

UNIVERSITÉ DU QUÉBEC À MONTRÉAL

LE CACHE DANS LES BASES DE DONNÉES MOBILES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

SOMIA BELAIDOUNI

octobre 2014

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.07-2011). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

Remerciements

Je tiens tout d'abord à remercier Professeur Abdellatif OBAID, mon directeur de recherche, pour sa disponibilité, ses précieux conseils et ses encouragements qui m'ont toujours poussé à m'améliorer.

Une pensée toute spéciale va aux membres de ma famille pour leurs encouragements durant mes longues études.

Mon père, Ma mère pour avoir suscité en moi l'envie de faire de la recherche et su me donner tant d'amour et de confiance en moi.

Mon mari pour avoir relu mon manuscrit et pour m'avoir donné tant de courage.

Ma très chère sœur pour m'avoir toujours dit « tu finiras bien par l'avoir ce mémoire ! »

J'adresse également mes sincères remerciements à tous mes amis du laboratoire.

TABLE DES MATIÈRES

TABLE DES MATIÈRES.....	iii
LISTE DES FIGURES.....	vi
LISTE DES TABLEAUX.....	vi
RÉSUMÉ.....	viii
INTRODUCTION.....	1
CHAPITRE I	
PROBLÉMATIQUE.....	4
1.1 Contexte.....	4
1.2 Méthodologie.....	5
1.3 Objectifs.....	6
1.4 Organisation du document.....	7
CHAPITRE II	
GESTION DE CACHE.....	8
2.1 Bases de données mobiles.....	8
2.2 Notion de cache.....	9
2.3 Mécanisme de cache.....	10
2.4 Hit ratio de cache.....	11
2.5 Gestion de cache.....	12
2.6 Granularité de cache.....	12
2.6.1 Cache en pages.....	13
2.6.2 Cache en tuples.....	14
2.6.3 Cache sémantique.....	14

2.7	Cohérence de cache.....	17
2.8	Remplacement dans le cache.....	17
2.9	Les réseaux de téléphonie mobile.....	19
2.9.1	Technologie GSM.....	19
2.9.2	Réseaux CDMA2000.....	21

CHAPITRE III

LE

	MÉCANISME DE CACHE.....	25
3.1	CONTEXTE ET ARCHITECTURE.....	25
3.2	PROCESSUS DE CACHE.....	26
3.2.1	Recherche locale.....	26
3.2.2	Recherche auprès du voisin.....	27
3.2.3	Recherche auprès du SBD.....	28
3.3	CONCEPT DE CACHE SÉMANTIQUE.....	29
3.3.1	Définitions formelles.....	30
3.4	ORGANISATION PHYSIQUE D'UN CACHE SÉMANTIQUE.....	31
3.5	TRAITEMENT DE REQUÊTES.....	32
3.6	PROCESSUS DE CACHE.....	33
3.7	PARTITIONNEMENT DE REQUÊTES.....	33
3.8	POLITIQUE DE REMPLACEMENT.....	45

CHAPITRE IV

PROTOTYPE ET

	ÉVALUATION.....	52
4.1	ARCHITECTURE DU PROTOTYPE.....	52
4.1.1	Unité du client.....	53
4.1.2	Cache du client.....	55
4.1.3	Unité du voisin.....	56
4.1.4	Cache du voisin.....	56
4.1.5	VLR.....	56

4.1.6	Serveur.....	56
4.2	CONCEPTION.....	58
4.3	EXPÉRIENCES ET RÉSULTATS.....	59
4.4	RÉSULTATS DU PROTOTYPE.....	70
	CONCLUSION.....	74
	RÉFÉRENCES.....	76

LISTE DES FIGURES

Figure	Page
2.1 Environnement d'une base de données mobile	9
2.2 Mécanisme de cache	10
2.3 Processus du hit ratio et de miss ratio	11
2.4 Principales phases dans la gestion de cache	12
2.5 Architecture d'un système GSM	20
2.6 Architecture du réseau CDMA2000	22
2.7 Réseau noyau PCN	23
3.1 Architecture générale du modèle utilisé	27
3.2 Recherche locale	27
3.3 Recherche auprès du voisin	28
3.4 Recherche auprès du serveur de base de données	28
3.5 Requête trouvable dans le cache	34
3.6 Requête introuvable dans le cache	35
3.7 Partitionnement vertical de requête	36
3.8 Partitionnement horizontal de requête	37
3.9 partitionnement hybride de requête	38
4.1 Architecture de prototype	53
4.2 Modèle relationnel de base de données utilisée [14]	58
4.3 Temps de réponse des requêtes issues de cache en pages et de cache sémantique selon LRU et MRU	64
4.4 Nombre de requêtes récupérées localement, chez le voisin et le serveur	66
4.5 Temps de réponses des requêtes trouvées localement, chez le voisin et le serveur	67
4.6 Hit ratio en fonction de la taille de cache	68
4.7 Miss ratio de cache sémantique LRU/MRU en fonction de la taille de cache	69
4.8 Résultat de requête trouvée localement	71
4.9 Résultat de requête trouvée auprès du voisin	72
4.10 Résultat de requête trouvée chez le serveur	73

LISTE DES TABLEAUX

Tableau	Page
3.1 Exemple d'un segment sémantique	32
3.2 Mémoire cache de quatre segments.	40
4.1 Paramètres d'évaluation	61
4.2 Temps de transmission pour des requêtes issues du serveur dans un cache en pages	62
4.3 Calcul du temps de transmission pour des requêtes issues du voisin dans un cache sémantique	63

RÉSUMÉ

Ces dernières années, la technologie des réseaux sans fils a connu un fort essor, ajouté au développement des bases de données mobiles qui a contribué à l'émergence d'un grand nombre d'applications. Une des conséquences directes de ce progrès est la nécessité d'utiliser le cache afin de réduire les coûts de transmission des données à travers le réseau. Le sémantique est un type de cache qui a prouvé de la performance par rapport à celui d'un cache en pages ou d'un cache en tuples. En effet, l'utilisation d'un cache sémantique au niveau des dispositifs mobiles est une approche assez cruciale pour augmenter la performance.

Ainsi, notre principal objectif dans cette mémoire est de proposer une solution de traitement de requêtes au niveau de cache d'un mobile basé sur une sémantique, en plus, de montrer l'utilité d'utiliser la mémoire cache du voisin pour récupérer les données, au lieu de les ramener près du serveur. D'abord, nous présentons l'organisation sémantique des segments du cache. Nous proposons ensuite les différents cas de partitionnement des requêtes ainsi que le traitement adéquat. D'un autre côté, nous concevons une architecture basée sur un réseau cellulaire 3G, tel que CDMA2000 et permettant de traiter les requêtes entre le client et son voisin ainsi que le serveur de base de données. Enfin, nous validons nos propositions par une mise en œuvre d'un prototype permettant d'exécuter et d'évaluer la performance des méthodes proposées.

Mots clés : bases de données mobiles, réseaux CDMA 200, cache sémantique, traitement de requêtes.

INTRODUCTION

De nos jours, les bases de données sont de plus en plus volumineuses et elles sont utilisées partout dans le monde sous différents intérêts. Elles constituent le support d'une variété de nouvelles recherches et applications. De même, les réseaux sans fils connaissent une évolution importante, ce qui a permis à l'utilisateur d'accéder à des données n'importe où, et n'importe quand via le dispositif mobile.

Toutefois, la transmission des données à travers les réseaux sans fils consomme beaucoup d'énergies et de ressources. Suite à ces conséquences, l'idée de cacher les données au niveau du mobile joue un rôle important pour réduire les coûts de transmission et augmenter la performance du système.

La technique du cache permet de stocker la donnée la plus fréquemment accessible dans le site du client au lieu de la rechercher à chaque fois auprès du serveur. Cette technique est utilisée de plus en plus dans les réseaux sans fils qui sont limités dans la bande passante. Elle joue un rôle important à faire augmenter les performances du système en terme de temps de réponse des requêtes et à réduire le trafic des données transférées entre le client et le serveur. Le pourcentage des requêtes exécutées qui trouvent leurs réponses (totalement ou partiellement) dans le cache du client se définit par le taux de réussite (Hit ratio). Donc, le but principal du système du cache est d'utiliser différents outils pour augmenter le taux de réussite.

De manière générale, les mécanismes de cache sont de trois types : cache en pages, cache en tuples et cache sémantique. Le mécanisme de cache sémantique est considéré comme l'un des mécanismes les plus avantageux pour le système. L'idée est simple, il se base sur une description sémantique des requêtes qui génèrent les données. C'est à dire, en premier temps, les requêtes exécutées seront sauvegardées d'une manière sémantique avec leur résultat jusqu'au l'épuisement de la mémoire cache. Ensuite, chaque requête exécutée, sera recherchée et traitée au niveau du cache. À la fin de la recherche, il y'aura trois principaux résultats, soit un résultat complet, un résultat partiel ou pas de résultat. Dans le contexte sémantique, il existe deux principales tâches : le traitement de la requête au niveau du cache et la gestion du cache. Cela signifie qu'un mécanisme de cache efficace dépend sur ces deux derniers facteurs.

Le traitement de requête dans le cache représente le processus qui recherche la requête et récupère son résultat. En principe, la requête sera divisée en deux sections, une représente la partie de requête trouvée dans le cache et l'autre représente la partie manquante qui sera recherchée auprès du serveur.

La gestion du cache représente le processus qui s'occupe à organiser le cache en groupes et à le mettre à jour tout en employant des méthodes de cohérence tel que les registres d'invalidation qui assure que les données au niveau du mémoire cache correspond à celles du serveur. Aussi, d'effectuer des remplacements des requêtes lorsque la taille du cache atteint son maximum selon des méthodes de remplacements efficaces.

L'objectif de ce mémoire est de proposer un mécanisme d'un cache sémantique permettant de traiter la requête dans le cache et de récupérer son résultat soit localement, dans le site du client, soit auprès du serveur. Ce mécanisme est employé avec deux méthodes de remplacement, LRU et MRU pour maintenir la cohérence du

cache. Aussi, une architecture est proposée pour implémenter l'environnement mobile où sera exécuté notre algorithme du cache. Dans le cadre de notre étude, cinq types de requêtes sont traitées et partitionnées dans le cache.

Dans un premier temps, nous présentons la problématique de notre mémoire. La partie l'état de l'art définit un aperçu des principaux travaux liés à notre problème.

La partie suivante a pour but de présenter notre travail c'est à dire l'architecture de notre système mobile utilisé. Nous détaillons l'architecture proposée ainsi que les différents composants. Nous détaillons aussi le mécanisme du cache sémantique proposé avec les principaux algorithmes. Un prototype est implémenté pour évaluer la performance du système.

Enfin, une conclusion qui met en évidence le travail réalisé au cours de ce mémoire, et une bibliographie qui récapitule l'ensemble des références bibliographiques utilisées tout au long de l'étude.

CHAPITRE I

PROBLÉMATIQUE

1.1 Contexte

Aujourd'hui, les téléphones mobiles connaissent un grand avancement technologique. Ils sont devenus de plus en plus sophistiqués, performants et accessibles par le grand public. De même, les technologies des réseaux sans fils accroissent d'une manière importante en terme de débit et de couverture.

Grâce à ces technologies, l'utilisateur du mobile peut utiliser les données et les services n'importe où et n'importe quand, ce qui a contribué à accroître le nombre d'opérations vers les serveurs des bases de données dédiés.

De plus, les données sont transmises des serveurs aux mobiles à travers des canaux sans fil qui subissent des déconnexions.

Dans ce contexte, l'utilisation de mémoire de stockage temporaire appelée *Cache* devient importante pour réduire les coûts de transmission et augmenter la performance. En effet, mettre en mémoire cache du mobile les requêtes et les données les plus utilisées évite d'interroger le serveur à chaque demande ce qui va alléger la charge du serveur et éviter les pertes des données causées par les déconnexions.

Les recherches liées à la mise en cache ont proposé plusieurs méthodes pour cacher les requêtes initialisées par l'utilisateur. Ces méthodes sont classées en trois grandes catégories. La première concerne le cache des requêtes en pages où la page sera utilisée comme unité de transfert. La deuxième représente le cache en tuples où les tuples seront considérés comme étant l'unité de transfert entre le serveur et le mobile, et la troisième est connue sous le nom de *cache sémantique*.

Cette dernière approche se base principalement sur une description sémantique des données et des requêtes au niveau du cache. Donc, chaque requête recherchée par l'utilisateur sera comparée de façon sémantique avec les données qui existent déjà dans le cache du mobile.

Le cache sémantique est un mécanisme qui a montré une meilleure performance par rapport aux autres approches.

Dans ce mémoire, on va adopter cette technique tout en lui ajoutant une présentation spécifique pour l'organisation du cache ainsi que les requêtes et ce, afin d'améliorer sa performance.

Par ailleurs, lorsque le cache est plein, et il n'y aura pas d'espace pour enregistrer les nouvelles requêtes, ce qui nécessite d'éliminer quelque données du cache et les remplacer par des nouvelles plus pertinentes.

Mais comment choisir ces données à remplacer? Et sur quels critères se baser?

1.2 Méthodologie

Nous allons aborder ces problèmes en créant une solution basée sur un cache sémantique qui permet de partitionner des requêtes initialisées par le client, et de chercher leurs résultats, soit localement, soit auprès du voisin mobile, ou auprès du serveur de base de données. Notre proposition doit garantir une gestion efficace pour le cache avec une cohérence sur les données qu'il contient.

Les principales extensions aux techniques existantes que nous considérons sont :

- a. La présentation des requêtes, et l'organisation adéquate de cache du mobile dans lequel nous allons stocker nos requêtes.
- b. L'étude des différents cas de comparaison entre la nouvelle requête et celles qui existent dans le cache en se basant sur une description sémantique.
- c. La récupération du résultat d'une requête dans le cache du mobile ou dans un mobile voisin dans le réseau ou auprès du serveur au bout du compte.
- d. L'étude des méthodes de remplacement dans le cas où le cache est plein. Ces méthodes considèrent le nombre d'utilisation des données comme critère de sélection. Elles sont connues sous les noms: le moins récemment utilisé LRU (*Least Recently Used*) et le plus récemment utilisé MRU (*Most Recently Used*).
- e. La mise en œuvre d'un prototype pour exploiter nos solutions sur une base de données de test [14], et des requêtes générées de façon aléatoire.
- f. La mise au point d'un algorithme de gestion de cache, son application et son évaluation sur la base de trois critères de performance qui sont le temps de réponse, le hit ratio et le taux d'erreurs.

1.3 Objectifs

Notre principal objectif est de proposer une solution traitant les problèmes de mise en cache, de partitionnement et de recherche des résultats pour des requêtes exécutées par un utilisateur mobile.

Le mécanisme de cache utilisé se base principalement sur une description sémantique.

Nous étudions également les méthodes de remplacement pour conserver la cohérence de cache.

De façon générale, les objectifs visés dans ce mémoire se résument comme suit :

1. Étudier globalement les méthodes de cache existantes.
2. Décrire une nouvelle structure pour la segmentation des données au niveau du cache.
3. Présenter la nouvelle approche de cache sémantique pour les requêtes.
4. Utiliser deux méthodes de remplacement telles que LRU et MRU pour le maintien de la cohérence de cache.
5. Mettre en oeuvre un prototype exploitant nos algorithmes.
6. Évaluer la performance de notre solution.

1.4 Organisation du document

Après avoir introduit le contexte de notre sujet, présenté aussi notre problématique et nos objectifs, nous organisons le reste de ce mémoire de la façon suivante :

Dans le chapitre II, nous aborderons de façon générale des notions sur le cache sémantique. Nous présentons ensuite une synthèse des principaux travaux liés au traitement des requêtes au niveau du cache.

Dans le chapitre III, nous nous concentrons sur le problème de cache des requêtes. Nous présentons notre architecture pour la solution proposée ainsi que les méthodes de partitionnement des requêtes au niveau du cache.

Dans le chapitre IV, nous décrivons notre prototype réalisé pour évaluer nos algorithmes de cache et présentons nos expériences menées sur des requêtes générées aléatoirement avec les résultats des évaluations de la performance.

Dans la conclusion, nous présentons une synthèse des aspects abordés dans ce mémoire. Ensuite, nous dévoilons le résultat des solutions proposées. À la fin, nous parlons des principales limitations ainsi que les travaux futurs.

CHAPITRE II

GESTION DE CACHE ET MOBILITÉ

Notre principal objectif dans ce chapitre est de présenter une synthèse des travaux relatifs aux caches des requêtes dans un environnement mobile.

D'abord, nous introduisons les bases de données mobiles, ensuite nous abordons le mécanisme de cache en détail ainsi que les travaux reliés. Cette étape permet d'éclaircir la notion de cache, d'identifier son objectif et d'étudier l'apport des approches déjà proposées pour savoir comment traiter le problème et comment le résoudre. Nous décrirons aussi les politiques de remplacement proposées dans la littérature et leurs rôles dans la gestion de cache.

Dans la deuxième partie, nous décrivons les réseaux sans fils de façon générale, précisant que nous utilisons le réseau CDMA2000 dans notre étude.

2.1 Bases de données mobiles

Récemment, les bases de données distribuées sont devenues un sujet de recherche important. Les nouvelles technologies des réseaux sans fil et les équipements mobiles permettent d'avoir l'accès à des ressources partout et à tout moment, rendant ce type de bases de données plus accessibles.

L'intérêt principal des bases de données distribuées est d'exécuter de façon continue les requêtes et les transactions avec une indépendance vis-à-vis du réseau ou

du matériel utilisé. Cette répartition joue un rôle primordial afin d'augmenter la performance de ces systèmes en rapprochant la donnée de l'utilisateur et en améliorant leur fiabilité et leur disponibilité.

Généralement, l'architecture d'une base de données répartie se constitue de plusieurs sites reliés par un réseau et contenant des parties différentes (appelées *fragments*) de données, sachant que l'ensemble de ces parties forment la totalité des données. La figure suivante illustre un environnement de base de données mobile :

Figure 2.1. Environnement d'une base de données mobiles.

2.2 Notion de cache

La notion de cache désigne une mémoire intermédiaire de taille limitée pour enregistrer de façon temporaire les données. Elle est plus petite que la mémoire principale mais plus rapide que celle-ci, ce qui réduit le nombre d'accès vers le processeur central et améliore la performance du système. La figure 2.2 montre la place de la mémoire cache au sein d'équipements matériels.

Généralement, l'exécution des données au niveau du cache suit deux principes:

1. Le principe de *localité temporelle* qui indique que les éléments qui viennent d'être accédés récemment n'aura plus de probabilité d'être référencés dans le futur proche.
2. Le principe de *localité spatiale* qui indique que les éléments qui seront sélectionnés prochainement sont ceux qui se trouvent à proximité (en termes d'adresses) ou dans la même région que les éléments qui viennent d'être utilisés.

Figure 2.2. Mécanisme de cache.

2.3 Mécanisme de cache

La mise en cache décrit la façon dont les données sont mises en cache.

Le cache des données est une nouvelle approche de plus en plus utilisée dans les environnements mobiles. En effet, il permet de mettre en cache les items des bases de données qui sont fréquemment utilisés par l'utilisateur mobile afin de réduire le nombre de connexions vers le serveur ou le site des bases de données [1]. Il permet également à l'utilisateur d'obtenir une réponse plus rapide pendant les durées de déconnexion.

Généralement, le mécanisme de cache des données au niveau des dispositifs mobiles se divise en deux parties: la partie physique et la partie logique. La première

partie décrit la façon d'organiser les données enregistrées. La deuxième partie étudie la représentation des requêtes ainsi que leurs résultats.

2.4 Hit ratio de cache

Taux de réussite ou *hit ratio* est un critère pour mesurer l'efficacité de cache. Il correspond aux nombres de fois où la donnée recherchée par l'utilisateur a trouvé son résultat dans le cache tel que montré dans la figure 2.3. À ce moment, chaque requête sera effectuée et traitée premièrement par la mémoire cache. Si son résultat se trouve dans le cache, il suffit de le récupérer, sinon, la même requête sera dirigée vers la mémoire entraînant ce qu'on appelle *miss ratio*.

Le hit ratio se définit par la formule suivante:

$$\text{Hit ratio} * 100$$

Pour un nombre de lectures en mémoire principale égal à 0, le taux de réussite atteint sa valeur maximale qui est de 1.

Figure 2.3. Processus du hit ratio et de miss ratio.

2.5 Gestion de cache

La gestion de cache représente le processus qui permet de diriger et d'organiser le cache physiquement et logiquement. La figure 2.4 montre que ce processus se compose généralement de trois phases : le niveau de granularité (niveau pages, niveau tuples, sémantique), les politiques de cohérence et les politiques de remplacements.

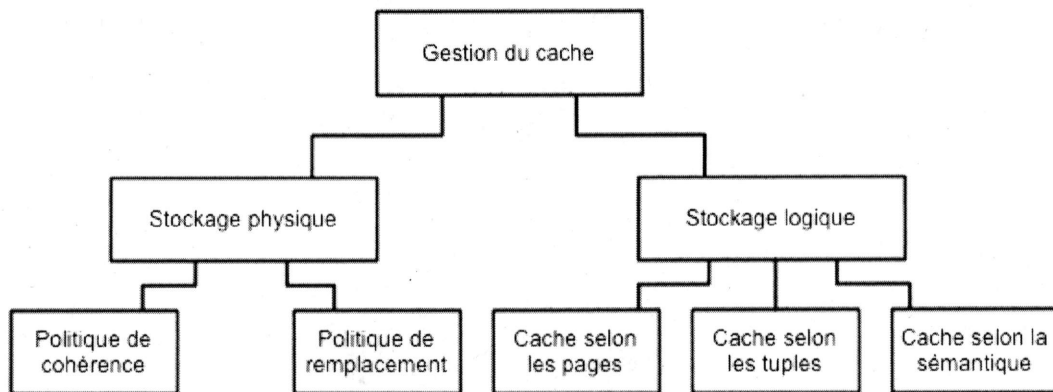


Figure 2.4. Principales phases dans la gestion de cache.

2.6 Granularité de cache

Le contenu de cache se présente par un certain niveau de granularité qui indique l'unité la plus fine pour représenter la donnée. Il diffère d'une application à une autre [2]. Les auteurs dans [4] ont opté pour trois niveaux de granularités : le cache selon les attributs, le cache selon les objets et le cache hybride sur des bases de données orientées objet (OODB). Le cache selon les attributs permet de stocker localement les attributs les plus accédés. Le cache selon les objets a pour principe de stocker les objets eux-mêmes. Le cache hybride est destiné à stocker les attributs des objets les plus fréquemment utilisés. Les expériences révèlent que le cache selon les attributs offre un minimum de temps de réponse, et un taux de stockage plus élevé.

Le cache hybride exploite les bénéfices des deux autres méthodes donnant un meilleur résultat. En plus, il offre aussi un meilleur résultat comparativement au cache en pages [4].

De façon générale, les applications qui se basent sur le mécanisme de cache des données reposent sur trois niveaux de granularités qui sont : le cache selon le niveau de pages, le cache selon le niveau de tuples et le cache selon la sémantique [2].

Le cache selon le niveau de pages représente ses données sous forme de groupe de tuples de manière statique dans des pages. Ce qui correspond aussi à la granularité de niveau en tuples, mais avec des tuples présentées de façon individuelle.

Le cache selon la sémantique a une représentation dynamique sous forme de groupes de tuples qui sont capables de se changer selon la requête et son résultat.

Dans notre travail, on considère que le cache est formé de plusieurs segments sémantiques. D'une façon logique, ces segments se composent de différentes requêtes enregistrées avec leurs résultats. D'une manière physique, les résultats sont sous forme d'une ou plusieurs pages reliées entre elles. La grande différence qu'on peut distinguer entre une mise en cache sémantique et celle qui se base sur un niveau de pages, est que les pages dans la première approche comportent seulement les réponses des requêtes alors que les pages dans la deuxième approche comportent les requêtes avec leurs résultats.

Dans la suite, on va définir les trois niveaux de cache de façon plus précise.

2.6.1 Cache en pages

Le cache selon une page est une ancienne approche largement utilisée dans les systèmes de bases de données avec le modèle client/serveur.

Dans une application qui exploite ce type de cache, le client émet une demande sous forme de requête SQL au serveur des bases de données qui va faire une recherche et envoyer son résultat au demandeur client. Une fois reçu par le client, ce résultat est sous forme de pages composées de requêtes et résultats qui vont être enregistrés au niveau du cache du client. Un pointeur de référence est créé pour accélérer la recherche d'une page.

2.6.2 Cache en tuples

Le cache en tuples est basé principalement sur une organisation en tuples. Cette approche a le même fonctionnement que la mise en pages sauf que le stockage ou l'échange des données sera sous forme de tuples individuelles au lieu de pages.

2.6.3 Cache sémantique

Afin de palier aux inconvénients des approches traditionnelles basées sur le niveau de pages ou de tuples, le cache sémantique garde une description sémantique de la requête.

Les auteurs dans [2] et [5] ont introduit le concept de *cache sémantique* en donnant une description formelle et une stratégie de cohérence. Cette technique a également été utilisée dans des bases de données mobiles pour améliorer les performances et réduire les coûts de transmission.

L'idée principale de cache sémantique repose sur le fait que le client maintient à la fois une description sémantique de la requête et son résultat dans son cache.

Par exemple, si un client veut exécuter une nouvelle requête, premièrement, il va la chercher dans son cache selon ses descriptions. Le client peut trouver toute la réponse au niveau du cache, donc il n'a pas besoin d'envoyer cette requête. Il peut également ne trouver qu'une partie seulement de sa requête. Dans ce cas, la

requête doit être divisée en deux parties: l'une comprend la moitié de requête avec son résultat dans le cache du client, et l'autre présente la partie manquante, qui sera envoyée au serveur pour récupérer son résultat.

Un cache sémantique est une collection de régions ou de segments qui comportent les descriptions des requêtes avec leurs résultats [5].

Généralement, un système de cache sémantique traite deux grandes parties qui sont la gestion de cache et le traitement de la requête [3]. La gestion de cache concerne notamment les stratégies de gestion de données, le remplacement, la cohérence, le regroupement et l'indexation des résultats.

Le traitement de requête se sert des descriptions sémantiques pour calculer et évaluer les données disponibles ou non dans le cache. En plus, il traite les requêtes au niveau de cache et celles qui sont envoyées ou récupérées du site du gestionnaire de base de données. Le processus de division de requêtes est une phase importante dans la phase de traitement des requêtes.

Les auteurs dans [3] exploitent un nouvel algorithme de modélisation basé sur des graphes.

Plusieurs applications exploitant la mise en cache sémantique, ont été proposées dans la littérature. L'objectif principal de ces travaux est de réduire le temps de réponse ainsi que le trafic vers le serveur de la base de données.

Les auteurs dans [7] proposent un cache sémantique selon la localisation de résultats des requêtes, où chaque résultat mis en cache est décrit par l'emplacement associé à la requête.

Les systèmes d'intégration [9] qui fournissent un accès transparent aux différentes sources de données tout en les présentant d'une manière unifiée, sont

munis de ce mécanisme qui tend à accélérer le traitement des requêtes dans de différentes sources de données.

Les systèmes OLAP [11], ont aussi employés le cache sémantique dans le but de réduire le temps de réponse des requêtes multidimensionnelles.

Une différence entre le cache sémantique et les méthodes traditionnelles est que la structure de cache est sous la forme d'une description de requêtes au lieu de pages ou tuples. Aussi, les approches traditionnelles utilisent les pointeurs de références pour identifier la donnée, alors que le cache sémantique utilise la description de la requête pour identifier la donnée.

Le tableau suivant montre les principaux points de comparaison entre les trois mécanismes de cache:

Tableau 2.1.

Comparaison entre les trois mécanismes de cache.

	Cache en pages	Cache en tuples	Cache sémantique
Infrastructure	Client/serveur	Client/serveur	Pas d'infrastructure
Stockage	Pages statiques	Tuples individuels	Pages dynamiques
Recherche	Pointeurs de références	Pointeurs de références	Description sémantique
Résultat	Complet	Complet	Partiel ou complet
Échec	Envoi de toute la requête au serveur	Envoi de toute la requête au serveur	Envoi complet ou partiel de la requête au serveur

2.7 Cohérence de cache

Un cache est un ensemble de données qui peuvent être modifiées ou mises à jour au niveau de gestionnaire de la base de données. Un changement au niveau de ce gestionnaire doit se refléter sur un changement au niveau du cache du client. Cette variation continue dans les données nécessite des mécanismes de préservation de cohérence au niveau du cache.

Les registres d'invalidation sont des mécanismes de cohérence, utilisés de plus en plus dans les environnements mobiles. Le principe général de cette approche est d'utiliser des canaux sans fils pour diffuser les données qui viennent d'être modifiées [8]. L'inconvénient majeur de cette approche réside dans les canaux utilisés qui subissent des moments de déconnexions, ce qui empêche de transmettre les données modifiées.

La technique de signature de cache est un autre mécanisme pour assurer la consistance des données. Précisons qu'une signature est une abstraction de données qui sert à comparer les informations diffusées auprès du serveur et celles qui existent au niveau de cache du mobile et qu'une signature se sert essentiellement des avis d'invalidation qui sont capables de déduire si cette dernière correspond à celle qui est envoyée par le serveur [9].

Dans ce mémoire, on suppose que nos données sont stables et qu'elles ne seront pas changées et par conséquent, le processus de cohérence sera ignoré.

2.8 Remplacement dans le cache

À chaque fois que la mémoire cache atteint sa taille limite, un algorithme de remplacement est utilisé pour supprimer une donnée existante dans le cache et la remplacer par une nouvelle donnée [6]

Le processus de remplacement au niveau du cache consiste à choisir la donnée victime qui sera remplacée par une autre en se basant sur des critères adéquats.

Généralement, le principe des processus de remplacement repose essentiellement sur le principe de localité, où la donnée à remplacer dans le cache sera choisie selon sa fréquence d'utilisation dans le passé. C'est le principe de la localisation temporelle, ou selon son emplacement physique au niveau de la mémoire cache, c'est le principe de la localisation spatiale.

Le choix des critères de la politique de remplacement est important. Si on fait le mauvais choix, la performance du système diminue en terme de taux de réussite, de temps de réponse et de bande passante [2].

L'algorithme dit le *Moins Récemment Utilisé* ou (*Least Recently Used* ou *LRU*) est un mécanisme de remplacement qui a prouvé son efficacité. Il consiste à garder les données récemment utilisées dans le cache et de supprimer les anciennes qui ne seront pas utiles.

Le remplacement suivant une file d'attente FIFO (*First In First Out*) est un autre algorithme de remplacement. Son principe est de supprimer la donnée selon l'ordre d'arrivée à la mémoire cache. Il est simple à concevoir. En revanche, il possède un pourcentage important de défauts comparativement à celui de MRU.

L'algorithme dit du *Plus récemment utilisé* ou (*Most Recently Used* ou *MRU*) est un autre algorithme de remplacement qui ressemble au LRU sauf qu'il s'intéresse à garder la donnée la plus utilisée. Le MRU a prouvé aussi un succès dans certaines

applications.

Une autre approche connue sous le nom du *Moins Fréquemment Utilisé* (*Least Frequently Used* ou *LFU*) considère que la donnée avec un nombre de fréquence d'accès minimum aura moins de chance à être utilisée dans le futur.

Les politiques de remplacement reposent sur le principe de probabilité d'accès. Cependant, d'autres catégories de remplacement s'articulent sur la distance de données. Elles sont conçues spécialement pour des requêtes qui dépendent de la localisation [7].

Les algorithmes que nous avons présentés ont connu un succès relatif qui dépend des critères de sélection choisis.

2.9 Les réseaux de téléphonie mobile

Plusieurs générations de réseaux de téléphonies mobiles ont été élaborées. Depuis la première génération on est passé à la deuxième nommée 2G telle que le GSM qui offre un service de voix et des services de données de base comme le SMS.

On est ensuite passé à la génération 2.5G qui a amélioré les services de données. Ces réseaux offraient ses services en utilisant des passerelles spécifiques pour la connexion entre la partie du réseau pour le transport de la voix à celle du transport des données.

Depuis quelques années, d'autres réseaux ont été implantés avec des services de données nettement améliorés en supportant des fonctions des réseaux IP.

2.9.1 Technologie GSM

Le premier standard international de téléphonie cellulaire 2G est le GSM (Global System for Mobile communication). C'est un réseau qui peut interfacer avec le réseau téléphonique commuté classique (*RTC* ou *PSTN* pour *Public Switched*

Telephony Network), tout en permettant aux abonnés mobiles de communiquer avec d'autres abonnés mobiles ou fixes.

L'espace couvert par ce réseau est divisé en cellules, chacune servant les utilisateurs qui s'y trouvent. Chaque cellule correspond à une zone géographique gérée par une *Station de base*. Cette station utilise des fréquences radio pouvant être partagées avec des cellules éloignées. La principale méthode d'accès est *F-TDMA* (*Frequency Time Division Multiple Access*) qui utilise la technique de division de fréquences *FDMA* (*Frequency Division Multiple Access*) pour partager le spectre en plusieurs canaux, chaque canal pouvant être partagé à son tour entre plusieurs utilisateurs avec la technique de division du temps *TDMA* (*Time Division Multiple Access*).

La figure 2.5 illustre les principaux éléments de l'architecture GSM.

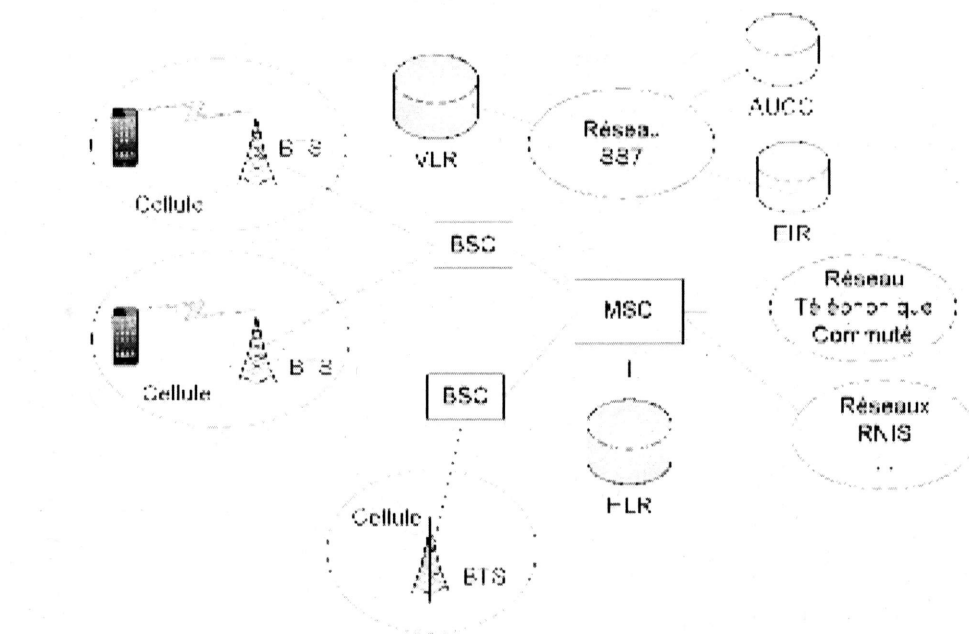


Figure 2.5. Architecture d'un système GSM.

Une BTS (*Base Station Transmission System*) est un ensemble d'émetteurs-récepteurs qui peut gérer plusieurs communications en parallèle.

Une BSC (*Base Station Controller*) agit comme concentrateur dans le traitement des fonctions de communication provenant de plusieurs BTS. De plus, elle se charge de gérer les transferts intercellulaires des utilisateurs appelés handover.

Le GSM offre également des modules de commutation et diverses bases de données, dont :

1. Le *Centre de commutation mobile* ou MSC (pour *Mobile Switching Center*) : il joue le rôle d'un commutateur entre les abonnés du réseau mobile et ceux du *Réseau téléphonique commuté* ou RTC (ou bien *PSTN* pour *Public Switched Telephone Network*). Il s'occupe des fonctions de la mobilité, le handover et la gestion des abonnés visiteurs grâce à son lien avec les bases de données *HLR* (*Home Location Registrar*) et *VLR* (*Visitor Location Register*). Ces bases sont accessibles à travers un réseau dédié appelé *SS7* (*System Signaling 7*).
2. Enregistrement de localisation nominale (*HLR*) : une base de données unique qui maintient des informations concernant les abonnés. Ces informations peuvent être statiques comme le numéro de l'abonné, le type d'abonnement, la clé d'authentification et d'autres dynamiques, telle que la position actuelle du mobile dans le réseau.
3. Enregistreur de localisation des visiteurs (*VLR*) : une base de données qui ne contient que les données dynamiques relatives aux abonnés. Elle est associée à un commutateur et a pour mission de localiser la cellule, dans laquelle existe l'abonné.

2.9.2 Réseaux CDMA2000

Ce standard déployé en Amérique du Nord, permet des services à haut débit pour l'échange de données multimédia et l'accès à des services dans les réseaux IP. La figure suivante décrit l'architecture globale du réseau CDMA2000.

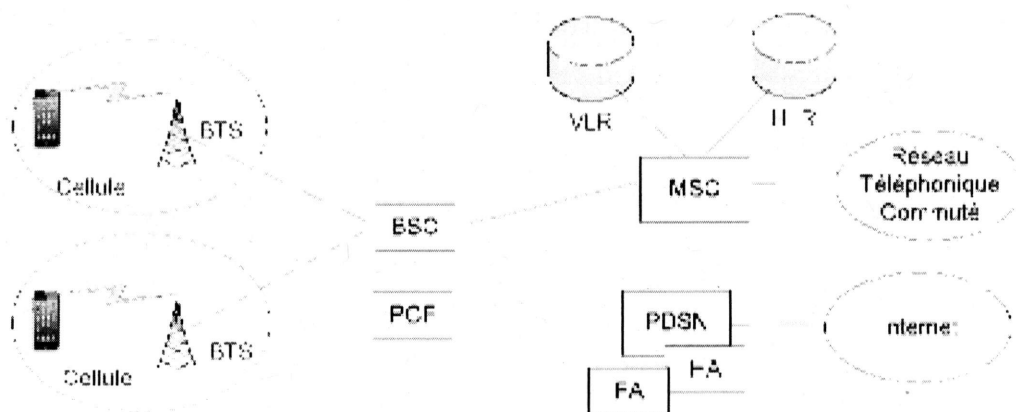


Figure 2.6. Architecture du réseau CDMA2000.

Une station de base (BS) est connectée à un contrôleur (BSC) et un émetteur (BTS) plus un contrôleur de paquets appelé *Paquet Control Function* (PCF). Un PCF est une entité logique qui fait router les paquets IP du PDSN vers la station mobile durant l'ouverture des sessions et contrôle les paquets de données.

Un PDSN est une passerelle entre le réseau radio et le réseau des paquets de données (public ou privé) qui permet d'établir et de maintenir une session PPP (*Point to Point Protocol*) avec la station mobile qui est un lien de données créé pour un abonné voulant échanger des paquets IP avec le PDSN.

Un réseau CDMA2000 apporte deux types d'accès IP : IP simple et IP mobile. Cette distinction dépend de l'adresse IP des nœuds mobiles et du routage des paquets. Chaque paquet routé entre deux nœuds comprend deux éléments à savoir : le serveur PDSN et le serveur d'adresses AAA qui permet d'authentifier et d'autoriser les requêtes des utilisateurs reçues près du PDSN.

Figure 2.7. Réseau noyau PCN.

Avec IP mobile, l'abonné garde son adresse IP même en changeant de réseau. En effet, c'est le rôle du réseau d'attribuer une adresse IP pour les abonnés pas le fournisseur d'accès aux services. Une fois que l'accès est basé sur IP mobile, le réseau accomplit deux tâches supplémentaires, l'agent étranger (Foreign agent ou FA) et l'Agent maison (home agent HA). Ces deux agents permettent de gérer cette mobilité.

Home agent :

HA représente l'agent du réseau d'origine (*Home Network*) de l'abonné, servi par son PDSN maison. Une fois l'abonné se déplace vers un nouveau réseau, l'agent HA doit lui transmettre ses paquets. Mais avant tout, il doit savoir son nouveau PDSN.

Foreign agent :

FA est une entité qui est considérée comme un nouveau PDSN pour un abonné qui visite un nouveau réseau. Dans ce cas, le FA/PDSN reçoit les paquets

destinés à l'abonné qui apparaît comme un visiteur et les transfère ensuite vers lui.
Pour accomplir cette tâche, le HA doit savoir la nouvelle adresse IP du mobile.

CHAPITRE III

LE MÉCANISME DE CACHE

Dans ce chapitre, nous présentons notre méthode de traitement des requêtes en tenant compte de la gestion du cache dans un environnement mobile.

3.1 Contexte et architecture

L'environnement mobile que nous considérons est composé d'unités mobiles qui communiquent avec un serveur de base de données à travers un réseau sans fil de troisième génération, soit le CDMA2000.

Le réseau CDMA2000 est choisi pour les raisons suivantes :

- C'est un standard qui est déployé en Amérique du Nord.
- Il supporte la pile TCP/IP avec IP mobile, permettant ainsi aux mobiles de se déplacer tout en gardant une adresse IP stable.
- Il supporte les services de voix et de données et offre un haut débit.
- Son architecture se prête à l'implantation des fragments de données nécessaires à notre solution.

L'architecture proposée se compose principalement de quatre types de nœuds :

a. Client mobile (CM)

Un nœud mobile possède une mémoire cache pour stocker ses requêtes les plus utilisées. Le client mobile initialise une requête et effectue ensuite la recherche d'abord dans son cache et dans le cas où il a trouvé le résultat, il le récupère de son

propre cache au lieu d'invoquer le serveur de base de données.

b. Nœud voisin (CV)

Un nœud voisin se trouve dans la même cellule que le client mobile et se sert lui aussi de sa mémoire cache pour stocker ses requêtes. Le nœud voisin participe aussi à répondre aux requêtes recherchées par le client mobile.

c. Packet Data Serving Node (PDSN)

Il contient le serveur de base de données local avec des pointeurs vers d'autres SGBD distants. Le serveur de base de données local ou le serveur distant reçoit des requêtes du client mobile, les traite et envoie le résultat correspondant.

d. Visitor Location Registrar (VLR)

C'est une base de données temporaire qui contient les informations des clients mobiles. Dans ce mémoire, le VLR est utilisé pour informer le client mobile de l'adresse de son voisin afin qu'il puisse lui acheminer la requête.

3.2 Processus de cache

De façon générale, le processus de recherche d'une requête initialisée par un client mobile suit trois chemins : recherche locale, recherche auprès du voisin et recherche auprès du serveur.

3.2.1 Recherche locale

Le client mobile initie une requête, vérifie s'il peut trouver son résultat localement dans son cache en utilisant les données qui y sont stockées. S'il y parvient, le processus de recherche de requête se termine ici. La figure 3.2 illustre une recherche locale.

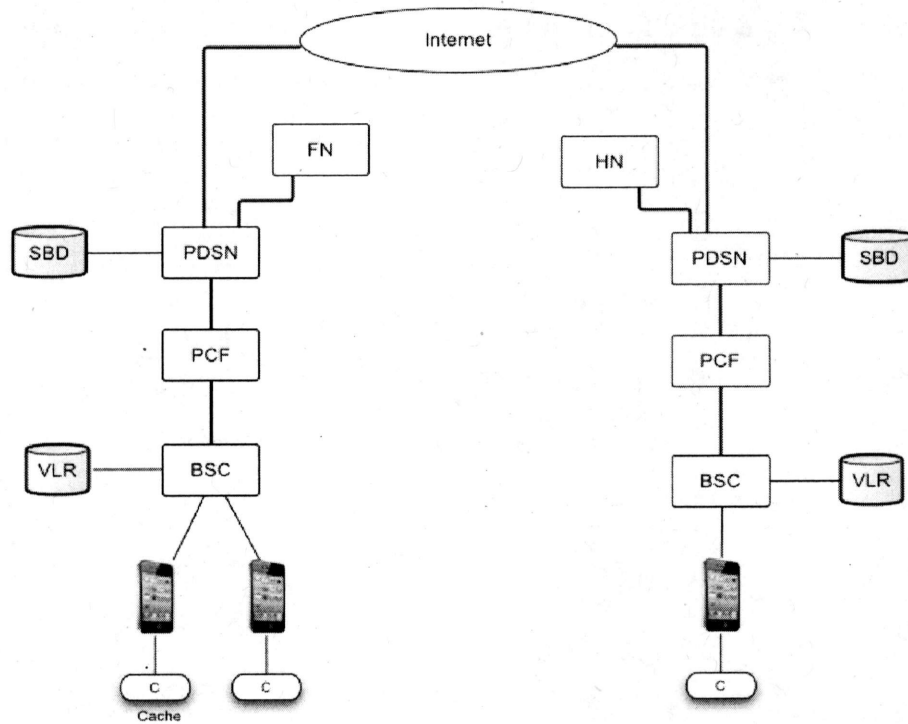


Figure 3.1. Architecture proposée de la base de données répartie.

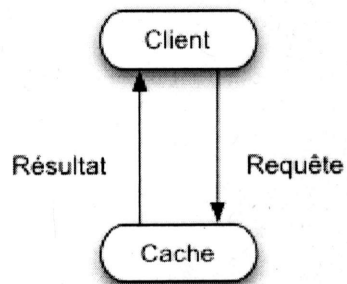


Figure 3.2. Recherche locale.

3.2.2 Recherche auprès du voisin

Dans le cas où la recherche locale a échoué, le client va tenter une recherche auprès de son voisin. Pour cela, il va récupérer l'adresse du voisin le plus proche de

lui à travers le VLR. Le nœud voisin, de son côté, effectue une recherche locale dans son cache et renvoie le résultat. La figure 3.3 illustre une recherche auprès d'un nœud voisin.

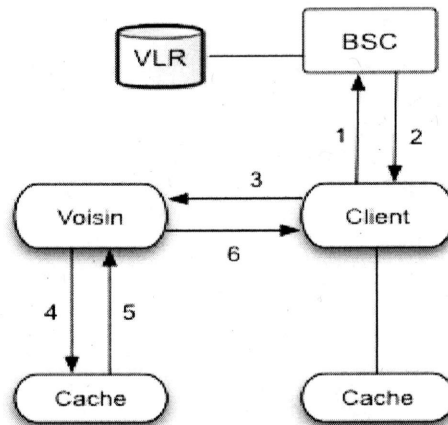


Figure 3.3. Recherche auprès du voisin.

3.2.3 Recherche auprès du SBD

La recherche auprès du serveur de base de données est le dernier processus de recherche effectué par le client mobile car l'objectif principal était de réduire l'accès au serveur pour augmenter la performance du système.

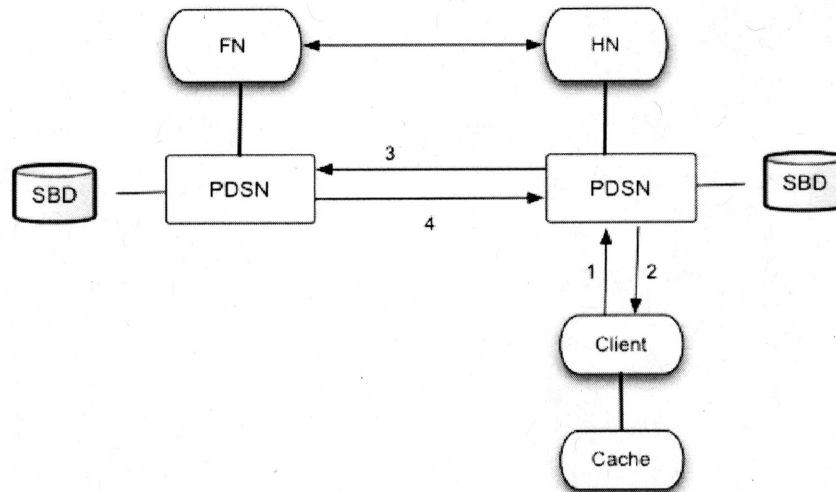


Figure 3.4. Recherche auprès du SBD.

Puisque la recherche locale et la recherche auprès du nœud voisin n'ont pas donné un résultat pour la requête, ce dernier envoie sa recherche au serveur de base de données qui peut se trouver soit au niveau du PDSN local, soit au niveau du PDSN distant tel que montré dans la figure 3.4.

3.3 Concept de cache sémantique

Le cache sémantique est un type de cache qui stocke la requête la plus utilisée par le client mobile avec son résultat dans sa mémoire cache. La façon dont les requêtes sont stockées est reconnue comme une description sémantique qui permet de déterminer si une nouvelle requête peut être répondue par le cache, ou il faut la rechercher auprès du serveur de bases de données. En effet, au moment où le client mobile initialise une requête, une comparaison se fera entre cette dernière requête et les anciennes requêtes stockées dans le cache. L'issue de cette comparaison détermine si le résultat est présent dans le cache ou non.

Le traitement de la requête au niveau du cache sémantique passe par différentes phases dont les plus importantes sont :

- a. La phase de décision : pour savoir si le résultat de la requête se trouve dans le cache.
- b. La phase de décomposition : pour décomposer la requête en deux sous-requêtes : une requête qui existe au niveau du cache et une autre inexistante.
- c. La phase d'extraction : pour extraire le résultat de la requête recherchée auprès du cache.
- d. La phase de jointure : pour joindre les résultats.

Nous allons définir formellement une requête et un segment, ensuite, nous nous focalisons sur le traitement et le partitionnement des requêtes dans le cache.

3.3.1 Définitions formelles

Définition1 (Base de données) :

Une BD est un ensemble Rel de relations $Rel_1, \dots, Rel_i$, tel que $BD = \{Rel_i, 1\}$. Pour chaque relation Rel_i , on définit l'ensemble de ses attributs $Attr(Rel_i)$ et l'ensemble de ses prédicats $Pred(Rel_i)$.

Définition 2 (Segment sémantique):

Considérons la base de données $BD = \{Rel_i\}$ et l'ensemble d'attributs $Attr = Attr(Rel_1) \cup Attr(Rel_2) \dots Attr(Rel_n)$. Un *segment sémantique* est défini comme un tuple sous la forme $Seg \langle Rel, Attr, Pred, Cont, temp, Nbr \rangle$, sachant que $Rel \in BD$, $Attr(seg) = Attr(Rel_i)$, $Pred(seg) = Pred_1 Pred_2 \dots Pred_m$, $Tem(seg)$ est le temps de chargement et $Nbr(seg)$: le nombre d'utilisation du segment.

Définition 3 (Requête) :

Une requête se présente comme un segment sémantique sous la forme d'un triplet $Req = \langle Rel, Attr, Pred \rangle$ avec :

- $Rel (Req)$: l'ensemble des relations de la requête.
- $Attr (Req)$: l'ensemble des attributs qui figurent dans la requête.
- $Pred (Req)$: l'ensemble des prédicats qui construisent la requête.

Définition 04 (Segments disjoints):

Un cache sémantique se définit comme un ensemble de segments sémantiques disjoints.

Soit $Seg_i < Rel_i, Attr_i, Pred_i, Cont_i, Temp_i, Nbr_i >$ et $Seg_j < Rel_j, Attr_j, Pred_j, Cont_j, Temp_j, Nbr_j >$. Ces deux segments sont disjoints si et seulement si :

$$Attr_i (Seg_i) \cap Attr_j (Seg_j) = (Pred_i (Seg_i) \cap Pred_j (Seg_j)) =$$

Définition 5 (Équivalence):

Soit les segments sémantiques $Seg_i < Rel_i, Attr_i, Pred_i, Cont_i, Temp_i, Nbr_i >$ et $Seg_j < Rel_j, Attr_j, Pred_j, Cont_j, Temp_j, Nbr_j >$. Ces segments sont équivalents si et seulement si :

$$Rel (Seg_i) = Rel (Seg_j), Attr_i (Seg_i) = Attr_j (Seg_j) \text{ et } Pred_i (Seg_i) = Pred_j (Seg_j).$$

3.4 Organisation physique d'un cache sémantique

Un cache sémantique peut être organisé de différentes manières. Nous avons opté pour une organisation en deux parties : l'indexation et le contenu.

L'indexation représente la partie spécifique pour chaque segment. Elle comprend les informations nécessaires de caractérisation, soit :

- La relation $Rel (Seg)$.
- L'ensemble d'attributs $Attr (Seg)$.
- L'ensemble des prédicats $Pred (Seg)$.

- Le contenu *Cont (Seg)* est un pointeur qui pointe vers la première page du résultat.
- Le timestamps *Temp (Seg)* indique le dernier temps d'accès au segment.
- Le nombre d'accès *Nbr (Seg)* présente le nombre d'utilisation du segment.
- Le contenu est sous forme d'une ou plusieurs pages liées entre elles.

Notons que le nombre d'utilisations pour chaque segment qui vient d'être enregistré dans le cache est nul.

Exemple :

Prenons comme exemple simple, un extrait d'une base de données de test [14] avec quatre relations :

- *Artist = (ArtistId, Name)*
- *Album = (AlbumId, ArtistId, Title)*
- *Track = (TrackId, AlbumId, GenreId, Name, Composer, UnitPrice, Milliseconds)*
- *Genre = (GenreId, Name)*

Considérons aussi, un cache d'un client qui se compose de trois segments, comme décrits dans le tableau ci-dessous.

Tableau 3.1

Exemple d'un segment sémantique.

Seg	Rel	Attr	Pred	Cont	Temp	Nbr
Seg1	Album	Title, Bytes	Album.AlbumId = Track. AlbumId, GenreId = '23'	Cont 1	T1	Nbr 1
	Track					
Seg2	Artist	Name	AlbumId = '84'	Cont 2	T2	Nbr 2

Seg3	Track Genre	GenreId, Composer	Track.GenreId = Genre. GenreId, Track.Name = 'Solitary',	Cont 3	T3	Nbr 3
------	----------------	----------------------	--	-----------	----	----------

3.5 Traitement des requêtes

Un cache sémantique se compose de segments sémantiques disjoints. En premier, l'utilisateur initialise sa requête et la cherche dans sa mémoire cache. La requête peut être trouvée soit complètement ou partiellement avec une partie manquante dans le cache.

On essaiera de répondre aux questions suivantes :

- Comment savoir si la requête est répondue partiellement ou complètement ?
- Quelle est la partie qui se trouve au niveau du cache ?
- Comment détecter la partie manquante ?

La section suivante présente les techniques et les algorithmes nécessaires pour répondre aux questions précédentes.

Proposition :

Soit $Seg_i = \langle Rel_i, Attr_i, Pred_i, Cont_i, Temp_i, Nbr_i \rangle$ un segment sémantique et $Req_i = \langle Rel_i, Attr_i, Pred_i \rangle$ une requête. Si $Rel(Req) = Rel(Seg)$ $Attr(Req) = Attr(Seg)$ $Pred(Req) = Pred(Seg)$, alors cette requête sera répondue totalement par ce segment.

3.6 Processus de cache

Chaque requête recherchée dans un cache peut avoir une réponse totale, une réponse partielle, ou pas de réponse du tout. Dans le cas où la requête est trouvée totalement, l'abonné récupère son résultat auprès du cache. Dans l'autre cas, où la

requête est trouvée partiellement, elle sera divisée en deux parties. La première partie, celle qui existe dans le cache et qui répond aux conditions demandées est nommée *Requête existante (ExistReq)*. L'autre partie manquante est nommée *Requête manquante (ManqReq)*. Elle sera envoyée au voisin. Ce dernier contient aussi un cache sémantique et suit la même procédure de traitement de requête au niveau du cache. Donc, le voisin de son côté, effectue une recherche de la requête manquante dans son propre cache et envoie ensuite une réponse au client mobile. Si celui-ci reçoit une réponse complète de sa requête recherchée, le processus de recherche sera terminé, sinon le client doit récupérer la réponse auprès de son serveur de base de données.

3.7 Partitionnement de requêtes

Le processus de partitionnement de la requête Req en deux sous requêtes :

- a. *Requête existante (ExistReq)* : définit la première partie de la requête trouvée localement dans le cache du client mobile.
- b. *Requête manquante (ManqReq)* : contient la deuxième partie qui manque de la requête. Elle est récupérée près du voisin si elle y existe, sinon à partir du serveur de base de données.

Après le parcours des segments du cache, le processus de recherche de requêtes se compose de cinq cas différents. La recherche d'une requête parcourt tous les segments du cache.

Pour expliquer les différents cas, on a considéré un segment *Seg* et une requête *Req*.

Cas 1: La requête se trouve dans la cache

Figure 3.5. Requête se trouvant dans le cache.

Le contenu de la requête recherchée *Req* existe dans un segment sémantique *Seg*:

- $Rel (Req) = Rel (Seg)$.
- $Attr (Req) = Attr (Seg)$
- $Pred (Req) = Pred (Seg)$

Dans ce cas, le processus de partitionnement de requête est inutile. Il suffit de récupérer le contenu du segment *Cont (Seg)* pour répondre à la requête.

Cas 2: Requête introuvable dans la cache

Figure 3.6. Requête introuvable dans le cache.

Dans cette situation, le contenu de la requête *Req* ne correspond à aucun segment sémantique du cache, alors :

La requête existante (*ExistReq*) ne définit aucun des segments du cache, donc elle sera exprimée de la façon suivante :

- *Rel (ExistReq) Rel (Seg)*.

- *Attr (ExistReq) Attr (Seg)*.

- *Pred (ExistReq) Pred (Seg)*.

La requête manquante (*ManqReq*) se traduit comme suit :

- *Rel (ManqReq) = Rel (Req)*.

- *Attr (ManqReq) = Attr (Req)*.

- *Pred (ManqReq) = Pred (Req)*.

Par conséquent, cette requête sera envoyée au mobile voisin qui va effectuer une recherche au niveau du cache en essayant de trouver le résultat. Si le mobile voisin échoue à trouver la réponse, alors la même requête sera transmise au serveur afin de récupérer son résultat.

Cas 3: Partitionnement vertical de requête

Figure 3.7. Partitionnement vertical de requête.

Ce cas représente une requête partitionnée verticalement par un segment sémantique. C'est-à-dire un segment parmi les segments de cache contient tous les éléments de la requête sauf au niveau des attributs. Autrement dit, la requête recherchée contient des attributs qui n'existent pas dans le cache.

Par conséquent, la requête existante (*ExistReq*) définit la partie du segment qui contient la requête de la façon suivante :

$$- Rel (ExistReq) = Rel (Seg).$$

$$- Attr (ExistReq) = Attr (Seg).$$

$$- Pred (ExistReq) = Pred (Seg).$$

La requête manquante (*ManqReq*) se traduit comme suit :

$$- Rel (ManqReq) = Rel (Req).$$

$$- Attr (ManqReq) = Attr (Req) - Attr (ExistReq).$$

$$- Pred (ManqReq) = Pred (Req).$$

Le client envoie la requête manquante *ManqReq* à son voisin. Si ce dernier possède le résultat dans son cache, il le lui envoie.

Dès que l'abonné reçoit le nouveau résultat *Cont (ManqReq)*, il le concatène avec celui qui est dans son cache pour avoir le résultat complet de la requête comme suit:

$$Contfinal (Req) = Cont (ExistReq) Cont (ManqReq).$$

Cas 4: Partitionnement horizontal d'une requête

Figure 3.8. Partitionnement horizontal de requête.

Ce cas révèle une requête divisée horizontalement par un segment de cache. Cela veut dire que la requête dépend d'autres prédicats qui n'existent pas dans le segment.

Pour cela, la sous-requête existante *ExistReq* représente l'intersection de la requête avec le segment, elle sera exprimée comme suit :

$$- Rel (ExistReq) = Rel (Seg).$$

$$- Attr (ExistReq) = Attr (Seg).$$

$$- Pred (ExistReq) = Pred (Seg).$$

Et la requête manquante (*ManqReq*) se traduit comme suit :

$$- Rel (ManqReq) = Rel (Req).$$

$$- Attr (ManqReq) = Attr (Req).$$

$$- Pred (ManqReq) = Pred (Req) - Pred (ExistReq).$$

Le résultat final sera :

$$Contfinal (Req) = Cont (ExistReq) Cont (ManqReq).$$

Cas 5: Partitionnement hybride de requête

Figure 3.9. Partitionnement hybride de requête.

Ce cas exprime le cas d'une requête qui manque d'attributs et de tuples. Elle est décomposée horizontalement et verticalement par un segment. La sous-requête existante s'exprime de la façon suivante :

$$- Rel (ExistReq) = Rel (Seg).$$

$$- Attr (ExistReq) = Attr (Seg).$$

$$- Pred (ExistReq) = Pred (Seg).$$

Et la requête manquante (*ManqReq*) se traduit comme suit :

$$- Rel (ManqReq) = Rel (Req).$$

$$- Attr (ManqReq) = Attr (Req) - Attr (ExistReq).$$

$$- Pred (ManqReq) = Pred (Req) - Pred (ExistReq).$$

Cette fois, la requête manquante se compose de deux autres sous-requêtes : l'une présente le manque d'attributs *Attr (ManqReq)* et l'autre présente le manque de tuples *Pred (ManqReq)*. Ensuite, les deux parties seront fusionnées pour construire un seul résultat.

$$- \text{Rel} (\text{ManqReq}) = \text{Rel} (\text{Req}).$$

$$- \text{Attr} (\text{ManqReq}) = \text{Attr} (\text{Req}) - \text{Attr} (\text{ExistReq}).$$

$$- \text{Pred} (\text{ManqReq}) = \text{Pred} (\text{Req}) - \text{Pred} (\text{ExistReq}).$$

La première partie de requête manquante sera :

$$- \text{Rel} (\text{ManqReq } 1) = \text{Rel} (\text{Req}).$$

$$- \text{Attr} (\text{ManqReq } 1) = \text{Attr} (\text{ManqReq}).$$

$$- \text{Pred} (\text{ManqReq } 1) = \text{Pred} (\text{Req}).$$

Et la deuxième requête manquante sera :

$$- \text{Rel} (\text{ManqReq2}) = \text{Rel} (\text{Req}).$$

$$- \text{Attr} (\text{ManqReq2}) = \text{Attr} (\text{Req}).$$

$$- \text{Pred} (\text{ManqReq2}) = \text{Pred} (\text{ManqReq}).$$

Le résultat final sera exprimé de la manière suivante :

$$\text{Contfinal} (\text{Req}) = [\text{Cont} (\text{ExistReq}) \text{ Cont} (\text{ManqReq1}) \text{ Cont} (\text{ManqReq2})].$$

Exemple :

Prenons la même base des données de test citée précédemment avec un cache d'un client mobile composé de quatre segments pour illustrer les différents cas de traitement de requêtes au niveau du cache.

Tableau 3.2.

Mémoire cache de quatre segments.

Seg	Rel	Attr	Pred	Cont	Temp	Nbr
-----	-----	------	------	------	------	-----

Seg1	Album Track	Title, Bytes	Album.AlbumId = Track. AlbumId, GenreId = '23'	Cont 1	T1	NU 1
Seg2	Artist	Name	AlbumId = '84'	Cont 2	T2	NU 2
Seg3	Track Genre	GenreId, Composer	Track.GenreId = Genre. GenreId, Track.Name = 'Solitary',	Cont 3	T3	NU 3
Seg4	Artist, Album	Name, Title	Artist.ArtistId = Album. ArtistId, ArtistId = '35'	Cont 4	T4	NU 4

Considérons plusieurs cas :

Premier cas : requête introuvable

- Requête 1 :

Select TrackId, Bytes

From Track

Where UnitPrice = '10'

Après le parcours des segments de cache, il s'est avéré que cette requête n'existe pas dans le cache local du client. Donc, les requêtes existantes *ExistReq* et manquantes *ManqReq* seront formulées de la façon suivante:

- *ExistReq* = .

- *ManqReq* = *Req*.

Deuxième cas : requête trouvable

Requête 2 :

Select Name
From Artist
Where AlbumId= '84'

En recherchant cette requête dans le cache du client, on trouve que cette dernière se trouve totalement dans le troisième segment. Alors la requête existante *ExistReq* représente la requête originale comme suit :

ExistReq = Req.

ManqReq = .

Troisième cas : requête trouvée partiellement (manque d'attributs)

Requête 3 :

Select Name, Composer, Milliseconds
From Track, Genre
Where (Track.GenreId =Genre.GenreId) (Track.Name = 'Solitary')

Après le parcours du cache, on constate que le deuxième segment partitionne la requête de façon verticale. C'est-à-dire que la partie d'attributs du segment n'a pas l'attribut 'Milliseconds' et qu'il existe au niveau de la requête.

Dans ce cas, la requête existante *ExistReq* s'exprime comme suit :

Select Name, Composer
From Track, Genre
Where (Track.GenreId =Genre.GenreId) (Track.Name = 'Solitary')

La requête manquante *ManqReq* s'exprime comme suit :

Select Milliseconds
From Track, Genre
Where (Track.GenreId = Genre.GenreId) (Track.Name = 'Solitary')

Tel que :

- $Rel (ManqReq) = Rel (Req) = \{Track, Genre\}.$
- $Attr (ManqReq) = Attr (Req) - Attr (ExistReq) = \{Milliseconds\}.$
- $Pred (ManqReq) = Pred (Req) = \{(Track.GenreID = Genre.GenreID) (Track.Name = 'Solitary')\}.$

Et le résultat final est:

$Contfinal = Cont (ExistReq) \text{ Cont } (ManqReq).$

Quatrième cas : requête trouvée partiellement (manque de tuples)

- Requête 4 :

Select Name, Title
From Artist, Album
Where (Artist.ArtistId = Album.ArtistId) (ArtistId = '35') (AlbumId = '230')

La quatrième requête expose une requête partitionnée horizontalement, où un manque de tuples existe. En effet, le quatrième segment contient la même requête mais avec une condition qui manque. Alors, la partie de requête existante se définit comme suit :

Select Name, Title
From Artist, Album
Where (Artist.ArtistId = Album.ArtistId) (ArtistId = '35')

Et la requête ManqReq est :

```
Select Name, Title
From Artist, Album
Where (Artist.ArtistId = Album.ArtistId)(AlbumId = '230')
```

Tel que :

- $Rel (ManqReq) = Rel (Req) = \{Artist, Album\}$.
- $Attr (ManqReq) = Attr (Req) = \{Name, Title\}$.
- $Pred (ManqReq) = Pred (Req) - Pred (ExistReq) = \{(AlbumId = '230')\}$.

Le résultat final de la requête initialisée sera :

$Contfinal (Req) = Cont (ExistReq) Cont (ManqReq)$.

Cinquième cas : requête trouvée partiellement (manque d'attributs et de tuples)

- Requête:

```
Select Title, Name, Bytes
From Album, Track
Where (Album.AlbumId = Track.AlbumId) (GenreId = '23')(UnitPrice =
'15')
```

Le dernier exemple illustre le cas d'une requête trouvée dans le premier segment du cache, mais avec un manque dans la partie d'attributs et des prédicats. Cette requête va subir un partitionnement hybride, c'est-à-dire, un partitionnement vertical sur les attributs et un partitionnement horizontal sur les conditions. Alors, la requête existante définit le segment comme suit :

Select Title, Bytes
From Album, Track
Where (Album.AlbumId = Track.AlbumId) (GenreId = '23')

Le processus de partitionnement de requête manquante sera :

- $Rel (ManqReq) = Rel (Req) = \{Album, Track\}$.
- $Attr (ManqReq) = Attr (Req) - Attr (ExistReq) = \{Name\}$.
- $Pred (ManqReq) = Pred (Req) - Pred (ExistReq) = \{UnitPrice = '15'\}$.

La première partie de la requête qui définit le manque d'attributs est :

Select Name
From Album, Track
Where (Album.AlbumId = Track.AlbumId) (GenreId = '23')(UnitPrice = '15')

Tel que :

- $Rel (ManqReq\ 1) = Rel (Req) = \{Album, Track\}$.
- $Attr (ManqReq\ 1) = Attr (ManqReq) = \{Name\}$.
- $Pred (ManqReq\ 1) = Pred (Req) = \{(Album.AlbumId = Track.AlbumId) (GenreId = '23')(UnitPrice = 15)\}$.

La deuxième partie de la requête qui définit le manque de tuples est :

Select Title, Name, Bytes
From Album, Track
Where (Album.AlbumId = Track.AlbumId) (UnitPrice = 15)

Tel que :

- $Rel (ManqReq\ 2) = Rel (Req) = \{Album, Track\}.$
- $Attr (ManqReq\ 2) = Attr (Req) = \{Title, Name, Bytes\}.$
- $Pred (ManqReq\ 2) = Pred (ManqReq) = \{(Album.AlbumId = Track.AlbumId)(UnitPrice = 15)\}.$

Le résultat final est exprimé comme suit :

$$Contfinal (Req) = [Cont (ExistReq) Cont (ManqReq1) Cont (ManqReq2)].$$

3.8 Politique de remplacement

La politique de remplacement est une phase importante dans le processus de cache des requêtes. Elle a pour but principal de gérer le cache et d'optimiser ses performances.

Lorsque la taille du cache est épuisée, et qu'il n'y a pas de place pour une nouvelle requête, c'est le rôle des stratégies de remplacement de sélectionner la bonne victime pour la faire remplacer. Cependant, le problème qui se pose réside dans le choix de cette victime.

Un algorithme de cache optimal doit prédire le futur pour désigner la donnée qui ne sera pas utilisée prochainement et pourra être remplacée par une nouvelle donnée. Ceci est irréalisable dans un système réel. Par conséquent, on se sert toujours d'une mesure pour faire un choix de remplacement.

Dans la littérature, il existe plusieurs approches de remplacement, chacune ayant ses avantages et ses inconvénients. La majorité de ces méthodes repose sur le principe de localité qui dépend ou non de l'utilisation des données existantes dans le cache.

Vu que notre stratégie s'appuie sur la sémantique, on a choisi d'implémenter deux méthodes de remplacement qui ont donné de bons résultats dans la gestion du cache sémantique. Notre objectif principal est d'étudier et de comparer l'effet de chaque méthode sur la performance du cache.

La première méthode est LRU (Least Recently Used), une méthode assez reconnue par sa performance dans la gestion de cache. Elle se base sur le principe de localité temporelle où la donnée qui a été moins récemment utilisée sera remplacée par une nouvelle et inversement, tout ce qui a été accédé récemment a de fortes chances d'être référé dans le futur.

La deuxième méthode est MRU (Most Recently Used). Son principe est l'inverse de celui de LRU. Elle suppose que la donnée la plus utilisée dans le passé aura moins de chance d'être référée dans le futur. Le remplacement sera effectué sur la donnée la plus récemment utilisée.

La réalisation de ces deux méthodes peut se faire de différentes manières. Dans notre travail, on se sert d'un bit de référence pour enregistrer les accès aux segments du cache ainsi que le temps de chargement de la donnée pour en déduire la ligne la moins récemment utilisée ou la plus récemment utilisée.

Pour évaluer la performance de nos méthodes de remplacement, nous avons choisi de nous servir des trois métriques les plus importantes qui sont : le temps de réponse, le hit ratio et le taux d'erreurs.

Comme nous l'avons déjà précisé dans la section précédente, chaque segment au niveau de notre cache possède un indicateur (NU) pour calculer le nombre d'utilisations de chaque requête et pour choisir le segment victime.

Dans la section suivante, nous présentons les algorithmes à implanter en pseudo code :

Algorithme

L'algorithme suivant est développé pour le but de partitionner la requête initialisée et de trouver son résultat final que ça soit au niveau du cache local, ou auprès du cache du voisin, sinon auprès du serveur de base de données.

Algorithme *partitionnerRequete* :

Entrée:

Une requête : Req < Rel (Req), Attr (Req), Pred (Req) >

Un cache : C ensemble de segments sémantiques Seg < Rel (Seg), Attr (Seg), Pred (Seg), Cont (Seg), Temp (Seg), Nbr (Seg) >

Sortie:

Résultat d'une requête (ResultReq)

Res1 vide;

Res2 vide;

Début

Tant que (cache n'est pas pleine) faire

// Requête trouvable

Cas1: $(Attr(Req) \text{ Attr}(Seg) = Attr(Req)) \wedge (\sigma_{Pred(Req)} \sigma_{Pred(Seg)} = Pred(Req))$

ExistReq Seg;

ManqReq vide;

ResultatReq $\leftarrow$ Cont (Seg);

// Requête introuvable

Cas2: $((Attr(Req) \text{ Attr}(Seg) =) \wedge (\sigma_{Pred(Req)} \sigma_{Pred(Seg)} =))$

```

ExistReq vide;
ManqReq Req;
EnvoiVoisin (ManReq);
Si VoisinResultReq = alors
    envoiServeur (ManqReq)
    ResultReq ServeurResultReq;
Sinon
    ResultReq VoisinResultReq;
Fin Si
ResultReq = ResultReq1;
enregistrerRequete;
// Partitionnement verticale
Cas3: (Attr (Req) Attr (Seg) Attr (Req)) ( $\sigma_{Pred (Req)}$   $\sigma_{Pred (Seg)}$  = Pred (Req))
    Cont (ExistReq) Cont (Seg);
    ManqReq (Rel (Req), Attr (Req) - Attr (Seg), Pred (Req));
    envoiVoisin (ManqReq);
    Si VoisinResultReq = alors
        envoiServeur (ManqReq)
        ResultReq1 ServeurResultReq;
    Sinon
        ResultReq1 VoisinResultReq;
    Fin Si
    ResultReq ResultReq1 Cont (Seg);
    enregistrerRequete;
//Partitionnement horizontale
Case4: (Attr (Req) Attr (Seg) Attr (Req)) ( $\sigma_{Pred (Req)}$   $\sigma_{Pred (Seg)}$  Pred (Req))
    Cont (ExistReq)Cont (Seg)
    ManqReq (Rel (Req), Attr (Req), Pred (Req) - Pred (Seg))

```



```

EnvoiVoisin (ManqReq)
Si VoisinResultReq = alors
    envoiServeur (ManqReq)
    ResultReq1 ServeurResultReq
Sinon
    ResultReq1 VoisinResultReq
Fin Si
ResultReq ResultReq1 Cont (Seg)
enregistrerRequete (Req)

// Partitionnement hybride
Cas5(Attr (Req) Attr (Seg) Attr (Req)) ( $\sigma_{Pred (Req)}$   $\sigma_{Pred (Seg)}$  Pred (Req))
    Cont (ExistReq)Cont (Seg)
    ManqReq (Rel (Req), Attr (Req) - Attr (Seg), Pred (Req))
    ManqReq (Rel (Req), Attr (Req), Pred (Req) - Pred (Seg))
    EnvoiVoisin (ManqReq1)
    Si VoisinResultReq = alors
        envoiServeur (ManqReq)
        ResultReq1 ServeurResultReq
    Sinon
        ResultReq1 VoisinResultReq
    Fin Si
    EnvoiVoisin (ManqReq2)
    Si VoisinResultReq = alors
        envoiServeur (ManqReq)
        ResultReq2 ServeurResultReq
    Sinon
        ResultReq2 VoisinResultReq
    Fin Si

```

ResultReq [ResultReq1 Cont (Seg)] ResultReq2
 enregistrerRequete (Req)

Fin Tant que

Cas 01: // Remplacement LRU

Min min (Nbr)

MinNbr Seg (1)

Pour indice de 1 jusqu'au dernier segment faire

Si Nbr [Seg (indice)] < MinNbr alors

MinNbr Nbr [Seg (indice)]

Fin Si

Fin Pour

Segment (MinNbr) vide

Cas 02 : // Remplacement MRU

Min min (Nbr)

MaxNbr Seg (1)

Pour indice de 1 jusqu'au dernier segment faire

Si Nbr [Seg (indice)] > MaxNbr alors

MaxNbr Nbr [Seg (indice)]

Fin Si

Fin Pour

Segment (MaxNbr) vide

Fin

Algorithme envoiVoisin:

Entrée:

Une requête : Req < Rel (Req), Attr (Req), Pred (Req) >

Un cache voisin : C ensemble de segments sémantiques Seg < Rel (Seg), Attr (Seg),

Pred (seg), Cont (Seg), temp (Seg), Nbr(Seg) >

Sortie :

Résultat de requête dans le cache voisin (VoisinResultReq)

Début

VoisinResultReq vide

Pour indice de 1 jusqu'au dernier segment faire

Si ($\text{Attr}(\text{Req}) \neq \text{Attr}(\text{Seg})$ $\text{Attr}(\text{Req})$) ($\sigma_{\text{Pred}(\text{Req})} \sigma_{\text{Pred}(\text{Seg})} = \text{Pred}(\text{Req})$) alors

VoisinResultReq cont (Seg)

Sinon

VoisinResultReq vide

Fin Si

Fin Pour

Envoyer VoisinResultReq;

Fin

CHAPITRE IV

PROTOTYPE ET ÉVALUATION

Après avoir défini le processus de partitionnement des différents types de requêtes au niveau du cache du client, nous allons passer aux détails de notre prototype et au dévoilement des résultats d'évaluation.

La conception de ce prototype vise à réaliser deux objectifs principaux. Le premier, concerne l'étude et l'implémentation des propositions et des algorithmes décrits dans le chapitre précédent. Le second objectif est d'exploiter notre prototype pour évaluer la performance du résultat.

L'environnement que nous considérons est composé de deux unités qui communiquent avec un serveur de base de données à travers le réseau sans fil CDMA 2000.

La première unité est présentée par un simulateur inclus dans un MacBook Pro et la deuxième est un iPhone Mac OS X 10.5.8.

Précisons que nous ne nous intéressons pas aux problèmes causés par la variation du délai du réseau et nous supposons avoir des garanties sur la bande passante.

4.1 Architecture du prototype

L'architecture logicielle de notre prototype s'articule autour de cinq principaux modules qui sont (voir Figure 4.1) : l'unité du client avec son cache,

l'unité du voisin et son cache et le serveur de base de données. La conception de ces modules sera détaillée dans la section suivante.

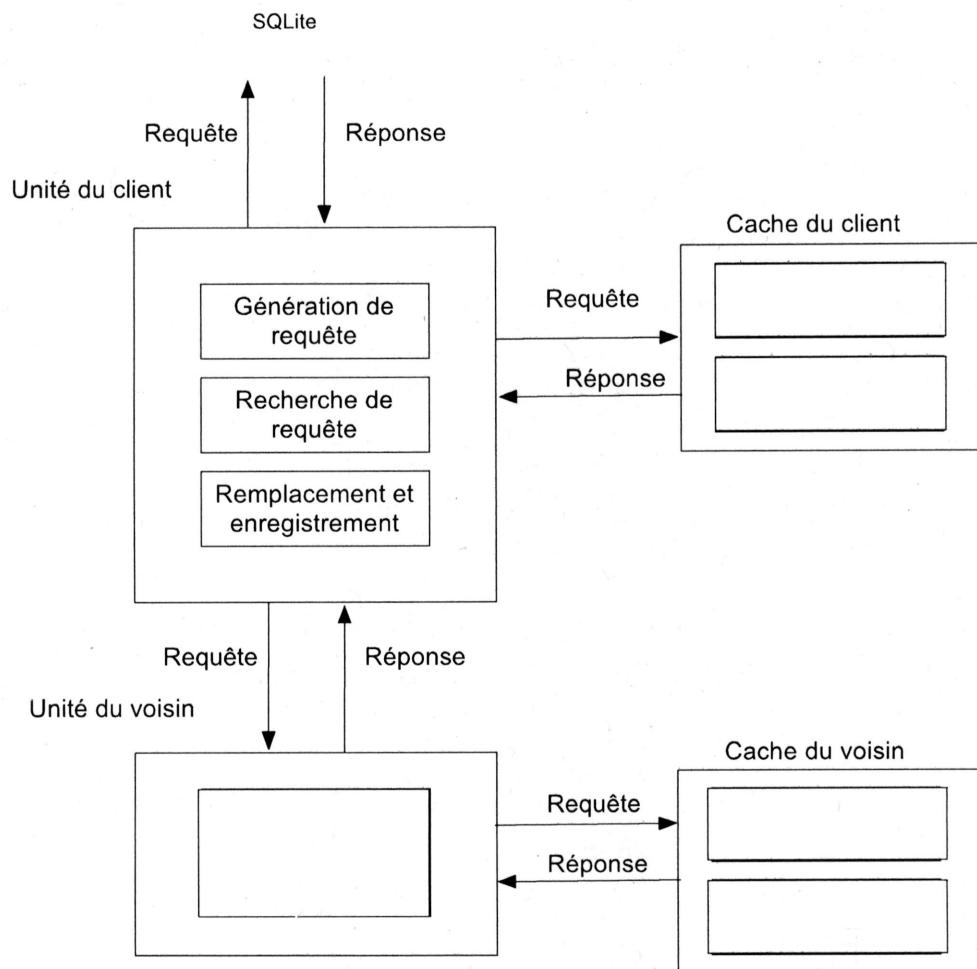


Figure 4.1. Architecture du prototype.

4.1.1 Unité du client

Le client est présenté par le simulateur IOS qui se trouve au cœur de l'environnement Xcode du MacBook pro utilisé. Cette station se sert de sa mémoire

cache pour enregistrer les requêtes afin d'accélérer le traitement des instructions par le processeur. Au début de cette expérience, on considère que le cache du simulateur est vide, ensuite il se remplira au fur à mesure qu'on enregistre des requêtes. Une fois remplie, le cache a besoin de remplacer les données inutiles par des nouvelles données à travers les méthodes de remplacement.

Tel qu'expliqué dans le chapitre précédant, on ne peut pas savoir qu'elles sont les données qui seront utiles ou non ultérieurement. Nous allons nous baser sur le nombre d'utilisations du segment comme mesure pour désigner la victime qui sera remplacée.

1. Génération des requêtes

Le premier module est la génération des requêtes. Il sert à générer aléatoirement quatre types de requêtes en langage SQL de types sélection et projection sur la base de données de test [14], comme suit :

Type 1:

```
SELECT Title  
FROM Album  
WHERE ArtistId = $p1
```

Type 2:

```
SELECT Name  
FROM Track  
WHERE (AlbumId = $p1) and (GenreId = $p2)
```

Type 3:

```
SELECT Name, Title from Artist  
FROM Artist  
WHERE Artist.ArtistId = $p1"
```

Type 4:

```
SELECT FirstName, LastName  
FROM Customer  
WHERE (supportRepId = $p1)
```

Sachant que p1 et p2 sont des entiers choisis de façon aléatoire.

2. Recherche de requêtes

Ce module a pour mission de rechercher chaque requête générée par le client. En premier lieu, la recherche est effectuée dans le cache du client, sinon elle est envoyée à l'unité voisine. Si cette dernière ne trouve pas la réponse de la requête dans son cache, ce module s'occupe d'envoyer la requête au serveur de la base de données.

3. Remplacement et enregistrement de requêtes

Le module de remplacement et d'enregistrement de résultat permet de recevoir les résultats *Cont (ManqReq)* et *Cont (ExistReq)*, de les fusionner en un seul résultat et de l'enregistrer dans le cache. Si ce dernier est plein, des méthodes de remplacement seront utilisées pour supprimer des données inutiles.

4.1.2 Cache du client

Le cache du client est l'unité principale pour dérouler nos algorithmes. Il reçoit les requêtes générées par le client et les enregistre.

1. Processus de cache

Chaque requête générée sera étudiée et partitionnée s'il le faut selon le type de partitionnement afin de récupérer la réponse.

2. Recherche de résultats

Ce sous-module fait le parcours des segments de cache pour retrouver le résultat des requêtes.

Précisons que ces différents modules se déroulent dans l'environnement Xcode. Ce dernier est un environnement de développement intégré conçu spécialement pour les systèmes Mac OS X. Cet outil est fourni avec une suite de logiciels nécessaires pour le développement et le fonctionnement d'une application. Xcode peut se télécharger gratuitement sur le Mac App Store. La version utilisée dans ce travail est la version 4.

4.1.3 Unité du voisin

La station voisine est représentée par un iPhone 4 qui est connecté au réseau CDMA2000. Cette station reçoit les requêtes auprès du client, les cherche dans sa mémoire cache, ensuite lui retourne leurs réponses si celles-ci existent.

4.1.4 Cache du voisin

Le cache du voisin a la même organisation que celui du client et joue le même rôle. En outre, il se charge de chercher le résultat de la requête reçue dans ses segments. On considère que la réponse sera soit complète, soit inexistante.

4.1.5 VLR

Le module VLR est une composante du réseau sans fils CDMA2000 qui contient les informations générales des abonnés mobiles qui appartiennent à une région donnée du réseau. Dans notre étude, on considère que le VLR fournit au client l'adresse de son voisin le plus proche.

4.1.6 Serveur

Le SQLite est le serveur de base de données choisi pour notre prototype. Ce choix est basé sur deux raisons principales : la première est que SQLite est un serveur très populaire dans les systèmes d'exploitation IOS. Il est simple, complet et ne nécessite aucune configuration. La seconde raison est qu'il est un système fiable, embarqué et qui ne demande pas le paradigme client-serveur.

Dans cette étude, on considère que notre serveur de base de données a deux localisations. La première localisation est considérée au sein du PDSN du réseau local. La deuxième est supposée dans un PDSN distant qui appartient à un autre réseau.

Notre langage de programmation côté serveur est le PHP. Ce choix est motivé par le fait que c'est un des langages les plus utilisés pour la programmation web. De plus, il est gratuit, simple et rapide.

Le rôle du serveur est de répondre aux requêtes émises par le client. Ces requêtes sont générées aléatoirement par un extrait d'une base de données de test [14]. La figure 4.2 montre le modèle relationnel de cette base de données.

Nos données sont transmises du serveur au simulateur en format Json. C'est un format de fichier alternatif à XML, se basant principalement sur la syntaxe des structures des tableaux de JavaScript. Il est rapide, léger, performant et surtout minimaliste par rapport au XML. Sa structure est compacte et formée de paires clé/valeur faciles à manipuler dans JavaScript ou en PHP.

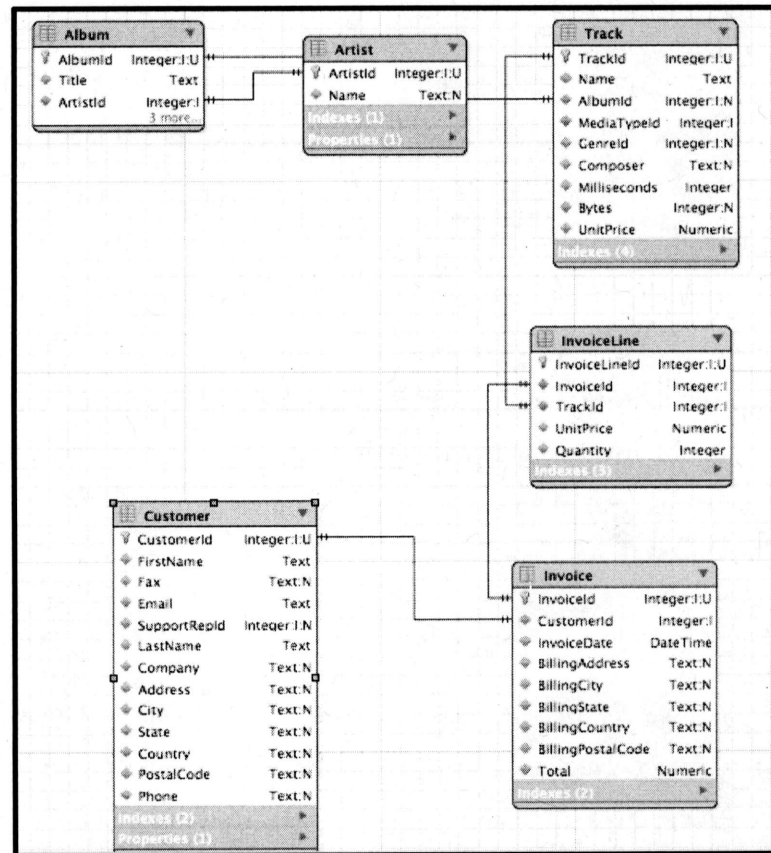


Figure 4.2. Modèle relationnel de base de données utilisée [14].

Pour employer le Json dans notre prototype, il suffit de télécharger sa librairie principale auprès du site web principal [15], et ensuite de l'inclure dans le projet principal.

4.2 Conception

La conception de notre prototype a nécessité le développement de six principaux objets, formés des classes Objective C, chacune étant définie dans un fichier d'entête `<classe.h>` et son implémentation `<classe.m>` :

- `<DB1Handler.h, DB1Handler.m>` : la classe qui permet d'ouvrir et de faire connecter notre base de données de test au serveur.

- <WebServiceClass.h, WebServiceClass.m> : ce processus traite l'envoi et la réception des requêtes entre les différentes unités : client, voisin et serveur. Rappelant que les données sont transitées en format json.
- <DemandeVlr> : une classe qui s'occupe de fournir l'adresse du voisin à la demande du client.
- <voisin.h, voisin.m> : la classe suivante traite le processus d'envoi et de réception de requêtes vers le voisin.
- <DemandeServeur.h, DemandeServeur.m> : la classe suivante décrit les étapes d'envoi et de réception des requêtes auprès du serveur.
- <DBRequestHandler.h, DBRequestHandler.m> : la classe qui définit les principales instructions pour la génération de requêtes.

Sachant que :

- <NSObjet> : La classe racine de tout programme Objective C.
- <AppDelegate.h, AppDelegate.m > : le fichier principal dans tout projet Objective C qui contient toutes les instructions de notre prototype.

4.3 Expériences et résultats

Notre système est conçu pour trois principaux objectifs. Il nous permet de :

- Étudier la différence de performance entre le cache selon la page et la mise en cache sémantique.
- Faire une évaluation sur notre mécanisme de cache proposée.
- Comparer le mécanisme de cache utilisé avec deux méthodes de remplacement pour déduire la gestion efficace du cache.
- Utiliser trois métriques pour évaluer la gestion du cache :

- *Le temps de réponse* : il reflète l'intervalle entre l'envoi de requête et la réception du résultat. Généralement, il dépend du temps de transmission ainsi que le temps d'exécution sous la formule suivante:

$$\text{Le temps de réponse} = \text{temps d'exécution} + \text{temps de transmission}$$

Le temps de transmission désigne le temps nécessaire pour transmettre une requête via le réseau depuis le début de l'envoi jusqu'à sa réception. Il est calculé à l'aide de la taille des tuples transférés et le débit du réseau, sous la formule suivante :

$$\text{Le temps de transmission} = \text{taille de message} / \text{débit}.$$

$$\text{Taille message} = \text{nombre tuples} * \text{taille tuple}.$$

- *Le taux de réussite (hit ratio)*: représente le pourcentage des données recherchées qui sont retrouvées dans le cache du client.

- *Le taux d'erreurs (miss ratio)* : définit le taux des requêtes non trouvées au niveau du cache du client et qui seront récupérées auprès de la mémoire. Il est déduit du hit ratio de la façon suivante :

$$\text{Miss ratio} = 1 - \text{hit ratio}.$$

Le tableau suivant liste les paramètres essentiels utilisés dans notre étude :

Tableau 4.1.

Paramètres d'évaluation.

Paramètres	Descriptions	Valeurs
tailledb	Taille de base de données	15607 tuples
tailletuple	Taille de tuple	800 octet
Taillepage	Taille de page	1024 octet
Taillecache	Taille de cache client	500 K.o
Débit	Débit du réseau	10 Mbit/s
NbrReq	Nombre de requêtes générées	[200 - 1000] requêtes

- **Expérience 01**

Notre première expérience a permis d'étudier l'intérêt d'utiliser le cache selon une sémantique au lieu d'un cache en pages. Nous avons généré aléatoirement de 200 à 1000 requêtes aléatoires pour chaque mécanisme utilisé avec deux méthodes de remplacement différentes, LRU et MRU.

Pour une mise en cache en pages, le client envoie la page qui contient la requête au serveur. Ce dernier prépare le résultat et le renvoie au client sous la forme de pages/résultats. Par contre, dans le cas où le cache maintient une description sémantique, le serveur reçoit seulement la requête auprès du client. Il la traite et transmet le résultat sous la forme de requête/résultats, ce qui va alléger le message transféré.

Le tableau suivant montre le calcul du temps de transmission pour des requêtes envoyées d'un serveur au client lors d'une mise en cache selon la page.

Rappelant que la taille d'une page est 1024 octets, et la taille d'un tuple est de 800 octets, c'est-à-dire que l'envoi d'une page contient un tuple.

Tableau 4.2.

T

emps de transmission pour des requêtes issues du serveur dans un cache en pages.

NbrReq	Nbrtuples transmis	Tailletuples (octet)	Temps de transmission (secondes)
100	200	$200 \times 1024 = 204800$	0,156
200	320	$320 \times 1024 = 327680$	0,249
300	415	$415 \times 1024 = 424960$	0,324
400	530	$530 \times 1024 = 542720$	0,414
500	612	$612 \times 1024 = 626688$	0,478
600	678	$678 \times 1024 = 694272$	0,529
700	716	$716 \times 1024 = 733184$	0,559
800	765	$765 \times 1024 = 783360$	0,597
900	800	$800 \times 1024 = 819200$	0,624
1000	850	$850 \times 1024 = 870400$	0,664

Le tableau 4.3 montre le calcul du temps de transmission pour des requêtes envoyées d'un serveur au client dans une affaire de cache sémantique. Les résultats transférés au client comportent seulement les tuples qui répondent aux requêtes. Alors, le temps de transmission sera calculé de la façon suivante :

Le temps de transmission est : $(\text{NbrTuples} * 800) * 7,629395 * 10^{-6} / 10$.

Tableau 4.3

Calcul du temps de transmission pour des requêtes issues du voisin dans un cache sémantique.

NbrReq	NbrTuples transmis	Tailletuples (octets)	Temps de transmission (seconde)
100	61	$61 * 800 = 12800$	0,009
200	111	$111 * 800 = 88800$	0,067
300	172	$172 * 800 = 137600$	0,104
400	237	$237 * 800 = 189600$	0,144
500	307	$307 * 800 = 245600$	0,187
600	370	$370 * 800 = 296000$	0,225
700	438	$438 * 800 = 348400$	0,265
800	496	$496 * 800 = 396800$	0,302
900	554	$554 * 800 = 433200$	0,338
1000	614	$614 * 800 = 491200$	0,374

Nous voyons que les requêtes qui n'ont pas eu de résultats localement, sont envoyées au serveur d'un nombre réduit. En plus, ces requêtes sont renvoyées au client avec un résultat minimum, comportant seulement les tuples nécessaires, ce qui explique la réduction dans le temps de transmission et le temps de réponse.

La figure 4.3 montre le temps de réponse des deux systèmes de mise en cache. On peut voir que la mise en cache en pages surpasse complètement le cache sémantique, ce qui peut se traduire par le nombre élevé des requêtes envoyées au serveur ainsi que les données volumineuses transférées au client sur les pages.

En effet, le traitement d'une requête, dans le cache en pages, nécessite de visiter et de traiter toutes les pages qualifiées, même si elles contiennent un petit nombre de données avec un trafic de plus vers le serveur qui comprend dans la plus pars des données inutiles. Tous ces facteurs jouent un rôle important dans l'augmentation du temps de réponse.

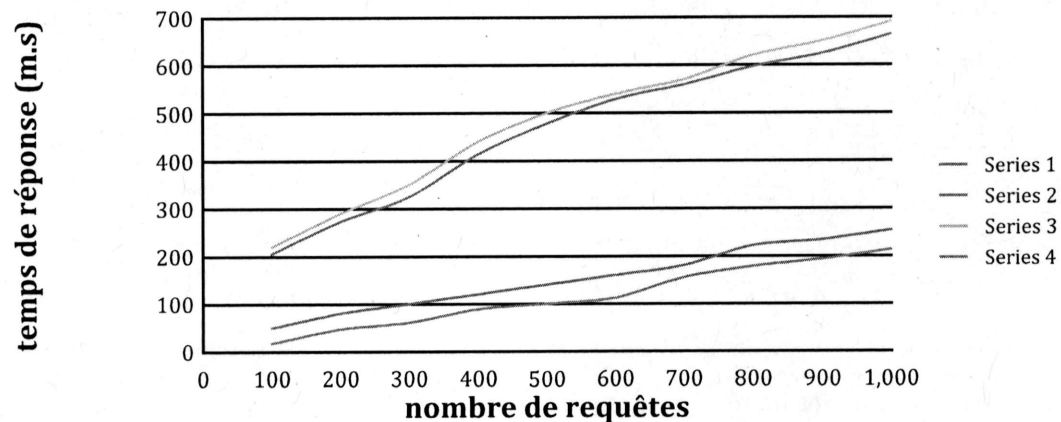


Figure 4.3. Temps de réponse des requêtes issues de cache en pages et de cache sémantique selon LRU et MRU.

On peut remarquer aussi que la différence entre le remplacement LRU et MRU dans un cache en pages est très faible comparé à celui du cache sémantique, où LRU donne un meilleur temps de réponse que celui du MRU. Cette différence peut

s'expliquer par le fait qu'un remplacement du moins récemment utilisé ou le plus récemment utilisé n'affecte pas la mise d'un cache en pages. Par contre, dans un processus sémantique, le remplacement LRU se comporte mieux dans la gestion d'un cache.

- **Expérience 02**

Dans une seconde expérience, nous étudions le profit de se servir d'un cache sémantique que ça soit localement, ou à distance auprès du nœud voisin. Nous procédons avec la même sélection précédente de requêtes. Cependant, ces dernières sont recherchées premièrement dans le cache du client, ensuite, pour celles qui n'ont pas eu de réponses, seront exécutées dans le cache du voisin proche, dont l'adresse est récupérée auprès du VLR. À la fin, le reste des requêtes qui n'ont toujours pas de réponse, sera envoyé au PDSN local qui est le dépôt de notre serveur de base de données.

Les résultats présentés dans la Figure 4.4 montrent à la fois, le nombre de requêtes trouvées localement, chez le voisin et auprès du serveur de base de données.

On remarque que l'utilisation du cache joue un rôle important à réduire le trafic vers le serveur et à diminuer l'utilisation des ressources réseau ainsi que de baisser le temps de réponse des requêtes.

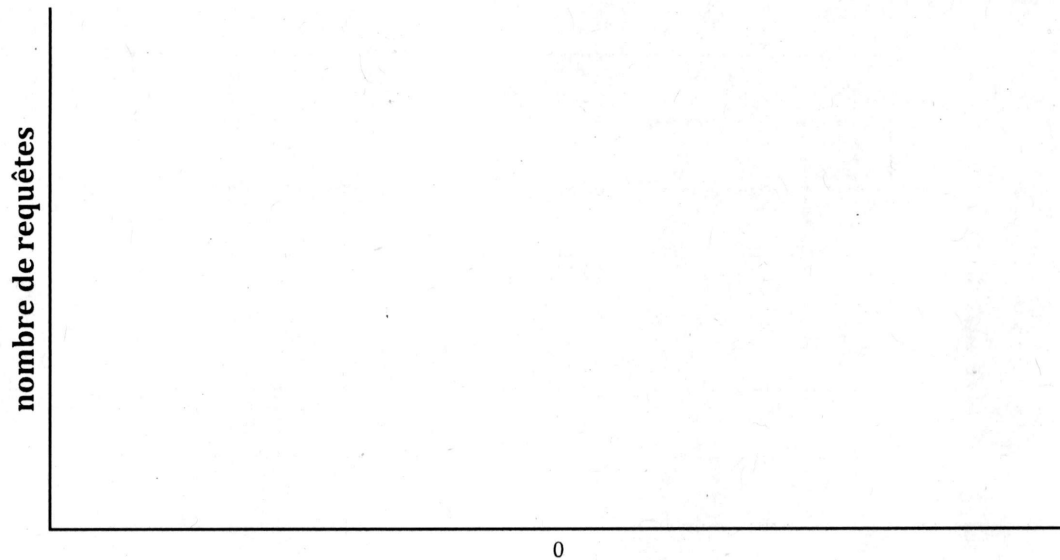


Figure 4.4. Nombre de requêtes récupérées localement, chez le voisin et le serveur.

Nous calculons aussi le temps de réponse des requêtes exécutées dans les trois cas. Notons que notre étude considère deux emplacements différents pour le serveur de base de données. Dans un premier cas, le serveur se trouve dans le PDSN local, ensuite il est considéré dans un PDSN distant d'un autre réseau avec une distance d'itinérance $d = 2$, c'est-à-dire que la distance entre les PDSN est de 2.

Dans ce deuxième cas, où le PDSN se trouve dans un réseau à distance, le temps de réponse sera calculé tel que montré précédemment, sauf que le temps de transmission sera multiplié par le facteur d'itinérance qui est de 2.

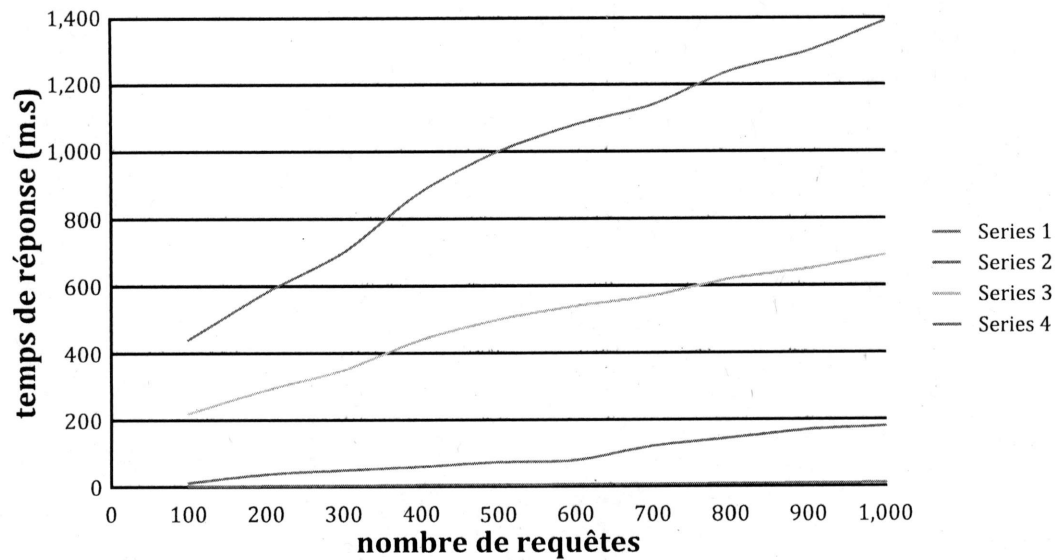


Figure 4.5. Temps de réponses des requêtes trouvées localement, chez le voisin et auprès du serveur.

D'une manière générale, on remarque que le temps de réponse moyen calculé dans les quatre cas augmente d'une manière proportionnelle au nombre de requêtes, tel que montré dans la Figure 4.5.

de plus, le temps de réponse moyen des requêtes trouvées localement est nettement meilleur que le cas où elles sont trouvées auprès du voisin. De même, les requêtes obtenues auprès du voisin donnent un temps de réponse amélioré que celui du serveur local ou distant, ce qui confirme l'importance d'utiliser le cache du voisin pour améliorer la performance du système.

Cette différence du temps de réponse s'explique par le nombre élevé de données transférées ainsi que la distance propagée entre le client et le serveur.

- **Expérience 03**

La troisième expérience montre une comparaison entre les deux méthodes de remplacement LRU et MRU employées dans le cache sémantique. En effet, on a mesuré le hit ratio et le taux d'erreur pour la même sélection précédente des requêtes recherchées localement mais avec une variance dans la taille de cache de 100 à 500 koctets.

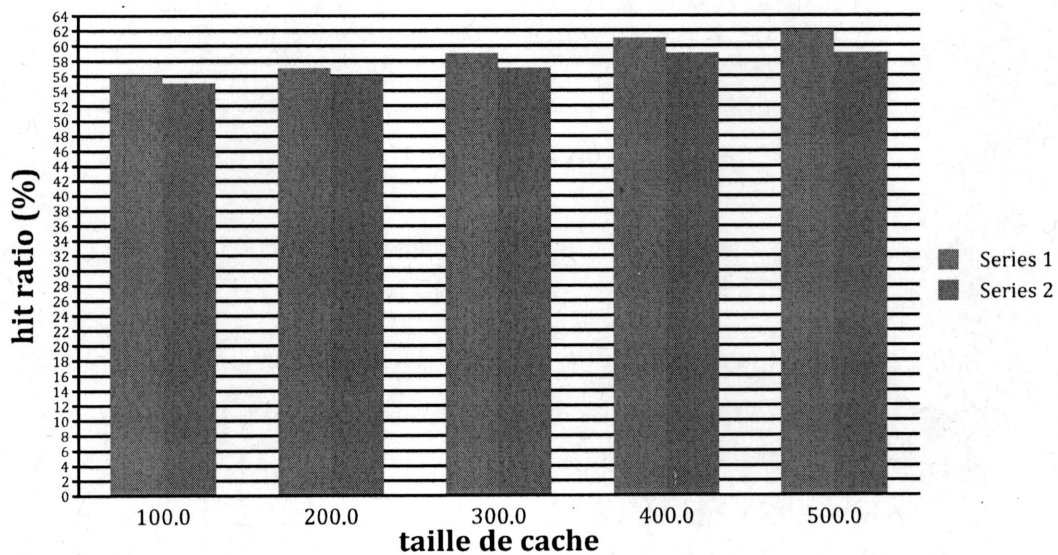


Figure 4.6. Hit ratio en fonction de cache.

De façon générale, nous constatons que l'intervalle de hit ratio varie entre 55% et 62%. Lorsque le remplacement LRU est utilisé, il procure un taux de hit ratio plus élevé que celui du MRU. Prenons par exemple le cas où le cache atteint sa taille maximale de 500 ko, le pourcentage des requêtes trouvées dans le cache atteint son maximum de 62% avec LRU et 59% avec MRU. Cette différence montre que remplacer le segment le moins récemment utilisé dans le passé est préférable, car les requêtes qui viennent d'être utilisées récemment ont plus de chance d'être référencées dans le futur.

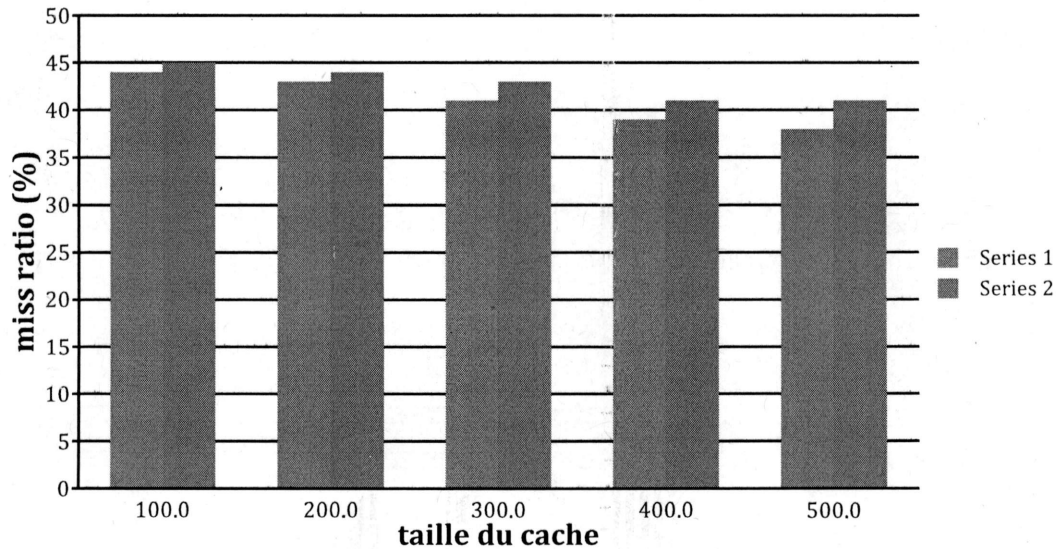


Figure 4.7. Miss ratio de cache sémantique LRU/MRU en fonction de cache.

La deuxième métrique mesurée est le taux d'erreur. La figure 4.7 montre que le taux des requêtes qui n'ont pas de réponse varie entre 38% et 45% selon la taille de cache. Entre autre, pour une taille de 100 ko ou de 200 ko, LRU présente un résultat légèrement meilleur que MRU. Toutefois, si la taille de cache grandit de 300 ko à 500 ko, le taux de miss ratio se réduit pour les deux cas. En plus, la différence entre les deux méthodes de remplacement utilisées commence à s'accroître, ce qui signifie que l'espace au niveau de cache est un facteur aussi important à considérer pour la gestion de cache.

À cet égard et au vu des résultats d'évaluation, l'algorithme de cache sémantique avec un remplacement LRU présente de meilleures performances que le MRU. Cela peut s'expliquer par le fait qu'une requête qui n'est pas utilisée dans le passé a moins de chances d'être référencée dans le futur. Donc, supprimer cette requête, et la remplacer par une nouvelle donne un aspect positif pour notre gestion de cache.

D'un autre côté, certaines limitations s'appliquent sur l'algorithme proposé, exactement au niveau des prédicats qui se restreignent dans les opérations de '=', ce qui conduit à développer d'autres algorithmes plus inclusifs.

4.4 Résultats du prototype

Nous présentons ici le résultat des trois différents cas de recherche des requêtes expérimentées à l'aide du prototype mis en œuvre.

Rappelons que les résultats des requêtes se présentent dans le format de Json.

- **Premier cas :**

Requête:

Select Name

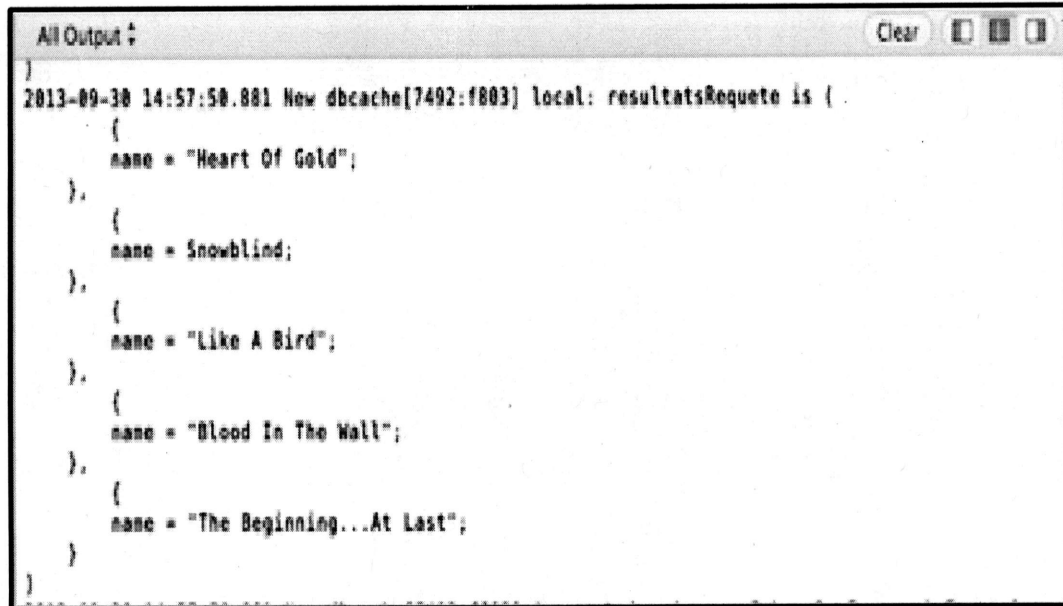
From Track

Where

(AlbumId = '23') and (GenreId = '14')

Ce premier exemple implique une requête SQL exécutée de type 4 choisie aléatoirement parmi les requêtes générées. En premier lieu, la recherche de requête sera effectuée dans le cache du client.

Le résultat retourné est montré dans la figure 4.8.



```

All Output
2013-09-30 14:57:50.881 New dbcache[7492:f803] local: resultsRequete is (
  {
    name = "Heart Of Gold";
  },
  {
    name = Snowblind;
  },
  {
    name = "Like A Bird";
  },
  {
    name = "Blood In The Wall";
  },
  {
    name = "The Beginning...At Last";
  }
)

```

Figure 4.8. Résultat de requête trouvée localement.

Deuxième cas :

Requête :

Select FirstName, LastName

From Customer

Where CustumerId = '53'

Le deuxième exemple présente le cas d'une requête qui n'a pas trouvé sa réponse dans le cache local du client, ce qui nécessite sa recherche auprès du voisin.

La figure 4.9 illustre le résultat de la requête chez le voisin.

```

All Output
2013-05-02 06:43:38.394 New dbcache[4377:f803] Voisin: resultatsRequete is (
  {
    FirstName = "Fran\U00e7ois";
    LastName = Tremblay;
  },
  {
    FirstName = Mark;
    LastName = Phillips;
  },
  {
    FirstName = Jennifer;
    LastName = Peterson;
  },
  {
    FirstName = Robert;
    LastName = Brown;
  },
  {
    FirstName = Edward;
    LastName = Francis;
  },
  {
    FirstName = Martha;
    LastName = Silk;
  },
  {
    FirstName = Aaron;
    LastName = Mitchell;
  },
  {
    FirstName = Ellie;
    LastName = Sullivan;
  }
)
2013-05-02 06:43:38.395 New dbcache[4377:f803] c:\Program Files\Internet Explorer\iexplore.exe http://localhost/~counia/

```

Figure 4.9. Résultat de requête trouvée auprès du voisin.

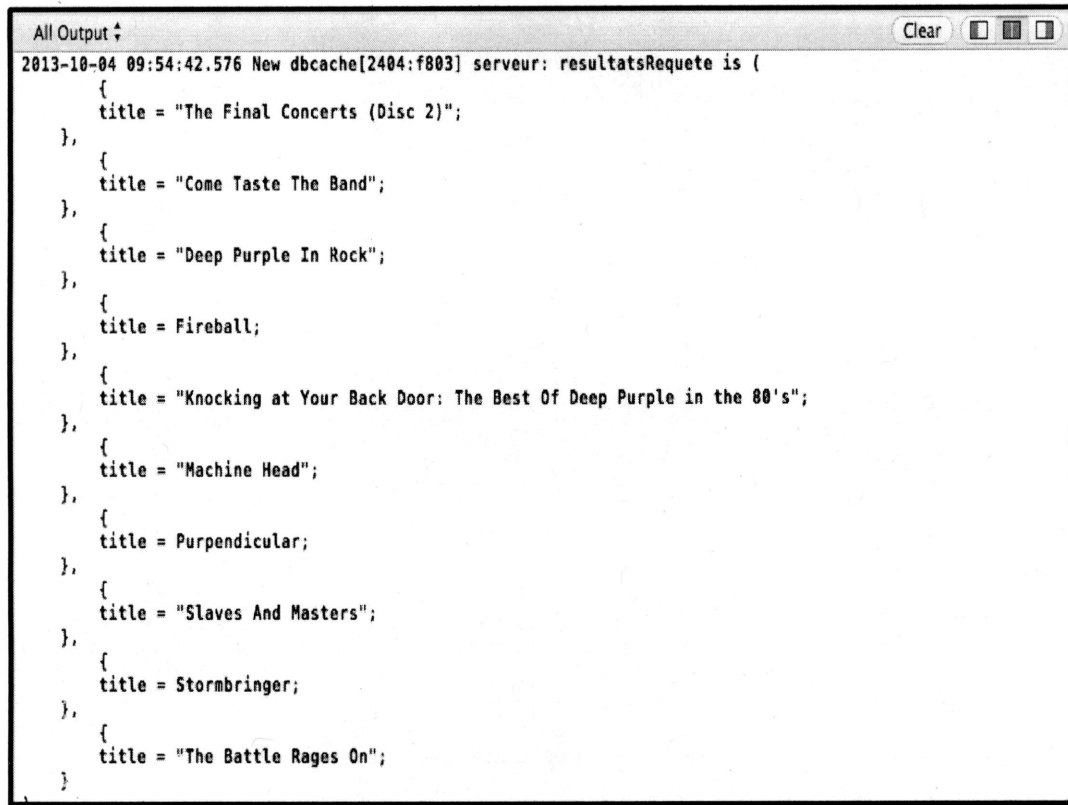
- Troisième cas :

Requête :

Select Title

From Album

Where ArtistId = '58'

A screenshot of a terminal window titled "All Output". The window shows a timestamp "2013-10-04 09:54:42.576" followed by the text "New dbcache[2404:f803] serveur: resultatsRequete is (". Below this, a JSON array is displayed, containing ten objects, each with a "title" property. The titles are: "The Final Concerts (Disc 2)", "Come Taste The Band", "Deep Purple In Rock", "Fireball", "Knocking at Your Back Door: The Best Of Deep Purple in the 80's", "Machine Head", "Purpendicular", "Slaves And Masters", "Stormbringer", and "The Battle Rages On". The terminal window has a "Clear" button and three window control icons in the top right corner.

```
All Output
2013-10-04 09:54:42.576 New dbcache[2404:f803] serveur: resultatsRequete is (
  {
    title = "The Final Concerts (Disc 2)";
  },
  {
    title = "Come Taste The Band";
  },
  {
    title = "Deep Purple In Rock";
  },
  {
    title = Fireball;
  },
  {
    title = "Knocking at Your Back Door: The Best Of Deep Purple in the 80's";
  },
  {
    title = "Machine Head";
  },
  {
    title = Purpendicular;
  },
  {
    title = "Slaves And Masters";
  },
  {
    title = Stormbringer;
  },
  {
    title = "The Battle Rages On";
  }
}
```

Figure 4.10. Résultat de requête trouvée chez le serveur.

Le troisième exemple implique le résultat d'une requête recherchée auprès du serveur, vu que la recherche locale et celle du voisin n'ont pas donné de résultat (figure 4.10).

CONCLUSION

Les progrès réalisés par les technologies des réseaux sans fil ainsi que les bases de données distribuées, ont créé un fort potentiel de développement de nouveaux services et d'applications. Il s'agit particulièrement des applications basées sur l'accès aux données par des usagers mobiles, particulièrement celles qui sont les produits de requêtes auprès des systèmes de gestion de données. Ce type d'accès nécessite de garder des données de manière temporaire dans une mémoire cache.

Un de ces mécanismes de cache consiste à utiliser le cache sémantique qui a généré plusieurs défis dans différents secteurs et qui a fait l'objet de ce mémoire.

Dans ce mémoire, nous nous sommes intéressés aux problèmes liés au traitement des requêtes au niveau du cache. Ainsi, nous avons abordé le principe du fonctionnement du cache.

Comme moyen d'accès aux bases de données nous avons utilisé les réseaux de téléphonie mobile 3G, notamment le réseau CDMA2000, pour supporter l'échange des requêtes entre les différents nœuds mobiles.

Nous avons abordé l'organisation des caches qui se base sur une description sémantique pour enregistrer les requêtes et les problèmes liés aux décompositions des requêtes dans le cache, et avons proposé notre solution de recherche et de partitionnement des requêtes.

Nous avons détaillé l'implémentation de notre prototype, ainsi que ses différents modules, permettant de montrer la faisabilité de nos solutions et d'évaluer leurs performances.

Notre prototype nous a permis de montrer le traitement des requêtes aléatoires dans un réseau de téléphonie mobile, entre deux unités mobiles et un serveur de base de données. Nous avons considéré deux emplacements pour le serveur, le premier au sein du PDSN local, le deuxième dans un PDSN distant. Nous avons employé le mécanisme de cache sémantique au niveau du cache du client ainsi que les nœuds voisins.

Nous nous sommes focalisés sur deux méthodes de remplacement qui sont LRU et MRU afin de choisir l'approche efficace pour la gestion du cache. En effet, lors de la mise en œuvre de notre prototype, nous avons constaté que le cache sémantique local ou du nœud voisin joue un rôle important de réduire le trafic vers le serveur des bases de données et d'augmenter la performance du système. D'un autre côté, nous avons observé que le remplacement LRU gère mieux les données du cache que le remplacement MRU.

L'emploi du cache sémantique au niveau du mobile ainsi que son voisin a prouvé de la performance au sein des réseaux sans fils. En revanche, l'algorithme traite seulement des requêtes avec des prédications simples, en plus si le voisin se trouve très loin du client ou dans le cas où les données enregistrées au niveau du cache du voisin n'ont pas de ressemblance avec celles qui existent au niveau du client, l'envoi de requêtes au voisin est inutile car la requête ne va pas trouver de résultat dans le cache du voisin. Cela nécessitera une prise en considération de l'emplacement du voisin ainsi que les données de son cache.

Pour les travaux futurs, nous pensons qu'il serait intéressant d'étudier les possibilités de partitionnement des différents types de requêtes au niveau du cache, par exemple, en se basant sur la localisation du client et d'employer d'autres méthodes de remplacement de données dans le cache.

RÉFÉRENCES

- [1] Zhankun, T. (2012). A cache based scheme of data management in mobile devices. Computer Science and Information Processing (CSIP), International Conference on.
- [2] Dar, S., et al. (1996). Semantic Data Caching and Replacement. Proceedings of the 22th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc.: 330-341.
- [3] Abbas, M. A., et al. (2011). Graph based query trimming of conjunctive queries in semantic caching. Emerging Technologies (ICET), 7th International Conference on.
- [4] Chan, B. Y., et al. (1998). Cache management for mobile databases: design and evaluation. Data Engineering, Proceedings, 14th International Conference on.
- [5] Qun, R., et al. (2003). Semantic caching and query processing. Knowledge and Data Engineering, IEEE Transactions on 15(1): 192-210.
- [6] Boris Y. Chan, Antonio Si, and Hong V. Leong. (2001). A Framework for Cache Management for Mobile Databases: Design and Evaluation. *Distrib. Parallel Databases* 10, 23-57.
- [7] Bashir, M.F.; Qadir, M.A. (2006). HiSIS: 4-Level Hierarchical Semantic Indexing for Efficient Content Matching over Semantic Cache, Multitopic Conference. INMIC '06. IEEE , vol., no., pp.211,214, 23-24
- [8] Sumalatha, M. R.; Vaidehi, V.; Kannan, A.; Rajasekar, M.; Karthigaiselvan, M. (2007). Dynamic Rule Set Mapping Strategy for the Design of Effective Semantic Cache, Advanced Communication Technology, The 9th International Conference on , vol.3, no., pp.1952,1957, 12-14
- [9] Ezziyyani, M.; Bennouna, M.; Essaaidi, M. (2005). A Quantitative Study to Evaluate the Performance of the Semantic Cache Management Methods," Computer as a Tool. EUROCON 2005. The International Conference on , vol.1, no., pp.697,700, 21-24
- [10] Abbas, M.A.; Qadir, M.A. (2009). Cross attribute knowledge: A missing concept in semantic cache query processing," Multitopic Conference, 2009. INMIC 2009. IEEE 13th International , vol., no., pp.1,6, 14-15 Dec. 2009
- [11] Ahmad, M.; Qadir, M.A.; Sanaullah, M. (2008). Query processing over relational databases with semantic cache: A survey," Multitopic Conference. INMIC 2008. IEEE International , vol., no., pp.558,564, 23-24

- [12] Ali, T.; Qadir, M.A.; Ahmad, M. (2010). Translation of relational queries into Description Logic for semantic cache query processing," Information and Emerging Technologies (ICIET), 2010 International Conference on , vol., no., pp.1,6, 14-16
- [13] Sumalatha, M. R.; Vaidehi, V.; Kannan, A.; Rajasekar, M.; Karthigaiselvan, M. (2007). Hash Mapping Strategy for Improving Retrieval Effectiveness in Semantic Cache System," Signal Processing, Communications and Networking. ICSCN '07. International Conference on , vol., no., pp.233,237, 22-24
- [15] Ahmad, M.; Qadir, M.A.; Sanaullah, M.; Bashir, M.F. (2009). An efficient query matching algorithm for relational data semantic cache," Computer, Control and Communication. IC4 2009. 2nd International Conference on , vol., no., pp.1,6, 17-18
- [16] Ni Qing -jun. (2010). Answering semantic caches in integration systems, Signal Processing Systems (ICSPS), 2010 2nd International Conference on , vol.3, no., pp.V3-461,V3-463, 5-7
- [17] Li Xiaoyong; Zhang Hui. (2010). Answering semantic caching for integration systems," Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on , vol.5, no., pp.V5-59,V5-61, 20-22
- [18] Lubbe, C.; Brodt, A.; Cipriani, N.; Großmann, M.; Mitschang, B., "DiSCO: A Distributed Semantic Cache Overlay for Location-Based Services," Mobile Data Management (MDM), 2011 12th IEEE International Conference on , vol.1, no., pp.17,26, 6-9 June 2011
- [19] Zhichao Li; Piliang He; Ming Lei. (2005). Research of semantic caching for LDQ in mobile network, e-Business Engineering, 2005. ICEBE 2005. IEEE International Conference on , vol., no., pp.511,517, 12-18 Oct. 2005
- [20] Ishikawa,Y.;Kitagawa,H. (1999). A semantic caching method based on linear constraints, Database Applications in Non-Traditional Environments. (DANTE '99) Proceedings.
- [21] Soundararajan,G.; Amza, C. (2005). Using semantic information to improve transparent query caching for dynamic content Web sites, Data Engineering Issues in E-Commerce, Proceedings. International Workshop on , vol., no., pp.132,138
- [22] Ren, Q. and M. H. Dunham. (2000). Using semantic caching to manage location dependent data in mobile computing. Proceedings of the 6th annual international conference on Mobile computing and networking. Boston, Massachusetts, USA ,ACM: 210-221.
- [23] Ahmad, M., et al. (2008). Query processing over relational databases with semantic cache: A survey. Multitopic Conference. IEEE International
- [24] Ni, Q.-j. (2010). Answering semantic caches in integration systems. Signal Processing Systems (ICSPS), 2nd International Conference on.

- [25] Deshpande, P. M., et al. (1998). Caching multidimensional queries using chunks. Proceedings of the 1998 ACM SIGMOD international conference on Management of data. Seattle, Washington, USA, ACM : 259-270
- [26] P. Godfrey and J. Gryz. (1997). Semantic Query Caching in Heterogeneous Databases, In Proceedings KRDB at VLDB'97
- [27] Q. and Dunham, M. H. (1999). Using clustering for effective management of a semantic cache in mobile computing. In Banerjee, *Proceedings of the 1st ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 94-101, New York, NY, USA. ACM Press.
- [28] W.-C. Lee and D. L. Lee. (1999). Signature caching techniques for information filtering in mobile environments. *Wireless Net-works*, 5(1): 57-67
- [29] Project Hosting for Open Source Software. (2012). Chinook Database Récupéré de <http://chinookdatabase.codeplex.com/>
- [30] GITBUB. (2013) json-framework récupéré de <https://github.com/stig/json-framework/>