query Performance through Freshness Control in a Database Cluster », *20ème Journées Bases de Données Avancées, Montpellier 2004*, 2004.

[OLS 00] OLSTON C., WIDOM J., « Offering a Precision-Performance Tradeoff for Aggregation Queries over Replicated Data », *Int. Conf. on VLDB*, 2000.

[PAC 03] PACITTI E., ÖZSU T., COULON C., « Preventive Multi-Master Replication in a Cluster of Autonomous Databases », *Int. Conf. on Parallel and Distributed Computing (Euro-Par)*, 2003.

[PU 91] PU C., LEFT A., « Replica Control in Distributed Systems : an Asynchronous Approach », *SIGMOD conference*, 1991.

[RÖH 02] RÖHM U., BÖHM K., SCHEK H.-J., SCHULDT H., « FAS - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components », *Int. Conf. on VLDB*, 2002.

[WU 92] WU K.-L., YU P. S., PU C., « Divergence Control for Epsilon-Serializability », *Int. Conf. On Data Engineering (ICDE)*, 1992.

[WU 95] WU K.-L., PU C., « Divergence Control Algorithms for Epsilon Serializability », *Distributed and Parallel Databases*, vol. 3, n° 1, 1995.

[YU 00a] YU H., VAHDAT A., « Design and Evaluation of a Continuous Consistency Model for Replicated Services », *Proceedings of Operating Systems Design and Implementation*, 2000.

[YU 00b] YU H., VAHDAT A., « Efficient Numerical Error Bounding for Replicated Network Services », *Int. Conf. on VLDB*, 2000.