République Algérienne Démocratique et Populaire
Ministère De L'Enseignement Superieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene
Faculté d'Electronique et d'Informatique

MÉMOIRE

Présenté pour l'obtention du diplôme de Magister en :

INFORMATIQUE

SPÉCIALITÉ : Intelligence Artificielle et Base de Données Avancées

Par
Mr Abderrahmane REMACHE

Thème :

OPTIMISATION ADAPTATIVE DES REQUETES SPATIO-TEMPORELLES CONTINUES

Soutenu publiquement le : 01 / 07 / 2008, devant le jury suivant :

Mme. A. AISSANI	Professeur, USTHB	Présidente
Mme. Z. ALIMAZIGHI	Professeur à USTHB	Directrice de Thèse
Mr. A. BELKHIR	Professeur, USTHB	Examinateur
Mr. A. BOUKRA	Docteur, USTHB	Examinateur
Mme. N. ABDAT	Chargé de Cours, USTHB	Invitée

PLAN

3. Modélisation des Données Spatio-Temporelles

4. Stockage & Indexation
    4.1. Indexation de données multidimensionnelles
    4.2. Indexation par R-Tree
    4.3. Indexation par QuadTree

5. Requêtes Spatio-Temporelles
    5.1. Introduction
    5.2. Classification
    5.3. Opérateurs Spatio-Temporels
    5.4. Requêtes Spatio-Temporelles Continues
    5.5. Conclusion

IV. Optimisation de Requêtes

1. Introduction

2. Revue de l'optimisation
    2.1. Plan d'évaluation
    2.2. Evaluation pipelinée
    2.3. Méthodes d'accès
    2.4. Optimisateur du System R

3. Les équivalences algébriques
    3.1. Sélections
    3.2. Projections
    3.3. Produit cartésien et jointures
    3.4. Sélections, Projections et Jointures
    3.5. Autres équivalences

4. Plans Alternatifs
    4.1. Poussée des sélections
    4.2. Utilisation des index

5. L'estimation du coût d'un plan
    5.1. La sélectivité
    5.2. Les histogrammes

6. Enumération des Plans Alternatifs
    6.1. Requêtes à une seule relation
    6.2. Requêtes à multiples relations

7. Les Sous-Requêtes Emboitées

8. Autres approches de l'optimisation

9. Optimisation des Requêtes Continues
    9.1. Introduction
    9.2. Opérateurs Unaires
    9.3. Opérateurs Binaires : Barrières de Synchronisation
    9.4. Opérateurs binaires : Moments de Symétrie
    9.5. Réordonnabilité des Algorithmes de Jointure
    9.6. Conculsion

V. Histogrammes Spatio-Temporels

1. Estimation de la Sélectvité Spatio-Temporelle

2. Principe et Architecture

3. Exécuteur de Requête

4. Gestionnaire d'Histogramme
    4.1. Construction de l'Histogramme
    4.2. Affinage de l'histogramme

5. Extensions des Histogrammes ST

6. Conclusion

VI. Optimisation Adaptative des Requêtes ST Continues

1. Motivations
    - 1.1. Défi temporel
    - 1.2. Défi spatial
    - 1.3. Défi spatio-temporel

2. Optimisation Spatio-Temporelle Adaptative
    - 2.1. Proposition
    - 2.2. Etapes d'optimisation
    - 2.3. Implémentation

VII. Défis

1. Optimisation des Requêtes Mutiples
2. Opérateurs Pipelinés

VIII. Conclusion

IX. Bibliographie

Introduction

1. Contexte et Motivations

Avec la prolifération des dispositifs mobiles et des réseaux sans fil, les services dépendants de la position suscitent aujourd'hui un vif intérêt. Ces systèmes transmettent à une entité centrale des informations précises sur la position, la direction et la vitesse des objets mobiles, tels que le GPS (*Global Positioning System*). Actuellement, on peut trouver ces derniers non seulement dans les avions, les bateaux, et les voitures, mais aussi dans certains téléphones mobiles et PDAs équipés d'appareils de détection de position.

De tels systèmes peuvent être utilisés pour supporter des services dépendants de la position permettant, par exemple, aux gens d'éviter les embouteillages, aux parents de s'assurer que leurs enfants sont sains et saufs, de fournir un service d'assistance, et d'informer les voyageurs sur les établissements qui leurs sont proches tout en se déplaçant.

Cette communication en temps réel avec des objets se déplaçant librement a rendu possible - et donc souhaitable - le développement de systèmes d'information gérant des données "mobiles" impliquant simultanément le domaine spatial et le domaine temporel, appelées « données spatio-temporelles ».

Ces SGBD-ST ou plutôt Systèmes de Gestion de Flux de Données Spatio-Temporelles (SGFD-ST) doivent être conçus afin de capturer un nombre massif d'objets mouvants dépendants de la position. Leurs applications ont deux caractéristiques principales qui les distinguent par rapport aux applications de bases de données conventionnelles :

1. Un environnement hautement dynamique où les données provenant des objets mobiles (ex. véhicules mouvants sur des réseaux de route) sont reçues continuellement.

2. Les requêtes dans ces applications spatio-temporelles sont pour la plupart continues (ex. requêtes de monitoring). Ce type de requêtes nécessite une évaluation continue de l'aire de la requête et/ou des données dont elle est sujette, et qui sont constamment en mouvement.

Par conséquent, les applications spatio-temporelles nécessitent de nouvelles techniques de traitement et d'optimisation des requêtes afin de gérer autant le domaine spatial que temporel.

2. Objectifs

L'objectif principal de ce mémoire est d'étudier en détail l'environnement entourant l'optimisation des requêtes spatio-temporelles continues et d'essayer de fournir un cadre de travail pour une implémentation d'une méthode plus ou moins générale pour cette optimisation.

Cette modeste contribution se veut être un tremplin pour un développement plus poussé d'une solution globale de traitement et d'optimisation des requêtes spatio-temporelles.

3. Plan

Le document est divisé en 6 grandes parties : la première évoque de manière générale le concept de mobilité des données, les systèmes mobiles et les systèmes de bases de données mobiles (SBDM), les architectures utilisées pour l'accès aux données, le traitement des requêtes dans ces SBDM et conclut avec un parallèle entre les bases de données mobiles et celles spatio-temporelles.

Dans la deuxième section, on expose les caractéristiques des bases de données spatio-temporelles ; les données spatio-temporelles, leur modélisation, leur stockage et indexation, ainsi que les requêtes spatio-temporelles et plus précisément celles continues et leurs différentes approches de traitement.

La troisième section aborde l'optimisation des requêtes de manière générale en se basant sur celles des SGBD Relationnels. Elle passe en revue tous les aspects de l'optimisation des requêtes, et conclut avec ceux des requêtes continues.

Dans la quatrième partie, on introduit le concept de l'estimation de la sélectivité spatio-temporelle. On présente une des approches majeures utilisées pour cette estimation à savoir : les histogrammes spatio-temporels.

Ce concept nous permettra d'exposer, dans la section suivante, notre contribution, à savoir : l'optimisation adaptative des requêtes spatio-temporelles continues. On avance d'abord les motivations qui nous ont poussé à développer ce sujet. On présente ensuite notre solution : la proposition, les hypothèses, les étapes d'optimisation et enfin une implémentation générale.

Avant de conclure, on énumère quelques points qui nous semblent n'avoir pas été suffisamment abordés et développés dans la littérature actuelle malgré leur influence essentielle sur le traitement et l'optimisation des requêtes spatio-temporelles continues.

Données et Mobilité

1. Les Systèmes Mobiles :

Un système dit mobile possède un ensemble de stations fixes, avec un emplacement connu et statique, et un ensemble de stations mobiles, ne possédant par conséquent d'emplacement exact. La communication entre ces deux types de stations est traitée à travers l'utilisation de communications sans fil.

Dans ces systèmes, l'utilisateur n'a pas besoin d'être toujours connecté au reste réseau, en d'autres termes, on autorise que l'utilisateur soit déconnecté pour certaines périodes de temps, en opérant dans ce cas seulement sur les données entreposées localement dans sa station. Les déconnexions peuvent être volontaires afin de réduire les coûts de communication et préserver les ressources, ou involontaires à chaque fois que se produit une panne dans le système. De plus, qu'il soit connecté ou pas, l'utilisateur peut se déplacer durant l'exécution de ses tâches.

Les unités mobiles possèdent généralement des ressources nettement inférieures à celles des stations fixes, car en augmentant la portabilité et la mobilité de ces unités, elles perdent de leurs capacités de stockage et de traitement de données, aussi bien que de l'autonomie de batterie. D'un autre coté, vu les communications utilisées, on peut facilement identifier les autres limitations possibles, tel que largeur de la bande limitée, niveau de confiance et de sécurité réduits et coûts de communication plus élevés. Les stations fixes, quand à elles, sont éventuellement de puissantes plates-formes communiquant entre eux à travers des communications filaires. Ces dernières présentent, comme on le sait, un plus grand niveau d'efficacité et de sécurité que les communications sans fils.

On donc peut conclure que dans un système mobile, une grande hétérogénéité existe au niveau des ressources disponibles dans les différentes stations.

2. Systèmes de Bases de Données Mobiles

Tel que l'indique son nom, un Système de Base de Données Mobile (SBDM) est un cas particulier d'un système mobile. Ces SBDMs sont une extension des Systèmes de Base de Données Distribuée (SBDD), puisqu'ils gèrent un ensemble de stations fixes avec des caractéristiques très identiques à celles des SBDDs. En revanche, les SBDMs contiennent des stations mobiles avec certaines caractéristiques non supportées habituellement par les SBDDs. Dans la partie fixe d'un SBDM, des stations servent de support aux utilisateurs mobiles, pouvant même fonctionner comme serveurs de certaines données.

Plusieurs auteurs ont défini l'architecture des SBDMs [46] [53] [54] [55]. Pourtant, ils convergent tous et sans surprise, vers un même modèle (Figure 1), dans lequel deux types de stations coexistent, fixes et mobiles. Les stations ou unités mobiles (UM) peuvent se déplacer, changeant fréquemment d'emplacement même pendant le traitement d'une tâche, pendant que les stations fixes possèdent un emplacement statique et connu. Certains d'entre-elles sont désignées par Stations de Support Mobile (SSM) ou Station de Base (*Mobile Support Stations* ou *Base Stations*). Ces dernières stations assurent la communication entre les stations mobiles et le réseau filaire fixe.

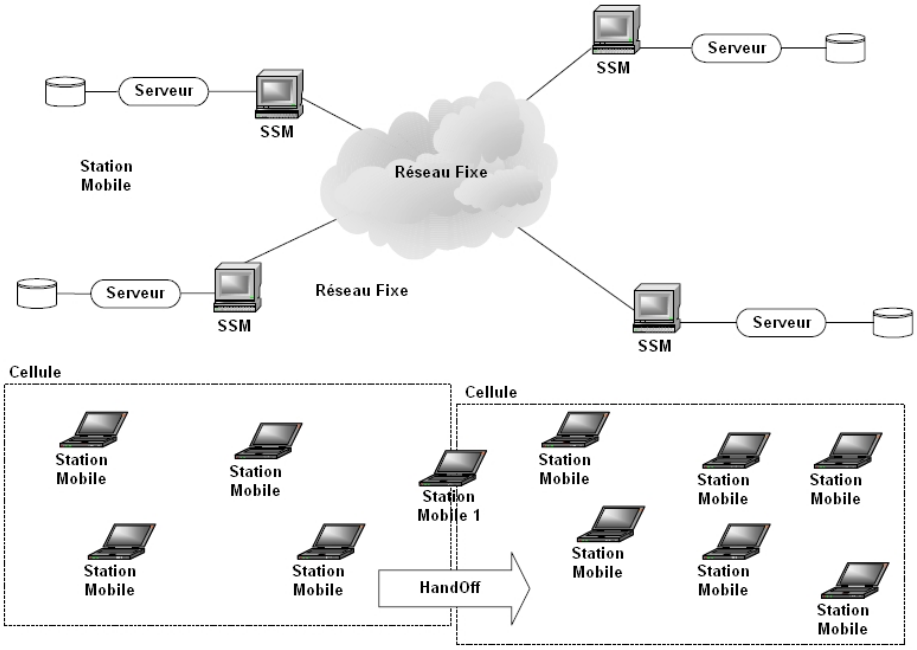


Figure 1 : Exemple d'une Architecture d'un Système de Base de Données Mobile.

A chaque SSM est attribué un secteur géographique limité désigné habituellement par cellule. Ce type de stations se charge de la gestion de toutes les UMs qui se trouvent à l'intérieur de sa cellule à un instant donné. Les cellules sont définies de manière à définir des ensembles disjoints, afin qu'une UM peut se trouver dans une seule cellule à chaque instant. La communication dans une cellule peut être réalisée à travers une connexion cellulaire, comme d'une connexion satellite ou d'un réseau local sans fils (*Wireless Local Area Network*). Les stations localisées dans des cellules distinctes peuvent communiquer entre elles à travers leurs SSMs respectives. Ainsi, une SSM agit comme l'interface entre les stations mobiles et les stations fixes et comme l'entité responsable de l'échange de messages et de données entre ces stations.

Etant donné la grande mobilité des stations actives, il peut arriver qu'une station dépasse les frontières d'une cellule pour entrer dans une autre (Figure 1, Station Mobile 1). Dans cette situation, la tâche d'acheminer les données entre l'UM et le réseau doit être transférée à la SSM de la nouvelle cellule. Ce processus est appelé *Handoff*.

Pour s'assurer que l'utilisateur ne se rend pas compte du changement de cellule, il est souhaitable que le processus de *Handoff* soit transparent et que la connexion soit maintenue, même pendant l'instant de changement. Pour cela, il est nécessaire de procéder à une reconfiguration de la topologie du réseau. D'un autre coté, quand l'UM exécute une tâche avec la coopération de la partie fixe du système, il peut être nécessaire que la tâche en cours soit transférée sur la SSM de la nouvelle cellule.

2.1. Caractéristiques de l'Environnement de Calcul d'un SBDM

2.2.1. Communication Sans Fils

Les environnements supportant ce genre de communications sont proies à de nombreux types de signaux et de bruits qui interfèrent sur ses opérations. Quelques-unes de ces caractéristiques et limitations sont :

- **Largeur de bande réduite**. La largeur de bande d'une cellule est une ressource limitée, du moment qu'elle est, à chaque instant, divisée par le nombre d'utilisateurs

qui s'y trouvent. Ainsi, dans le but de réduire la largeur de la bande utilisée et optimiser son attribution, doivent être utilisées autant que possible, des techniques de compression, comme la filtration et le buffering, avant toute transmission.

- **Grande variation de la largeur de la bande**. La disponibilité de la largeur de la bande offerte à un utilisateur peut souffrir de grandes variations sur le temps, suivant la position où il se trouve. Ceci est dû au fait que la largeur de la bande disponible diffère d'une cellule à une autre, ou même à l'intérieur d'une cellule, car influencée par le nombre d'utilisateurs qui s'y trouvent. Dans cette optique, un utilisateur mobile peut voir diminuer sa largeur de bande, simplement par l'arrivée de nouveaux utilisateurs à la cellule ou par le changement de cellule.

- **Déconnexions fréquentes**. Dans les SBDMs, on autorise que certaines stations soient déconnectées, volontairement ou pas. Les déconnexions involontaires peuvent être causées par des variations de signaux comme les variations de la largeur de la bande disponible, ou par des erreurs sur le système. Quand ce type de déconnexions se produit, on ne peut qu'attendre le rétablissement de la communication quand les conditions de l'environnement l'autoriseront. Les déconnexions volontaires, quant à elles, se produisent par décision de l'utilisateur afin de réduire les coûts de communication ou d'économiser l'énergie des stations.

- **Les cellules supportent la diffusion**. Une SSM doit posséder des infrastructures lui permettant la diffusion de messages à toutes les stations qui se trouvent dans sa cellule. De cette façon, on minimise la communication de l'UM avec la SSM, car elle est plus chère que la communication dans le sens inverse (de la SSM à l'UM). La SSM doit avoir aussi la possibilité de spécifier la station à laquelle un message est destiné.

Il convient alors, d'améliorer les deux types de communication possible dans les SBDMs : de l'UM à la SSM, désigné d'*uplink*, et celle vers l'UM, désigné de *downlink*. Dans le premier cas, la communication est assez chère à cause des ressources faibles de l'UM, il devient par conséquent recommandé d'utiliser autant que possible la communication du type *downlink*.

Certaines des caractéristiques mentionnées au-dessus sont traitées par les SBDDs comme des anomalies ou des exceptions, comme c'est le cas des variations de la largeur de la bande et

des déconnexions. Toutefois, les SBDMs doivent prendre en compte ces caractéristiques et ne pas les traiter comme des anomalies, ce qui impliquerait par conséquent, un grand nombre de retraitements de tâches et, subséquemment, une diminution du travail utile.

2.2.2. Mobilité

La mobilité fait référence à la capacité des utilisateurs à changer d'emplacement, sans perdre la connexion avec le reste du système. La mobilité est indubitablement le facteur fondamental dans les SBDMs. Néanmoins, cette mobilité peut augmenter la volatilité de certaines informations puisque les données considérées comme statiques deviennent dynamiques pour les stations mobiles. De plus, la mobilité cause :

- **Migration d'adresse**. Cette situation est due au fait que l'adresse réseau d'une station mobile peut se modifier dynamiquement, et pourrait même passer pour une nouvelle cellule.

- **Information dépendante de la position**. Il existe certaines informations dont les paramètres de configuration sont influencés par l'emplacement actuel, modifiant ainsi le résultat de certaines requêtes et transactions.

- **Réseau dynamique**. Les réseaux ne sont plus statiques avec des liaisons fixes et connues. Dans les environnements mobiles, les modifications sont fréquentes tant au niveau des liaisons qu'au niveau de leurs intervenants.

2.2.3. Portabilité

La portabilité, qui correspond à la facilité de transport et d'autonomie d'une station mobile, est un des autres aspects que l'on doit considérer dans la construction d'un SBDM. On peut dire que les stations mobiles sont autant plus portables que leurs tailles et poids sont petits.

Cependant, fonctionner avec des stations de plus en plus légères et petites peut causer quelques limitations, comme :

- **Temps de “vie” limité**. La batterie influence beaucoup le poids des UMs. La réduction de son poids est donc un aspect important pour l'augmentation de la portabilité. Cependant, une grande réduction du poids de la batterie peut être traduite par une perte de portabilité et non pas une augmentation, car impliquerait des recharges plus fréquentes.

- **Interface utilisateur limitée**. La réduction des tailles du clavier et du moniteur peut réduire le poids et la dimension des stations. Néanmoins, quand ils sont réduits dans excès, ils peuvent compromettre la portabilité, du moment qu’il y a une réduction de la quantité d'information présentée à l'utilisateur, et par conséquent la visualisation de toute l'information sera plus lente.

- **Capacité de stockage limitée**. Une réduction de la taille des UMs peut causer une baisse de leur capacité de stockage primaire et secondaire et réduire ainsi leur autonomie.

La portabilité ne se limite pas seulement au respect de la dimension et du poids des stations, mais aussi à leur autonomie. Il faut alors trouver un compromis entre tous ces aspects. Par exemple, on peut réduire la taille de la batterie jusqu'à obtenir un temps de traitement raisonnable, en utilisant en simultané des technique minimisant la consommation d'énergie, comme d’éteindre les composants individuels quand ils sont au repos (*idle*) ou de concevoir les applications de sorte à solliciter le plus petit nombre de calculs et de communications possibles.

Des caractéristiques susmentionnées, on peut conclure que les stations mobiles sont plus petites et légères que les stations fixes afin qu'elles puissent être transportées plus facilement. Chose qui en conjonction avec les coûts de la technologie, fait que les stations mobiles possèdent moins de ressources que celles fixes, y compris en mémoire, capacité du disque et même la taille du moniteur. En conséquence, il devient évident qu’il existe une grande asymétrie entre les stations mobiles et les fixes.

3. Architecture d'Accès dans les Systèmes Mobiles

Comme dans les SBDDs, les systèmes mobiles possèdent des architectures autorisant aussi bien la communication que l'échange et l'accès d'information entre les divers membres du réseau. Cependant, dans les SBDMs, l'architecture doit aussi permettre aux utilisateurs de posséder une certaine mobilité, de manipuler une information dépendante de la localisation et de se relier à des différents points, acceptant ainsi une reconfiguration dynamique du système.

Une architecture d'un SBDM doit être transparente à l'utilisateur et adaptable dynamiquement aux caractéristiques de l'environnement de calcul. En outre, elle doit aussi :

- Fournir les inter-opérations entre les différentes infrastructures, qu'elles soient fixes ou mobiles.
- Travailler avec des résultantes imprévisibles, autant pour les comportements de l'utilisateur que pour la capacité et la disponibilité du réseau.
- Fournir une scalabilité, en prenant en compte l'espace d'adresse, la qualité du service et le nombre d'utilisateurs intervenants.
- Fournir un accès intégré aux divers services.
- Garantir l'indépendance entre les applications et le réseau.
- Offrir les outils de coopération entre tous les éléments du réseau.

On va présenter quelques modèles utilisés dans les SBDDs adaptés ou étendus pour les SBDMs, ainsi que d'autres nouveaux modèles créés spécifiquement pour supporter les caractéristiques des systèmes mobiles.

3.1. Modèle Client/Serveur :

Plusieurs types d'architectures sont guidés par le modèle Client/Serveur. La variante la plus simple de ce modèle est définie pour un serveur unique accédé par plusieurs clients. Du point de vue de la gestion des données, ce type d'architecture ne diffère pas beaucoup des systèmes centralisés, du moment que les données sont gardées dans une station unique et que seule change la forme d'exécution des transactions.

Néanmoins, plusieurs serveurs peuvent exister, accédés par plusieurs clients. Dans ce cas, deux solutions sont envisageables pour l'accès aux données : chaque client est relié directement au serveur qui possède les données dont il a besoin, ou alors, on autorise que chaque client soit relié juste à un serveur, le rendant responsable de la recherche et la fourniture des données demandées par le client [43].

Le modèle Client/Serveur utilisé dans les SBDDs suppose que la position des stations est statique et bien connue. Pour cette raison, ils ne peuvent être appliqués directement aux SBDMs. Dans [55] est présentée une adaptation possible du modèle Client/Serveur pour les environnements mobiles (Figure 2. a), qui consiste en un système désigné par Client, demandant un service à un composant de l'application, exécuté dans autre système, désigné par Serveur.

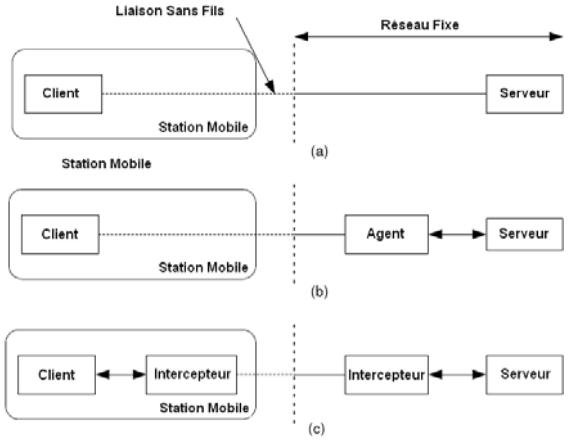


Figure 2 : Modèles Basés sur le Client/Serveur

[55] considèrent que ce modèle possède deux variantes : Client/Agent/Serveur (2. b) et Client/ Intercepteur/Serveur (2. c). Le modèle Client/Agent/Serveur utilise les trois entités: Client, Agent et Serveur. Les Agents se trouvent toujours dans les serveurs et fonctionnent comme intermédiaires entre les clients et le serveur, empêchant qu'un client communique directement avec un serveur. Cette architecture essaye de soulager l'impact de certaines caractéristiques des SBDMs, à savoir la largeur de la bande réduite et la faible fiabilité des moyens de communication sans fils. L'Agent doit inclure au moins un support pour les messages, de manière à rendre possible la communication entre les différentes entités.

[59] considèrent que les agents de l'architecture du Client/Agent/ Serveur fonctionnent du comme des intermédiaires utilisés aussi pour filtrer ou différer les articles de données sur les liaisons lentes. Leur performance dépend énormément de la largeur de bande disponible.

Le modèle Client/Agent/Serveur est plus approprié pour les systèmes avec une énergie et des ressources limitées, du moment qu'il y a transfert de certaines responsabilités du Client pour l'Agent. Malgré les avantages innombrables de ce modèle, il présente un grand inconvénient pour les SBDMs, puisqu'il ne peut soutenir le traitement des Clients pendant les déconnexions. Le Client mobile perd donc la possibilité de continuer l'exécution des ses tâches, quand une déconnexion se produit.

Contrairement au schéma précédent, le modèle Client/Intercepteur/Serveur utilise pour la communication deux Agents Intercepteurs, un du côté Client et un autre du côté Serveur. Cependant vu les limitations de ressources que présentent les UMs, ces dernières peuvent ne pas supporter les Agents Intercepteurs. Un autre inconvénient de ce modèle est que toutes les applications requièrent un traitement autant sur la partie Serveur que sur la partie Client, nécessitant ainsi la création d'une paire d'agents pour chaque instance d'application.

3.2. Modèle Client/Serveur avec Clients *Hoard*:

Une autre variante du modèle Client/Serveur pour les environnements mobiles a été présentée dans [56]. Dans cette approche, les clients peuvent opérer pendant les périodes de déconnexion, après avoir accumulé localement des données dans leurs disques à travers la création de répliques locales. Dans ce modèle les clients sont désignés par Clients *Hoard*.

Durant les déconnexions, les UMs peuvent exécuter des tâches sur les données locales, en réconciliant lors du rétablissement de connexion les modifications effectuées. Les Clients *Hoard* n'ont pas besoin de répliquer les tables de données complètes, ils peuvent en répliquer juste des fragments.

Le serveur est le point de synchronisation de cette architecture. Toutes les actualisations réalisées doivent se refléter dans le serveur, qui de plus contrôle les accès aux données effectués pour les clients. Le serveur, contrairement aux clients, stocke les relations

complètes. Cependant, l'organisation physique de ses données est basée sur des fragments, de sorte que les opérations de *hoarding* et de réintégration soient accomplies de la façon la plus rapide. Les serveurs doivent être conçus soigneusement, afin de capturer correctement la position d'accès des clients *hoard*.

Du côté du serveur, toutes les opérations de réintégration et de *hoarding* sont exécutées sur les fragments. Le serveur n'a pas besoin de garder les données entreposées dans chaque client. Dans ce modèle, les fragments physiques sont des unités de *hoarding*, de réintégration, de contrôle de concurrence et de contrôle d'accès. Chaque client *Hoard* doit informer le serveur sur les fragments dont il projette d'utiliser les données. Malgré que les clients *Hoard* ne stockent pas le fragment complet, du point de vue du serveur les clients stockent le fragment entier. Le serveur a aussi la responsabilité de résoudre tous les conflits qui surgissent.

3.3. Modèle Client/Serveur Etendu :

Afin que le modèle Client/Serveur traditionnel puisse être utilisé dans les systèmes mobiles, [57] proposent un modèle Client/Serveur étendu, où on autorise dans certains cas, que les clients exécutent des fonctions des serveurs et qu'en même temps, les serveurs effectuent les fonctions des clients. Ceci est intéressant dans la mesure où les clients du SBDMs qui possédant, généralement, peu de ressources, exécutent sur le serveur certaines de leurs tâches nécessitant plus de ressources. Cependant, des situations peuvent survenir dans lesquelles les communications sont très faibles ou simplement impossibles. Dans ces cas, les clients peuvent exécuter certains de leurs travaux, devenant ainsi serveur de leurs tâches.

Ce modèle présente quelques variantes. Dans les cas extrêmes, il offre toutes les fonctionnalités du serveur à un client (architecture de clients complets - *Full Client Architecture*) ou il autorise l'implémentation d'architectures qui réduisent au maximum le nombre d'opérations exécutées dans les clients (les architectures de clients pauvres - *Thin Client Architecture*). Entre ces deux cas, une solution intermédiaire existe (architecture Client/Serveur flexible) qui permet au client d'accomplir certaines de ses tâches, avec quelques restrictions, mais sans empêcher le traitement des clients, comme c'est le cas dans les architectures avec les clients pauvres.

3.4. Modèle *Point-à-Point* :

Par contraste avec les architectures Client/Serveur, dans l'architecture Point à point distribuée, il n'y a aucune distinction entre clients et serveurs, en d'autres termes, chaque composant du système peut travailler autant comme client que comme serveur. Ce modèle fournit indépendance de données et transparence sur leur localisation et réplication. De la perspective logique des données, l'architecture Client/Serveur et Point à Point fournissent la même vue du système, puisque les deux offrent l'idée d'une base de données unique, alors qu'en réalité, les données sont physiquement distribuées.

En plus d'être inapproprié pour les clients avec peu de ressources, ce modèle nécessite aussi une connexion continue de l'UM, ce qui n'est pas très pratique dans ces environnements. Il utilise ainsi des mécanismes qui initient automatiquement les applications des stations quand il y a des demandes de clients.

3.5. Modèle Basé Agents Mobiles :

[45] présentent un modèle basé sur des processus appelés Agents Mobiles envoyés par une station pour accompagner une tâche donnée. Dans le fond, ce sont des modules qui encapsulent données et code pour faciliter la coopération dans la résolution de tâches complexes et leur exécution dans des stations distantes avec la plus petite interaction possible de l'utilisateur. Après avoir été soumis, un Agent procède de façon autonome et indépendante du Client et du Serveur. Quand l'Agent Mobile arrive au Serveur, un agent d'exécution est envoyé initiant sa partie exécutable. Quand il achève le calcul, l'Agent envoie le résultat au Client. Les Agents Mobiles peuvent être utilisés conjointement avec des agents localisés dans le réseau fixe.
Certains avantages de ce modèle relié à l'utilisation des agents mobiles, sont :

- **Communication asynchrone**, du moment que les agents peuvent opérer indépendamment, même pendant les déconnexions des UMs.
- **Communication autonome**, car les agents peuvent prendre de manière autonome certaines décisions pour l'utilisateur comme de relancer l'exécution d'une transaction qui a échoué.

- **Communication distante**, car les agents exploitent à distance avec les serveurs.

Deux autres raisons justifiant l'utilisation des agents mobiles, présentées dans [58], résident dans fait qu'ils fournissent une recherche de services d'information efficace, asynchrone et flexible, et qu'ils supportent la connectivité intermittente, les réseaux lents ou les composants réseau lourds.

Pour ces raisons, l'usage des agents mobiles dans les SBDMs devient très attractif, vu que dans ces systèmes les ressources sont habituellement rares et les composants un peu lourd. Les agents pourraient contribuer à libérer les stations mobiles des éventuels *overheads*.

4. Traitement des Requêtes dans les SBDMs

L'extraction d'information dans les systèmes de base de données traditionnels est réalisée à travers le traitement des requêtes. Dans ce type de traitement, une interrogation - requête -, habituellement écrite dans un langage de haut niveau, est transformée dans une autre interrogation équivalente, écrite dans un langage de niveau inférieur. Cette nouvelle interrogation plus efficiente doit traduire correctement le sujet initial, de sorte à obtenir l'information désirée. De plus, les méthodes de transformation des requêtes doivent être rapide et efficace dans le souci de minimiser la consommation des ressources système. Ces méthodes sont désignées, habituellement, par méthodes d'optimisation des requêtes.

Dans les systèmes de base de données traditionnels, ces méthodes pourraient être partagées en quatre phases principales :

- **Décomposition**. La phase de la décomposition inclut les tâches d'analyse grammaticale (*parsing*) et de compilation de l'interrogation. Le résultat de cette phase est une expression en algèbre relationnel équivalente à l'interrogation initiale.

- **Optimisation**. Pendant cette phase, le plan d'exécution de l'interrogation est créé.

- **Génération du code**. Ici, le code exécutable est produit à partir du plan précédent.

- **Exécution de la requête**. Pendant cette phase, le code produit dans la phase antérieure est exécuté et on obtient le résultat final de l'interrogation.

Dans les systèmes distribués, le traitement des requêtes et leur optimisation conséquente sont des tâches nettement plus complexes, puisqu'un plus grand nombre de paramètres affecte l'exécution et l'efficacité d'une requête. Comme on le sait, dans les systèmes distribués, les données sont fréquemment fragmentées et répliquées, ce qui pourrait causer une augmentation du nombre de communications entre les différentes stations pour accomplir avec succès la requête.

La phase d'optimisation des requêtes pour les SBDDs est subdivisée en quatre étapes différentes :

- **Décomposition de la requête**, en préservant les relations globales.
- **Localisation des données**, en utilisant l'information sur la distribution des données.
- **Optimisation globale**. Afin de trouver la forme d'exécution la plus efficiente pour la requête.
- **Optimisation locale**. Cette dernière étape est exécutée par toutes les stations possédant les données nécessaires à l'exécution de la requête. Chacun de ces stations exécute une sous requête désignée par requête locale.

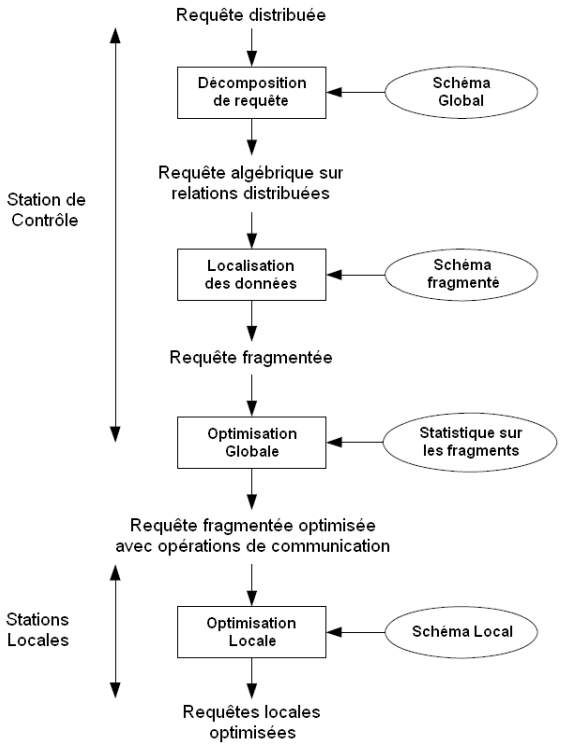


Figure 3 : Etapes d'optimisation des requêtes dans les SBDDs

Le traitement des requêtes devient encore plus complexe dans les SBDMs, du fait de la distribution et la réplication de données et des stations pouvant se déplacer ou se trouver temporairement indisponible. Ce mouvement peut influencer le résultat de certaines requêtes. Par exemple, si on veut savoir quelles sont les pharmacies disponibles, la réponse à ce sujet

dépend, naturellement, de l'emplacement de l'utilisateur, ou encore de l'heure à laquelle le sujet a été posé.

Le traitement des requêtes dans les SBDDs, comme reporté, peut être divisé en plusieurs phases. La phase d'optimisation des requêtes se base essentiellement sur des fonctions de coûts et les accès disque et mémoire afin de choisir l'exécution la moins coûteuse. Cependant, dans les SBDMs, ces critères ne suffisent plus pour calculer les coûts associés au traitement d'une requête. Doivent alors, pris en compte les aspects particuliers des environnements mobiles :

- **Les ressources limitées des stations mobiles**, énergie, largeur de bande, entre autres.
- **Localisation des stations mobiles**, du moment qu'il peut être nécessaire de connaître l'emplacement des stations qui ont produit la requête ou des stations qui possèdent des informations indispensables pour l'obtention des résultats.

Afin d'obtenir un traitement efficace de requêtes dans un SBDM, on doit prendre en compte les aspects suivants :

- Incorporation de l'acquisition des données dans le traitement des requêtes.

- Existence de nouveaux types de requêtes :
    - Dépendant de l'emplacement de l'utilisateur.
    - Dépendant de la direction du déplacement de l'utilisateur.
    - Requêtes jointes, comme par exemple les requêtes qui mesurent la circulation globale d'une certaine zone pour aider dans la tâche d'allocation de ressources.

- Exécution des requêtes sur des données temporaires, du moment que les modifications d'information sont presque constantes dans ce type d'environnements. L'information pourrait même se modifier durant le traitement d'une requête.

- Traitement d'une grande quantité de requêtes sur un certain ensemble de données, et que d'autres données ne soient presque sans traitement.

4.1. Optimisation des Requêtes :

Le traitement de certaines requêtes dans les SBDMs peut exiger la localisation de l'emplacement de certaines stations. Plusieurs stratégies peuvent être utilisées pour la localisation de stations. Cependant cette recherche implique des coûts supplémentaires qui doivent être considérés dans la phase d'optimisation des requêtes. En outre, quand il est fait la référence à une station donnée, on vise soit à accéder : (1) à sa localisation; ou (2) aux données entreposées localement. Reste qu'il est seulement possible d'accéder aux données entreposées dans une station mobile dont on connaît concrètement la position.

Dans [44], un plan d'optimisation des requêtes est présenté qui, en plus des coûts de localisation et des facteurs internes (comme la structure des données et metadonnées), prend en compte les facteurs externes à la base de données pouvant influencer les performances de traitement du système. Parmi ces facteurs, on peut souligner : le hardware comme le temps d'accès au disque et les limitations de mémoire et d'énergie, l'architecture comme la distribution et la réplication des données, les communications sans fils, la mobilité : aussi bien des stations que des données, et enfin la préférence d'utilisateurs comme le fait de donner la priorité aux requêtes des utilisateurs mobiles, afin que ces derniers obtiennent des performances identiques à celles des utilisateurs fixes. Un autre point important de l'optimisation est celui de déterminer la station devant exécuter la requête. Ainsi, pendant le processus d'optimisation, il est essentiel de déterminer quelles stations exécuteront chacune des phases de traitement des requêtes. Cette décision doit être flexible et révisable autant que nécessaire, vu les caractéristiques de ces environnements.

Le traitement des requêtes mobiles est divisé en trois phases: traduction, optimisation et exécution, avec la possibilité que chacune de ces phases soit exécutée dans une station différente. Néanmoins, certaines stations sont incapables d'exécuter certaines de ces phases.

Pendant la phase d'optimisation, l'information du système est utilisée pour déterminer le plan considéré comme optimal, qui sera ensuite transmis aux stations exécutant les phases restantes. Pour éviter l'énorme échange de messages entre la station qui traite la phase d'optimisation et celle qui traite la phase d'exécution, ce modèle propose que ces deux phases soient exécutées par la même station.

Dans la Figure 4, sont schématisées trois stratégies possibles pour le traitement d'une requête dans un environnement mobile. Dans le cas 1, le traitement doit se poursuivre en dépit de la rareté des ressources et la requête est envoyée au serveur. Ici, le serveur doit effectuer autant la phase de traduction que les phases d'optimisation et d'exécution, afin que les ressources de la station mobile soient préservées. Quand la station possède suffisamment de ressources pour le traitement, deux stratégies peuvent être utilisées : la phase de traduction peut être initiée dans la station mobile et s'il est conclut que le traitement consommera plus de ressources que la station n'en possède, alors cette phase est abandonnée et la requête est envoyée au serveur qui effectuera aussi les phases d'optimisation et exécution (cas 2). En revanche, si l'analyse de la phase de traduction conclut que la station mobile possède bien les ressources pour continuer le traitement de la requête, alors la station mobile accomplit les phases restantes du traitement (cas 3). La station mobile peut avoir aussi recours au cas 3 quand la station est déconnectée et qu'elle envisage d'exécuter des requêtes.

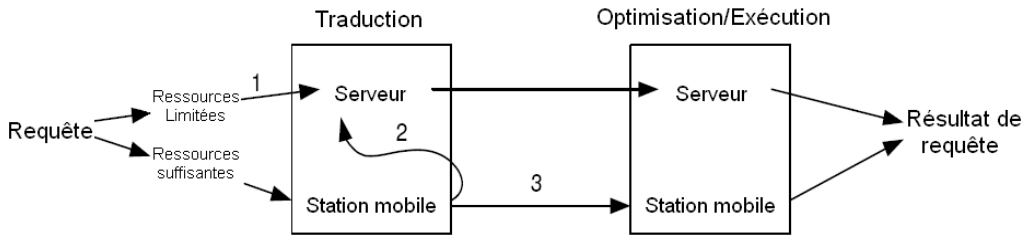


Figure 4 : Stratégies des Traitements Possibles des Requêtes.

L'évaluation des coûts et des ressources exigées pour l'exécution des requêtes, est aussi incluse dans le plan d'optimisation. Les principaux facteurs utilisés sont l'usage du CPU et le nombre d'accès au disque, aussi bien que la consommation d'énergie, mémoire disponible et vitesse du CPU.

5. SGBD-ST et mobilité

Les Systèmes de Gestion de Bases de Données Spatio-Temporelles (SGBD-ST), traités dans le chapitre suivant, peuvent naturellement être des systèmes mobiles et exploiter les architectures d'accès décrites précédemment utilisées dans le cas d'un environnement mobile. On pourrait, par exemple, imaginer un SGBD-ST, où les objets mobiles ou une partie d'eux

sont des clients de la BD, ou même des serveurs répondant à leurs requêtes où à celles des clients, éventuellement, d'autres objets mobiles.

Néanmoins, il faudrait garder en tête que les SGBD-ST fonctionnent dans un environnement hautement dynamique et traitent souvent des flux continues d'information de positions relatives aux objets mobiles. Leurs applications sont le plus souvent en temps-réel. Un objet mobile hébergeant une partie des données de la BD pourrait, par exemple, se trouver déconnecté lors de mises à jour de position d'objets mobiles ou lors de requêtes émises au SGBD-ST impliquant cette partie des données. Ce cas de figure, parmi d'autres relatifs à la mobilité des données, pourrait entraver le bon fonctionnement du SGBD-ST, et affecter sa vocation même.

Ces aspects doivent, alors, être pris en compte lors de l'implémentation d'un SGBD-ST mobile. Si l'environnement de calcul et le type de communication sans fil sont adéquats et que les contraintes imposées au SGBD-ST sont respectées, on pourrait facilement exploiter le concept de mobilité des BD et l'appliquer au SGBD-ST.

Les Bases de Données Spatio-Temporelles

1. Introduction

Les Systèmes de Gestion de Bases de Données (SGBD) actuels ne permettent pas la représentation d'informations évoluant de manière continue. Une des caractéristiques principales des SGBD est que les valeurs des attributs, entre deux mises à jour, sont constantes, et que l'évolution de ces valeurs s'effectue de manière discrète en fonction des modifications explicitement demandées au système. A moins d'effectuer des mises à jour extrêmement fréquentes, on ne peut donc obtenir qu'une représentation approximative de la position d'un objet, et il est encore plus difficile de représenter des positions successives formant une trajectoire complète.

Les applications n'ont pas attendu le développement de technologies propres aux bases de données pour gérer des objets mobiles. Cependant, la croissance rapide du volume d'informations à traiter et la proximité de la problématique avec celles, plus traditionnelles, des bases de données spatiales [11, 12] et des bases de données temporelles laissent à penser que les SGBD tiendront une place importante à l'avenir dans les systèmes dédiés aux objets mobiles. C'est, en tout cas, la conviction de nombreux chercheurs, souvent issus de l'une des deux communautés citées ci-dessus, et qui sont à l'origine d'un nombre croissant de publications proposant soit l'extension de modèles et techniques initialement conçues pour les données spatiales ou temporelles, soit leur intégration.

Ces propositions visent en premier lieu à élaborer des solutions pour fournir aux applications impliquant des objets mobiles, les fonctionnalités principales d'un SGBD, à savoir :

- La capacité à représenter l'information et à associer à cette représentation des opérations qui permettent d'interroger la base ;
- Des structures d'accès aux données qui garantissent des temps de réponse acceptables même en présence de volumes très importants.

Ces aspects traditionnels sont bien représentés dans les SGBD classiques par le modèle relationnel (doté du langage SQL) et les structures de hachage ou d'arbre B qui offrent, toutes, des propriétés optimales en termes d'occupation d'espace et de performance. Alors que ces questions ont été partiellement résolues au cours des dix dernières années pour les données spatiales ou temporelles, elles restent totalement à explorer pour les données *spatio-temporelles*. Il faut aussi noter, à côté de ces motivations classiques, certains aspects particuliers à ce dernier type de données, comme la gestion de l'incertitude de la position exacte d'un objet à un instant donné, qui font l'objet de recherches spécifiques.

Deux approches majeures peuvent être utilisées pour implémenter un serveur de base de données spatio-temporelle :

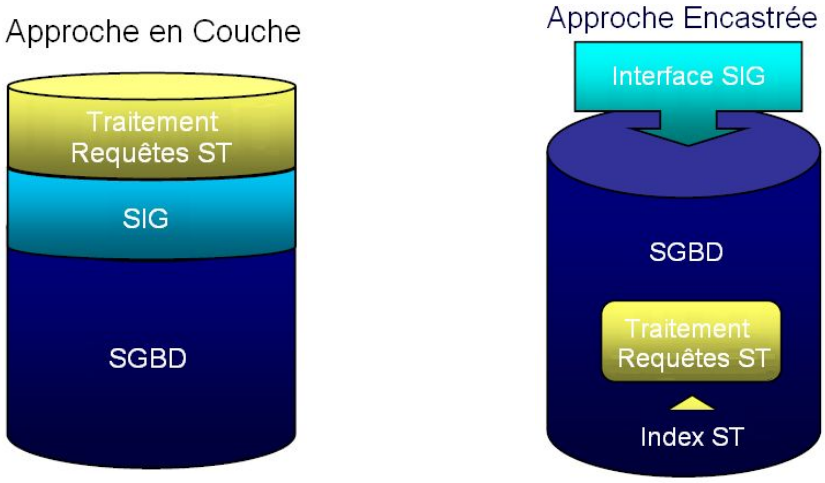


Figure 0 : Serveurs de Base de Données ST

- La première se fait par couche : toutes les données, que se soit les données de position ou celles des attributs sont accédées par le biais du SGBD. Avec des SGBD commerciaux puissants (comme Oracle, Sybase, Informix), les ensembles de données dans ce format peuvent être optimisés pour des opérations ultra rapides de récupération, vision, et mise en requête. Cependant, les fonctions d'analyse des données ST et de traitement avancé des requêtes ST (comme l'optimisation ST spécifique) sont généralement absentes ou limitées avec cette approche, car elles sont reléguées au niveau supérieur.

- Dans la deuxième : une partie des données (généralement les attributs des tables et les relations) est accédée par le SGBD car elle convient bien à son modèle, mais les données

ST sont accédées directement par le Système d'information. Ainsi, dans cette approche encastrée, le traitement des requêtes ST se fait à un niveau plus bas ce qui permet une optimisation plus poussée. Cette approche est plus commune pour les Systèmes d'Information Géographiques (SIG).

Notre travail se veut le plus général possible. On a évité, par conséquent, toute dépendance à un modèle spécifique, que ça soit au niveau de la représentation des données spatiales (raster/vecteur) ou des relations qui associent ces données (topologie). Ainsi, ces volets ne seront pas abordés dans ce qui suit.

2. Données Spatio-Temporelles

Passons maintenant aux problèmes spécifiques soulevés par les données spatio-temporelles. Si on considère qu'un objet spatial est caractérisé, entre autres, par un attribut géométrique décrivant sa forme et sa position, on peut définir un objet spatio-temporel comme un objet spatial dont la forme et/ou la position varient au cours du temps. En fait nous nous limitons (comme la plupart des articles) aux mouvements d'objets dont on ignore la forme (que l'on peut donc assimiler à un point). Cela revient à ignorer les zones à surface variables (incendies, déserts, marées, évolution des paysages).

On peut distinguer deux types d'applications parmi celles qui gèrent des objets mobiles. Les premières s'intéressent principalement à l'analyse des flux d'une population donnée, et considèrent des trajectoires décrivant l'historique, pour chaque objet, de ses déplacements successifs au cours d'un intervalle de temps qui peut être arbitrairement grand. Les requêtes envisageables sur ce type de données sont par exemple :

- Donner la trajectoire de l'objet o entre t_1 et t_2 (requête temporelle) ;
- Donner tous les objets dont la trajectoire passe dans la région R (requête spatiale) ;
- Donner tous les objets dont la trajectoire passe dans la région R entre t_1 et t_2 (requête spatio-temporelle).

Ces requêtes requièrent une connaissance de la trajectoire complète des objets, ce qui suppose soit que l'on interroge en fait le passé (les mouvements ayant été enregistrés et stockés au fur et à mesure), soit que l'on fait des hypothèses sur la trajectoire future des objets.

Le second type d'application est beaucoup plus orienté vers le suivi (*tracking*) des objets en temps réel. On s'intéresse par exemple aux mouvements d'une flotte de taxis, des avions s'approchant d'un aéroport ou aux engins militaires sur un champ de bataille.

Dans de tels cas, on cherche à satisfaire des besoins nouveaux ou à prévoir la situation dans un avenir proche plutôt qu'à faire une analyse a posteriori des événements. Les requêtes typiques de ce genre d'application sont [13] :

- Trouver les taxis les plus proches du point *p* (un client qui attend).
- Trouver les hôtels les plus proches de l'objet *o* (un voyageur qui voit la nuit approcher!).
- Trouver tous les avions qui vont entrer dans la région *R* dans moins de 10 minutes.

C'est donc la position courante des objets qui est primordiale, et éventuellement la position prévisible des objets dans un futur très proche.

Les deux types d'application diffèrent sur quelques points importants. Le premier cas semble plus complexe puisqu'on ne peut se contenter de connaître la position à un instant *t*. Il faut conserver toute la trajectoire, ce qui implique une structure plus complexe et des données plus volumineuses. En revanche, les données étant connues complètement, on se trouve dans une situation assez confortable où la base peut être considérée comme fixe. Ce n'est pas le cas du deuxième type d'application où la position des objets est sans cesse remise en question par des mises à jour successives. Le problème est alors de s'assurer que l'on est capable de prendre en compte le flux des mises à jour qui peut être important, et de garantir que le résultat d'une requête n'est pas déjà obsolète au moment où on le transmet à l'utilisateur.
Dans les deux cas, on trouve un ensemble commun de problèmes spécifiques à la gestion de trajectoires.

2.1. Représentation

Comment représenter une valeur qui varie constamment ? La solution évidente consistant à effectuer des modifications aussi fréquentes que possibles revient à adopter une représentation discrète et est globalement insatisfaisante, aussi bien pour des raisons de précision que de surcharge du système.

Le problème est similaire à celui consistant à représenter des données géographiques linéaires (routes, rivières) : on a à faire à un ensemble infini de points que l'on ne peut représenter de manière finie qu'à l'aide de structures plus ou moins complexes. Le cas des trajectoires introduit un degré de complexité supplémentaire puisque ces données linéaires sont décrites non pas dans un espace à deux dimensions (x et y correspondant à la longitude et à la latitude), mais dans un espace à trois dimensions où la troisième coordonnée, le temps t, joue un rôle essentiel. Il n'y a pas de support dans les SGBD actuels pour ce type d'information.

2.2. Interrogation

Le langage traditionnel d'interrogation est SQL, de plus en plus étendu avec diverses fonctionnalités. De telles extensions ont été proposées pour des données spatiales ou temporelles, mais pas pour des données intégrant étroitement le temps et l'espace.

2.3. Indexation

Les structures traditionnelles comme l'arbre B ne sont pas adaptées aux données multidimensionnelles car elles reposent sur une structure d'ordre qui n'existe pas dans un espace dense. Des solutions ont été définies dans le cas des données spatiales qui permettent de s'approcher d'un temps de recherche logarithmique, mais ces structures (*R-tree, Quadtree*) supposent des données statiques, et ne semblent pas adaptées à des données mobiles.

2.4. Incertitude

La notion d'incertitude est très importante à prendre en compte pour les données spatio-temporelles. La position d'un objet n'est connue qu'avec une précision relativement limitée, et surtout, sa trajectoire est soumise à diverses fluctuations qui tiennent au caractère accidenté du réseau parcouru (route) et aux variations de vitesse. Il est essentiel de tenir compte de ces marges de tolérance quand on cherche à évaluer une requête qui spécifie des critères de recherche précis.

3. Modélisation des Données ST

La motivation principale des modèles de données ST est de définir, pour un objet qui se déplace de manière continue, une représentation sur laquelle on puisse construire un ensemble d'opérations intégrables à un langage d'interrogation.

Quelques idées de base sont communes à tous les modèles :

- En premier lieu, on assimile la trajectoire d'un objet à une courbe dans un espace défini par trois variables représentant le temps t, et l'espace x et y avec l'interprétation habituelle. De plus, on ne considère le plus souvent qu'une approximation linéaire de la courbe, pour des raisons de simplicité de description, et également d'efficacité algorithmique.

- Ensuite, on se place dans un espace dense, disons R^3, ce qui permet d'obtenir une continuité du mouvement. Dans une telle interprétation, la trajectoire d'un objet mobile est constituée d'un ensemble infini de points, ensemble pour lequel on doit définir une représentation finie. Le problème est identique dans les bases de données géographiques: un département, une route, considérés dans un espace dense, forment des ensembles infinis que l'on manipule en les représentant de manière finie par des structures telles que la ligne brisée formant le contour d'un polygone.

Cette approche débouche sur des outils relativement complexes pour l'utilisateur qui doit connaître les structures utilisées, maîtriser la syntaxe des opérations pour chaque structure en particulier, savoir quel est le type du résultat qui lui est fourni, etc. L'extension de cette approche à des données spatio-temporelles semble donc mener à des modèles complexes. Heureusement le caractère très particulier d'une trajectoire d'objet mobile permet de définir deux niveaux de représentation :

- Au niveau abstrait, on peut définir un objet mobile comme une fonction continue du temps vers l'espace à deux dimensions ;
- Au niveau symbolique, on utilise des structures qui permettent une représentation compacte de ces fonctions.

La représentation abstraite est simple à appréhender et permet de définir une interface avec l'utilisateur qui est indépendante de la représentation symbolique choisie. Cette approche fondée sur plusieurs niveaux d'abstraction est classique et associe chaque acteur confronté au SGBD (l'utilisateur, le programmeur, l'administrateur) à un mode de présentation de l'information approprié.

Au niveau le plus haut on doit trouver un formalisme qui respecte la continuité de l'espace et du temps. Le modèle doit, conceptuellement, manipuler des ensembles infinis à l'aide d'un langage de requêtes déclaratif proche de SQL. A un niveau plus bas, proche de l'implémentation, le modèle propose une représentation finie des données afin de les stocker et de les manipuler.

L'exemple de la Figure 1 représente l'approximation linéaire d'une trajectoire. On peut voir, au niveau abstrait, cette trajectoire comme une fonction par morceaux qui, à chaque valeur de t sur l'intervalle [0, 5], associe une paire [x, y]. Une vitesse v relative au segment peut être obtenue à partir de ces données.

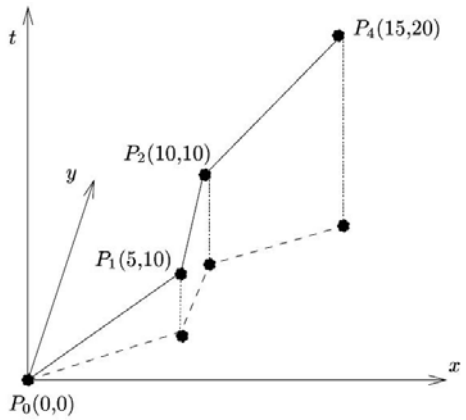


Figure 1 : Une trajectoire

Il existe plusieurs représentations symboliques possibles pour ces fonctions. On peut par exemple stocker les points définissant les segments, et reconstruire ensuite les fonctions par interpolation. La forme générale d'une fonction donnant une coordonnée en fonction du temps est :

$$x(t) = x_0 + v*(t - t_0)$$

On peut donc, la représenter par un triplet $[x_0, t_0, v]$. Noter que la restriction à des données linéaires implique que la vitesse est supposée constante sur un segment. C'est cette forme que nous considérerons dans tout ce qui suit.

4. Stockage et indexation

Cette section est consacrée à un deuxième problème fondamental : l'indexation des objets mobiles. Les deux techniques les plus couramment utilisées pour indexer des données multidimensionnelles, à savoir le *R-tree* et la *Quadtree linéaire*, ont été récemment adaptées aux objets mobiles.

4.1. Indexation de données multidimensionnelles

Il existe une multitude de techniques d'indexation d'objets multidimensionnels [14], dont la plupart sont des variantes de l'une des deux catégories suivantes :

- *Partitionnement du jeu de données en fonction de leur répartition dans l'espace* : Essentiellement, on cherche à regrouper les objets proches dans l'espace dans les mêmes pages du disque. La principale structure de cette première catégorie, et qui sera détaillée plus bas, est le *R-tree*.

- *Découpage régulier de l'espace* : Dans ce cas, on ne tient pas compte de la distribution du jeu de données à indexer. L'espace de référence est découpé à priori en cellules, régulières ou non. Les objets sont ensuite affectés aux cellules avec lesquelles ils ont une intersection. Cette deuxième structure largement utilisée (dans ORACLE, par exemple) sous une variante ou une autre est le *Quadtree*.

4.2. Le *R-Tree*

Le *R-tree*, dont un exemple est donné dans la Figure 2, est construit sur une hiérarchie de rectangles contenus les uns dans les autres, le niveau le plus bas étant constitué des rectangles minimaux englobant la géométrie des objets spatiaux. A chaque niveau, les rectangles sont groupés en "paquets" pouvant être stockés sur une page du disque.

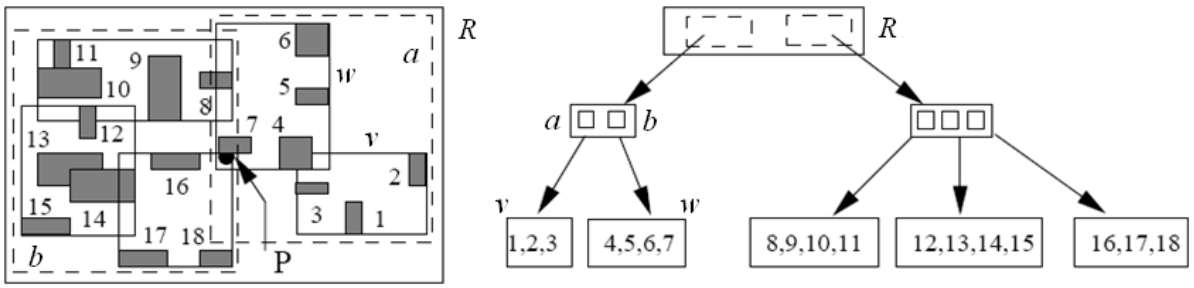


Figure 2 : Indexation par R-Tree

Dans l'exemple de la Figure 2, on suppose qu'une page peut stocker au plus 4 objets, ce qui donne les groupes {1, 2, 3}, {4, 5, 6, 7}, etc. Le regroupement des objets est basé sur leur proximité dans l'espace. On cherche en fait à minimiser le recouvrement des rectangles englobant de chaque groupe (les rectangles dessinés en traits fins sur la figure). Ces rectangles sont à leur tour regroupés pour former le niveau supérieur, et ainsi de suite jusqu'à ce que l'on obtienne un dernier groupe de moins de 4 éléments, stockés sur une page qui forme la racine de l'arbre.

Le *R-tree* a des propriétés comparables à celles de l'arbre-B : l'arbre est équilibré, sa hauteur est logarithmique dans la taille du jeu de données, et sa complexité en espace est linéaire. La recherche (pointé par exemple) basée sur un *R-tree* consiste à parcourir, à chaque niveau (en partant de la racine) le sous-arbre dont le rectangle englobant contient le point argument *P*.

Malheureusement le *R-tree* ne garantit pas un temps de recherche logarithmique. La recherche des objets contenant le point *P* par exemple, devra parcourir 5 pages (la racine, les deux nœuds de niveau 1, et deux feuilles), sans ramener un seul objet. Un autre inconvénient de la structure est le coût de l'algorithme qui divise les objets d'un nœud quand celui-ci est trop plein.

Indexation par R-Tree : le TPR-Tree [17]

Le TPR-tree (*Time-Parameterized R-tree*) est une extension du R-tree pour indexer les positions actuelles et futures d'objets mobiles, et non leurs historiques. L'index s'applique donc à un modèle de données comme celui de [16], et pas à ceux qui permettent d'interroger la trajectoire passée. Chaque objet est décrit par une position de référence $x^i(t_{ref})$, $v^i(t_{ref})$, i variant entre 1 et la dimension de l'espace considéré. La position sur l'axe i à un instant $t > t_{ref}$ est obtenue par $x^i(t) = x^i(t_{ref}) + v^i(t_{ref} * (t - t_{ref}))$.

Il y a également 3 paramètres qui affectent l'indexation et ses performances. Le premier est l'intervalle de temps dans le futur *W* qu'une requête peut interroger, intervalle ayant pour borne inférieure l'instant présent. Le deuxième est l'intervalle de temps *U* pendant lequel on suppose que l'index peut-être utilisé, qui a donc pour borne inférieure l'instant de création de l'index. Enfin, le dernier est l'intervalle *H* qui contient tous les temps spécifiés dans les requêtes ($H = W+U$).

Le temps correspondant à la position de référence d'un objet mobile t_{ref} est choisi égal au temps de chargement de l'index. De plus, la période de validité d'un index doit être fixée, afin d'avoir une indexation pertinente, en fonction des caractéristiques des objets étudiés.

a. Construction

Dans un arbre R, chaque rectangle à n'importe quel niveau de l'arbre englobe un ensemble d'objets. Cette propriété est essentielle pour garantir la possibilité d'effectuer correctement une recherche. L'idée de base de l'index TPR est que les rectangles doivent évoluer avec le temps de manière à préserver cette propriété à tout instant.

La Figure 3 illustre l'intuition. On considère 6 objets mobiles, et on suppose qu'une page peut contenir au plus trois objets. La partie haute représente les positions à l'instant T1, et la partie basse les mêmes objets à l'instant T2 (T2 > T1).

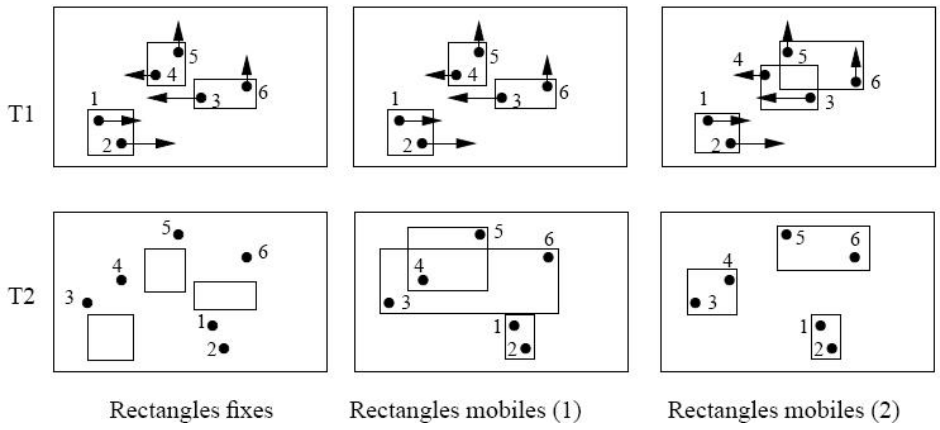


Figure 3 : Rectangles Mobiles dans le *TPR-Tree*

Clairement, un rectangle fixe est obsolète à T2 (partie gauche de la Figure), et ne peut plus servir à la recherche. Si on prend maintenant des rectangles mobiles, le choix des groupes d'objets ne doit plus seulement tenir compte de la proximité dans l'espace, mais aussi de leur position future. Par exemple (partie centrale) grouper les objets d'après leur proximité donne un bon résultat pour le groupe {1, 2}, mais pas pour {4, 5} et {3, 6} puisque les directions sont divergentes au sein de chaque groupe, ce qui aboutit à de très larges rectangles au temps T2. En revanche, un regroupement qui tient compte des deux paramètres (partie droite) donne de bons résultats.

Pour des raisons de coûts élevés de stockage, on ne peut avoir à tout instant le rectangle englobant minimal. Les auteurs proposent un rectangle englobant qui est minimal pour $t = t_{ref}$ et qui croit suivant les vitesses maximales des objets qu'il contient. Ce qui signifie que la

taille d'un rectangle croît toujours, et ne constitue pas le rectangle minimal des objets à tout instant. La Figure 4 donne un exemple de l'évolution du rectangle englobant entre l'instant t_{ref} et un instant $t > t_{ref}$. A l'instant de référence t_{ref}, qui correspond donc à l'instant où est chargé l'index, les différents objets *a*, *b*, *c*, *d*, *e* et *f* et ont une position et une vitesse donnée. On notera $(x_a(ref), y_a(ref))$ et $(v_{a,x}(ref), v_{a,,y}(ref))$ la position et la vitesse de l'objet *a* à un instant t_{ref}. Ainsi, à l'instant $t > t_{ref}$, l'objet *a* a pour position $(x_a(ref) + v_{a,x}(ref).(t - t_{ref}), y_a(ref) + v_{a,y}(ref).(t - t_{ref}))$.

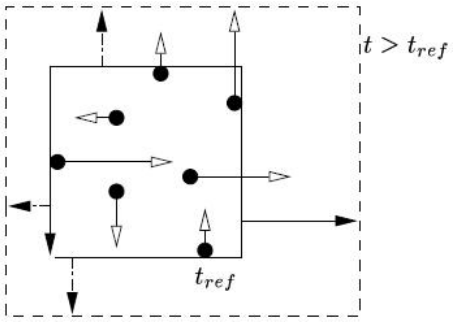


Figure 4 : Croissance d'un Rectangle Mobile

La vitesse de croissance du rectangle englobant est obtenue quant à elle en cherchant les vitesses minimales et maximales en abscisse et ordonnées, en valeurs algébriques, des objets qu'il contient. Ainsi, par exemple pour les abscisses, la vitesse du bord droit du rectangle se déplacera à la vitesse correspondant à la valeur maximale des différentes composantes horizontales des vitesses des objets contenus, soit dans notre cas $v_{a,x}(ref)$, et la vitesse du bord gauche à la valeur minimale, soit ici $v_{b,x}(ref)$. L'exemple de la Figure 4 illustre bien le caractère non minimal du rectangle englobant pour un instant $t > t_{ref}$.

Les algorithmes d'insertion et suppression d'objets dans un R-tree classique visent à minimiser certains paramètres qui conditionnent la qualité de la structure obtenue. Ces paramètres sont la surface occupée par les rectangles, le recouvrement des rectangles, le périmètre (qui indique si le rectangle est plus ou moins carré). Dans le cas du TPR-tree qui utilise des rectangles variant avec le temps, on doit chercher à minimiser les paramètres ci-dessus pour l'ensemble de l'intervalle de temps t_{ref}, $t_{ref} + H$ pendant lequel l'arbre va être utilisé. On va donc utiliser l'intégrale de la fonction donnant la valeur de ce paramètre. Par exemple on cherchera à minimiser l'expression suivant, donnant la surface d'un rectangle au cours de son déplacement :

$$\int_{t_1}^{t_1+H} Aire(t).dt$$

La généralisation des algorithmes du R-tree (en fait d'une de ses variantes, le R*tree [18]) en remplaçant les paramètres habituels par une intégrale est relativement directe. Nous renvoyons à l'article [17] pour plus de détails.

b. Mise-à-jour

Là encore, une des principales limites de la structure est la mise-à-jour de l'index. La solution proposée par les auteurs est différente de la précédente : l'index créé est valable, en théorie, pour un temps infini. Néanmoins, ses performances se détériorent avec le temps. Cet index a un intervalle de validité: il est construit pour une période de temps déterminée, fixée par le paramètre U, où il est optimisé. Au delà de cette période, il est préférable pour l'exactitude des résultats de reconstruire l'index.

c. Recherche

L'index proposé par les auteurs doit permettre de résoudre trois types de requêtes:

1. les requêtes correspondant à un fenêtrage spatial R à un temps t donné.
2. les requêtes correspondant à un fenêtrage spatial lors d'un intervalle de temps [t_{deb}, t_{fin}].
3. les requêtes correspondant à un fenêtrage mobile, valant R_{deb} à t_{deb} et R_{fin} à t_{fin}.

Voyons maintenant comment sont évalués les différents types de requêtes et notamment comment s'effectue la recherche dans le TPR-tree. Dans le cas d'une requête (R, t_0), donc de type 1, la recherche s'effectue de manière assez semblable au R-tree classique: on sélectionne un rectangle englobant $mbb(t)$ si à l'instant t_0 où la requête est évaluée, la fenêtre de la requête R intersecte le rectangle englobant à l'instant t_0 $mbb(t_0)$. Ainsi, en dimension 1 par exemple, si $R = (a_{min}, a_{max})$ et le rectangle englobant est défini par $(x_{min}, x_{max}, v_{min}, v_{max})$, on sélectionne ce rectangle englobant si $a_{min} \leq x_{max} + v_{max}.(t_0 - t_{ref})$ et $a_{max} \geq x_{min} + v_{min}.(t_0 - t_{ref})$.

Pour répondre aux requêtes de types 2 et 3, définies sur un intervalle de temps [t_{deb}, t_{fin}], il faut sélectionner les rectangles englobants qui intersectent la fenêtre de requête entre t_{deb} et t_{fin}. Pour les requêtes de type 3, les auteurs proposent un algorithme qui s'appuie sur l'observation que 2 rectangles mobiles s'intersectent si il y a un instant t, $t_{deb} \leq t \leq t_{fin}$, où leurs projections dans chacune des dimensions s'intersectent.

Ainsi l'algorithme présenté va calculer pour chaque dimension l'intervalle de temps pour lequel les deux rectangles s'intersectent. Si l'intersection de tous ces intervalles est nulle, alors les deux rectangles n'intersectent pas. Sinon, l'intervalle résultat correspond à l'intervalle de temps pendant lequel les deux rectangles intersectent.

d. Résultats observés

Le premier résultat observé par les auteurs est lié à l'importance de l'intervalle *H*. Les meilleurs résultats sont obtenus pour une valeur proche de l'intervalle moyen des mises à jour des objets mobiles.

Ils remarquent également que plus les données contenues dans un rectangle englobant ont un comportement proche, plus l'indexation avec un R-tree ou mieux encore un TPR-tree est performante au niveau du nombre d'entrées-sorties. En effet, le rectangle englobant aura une expansion au cours du temps limité et restera proche du rectangle englobant minimal.

Les performances de l'indexation proposée dépendent également de la taille de l'intervalle temporel qui correspond au futur du moment où est exprimé la requête, que l'on peut interroger. Plus cet intervalle augmente, moins les performances sont bonnes. Néanmoins, elles restent supérieures à celle d'un R-tree classique.

Enfin, les auteurs notent que le nombre d'opérations d'entrées-sorties ne croit que très lentement en fonction du nombre d'objets représentés. De plus, si les performances se dégradent au début avec le temps, elles stagnent ensuite car l'arbre se stabilisera.

4.3. Le *QuadTree*

Dans la variante la plus simple du *QuadTree*, on découpe l'espace en un ensemble de cellules régulières et on associe une page à chaque cellule. Chaque objet est alors inséré dans toutes les cellules qu'il intersecte, ce qui peut mener à référencer plusieurs fois un même objet.

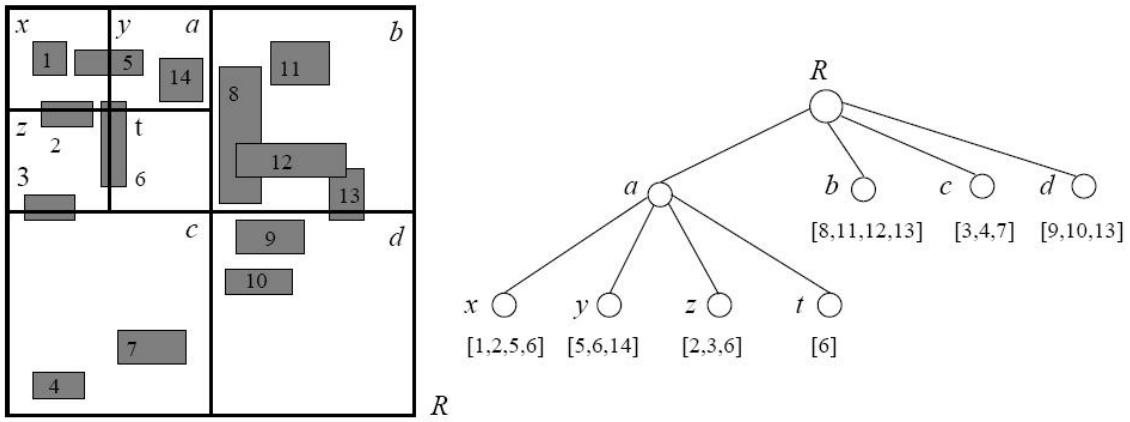


Figure 5 : Indexation par QuadTree

Quand une page déborde, on la divise en quatre sous-cellules égales, et on répartit les objets entre ces quatre pages/cellules. Un exemple est donné dans la Figure 5, en supposant qu'une page contient 4 objets. Le coin supérieur gauche est plus divisé puisqu'il contient 5 objets.

Cette méthode est très simple, et comprend de nombreuses variantes intéressantes puisqu'elles permettent de réutiliser l'arbre B du SGBD. L'inconvénient est la redondance qui impose un tri pour éliminer les doublons après chaque requête.

Indexation par Quadtree [15]

Comme le précédent, cet index vise à indexer les positions futures, et se rapproche donc du modèle [16] auquel il se réfère d'ailleurs explicitement. Chaque trajectoire est donc un segment de droite commençant à t_0 et s'étendant sur un intervalle de temps prédéfini. Elle est représentée par deux fonctions linéaires $x = v_x * t + x_0$ *et* $y = v_y * t + y_0$.

a. Construction

Les trajectoires sont indexées par une variante du *quadtree* dite *PMR-quadtree*. Afin de se ramener au cas où on indexe des segments dans un espace 2D, on utilise en fait deux quadtrees : un pour la fonction donnant *x(t)* et l'autre pour *y(t)*. Lors d'une recherche, il faut utiliser séparément les deux index, puis effectuer l'intersection des résultats obtenus.

La construction de l'index est basée sur le découpage récursif classique du Quadtree. Etant donné un segment *s* décrit par (v_x, x_0), on recherche tous les quadrants qui intersectent *s* et on y insère l'information (id, v_x, x_0) où *id* est l'identifiant permettant de rechercher l'objet sur le disque. Comme toutes les structures basées sur un découpage régulier de l'espace, le même objet est référencé plusieurs fois (redondance).

Lorsqu'il y a un dépassement de capacité de la page d'indexation, le quadrant est divisé en 4 régions (qu'on appellera suivant leur position géographique *NW*, *SW*, *NE* et *NW*). L'enregistrement $<id, v_x, x_0>$ est ajouté à tous les nouveaux quadrants traversés. La Figure 6 montre un exemple de l'indexation proposée par les auteurs. Dans cet exemple la capacité de stockage d'une page est fixée à 2. L_1, L_2, L_3 et L_4 sont 4 trajectoires d'objets mobiles traversant l'espace d'indexation. Par exemple le quart en bas à gauche de notre espace est traversé par 3 trajectoires $(L_1, L_2$ et $L_3)$. Il a donc fallu le découper en quatre quadrants et associer autant de pages du disque $(P_0, P_1, P_2$ et $P_3)$.

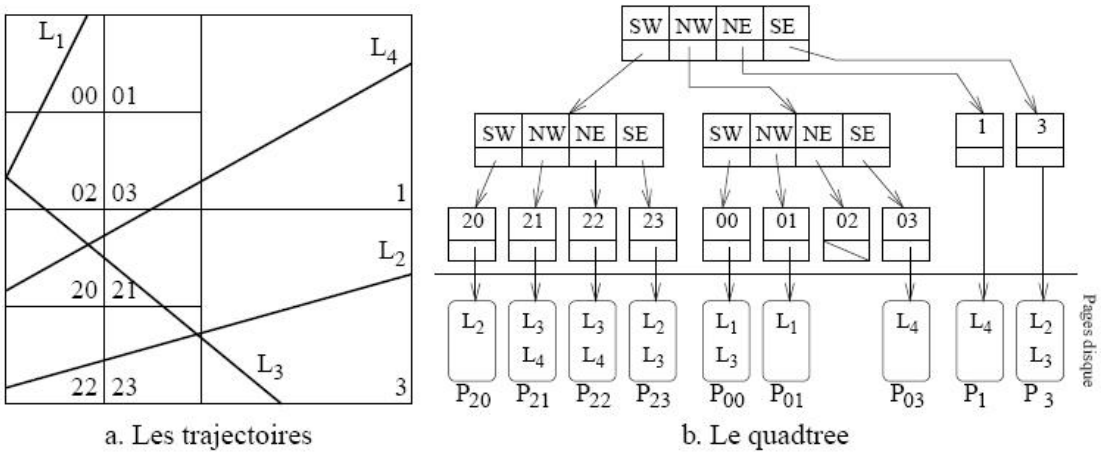


Figure 6 : Construction du quadtree

b. Mise-à-jour

Il s'agit d'une des principales limites de la structure : que faire quand un objet change de direction ? La solution proposée par les auteurs est assez brutale : un index est construit pour une période de temps déterminée ΔT et devient obsolète à l'issue de cette période. Il faut alors reconstruire complètement un nouvel index pour la période suivante, et ainsi de suite.

c. Recherche

L'index supporte les fenêtrages spatio-temporels : étant donnée les intervalles [x_{min}, x_{max}, y_{min}, y_{max}, t_{min}, t_{max}], trouver les objets qui vont passer dans cette fenêtre (NB : l'intervalle temporel doit être compris dans l'intervalle de validité de l'index). On effectue deux fenêtrages sur les deux quadtree, le premier avec le rectangle [x_{min}, x_{max}, t_{min}, t_{max}], le second avec [y_{min}, y_{max}, t_{min}, t_{max}], l'algorithme consistant à parcourir récursivement, en partant de la racine, les quadrants intersectant la fenêtre.

d. Résultats

La première remarque faite par les auteurs concerne la taille du stockage de cette indexation. Le nombre de pages du disque nécessaire croit en fonction du nombre d'objets par paliers. La valeur d'un palier est 4 fois plus grande que celle du palier précédent. Cela est du au fait que les quadrants en cas de dépassement de capacité sont coupés en 4 et que dans les résultats présentés on a supposé une équi-répartition des données (par conséquent le dépassement de capacité à lieu pour tous les quadrants à peu près au même moment).

Les tests effectués pas les auteurs permettent également d'évaluer le taux de remplissage de l'arbre Quadtree. Ce taux est une fonction du nombre d'objets que l'on souhaite indexer. La valeur de celui-ci est comprise, d'après les auteurs, entre 50% et 90%. Cette valeur de 90% s'explique par le fait que la $i^{ème}$ "vague" de segmentation des quadrants, correspondant au $i^{ème}$ palier cité au paragraphe précédent, commence avant que la $(i-1)^{ème}$ soit finie.

4.4. Synthèse :

En résumé, il n'existe pas de structure optimale pour indexer des objets dans un espace de dimension 2. Les solutions ci-dessus apparaissent cependant comme de bons compromis qui se rapprochent du comportement souhaité (temps de recherche logarithmique, occupation de l'espace linéaire) dans le cas de données ayant des propriétés statistiques acceptables (et notamment une répartition dans l'espace à peu près uniforme). Si on considère maintenant que les objets sont mobiles, aucune des deux structures ci-dessus ne semble tenir la route. Dans l'un et l'autre cas, l'indexation est basée sur l'hypothèse que la position des objets est fixe.

5. Requêtes Spatio-temporelles

5.1. Introduction

Avec l'augmentation du nombre d'applications exploitant de grands ensembles de données spatio-temporelles, il devient essentiel de fournir des techniques efficaces de traitement des requêtes dans les bases de données spatio-temporelles.

Les requêtes ST vont permettre une analyse sémantique et spatiale des données de la base. On peut distinguer les requêtes tributaires d'informations sémantiques et qui seront traitées dans la couche supérieure, et celles de bas niveau exclusivement spatiales. Par exemple, une requête d'un système d'information géographique est souvent une combinaison d'un ensemble de requêtes sémantiques et spatiales.

5.2. Classification

Contrairement aux requêtes traditionnelles et spatiales, dans les requêtes spatio-temporelles, les objets et les régions de requêtes peuvent changer leurs positions à travers le temps. Outre le type (requête d'étendue, requête du plus proche voisin, etc.…), on peut classer une requête spatio-temporelle suivant plusieurs autres critères comme :

- la mobilité de l'objet sujet à requête,
- la mobilité de la requête (son émetteur),
- le temps en question (passé, présent ou futur), et enfin,
- la durée de la requête dans le temps (instantanée ou continue).

Pour illustrer cette classification, voici quelques exemples de requêtes spatio-temporelles qu'on peut rencontrer dans la vie réelle, avec leurs paramètres respectifs :

Reporter continuellement le nombre de voitures sur le tronçon 2 de l'autoroute A4.

- Type : Requête d'étendue (*Range Query*).
- Temps : Présent (instantané).
- Durée : Requête Continue.
- Mobilité de l'émetteur de requête : Stationnaire.
- Mobilité de l'objet sujet à requête : Mouvant.

Quels seront les restaurants les plus proches pour la prochaine heure ?

- Type : Requête des *K* plus proches voisins (*K-NN : K-Nearest Neighbors Query*).
- Temps : Futur.
- Durée : Requête Continue.
- Mobilité de l'émetteur de requête : Mouvante.
- Mobilité de l'objet sujet à requête : Stationnaire.

Envoyer une invitation à toutes les voitures dont je suis le plus proche restaurant.

- Type : Requête inverse des plus proches voisins (*Reverse K-Nearest Neighbors Query*).
- Temps : Présent.
- Durée : Requête Instantanée.
- Mobilité de l'émetteur de requête : Stationnaire.
- Mobilité de l'objet sujet à requête : Mouvant.

Quelle a été la distance la plus proche entre les taxis A et B, hier ?

- Type : Requête du point le plus proche (*Closest Point Query*).
- Temps : Passé.
- Durée : Instantanée.
- Mobilité de l'émetteur de requête : Mouvante.
- Mobilité de l'objet sujet à requête : Mouvant

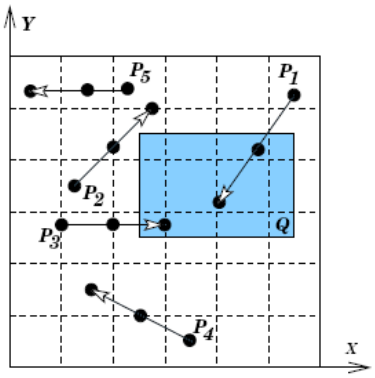
5.3. Opérateurs Spatio-temporels

On peut distinguer deux types de prédicats spatiaux :

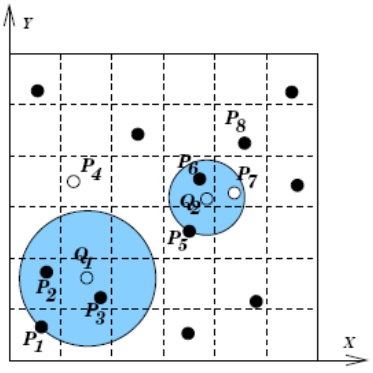
- Le premier représente le type de base, et inclut les opérations impliquant des attributs spatiaux comme la sélection ou la jointure spatiale. Le traitement de ces prédicats dépendra principalement de l'indexation utilisée pour les données (voir Section 4).

- Le deuxième type englobe des prédicats plus évolués et spécifiques au domaine spatial, comme la requête d'étendue ou la requête du plus proche voisin :

    - **La requête d'étendue** : renvoie les ID des objets spatiaux se trouvant à l'intérieur de la zone indiquée. Les requêtes d'étendue peuvent être rectangulaire ou circulaire stationnaire en spécifiant les coins de l'étendue ou son centre et son rayon.



    - **La requête du plus proche voisin** : comme son nom l'indique, renvoie l'ID de l'objet spatial le plus proche du point indiqué. Ce point représente souvent la position de l'émetteur de la requête. Une variante de ce type de requête permet de récupérer un ensemble des plus proches voisins au lieu d'un seul. En paramètre, le point sera alors associé à un nombre *k* précisant le nombre voulu des voisins les plus proches.



De tels opérateurs peuvent être exploités autant par les requêtes stationnaires que celles mobiles. De plus, les requêtes spatio-temporelles peuvent être associées à une fenêtre de temps permettant d'interroger la base sur le passé si le système gère l'historique du mouvement des objets, et éventuellement sur le futur immédiat si ce dernier permet la prédiction de leurs mouvements en exploitant cet historique.

5.4. Requêtes Spatio-temporelles Continues

Les applications spatio-temporelles, comme les services dépendants de la position par exemple, possèdent les caractéristiques spécifiques suivantes :

1. Un grand nombre d'objets mobiles et stationnaires, et par conséquent autant de requêtes mobiles et stationnaires.

2. Les requêtes spatio-temporelles sont continues par nature. A l'inverse des requêtes instantanées qui sont évaluées une fois, les requêtes continues nécessitent une évaluation continue puisque le résultat d'une requête devient invalide avec le changement d'informations de la requête ou des objets de la base de données.

3. N'importe quel retard du résultat de la requête aboutit à une réponse obsolète. Considérons, par exemple, une requête qui recherche les objets mouvants qui se trouvent dans une certaine région. Si la réponse de la requête est retardée, la réponse peut être périmée dans un environnement où les objets changent continuellement leurs positions.

Ces caractéristiques spécifiques appellent à des algorithmes spéciaux de traitement continu des requêtes spatio-temporelles afin d'obtenir une scalabilité et une évaluation efficace des requêtes spatio-temporelles continues. Ces algorithmes suivent cinq grandes approches de traitement :

5.4.1. Approche Directe (Naïve)

C'est l'approche intuitive et la plus simple. Elle se résume à effectuer une abstraction des requêtes continues en une série de requêtes instantanées évaluées périodiquement. Cette approche peut générer, cependant, une grande surcharge lorsque la fréquence de réévaluation est grande. On peut réduire cette surcharge pour chaque requête, au détriment de la précision des résultats, en augmentant la durée séparant deux réévaluations consécutives. Cette approche est, par conséquent, très inappropriée pour les applications temps réel nécessitant des réponses instantanées et exactes.

5.4.2. Approche par Validation de Résultat

Ici, la réponse à une requête dispose d'un paramètre additionnel utilisé pour valider le résultat de cette dernière. Ce paramètre est soit temporel (ex. temps valide -*valid time*- [19]) soit spatial (ex. une période sûre -*safe period*- [20], une région valide -*valid region*- [21], ou une région sûre -*safe region*- [22]) :

- Temps de validité (t) : La réponse à la requête est valide pour les t prochaines unités de temps.
- Région de validité (R) : La réponse à la requête est valide tant qu'on est à l'intérieur d'une région R.

A chaque réponse de requête, le serveur associe le temps de validité, ou la région de validité, de la réponse. Une fois que le temps de validité expire ou que le client sort de la région de validité (expiration de la condition de validité associée), le client doit re-soumettre la requête continue pour sa réévaluation. Le temps T entre deux évaluations consécutives dépend de la précision du temps de validité et de la région de validité.

Le calcul d'un temps ou d'une région de validité pour les requêtes sur des objets mouvants est un énorme défi, puisque les structures utilisées ici et qui supposent que les objets sujets à requêtes sont fixes, nécessitent beaucoup de calcul pour leur reconstruction en cas de modification de la position d'un de ces derniers.

5.4.3. Approche par « *Caching* » du Résultat

Cette approche, relative à une architecture client/serveur, est venue de l'observation du fait que les évaluations consécutives d'une requête continue produisent souvent des résultats très similaires. L'idée était alors de récupérer plus de données pouvant être utilisées plus tard (prochaines évaluations), au lieu de l'évaluation exacte de la requête continue. Par exemple, pour les :

- Requêtes *K-NN* : Initialement, récupérer plus de K plus proches voisins.
- Requête d'étendue : Evaluer la requête avec une étendue plus large.

Les résultats antérieurs sont soit cachés du coté serveur (ex. [23]), soit du coté client (ex. [24]) si ce dernier possède les capacités de stockage et de calcul nécessaires. Les résultats précédemment mis en cache sont utilisés pour élaguer la recherche pour les nouveaux résultats des requêtes subséquentes du plus proche voisin et des requêtes d'étendue.

Cette approche pose, cependant, de nouvelles questions relatives notamment à :

- La quantité d'information à pré-calculer : En plus de la taille mémoire limitée du client, on peut largement pré-calculer le résultat d'une requête mais n'en utiliser qu'une petite fraction (ex. le client peut changer de direction et de vitesse à n'importe qu'elle instant). Ainsi, une bonne appréciation de la quantité d'information à pré-calculer pour une requête donnée est primordiale.

- La politique de remplacement du cache (*re-caching policy*) : du moment que la mémoire du client est très réduite, il crucial de connaître (ou de prédire) les données qui ne seront plus utilisées par la suite, et de les remplacer par de nouvelles données.

5.4.4. Approche par Prédiction du Résultat

Si la trajectoire du mouvement de la requête est connue à priori, alors en utilisant la géométrie calculatoire pour les objets stationnaires (ex. [25]) ou les informations de vitesse pour les objets mouvants (ex. [23]), on peut identifier quels objets seront à l'intérieur d'une étendue ou les plus proches voisins de la trajectoire de la requête, et qui feront partie de sa réponse. La réponse à une requête dont la trajectoire est donnée ou même présumée peut, donc, être pré-calculée à l'avance.

Ainsi, dans cette approche, le mouvement dans la trajectoire est divisé en N intervalles, chacun associé à sa propre réponse à la requête.

La requête est évaluée une seule fois (comme une requête instantanée). La réponse est alors valide pour de longues périodes de temps.

Si les informations sur la trajectoire changent, alors la requête nécessite d'être réévaluée. Le temps *T* entre deux évaluations consécutives dépend de la précision de détermination de la trajectoire future.

5.4.5. Approche par Evaluation Incrémentale du Résultat

Au lieu de la réévaluation de la requête et la production de la réponse entière quand cette dernière change, l'exécuteur de la requête renvoie deux types de mises à jour : positives et négatives pour la réponse précédemment reportée [26, 27]. Comme dans l'approche précédente, la requête est évaluée seulement une fois. Ensuite, seules les mises à jour de la réponse de la requête sont évaluées et envoyées au client.

Une mise à jour positive se réfère à un objet devant être ajouté à la réponse de la requête, alors qu'une mise à jour négative indique un objet devant y être retiré [26].

Seuls les objets traversant les frontières de la requête sont pris en compte dans cette approche. Cependant, il est nécessaire d'écouter continuellement les notifications relatives à la traversée des frontières de la requête.

5.5. Conclusion

En résumé, l'adoption d'une approche d'exécution continue des prédicats ST autre que celle de l'approche directe, comme l'approche par validation, par *caching* ou par celle de l'évaluation incrémentale dépendra du contexte et du type de l'application exploitée ainsi que de la nature du mouvement des objets et/ou des requêtes. Ce choix influencera grandement les performances des requêtes spatio-temporelles continues.

Optimisation des Requêtes

1. Introduction

Il y a plusieurs manières d'évaluation et cela même pour une simple requête, où chacune est meilleure dans certaines situations, et le SGBD doit considérer ces alternatives et choisir celle avec le plus petit coût estimé. Les requêtes composées de plusieurs opérations ont beaucoup plus d'options d'évaluation, et leur trouver un bon plan peut représenter un défi significatif.

Une vue détaillée de l'optimisation de requête et de la couche d'exécution dans l'architecture SGBD est montrée en Figure 1. Les requêtes sont analysées grammaticalement (*parsing*) et présentées ensuite à **l'optimisateur de requête**, qui est responsable de l'identification d'un plan d'exécution efficace pour l'évaluation de la requête. L'optimisateur génère des plans alternatifs et choisit le plan avec le plus petit coût estimé, en utilisant, en autres, les informations des catalogues du système.

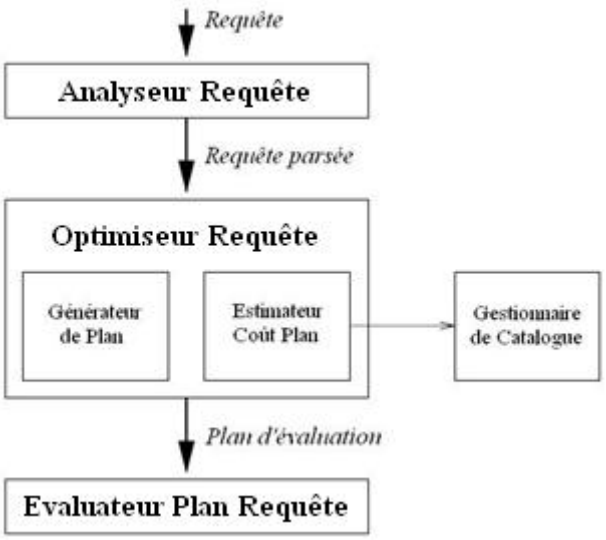


Figure 1 : Analyse de Requête, Optimisation, et Exécution

L'optimisation a pour mission d'établir le moyen le plus efficace pour exécuter les requêtes soumises. L'optimiseur exprime la méthode choisie sous la forme d'un plan d'accès. Ce plan décrit les tables à balayer, l'index à utiliser, le cas échéant, pour chaque table, la stratégie de jointure à adopter et l'ordre de lecture des tables. Il existe souvent plusieurs plans pour parvenir au même objectif. D'autres variables peuvent encore augmenter le nombre de plans d'accès possibles.

En exploitant de puissants algorithmes, parfois propriétaires, l'optimiseur commence sa sélection parmi les choix disponibles. Il fonde ses décisions sur les prévisions des ressources nécessaires à chaque requête. L'optimiseur tient aussi bien compte du coût en termes d'accès disque qu'en termes de temps CPU estimé pour chaque opération.

La plupart des requêtes peuvent être exprimées de plusieurs façons dans le langage utilisé. Ces expressions sont sémantiquement équivalentes, c'est-à-dire qu'elles accomplissent la même tâche, mais leur syntaxe peut être sensiblement différente. A quelques rares exceptions près, l'optimiseur conçoit un plan d'accès uniquement en fonction de la sémantique de chaque instruction. Même si elles semblent importantes, les différences de syntaxe n'ont généralement aucun effet. Par exemple, des différences dans l'ordre des prédicats, des tables et des attributs dans la syntaxe de la requête n'entrent strictement pas en ligne de compte lors du choix d'un plan d'accès.

Idéalement, l'étape d‘optimisation doit identifier le plan d'accès le plus efficace possible, mais cet objectif est souvent peu réaliste. Avec une requête relativement complexe, il peut exister un grand nombre de possibilités.

Aussi efficace que peut l’être l'optimisation, l'analyse de chaque option consomme du temps et des ressources. L'optimiseur compare le coût de l'optimisation suivante avec le coût d'exécution du meilleur plan trouvé jusque là. Si un des plans conçus présente un coût relativement faible, l'optimiseur s'arrête et autorise l'exécution de ce plan. L'optimisation suivante risque en effet de consommer plus de ressources que l'exécution du plan déjà trouvé.

La première partie de ce chapitre présente une revue de l’optimisation de requête, quelques informations de fond pertinentes, et un cas d’étude qui illustre et motive l’optimisation de requête.

Dans la suite, on considérera un nombre d'exemples de requête utilisant le schéma suivant :

Sailors (sid: integer, sname: string, rating: integer, age: real)
Reserves (sid: integer, bid: integer, day: date, rnames: string)

On assumera que chaque tuple de *Reserves* est d'une longueur de 40 octets, qu'une page peut contenir 100 tuples de *Reserves*, et que nous avons 1000 pages de tuples semblables. De manière similaire, on assumera que chaque tuple de *Sailors* est d'une longueur de 50 octets, qu'une page peut contenir 80 tuples, et que nous avons 500 pages de tuples identiques.

2. Revue de l'Optimisation des Requêtes Relationnelles

L'espace des plans considérés par un optimisateur relationnel typique peut être trouvé en admettant qu'*une requête est traitée essentiellement comme une expression algébrique de restriction (σ) - projection (π) - jointure (×)*, avec les opérations restantes (si il y en a) accomplies sur le résultat de l'expression σ - π - ×. Optimiser une telle expression algébrique relationnelle implique deux étapes de base :

- L'énumération des plans alternatifs pour l'évaluation de l'expression ; en général, un optimisateur considère un sous-ensemble de tous les plans possibles car le nombre de plans possibles peut être très grand.

- L'estimation de chaque plan énuméré, et le choix du plan avec le plus petit coût.

2.1. Plans d'Evaluation de Requête

Un plan d'évaluation de requête (ou simplement plan) consiste en un arbre algébrique relationnel étendu, avec des annotations supplémentaires à chaque nœud indiquant les méthodes d'accès à employer pour chaque relation et la méthode d'implémentation à utiliser pour chaque opérateur relationnel.

Considérez la requête SQL suivante :

```
SELECT      S.name
FROM        Reserves R, Sailors S
WHERE       R.sid = S.sid
            AND R.bid = 100 AND S.rating > 5
```

Cette requête peut être exprimée dans l'algèbre relationnelle comme suit:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \times_{sid=sid} Sailors))$$

Cette expression est montrée sous la forme d'un arbre dans la Figure 2. L'expression algébrique spécifie partiellement comment évaluer la requête : on calcule en premier la jointure naturelle de *Reserves* et de *Sailors*, ensuite on effectue les sélections, et on projette finalement l'attribut *sname*.

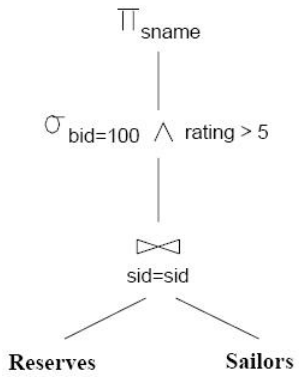


Figure 2 : Requête Exprimée comme un Arbre Relationnel Algébrique

Pour obtenir un plan d'évaluation entièrement spécifié, on doit décider d'une implémentation pour chacune des opérations algébriques impliquées. Par exemple, on peut utiliser une jointure par boucles emboîtées simple orientée page, avec *Reserves* comme relation extérieure et appliquer les sélections et les projections à chaque tuple du résultat de la jointure comme il est produit ; le résultat de la jointure avant les sélections et les projections n'est jamais stocké dans sa totalité. Ce plan d'évaluation est montré dans la Figure 3.

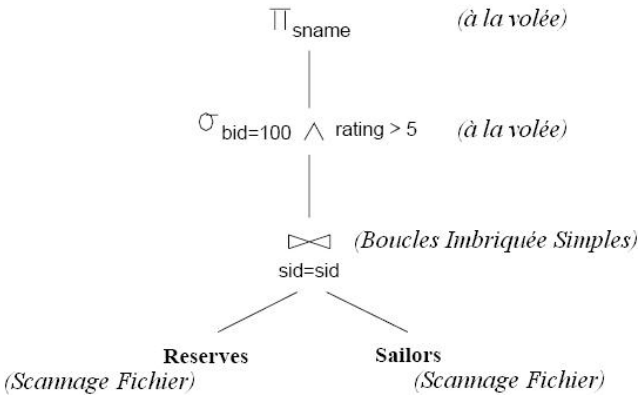


Figure 3 : Plan d'Evaluation pour une Requête Simple

En dessinant le plan d'évaluation de la requête, on a utilisé la convention selon laquelle la relation extérieure est le fils gauche de l'opérateur de jointure. On adoptera cette convention dorénavant.

2.2. Evaluation Pipelinée

Lorsqu'une requête est composée de plusieurs opérateurs, le résultat d'un opérateur est parfois **pipeliné** à un autre opérateur sans la création d'une relation temporaire pour contenir le résultat intermédiaire. Le plan dans la Figure 3 pipeline la sortie de la jointure de *Sailors* et de *Reserves* vers les sélections et projections qui suivent. Pipeliner la sortie d'un opérateur vers l'opérateur suivant préserve le coût d'écriture du résultat intermédiaire et de sa relecture, et les économies de coût peuvent être considérables. Si la sortie d'un opérateur est enregistrée dans une relation temporaire pour son traitement par le prochain opérateur, on dit que les tuples sont matérialisés. L'évaluation pipelinée a moins de coûts de surcharge que la matérialisation et est choisie à chaque fois que l'algorithme de l'évaluation de l'opérateur le permet.

Il y a beaucoup d'opportunités de pipelining dans les plans de requête typiques, même ceux simples impliquant seulement des sélections. Prenez le cas d'une requête de sélection dans laquelle une partie seulement de la condition de sélection coïncide avec un index. On peut imaginer une telle requête comme contenant deux instances de l'opérateur de sélection: La

première contenant la partie primaire, ou concordante, de la condition de sélection originale, et la seconde contenant le reste de la condition de sélection. On peut évaluer une telle requête en appliquant la sélection primaire et en écrivant le résultat dans une relation temporaire, et en appliquant par la suite la deuxième sélection à la relation temporaire. Par opposition, une évaluation pipelinée consiste à appliquer la seconde sélection à chaque tuple du résultat de la sélection primaire, tels qu'ils sont produits, et à ajouter ceux correspondants au résultat final. Lorsque la relation d'entrée d'un opérateur unaire (par exemple, la sélection ou la projection) y est pipelinée, on dit parfois que l'opérateur est appliqué *à la volée*.

Comme exemple plus général, considérez la jointure de la forme $(A \otimes B) \otimes C$, montrée dans la Figure 4 comme un arbre d'opérations de jointure.

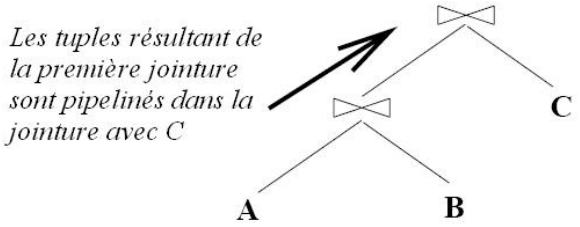


Figure 4 : Un Arbre de Requête illustrant le *Pipelining*

Les deux jointures peuvent être évaluées de manière pipelinée en utilisant une version de la jointure par boucles emboîtées. Conceptuellement, l'évaluation est initiée de la racine, et le noeud qui joint A et B produit les tuples au fur et à mesure qu'ils sont demandés par son noeud parent. Quand le noeud de la racine obtient une page de tuples de son enfant gauche (la relation extérieure), tous les tuples intérieurs correspondants sont récupérés (en utilisant soit un index ou un scanner) et joints avec les tuples extérieurs correspondants; la page courante des tuples extérieurs est abandonnée ensuite. Une demande est alors faite à l'enfant gauche pour la prochaine page de tuples, et le processus est répété.

L'évaluation pipelinée est par conséquent une stratégie de contrôle qui gouverne le taux auquel les différentes jointures dans le plan sont traitées. Elle évite ainsi d'écrire le résultat des jointures intermédiaires dans un fichier temporaire puisque les résultats sont produits et consommés, une page à la fois.

2.3. Interface Itérateur pour les Opérateurs et Méthodes d'Accès

Un plan d'évaluation de requête est un arbre d'opérateurs relationnels. Il est exécuté en sollicitant les opérateurs dans un certain ordre (éventuellement imbriqué). Chaque opérateur dispose d'une ou plusieurs entrées ainsi que d'une sortie, représentant les nœuds du plan. Les tuples doivent être passés entre les opérateurs suivant la structure de l'arbre du plan.

Dans le souci de simplifier le code responsable de la coordination de l'exécution d'un plan, les opérateurs relationnels qui forment les nœuds de l'arbre du plan (lequel est à évaluer en utilisant le pipelining) supportent généralement une interface uniforme d'itérateur, dissimulant les détails d'implémentation interne de chaque opérateur. L'interface d'itérateur d'un opérateur inclut les fonctions **open**, **get_next**, et **close**.

La fonction *open* initialise l'état de l'itérateur en allouant les tampons pour ses entrées et sa sortie, et est aussi utilisée pour faire passer les arguments tels que les conditions de sélection qui modifient le comportement de l'opérateur. Le code pour la fonction *get_next* appelle la fonction *get_next* sur chaque nœud d'entrée et appelle un code spécifique à l'opérateur pour traiter les tuples entrants.

Les tuples sortants générés par le traitement sont placés dans le tampon de sortie de l'opérateur, et l'état de l'itérateur est mis à jour pour garder trace du nombre de tuples consommés. Lorsque tous les tuples de sortie ont été produits à travers des appels répétés à *get_next*, la fonction *close* est appelée (avec le code qui a initié l'exécution de cet opérateur) pour désallouer les informations d'état.

L'interface de l'itérateur supporte naturellement le pipelining des résultats ; la décision de pipeliner ou de matérialiser les tuples d'entrée est encapsulé dans le code spécifique à l'opérateur qui traite les tuples d'entrée. Si l'algorithme implémenté pour l'opérateur autorise que les tuples d'entrée soient traités complètement lorsqu'ils sont reçus, les tuples d'entrée ne sont pas matérialisés et l'évaluation est pipelinée. Si l'algorithme examine les mêmes tuples d'entrée plusieurs fois, alors ils sont matérialisés. Cette décision, comme d'autres détails de l'implémentation de l'opérateur, est dissimulée par l'interface itérateur de l'opérateur.

L'interface itérateur est aussi utilisée pour encapsuler les méthodes d'accès telles que les arbres B^+ et les index basés sur le hachage. Extérieurement, les méthodes d'accès peuvent être vues simplement comme des opérateurs qui produisent un flux de tuples de sortie. Dans ce cas, la fonction *open* peut être utilisée pour passer les conditions de sélection qui correspondent au chemin d'accès.

2.4. L'optimisateur du System R

Les optimisateurs des SGBDRs actuels sont des pièces complexes de logiciel avec beaucoup de détails étroitement protégés et représentent typiquement 40 à 50 ans d'effort de développement. Ces derniers ont été largement influencés par les choix faits dans la conception de l'optimisateur de requête du System R d'IBM. Parmi les choix importants, on retrouve :

- L'utilisation de statistiques sur le contexte de la BD pour estimer le coût d'un plan d'évaluation de requête.

- La considération des plans avec les jointures binaires dans lesquelles la relation intérieure est une relation de base uniquement (c-à-d, pas une relation temporaire). Cette heuristique réduit le nombre (potentiellement très grand) de plans alternatifs devant être considérés.

- La focalisation de l'optimisation sur la classe de requêtes SQL sans imbrication et le traitement des requêtes imbriquées de manière relativement *ad hoc*.

- Ne pas effectuer d'élimination de doublons pour les projections (sauf comme étape finale dans l'évaluation de la requête lorsque c'est explicitement exigé par la clause DISTINCT).

- Un modèle de coût prenant en compte aussi bien les coûts CPU que les coûts E/S.

Notre discussion sur l'optimisation reflète ces choix de conception, sauf pour le dernier point de la liste précédente, que l'on ignore afin de maintenir un modèle de coût simple basé sur le nombre d'E/S de pages.

3. Equivalences Algébriques Relationnelles

Deux expressions algébriques relationnelles sur le même ensemble de relations d'entrée sont dites équivalentes si elles produisent le même résultat sur toutes les instances des ces relations. Les équivalences algébriques relationnelles jouent un rôle central dans l'identification des plans alternatifs. Comme on va le voir dans la section suivante, la poussée des sélections devant les jointures peut conduire à une amélioration significative du plan d'évaluation ; cette poussée est basée sur les équivalences algébriques impliquant les opérateurs de sélection et du produit cartésien.

Dans le fond, un bloc de requête SQL basique peut être vu comme une expression algébrique constituée du produit cartésien de toutes les relations dans la clause FROM, les sélections dans la clause WHERE, et des projections dans la clause SELECT. L'optimisateur peut choisir d'évaluer n'importe quelle expression équivalente et avoir toujours le même résultat. Ces équivalences nous permettent de convertir les produits cartésiens en jointures, de choisir des ordres différents de jointures, et de pousser les sélections et les projections devant les jointures.

3.1. Sélections

Il y a deux équivalences importantes qui impliquent l'opération de sélection. La première implique **la cascade de sélections** :

$$\sigma_{c1 \wedge c2 \wedge \cdots cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$$

Allant de la droite à la gauche, cette équivalence permet de combiner plusieurs sélections en une seule sélection. Intuitivement, on peut tester si un tuple coïncide avec chacune des conditions $c_1, \ldots, c_n$ en même temps. Dans l'autre direction, cette équivalence permet de prendre une condition de sélection impliquant plusieurs conjonctions et de la remplacer par plusieurs opérations de sélection plus petites. Ce remplacement s'avère être très utile en combinaison avec d'autres équivalences, spécialement, la commutation des sélections avec les jointures ou les produits cartésiens. Intuitivement, un tel remplacement est utile dans les cas où seule une partie d'une condition de sélection complexe peut être poussée.

La seconde équivalence déclare que les sélections sont **commutatives** :

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

En d'autres termes, on peut tester les conditions c_1 et c_2 dans l'un ou l'autre ordre.

3.2. Projections

La règle de la cascade des projections dit que le fait d'éliminer successivement des colonnes d'une relation est équivalent à simplement éliminer toutes les colonnes sauf celles retenues par la projection finale :

$$\pi_{a_1}(R) \equiv \pi_{a1}(\pi_{a2}(...(\pi_{an}(R))...))$$

Chaque a_i est un ensemble d'attributs de la relation R, et $a_i \subseteq a_{i+1}$ pour $i=1...n-1$. Cette équivalence est utile en conjonction avec d'autres équivalences comme la commutation des projections avec les jointures.

3.3. Produits Cartésiens et Jointures

Il y a deux équivalences importantes impliquant les produits cartésiens et les jointures. On les présente en termes de jointures naturelles pour simplifier, mais elles sont aussi bien valables pour les jointures générales.

En premier, en supposant que ces champs sont identifiés par le nom au lieu de la position, ces opérations sont commutatives :

$$R \times S \equiv S \times R$$

$$R \otimes S \equiv S \otimes R$$

Cette propriété est très importante. Elle nous permet de décider de la relation intérieure et celle extérieure dans la jointure de deux relations.

La seconde équivalence permet de dire que les jointures et les produits cartésiens sont associatifs :

$$R \times (S \times T) \equiv (R \times S) \times T$$
$$R \otimes (S \otimes T) \equiv (R \otimes S) \otimes T$$

Ainsi, on peut soit joindre *R* et *S* en premier et puis joindre *T* au résultat, ou joindre *S* et *T* ensuite joindre *R* au résultat. L'intuition derrière l'associativité des produits cartésiens est que quelque soit l'ordre dans lequel les trois relations sont considérées, le résultat final contient les mêmes colonnes. L'associativité de la jointure est basée sur la même intuition, avec l'observation supplémentaire que les sélections spécifiant la jointure peuvent être cascadées.

Associée à la commutativité, l'associativité dit principalement qu'on peut choisir de joindre n'importe quelle paire de ces relations, et de joindre ensuite le résultat avec la troisième relation, et d'obtenir toujours le même résultat final. Par exemple, vérifions que :

$$R \otimes (S \otimes T) \equiv (T \otimes R) \otimes S$$

Pour la commutativité, on a :

$$R \otimes (S \otimes T) \equiv R \otimes (T \otimes S)$$

Pour l'associativité, on a :

$$R \otimes (T \otimes S) \equiv (R \otimes T) \otimes S$$

En utilisant encore la commutativité, on a :

$$(R \otimes T) \otimes S \equiv (T \otimes R) \otimes S$$

En d'autres termes, lors de la jointure de plusieurs relations, on est libres de les joindre dans n'importe quel ordre. Cette indépendance d'ordre est fondamentale à la manière avec laquelle un optimisateur de requête génère les plans alternatifs d'évaluation de requêtes.

3.4. Sélections, Projections et Jointures

Quelques équivalences importantes impliquent deux opérateurs ou plus :

On peut commuter une sélection avec une projection si l'opération de sélection implique seulement des attributs retenus par la projection :

$$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$

Chaque attribut mentionné dans la condition de sélection *c* doit être inclus dans l'ensemble des attributs *a*.

On peut combiner une sélection avec un produit cartésien pour former une jointure, comme par la définition de la jointure :

$$R \otimes_c S \equiv \sigma_c(R \times S)$$

On peut commuter une sélection avec un produit cartésien ou une jointure si la condition de sélection implique seulement des attributs d'un des arguments du produit cartésien ou de la jointure :

$$\sigma_c(R \times S) \equiv \sigma_c(R) \times S$$

$$\sigma_c(R \otimes S) \equiv \sigma_c(R) \otimes S$$

Les attributs mentionnés dans *c* doivent apparaître seulement dans *R*, et pas dans *S*. Les équivalences similaires sont valables si *c* implique seulement des attributs de *S* et pas de *R*.

En général, une sélection sur *R x S* peut être remplacée par une cascade de sélections σ_{c1}, σ_{c2} et σ_{c3} tel que c_1 implique des attributs de *R* et *S*, c_2 implique seulement des attributs de *R*, et c_3 implique seulement des attributs de *S* :

$$\sigma_c(R \times S) \equiv \sigma_{c1 \wedge c2 \wedge c3}(R \times S)$$

Utilisant la règle de cascade pour les sélections, cette expression est équivalente à :

$$\sigma_{c1}(\sigma_{c2}(\sigma_{c3}(R \times S)))$$

Utilisant la règle de la commutation des sélections et des produits cartésiens, cette expression est équivalente à :

$$\sigma_{c1}(\sigma_{c2}(R) \times \sigma_{c3}(S))$$

Ainsi, on peut pousser une partie de la condition de sélection *c* devant le produit cartésien. Cette observation est aussi valable pour les sélections en combinaison des jointures, bien sur.

On peut commuter une projection avec un produit cartésien :

$$\pi_a(R \times S) \equiv \pi_{a1}(R) \times \pi_{a2}(S)$$

a_1 est un sous ensemble d'attributs dans a qui apparaît dans R, et a_2 est un sous ensemble d'attributs dans a qui apparaît dans S. On peut commuter une projection avec une jointure si la condition de jointure implique seulement des attributs retenus par la projection :

$$\pi_a(R \otimes_c S) \equiv \pi_{a1}(R) \otimes_c \pi_{a2}(S)$$

De plus, chaque attribut mentionné dans la condition de jointure c doit apparaître dans a.

Intuitivement, on a besoin de retenir seulement ces attributs de R et S qui sont mentionnés soit dans la condition de jointure c ou inclus dans l'ensemble des attributs a retenus par la projection. Clairement, si a inclut tous les attributs mentionnés dans c, les règles de commutation au dessus sont valables. Si a n'inclut pas tous les attributs mentionnés dans c, on peut généraliser les règles de commutation en projetant dehors les attributs qui ne sont pas mentionnés dans c ou a, en effectuant la jointure, et en projetant dehors ensuite tous les attributs qui ne sont pas dans a :

$$\pi_a(R \otimes_c S) \equiv \pi_a(\pi_{a1}(R) \otimes_c \pi_{a2}(S))$$

A présent, a_1 est sous ensemble d'attributs de R qui apparaissent soit dans a ou c, et a_2 est sous ensemble d'attributs de S qui apparaissent soit dans a ou c.

En fait, on peut dériver la règle de commutation la plus générale en utilisant la règle de cascade des projections et la règle de commutation simple.

3.5. Autres équivalences

Des équivalences supplémentaires sont valables lorsqu'on considère des opérations telles que la différence d'ensembles, l'union et l'intersection. Ces deux dernières sont associatives et commutatives. Les sélections et les projections peuvent être commutées avec chacune des opérations d'ensembles (différence d'ensembles, union et intersection).

4. Plans Alternatifs : Un Exemple Motivant

Considérons la requête exemple précédente et intéressons nous au coût d'évaluation du plan montré dans la Figure 3. Le coût de la jointure est de 1.000 + 1.000 * 500 = 501.000 E/Ss de page. Les sélections et la projection sont faites à la volée et n'induisent pas d'E/Ss supplémentaires. Pour simplifier, on ignore le coût d'écriture du résultat final. Le coût total de ce plan est par conséquent de 501.000 E/Ss de page. Il faut convenir que ce plan est naïf, cependant, il est possible d'être encore plus naïf en traitant la jointure comme un produit cartésien suivi d'une sélection !

On considère maintenant divers plans alternatifs pour l'évaluation de cette requête. Chaque alternative améliore le plan original d'une manière différente et introduit quelques idées d'optimisation examinées plus en détail dans le reste de cette section.

4.1. Poussée des Sélections

Une jointure est une opération relativement chère, et une bonne heuristique serait de réduire les tailles des relations à joindre autant que possible. Une approche est d'appliquer les sélections plutôt ; si un opérateur de sélection se trouve après un opérateur de jointure, il serait intéressant d'examiner si la sélection peut être 'poussée' devant la jointure.

Comme exemple, la sélection *bid=100* implique seulement les attributs de *Reserves* et peut être appliquée à *Reserves avant* la jointure. De manière analogue, la sélection *rating>5* implique seulement les attributs de *Sailors* et peut être appliquée à *Sailors* avant la jointure. Supposons que les sélections sont réalisées en utilisant un simple scannage de fichier, que le résultat de chaque sélection est écrit dans une relation temporaire sur le disque, et que les relations temporaires sont ensuite jointes en utilisant une jointure par fusion. Le plan d'évaluation de la requête résultant est montré dans la Figure 5.

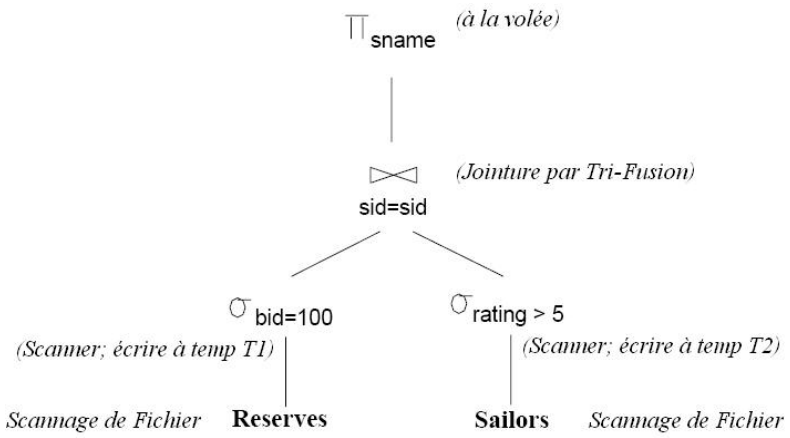


Figure 5 : Un Second Plan d'Evaluation de la Requête

Supposons que cinq pages tampon sont disponibles et estimons le coût de ce plan d'évaluation de requête. (Plus de pages tampon seront vraisemblablement disponibles dans la pratique. On a choisi un petit nombre dans cet exemple simplement à des fins d'illustrations). Le coût d'appliquer *bid=100* à *Reserves* est le coût de scannage de *Reserves* (1.000 pages) plus le coût d'écriture du résultat dans une relation temporaire, disons *T1*. Notez que le coût d'écriture de relation temporaire ne peut être ignoré – on peut seulement ignorer le coût d'écriture du résultat *final* de la requête, qui est le seul composant du coût qui est le même pour tous les plans.

Pour estimer la taille de *T1*, on exige quelques informations complémentaires. Par exemple, si on assume que le nombre maximum de réservations d'un bateau donné est de un, un tuple seulement apparaîtra dans le résultat. Alternativement, si on sait qu'il y a 100 bateaux, on peut supposer que les réservations sont distribuées uniformément à travers tous les bateaux et estimer le nombre de pages dans *T1* à 10. Pour être exacts, supposons que le nombre de pages dans *T1* est effectivement 10.

Le coût d'application de *rating>5* à *Sailors* est le coût de scannage de *Sailors* (500 pages) plus le coût d'écriture du résultat dans une relation temporaire, disons *T2*. Si on assume que les estimations sont uniformément distribuées sur l'intervalle 1 à 10, on peut approximativement estimer la taille de *T2* à 250 pages.

Pour effectuer une jointure par fusion de *T1* et *T2*, supposons qu'une implémentation directe est utilisée dans laquelle les deux relations sont en premier complètement triées et par la suite

fusionnées. Du moment que cinq pages tampon sont disponibles, on peut trier *T1* (qui a 10 pages) en deux passes. Deux parcours de cinq pages chacun sont produits dans la première passe et sont ensuite fusionnés dans la deuxième. A chaque passe, on lit et écrit 10 pages ; ainsi, le coût de tri de *T1* est 2 * 2 * 10 = 40 E/Ss de page. On a besoin de quatre passes pour trier *T2*, qui a 250 pages. Le coût est de 2 * 4 * 250 = 2.000 E/Ss de page. Pour fusionner les versions triées de *T1* et *T2*, on a besoin de scanner ces relations, et le coût de cette étape est de 10 + 250 = 260. La projection finale est faite à la volée, et par convention on ignore le coût d'écriture du résultat final.

Le coût total du plan montré dans la Figure 5 est la somme du coût de la sélection (1.000 + 10 + 500 + 250 = 1.760) et du coût de la jointure (40 + 2.000 + 260 = 2.300), ce qui fait 4.060 E/Ss de page :

- La jointure par fusion est l'une des diverses méthodes de jointure. On peut être capable de réduire le coût de ce plan en choisissant une méthode de jointure différente. Comme alternative, supposez l'utilisation d'une jointure par boucles emboîtées à bloc au lieu de la jointure par fusion. Utilisant *T1* comme relation extérieure, pour chaque bloc à trois pages de *T1*, on scanne entièrement *T2* ; ainsi, on scanne *T2* quatre fois. Le coût de la jointure est par conséquent le coût de scannage de *T1* (10) plus le coût de scannage de *T2* (4 * 250 = 1.000). Le coût du plan est à présent égal à 1.760 + 1.010 = 2.770 E/Ss de page.

- Un affinage supplémentaire consisterait à pousser les projections, de la même manière que nous avons poussé les sélections devant la jointure. Notez que seuls les attributs *sid* et *sname* de *T2* et l'attribut *sid* de *T1* sont réellement requis. Pendant qu'on scanne *Reserves* et *Sailors* pour effectuer les sélections, on pourrait aussi éliminer les colonnes indésirables. Cette projection à la volée réduit les tailles des relations temporaires *T1* et *T2*. La réduction de la taille de *T1* est substantielle car un seul champ d'entier est retenu. En fait, *T1* tiendra maintenant à l'intérieur de trois pages tampon, et on peut effectuer une jointure par boucles emboîtées avec un seul scannage de *T2*. Le coût de l'étape de jointure baisse ainsi sous 250 E/Ss de page, et le coût total du plan diminue à environ 2.000 E/Ss.

4.2. Utilisation des Index

Si les index sont disponibles sur les relations *Reserves* et *Sailors*, de meilleurs plans d'évaluation de requête peuvent même être disponibles. Par exemple, supposons qu'on a un index de hachage statique clustérisé sur le champ *bid* de *Reserves* et un autre index de hachage sur le champ *sid* de *Sailors*. On peut ensuite utiliser le plan d'évaluation de requête montré en Figure 6.

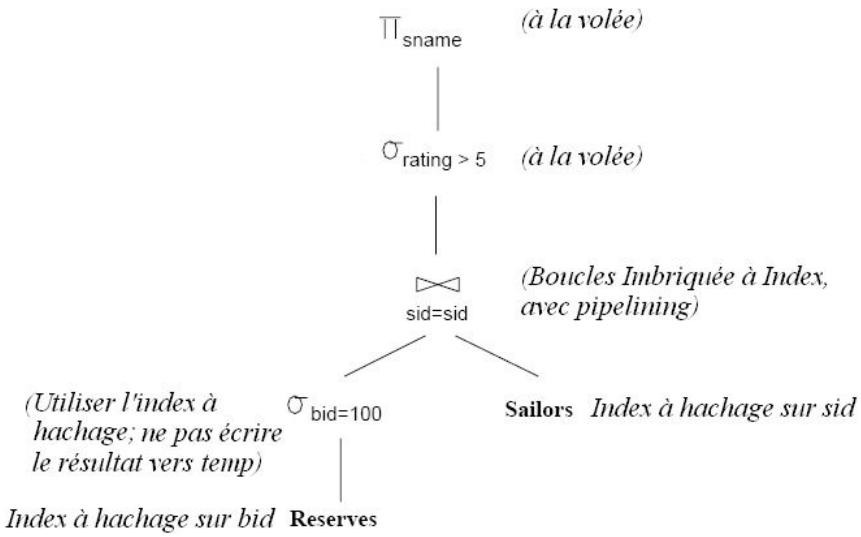


Figure 6 : Un Plan d'Evaluation de Requête Utilisant des Index

La sélection *bid=100* est effectuée sur *Reserves* en utilisant l'index d'hachage sur *bid* afin de récupérer seulement les tuples correspondants. Comme précédemment, si on sait que 100 bateaux sont disponibles et on suppose que les réservations sont distribuées uniformément à travers tous les bateaux, on peut estimer le nombre de tuples sélectionnés à 100.000/100 = 1.000. Puisque l'index sur *bid* est clustérisé, ces 1.000 tuples apparaissent consécutivement à l'intérieur du même seau (bucket) ; ainsi, le coût est de 10 E/Ss de page.

Pour chaque tuple sélectionné, on retrouve les tuples *Sailors* concordants en utilisant l'index d'hachage sur le champ *sid* ; les tuples *Reserves* sélectionnés ne sont pas matérialisés et la jointure est pipelinée. Pour chaque tuple dans le résultat de la jointure, on effectue la sélection *rating>5* et la projection de *sname* à la volée. Il y a plusieurs points importants à noter ici :

1. Puisque le résultat de la sélection sur *Reserves* n'est pas matérialisé, l'optimisation de la projection des champs non nécessaires subséquemment est inutile (et n'est pas utilisée dans le plan montré en Figure 6).

2. Le champ de jointure *sid* est une clé pour *Sailors*. Par conséquent, au plus un tuple de *Sailors* correspond à un tuple donné de *Reserves*. Le coût de récupération de ce tuple concordant dépend de l'occupation de la mémoire par le répertoire de l'index d'hachage sur la colonne de *Sailors* et de la présence d'éventuelles pages de débordement. Cependant, le coût ne dépend pas de la clusterisation de cet index car il y a au plus un tuple de *Sailors* correspondant et les requêtes pour les tuples de *Sailors* sont faites dans un ordre aléatoire par *sid* (car les tuples de *Reserves* sont récupérés par *bid* et sont en conséquence considérés dans un ordre aléatoire par *sid*). Pour un index d'hachage, 1,2 E/S de page (en moyenne) est une bonne estimation du coût de récupération d'une entrée de donnée. En supposant que l'index d'hachage de *sid* sur *Sailors* utilise l'Alternative (1) pour les entrées de données, 1.2 E/S est le coût pour récupérer un tuple correspondant de *Sailors* (et si une des deux autres alternatives est utilisée, le coût serait de 2,2 E/Ss).

3. On a choisi de ne pas pousser la sélection *rating>5* devant la jointure, et il y a une raison importante pour cette décision. Si on a effectué la sélection avant la jointure, la sélection impliquerait le scannage de *Sailors*, en supposant qu'il n'y a pas d'index disponible sur le champ rating de *Sailors*. De plus, que l'index soit disponible ou pas, une fois qu'on applique une telle sélection, on n'a pas un index sur le champ *sid* de du résultat de sélection (à moins qu'on choisisse de construire un tel index uniquement pour les besoins de jointure subséquente). Ainsi, pousser les sélections devant les jointures est une bonne heuristique, mais pas toujours la meilleure stratégie. Typiquement, comme dans cet exemple, l'existence d'index utiles est la raison pour qu'une sélection ne soit pas poussée. (Autrement, les sélections sont poussées).

Dans la Figure 6, la sélection des tuples de *Reserves* coûte 10 E/Ss, comme vu précédemment. Il y a 1.000 tuples pareils, et pour chaque, le coût pour trouver le tuple correspondant de *Sailors* est de 1,2 E/Ss, en moyenne. Le coût pour cette étape (la jointure) est par conséquent égal à 1.200 E/Ss. Toutes les sélections et projections restantes sont effectuées à la volée. Le coût total du plan est de 1.210 E/Ss.

Comme noté plutôt, ce plan n'utilise pas de *clustering* de l'index de *Sailors*. Le plan peut davantage être affiné si l'index sur le champ *sid* de Sailors est clustérisé. Supposez qu'on matérialise le résultat de l'exécution de la sélection *bid=100* sur *Reserves* et qu'on trie cette relation temporaire. Cette relation contient 10 pages. La sélection les tuples coûte 10 E/Ss de page (comme avant), l'écriture du résultat vers une relation temporaire coûte encore 10 E/Ss, et avec cinq pages tampon, le tri de celle-ci coûte 2 * 2 * 10 = 40 E/Ss. (Le coût de cette étape est réduit si on pousse la projection sur *sid*. La colonne *sid* des tuples matérialisés de *Reserves* nécessite seulement trois pages et peut être triée en mémoire avec cinq pages tampon). Les tuples sélectionnés de *Reserves* peuvent maintenant être récupérés en ordre par *sid*.

Si un capitaine a réservé le même bateau plusieurs fois, tous les tuples de *Reserves* correspondants sont récupérés consécutivement ; le tuple de Sailors correspondant sera trouvé dans la zone tampon pour tous mais le premier le requiert. Ce plan amélioré démontre aussi que le pipelining n'est pas toujours la meilleure stratégie.

La combinaison de la poussée des sélections et l'utilisation des index illustré par ce plan est très puissante. Si les tuples sélectionnés de la relation extérieure sont joints avec un seul tuple intérieur, l'opération de jointure peut devenir triviale, et on gagne en performance en considérant le plan naïf montré en Figure 5. La variante suivante de notre requête exemple illustre cette situation :

```
SELECT      S.name
FROM        Reserves R, Sailors S
WHERE       R.sid = S.sid
            AND R.bid = 100 AND S.rating > 5
            AND R.day = '8/9/94'
```

Une légère variante du plan montré en Figure 6, désigné pour répondre à cette requête, est montrée en Figure 7. La sélection *day='8/9/94'* est appliqué à la volée au résultat de la sélection *bid=100* sur la relation *Reserves*.

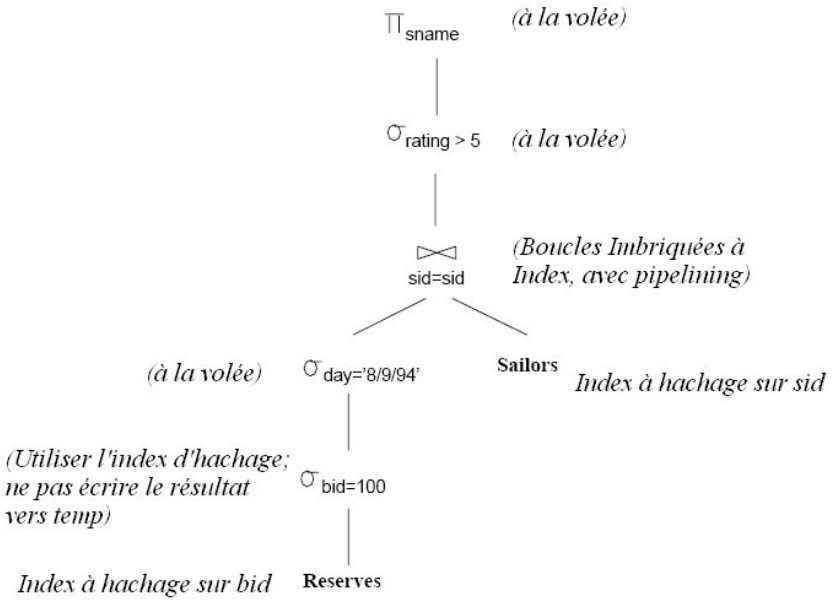


Figure 7 : Un Plan d'Evaluation pour le Second Exemple.

Supposez que *bid* et *day* forment une clé pour *Reserves*. Estimons maintenant le coût du plan montré en Figure 7. La sélection *bid=100* coûte 10 E/Ss de page, comme avant, et la sélection supplémentaire *day='8/9/94'* est appliquée à la volée, éliminant tout sauf (au plus) un tuple de *Reserves*. Il y a au plus un tuple de *Sailors* correspondant, et celui-ci est récupéré en 1,2 E/Ss (un nombre moyen !). La sélection sur rating et la projection sur *sname* sont alors appliquées à la volée sans coûts additionnels. Le coût total du plan en Figure 7 est ainsi d'environ 11 E/Ss. En contraste, si on modifie le plan naïf de la Figure 5 pour effectuer la sélection supplémentaire sur *day* en même temps que la sélection *bid=100*, le coût reste à 501.000 E/Ss.

4.3. Conclusion

Le but de l'optimisation de requête est en général d'éviter les plus mauvais plans d'évaluation et de trouver un bon plan, plutôt que de trouver le meilleur plan. Pour optimiser une requête SQL, on l'exprime en premier en algèbre relationnelle, on considère plusieurs plans d'évaluation de requête pour l'expression algébrique, et on choisit celui avec le plus petit coût estimé.

Un *plan d'évaluation de requête* est un arbre avec des opérateurs relationnels dans les nœuds intermédiaires et des relations sur les nœuds feuilles. Les nœuds intermédiaires sont annotés avec l'algorithme choisi pour exécuter l'opérateur relationnel et les nœuds feuilles sont annotés avec la méthode d'accès utilisée pour récupérer les tuples depuis la relation.

Les résultats d'un opérateur peuvent être pipelinés dans un autre opérateur sans la matérialisation du résultat intermédiaire. Si les tuples d'entrée d'un opérateur unaire sont pipelinés, cet opérateur est dit être appliqué à la volée. Les opérateurs ont une interface uniforme d'itérateur avec les fonctions *open, get_next*, et *close*.

Un SGBD maintient des informations (appelées métadonnées) sur les données dans un ensemble spécial de relations appelées le catalogue (appelées aussi catalogue système ou dictionnaire de données). Le catalogue système contient des informations à propos de chaque relation. Index, et vue. En addition, il contient des statistiques sur les relations et les index. Puisque le catalogue système est lui-même stocké comme un ensemble de relations, on peut utiliser la puissance du SQL pour le mettre en requête et le manipuler.

Les plans alternatifs peuvent différer substantiellement dans leur coût total. Une heuristique est d'appliquer les sélections aussitôt que possible afin de réduire la taille des relations intermédiaires. Les index existants peuvent utilisés comme des chemins d'accès concordants pour une condition de sélection. En addition, lors de la considération du choix d'un algorithme de jointure, l'existence des index sur la relation intérieure affecte le coût de la jointure.

5. Estimation du coût d'un plan

On doit estimer le coût de chaque plan énuméré. Il y a deux étapes dans l'estimation du coût d'un plan d'évaluation pour un bloc de requête :

1. Pour chaque nœud dans l'arbre, on doit estimer le coût d'exécution de l'opération correspondante. Les coûts dépendent significativement de l'utilisation du pipelining ou de relations temporaires pour faire passer la sortie d'un opérateur à son parent.

2. Pour chaque nœud dans l'arbre, on doit estimer la taille du résultat, et savoir s'il sera trié. La taille et l'ordre de tri du résultat du nœud courant affecteront à leur tour l'estimation de la taille, du coût, et l'ordre de tri pour le nœud parent.

L'estimation des coûts nécessite la connaissance de plusieurs paramètres sur les relations d'entrée, comme le nombre de pages et les index disponibles. De telles statistiques sont maintenues dans les catalogues système du SGBD. Dans cette section on décrit les statistiques maintenues par un SGBD typique et on discute de la manière d'estimer les tailles des résultats.

Les estimations utilisées pour les tailles de résultat et les coûts sont au mieux des approximations des tailles et coûts réels. Il est irréaliste d'espérer d'un optimisateur de trouver le meilleur plan ; il est important d'éviter les plus mauvais plans et de trouver un bon plan.

5.1. Estimation de la Taille des Résultats : La Sélectivité

On discute à présent de la manière avec laquelle un optimisateur typique estime la taille du résultat d'un opérateur sur des entrées données. L'estimation de la taille joue un rôle important dans l'estimation du coût, aussi bien du fait que la sortie d'un opérateur peut être l'entrée d'un autre opérateur, que du fait que le coût d'un opérateur dépend de la taille de ses entrées.

Sélectivité des opérateurs :

La sélectivité d'un prédicat est définie comme la fraction des tuples de la relation d'entrée satisfaisant ce prédicat. Ainsi, un prédicat avec une sélectivité *S* sélectionnera *S*CARD(T)* lignes d'une table *T*.

En déterminant le plan d'évaluation pour une requête à une relation impliquant plusieurs prédicats (sous forme conjonctive), l'optimisateur effectue une estimation de la sélectivité sur chaque prédicat. Cette estimation de la sélectivité fournit le nombre approximatif de tuples de la relation cible qui vont satisfaire le prédicat.

En supposant que des index existent pour tous les prédicats impliqués dans la requête, l'optimisateur choisira le prédicat qu'il juge être le plus sélectif (c-à-d, satisfait par le plus petit nombre de tuples), utilisera l'index de ce prédicat pour obtenir les ID des tuples satisfaisant réellement le prédicat, lira ensuite ces tuples et vérifiera enfin qu'ils concordent avec les prédicats restants. Ces derniers tuples composeront le résultat final de la requête.

Considérez un bloc de requête de la forme :

```
SELECT      liste d'attributs
FROM        liste de relation
WHERE       terme1 & terme2 & … & terme3
```

Le nombre maximum de tuples dans le résultat de cette requête (sans élimination des doublons) est le produit des cardinalités des relations dans la clause *FROM*. Chaque terme dans la clause *WHERE* élimine cependant certains tuples potentiels du résultat. On peut modéliser l'effet de la clause *WHERE* sur la taille du résultat en associant un facteur de réduction avec chaque terme, qui est le ratio de la taille (attendue) du résultat sur la taille d'entrée en considérant seulement la sélection représentée par le terme. La taille réelle du résultat peut être estimée comme la taille maximum multipliée par le produit des facteurs de réductions pour les termes dans la clause *WHERE*. Bien sur, cette estimation reflète la supposition – irréaliste mais simplifiée – que les conditions testées par chaque terme sont statistiquement indépendantes.

On considère maintenant le calcul des facteurs de réduction pour différents types de termes dans la clause *WHERE* en utilisant les statistiques disponibles dans les catalogues :

- *colonne = valeur* : Pour un terme de cette forme, le facteur de réduction peut être approximé par *1/NKeys(I)* si il y a un index *I* sur la colonne de la relation en question. Cette formule suppose une distribution uniforme des tuples entre les valeurs de clé de l'index ; cette supposition de distribution uniforme est fréquemment faite lors des estimations de coût dans un optimisateur de requêtes relationnelles typique. S'il n'y a pas d'index sur colonne, l'optimisateur du System R suppose arbitrairement que le facteur de réduction est de *1/10* ! Bien sur, il est possible de maintenir des statistiques comme le nombre de valeurs distinctes présentes pour n'importe quel attribut qu'il y est un index sur cet attribut ou pas. Si de telles statistiques sont maintenues, on peut faire mieux que le choix arbitraire de *1/10*.

- $colonne_1 = colonne_2$: Dans ce cas, le facteur de réduction peut être approximé par $1/MAX(NKeys(I_1), NKeys(I_2))$ si il y a des index I_1 et I_2 sur $colonne_1$ et $colonne_2$, respectivement. Cette formule assume que chaque valeur de clé dans l'index plus petit, disons I_1, a une valeur concordante dans l'autre index. Pour une valeur donnée de la $colonne_1$, on assume que chacune des valeurs de $NKeys(I_2)$ pour $colonne_2$ est aussi probable. Ainsi, le nombre de tuples ayant la même valeur dans $colonne_2$ pour une valeur donnée dans la $colonne_1$ est de $1/NKeys(I_2)$. Si une seulement des deux colonnes a un index, on prend le facteur de réduction comme étant *1/NKeys(I)* ; si aucune colonne n'a d'index, on l'approxime par l'omniprésent *1/10*. Ces formules sont utilisées que les deux formules apparaissent dans la même relation ou pas.

- *colonne > value* : Le facteur de réduction est approximé par *(High(I) – valeur)/ (High(I) – Low(I))* si il y a un index *I* sur *colonne*. Si la colonne n'est pas d'un type arithmétique ou s'il n'y pas d'index, une fraction plus petite qu'un demi est arbitrairement choisie. Des formules similaires pour le facteur de réduction peuvent être dérivées pour d'autres sélections d'étendue.

- *colonne IN (liste de valeurs)* : Le facteur de réduction est pris comme étant le facteur de réduction pour *colonne = valeur* multiplié par le nombre d'articles dans la liste. Cependant, il est permis qu'il soit au plus un demi, reflétant l'idée de l'heuristique que chaque sélection élimine au moins la moitié des tuples candidats.

Ces estimations des facteurs de réduction sont des approximations qui se basent sur les suppositions telles que la distribution uniforme des valeurs et la distribution indépendante des valeurs dans les différentes colonnes. Dans les dernières années, des techniques plus sophistiquées basées sur le stockage de statistiques plus détaillées (ex. histogrammes des valeurs dans une colonne) ont été proposées et sont entrain de trouver leur chemin dans les systèmes commerciaux.

Les facteurs de réduction peuvent aussi être approximés pour les termes de la forme *colonne IN sous requête* (ratio de la taille estimée du résultat de la sous requête sur le nombre de valeurs distinctes dans colonne dans la relation extérieure) ; *NOT condition* (1 – facteur de réduction pour condition) ; $valeur_1 < colonne < valeur_2$; la disjonction des deux conditions ; et ainsi de suite, mais on ne discutera pas sur de tels facteurs de réduction.

On peut estimer la taille du résultat final, sans tenir compte du plan choisi, en prenant le produit des tailles des relations dans la clause *FROM* et les facteurs de réduction pour les termes de la clause *WHERE*. On peut de manière similaire estimer la taille du résultat de chaque opérateur dans un arbre du plan en utilisant les facteurs de réduction, du moment que le sous-arbre dont la racine est le nœud de l'opérateur est lui-même un bloc de requête.

Notez que le nombre de tuples dans le résultat n'est pas affecté par les projections si une élimination de doublons n'est pas effectuée. Cependant, les projections réduisent le nombre de pages dans le résultat puisque les tuples dans le résultat d'une projection sont plus petits que les tuples originaux ; le ratio des tailles des tuples peut être utilisé comme un facteur de réduction pour la projection afin d'estimer la taille du résultat en pages, donnée la taille de la relation d'entrée.

5.2. Statistiques Améliorées : les Histogrammes

Considérez une relation avec *N* tuples et une sélection de la forme *colonne* > *valeur* sur une colonne avec un index *I*. Le facteur de réduction *r* est approximé par *(High(I) – valeur)/(High(I) –Low(I))*, et la taille du résultat est estimé à *r*N*. Cette estimation se base sur la supposition que la distribution des valeurs est uniforme.

Les estimations peuvent être considérablement améliorées en maintenant plus de statistiques détaillées au lieu des valeurs haute et basse dans l'index *I*. Intuitivement, on veut approximer la distribution des valeurs de clé *I* aussi précisément que possible. Considérez les deux distributions de valeurs montrées en Figure 8. La première est distribution non uniforme de valeurs *D* (disons, pour un attribut appelé *age*). La fréquence d'une valeur est le nombre de tuples avec cette valeur d'age ; une distribution est représentée en montrant la fréquence pour chaque valeur possible d'*age*.

Dans notre exemple, la plus petite valeur d'*age* est *0*, la plus grande est *14*, et toutes les valeurs d'*age* enregistrées sont des entiers dans l'intervalle *0* à *14*. La seconde distribution approxime *D* en supposant que chaque valeur d'*age* dans l'intervalle *0* à *14* apparaît aussi souvent dans la collection des tuples. Cette approximation peut être stockée de manière compacte car on a seulement besoin d'enregistrer les valeurs haute et basse de l'intervalle d'*age* (*0* et *14* respectivement) et le compte total de toutes les fréquences (qui est de *45* dans notre exemple).

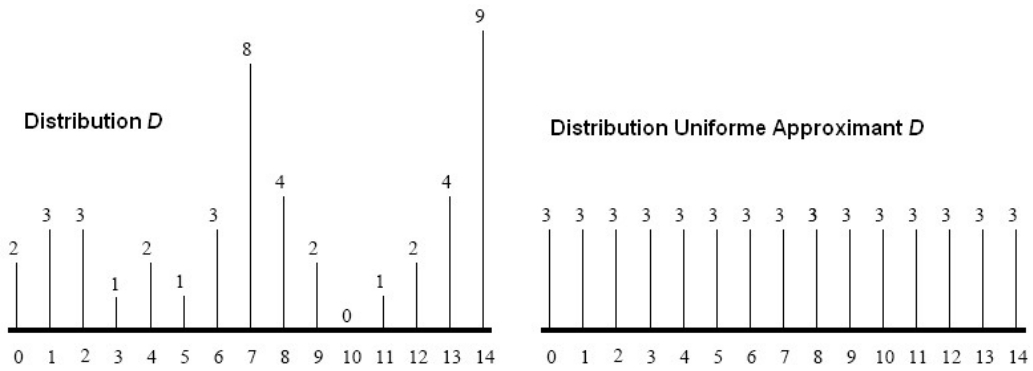


Figure 8 : Distribution Uniforme et Non Uniforme

Prenez comme exemple, la sélection *age > 13*. De la distribution *D* dans la Figure 8, on voit que le résultat a 9 tuples. En utilisant l'approximation de la distribution uniforme, d'un autre coté, on estime la taille du résultat par 1/15 * 45 = 3 tuples. L'estimation est clairement très inexacte.

Un **histogramme** est une structure de données maintenue par un SGBD pour approximer une distribution de données. Dans la Figure 9, on montre comment la distribution de données de la Figure 8 peut être approximée en divisant l'intervalle des valeurs d'âge en sous intervalles appelés seaux (*buckets*), et pour chaque seau, compter le nombre de tuples dont les valeurs d'âge se trouvent à l'intérieur de ce seau. La Figure 9 montre deux types différents d'histogrammes, appelés **équi-largeur** et **équi-profondeur**, respectivement.

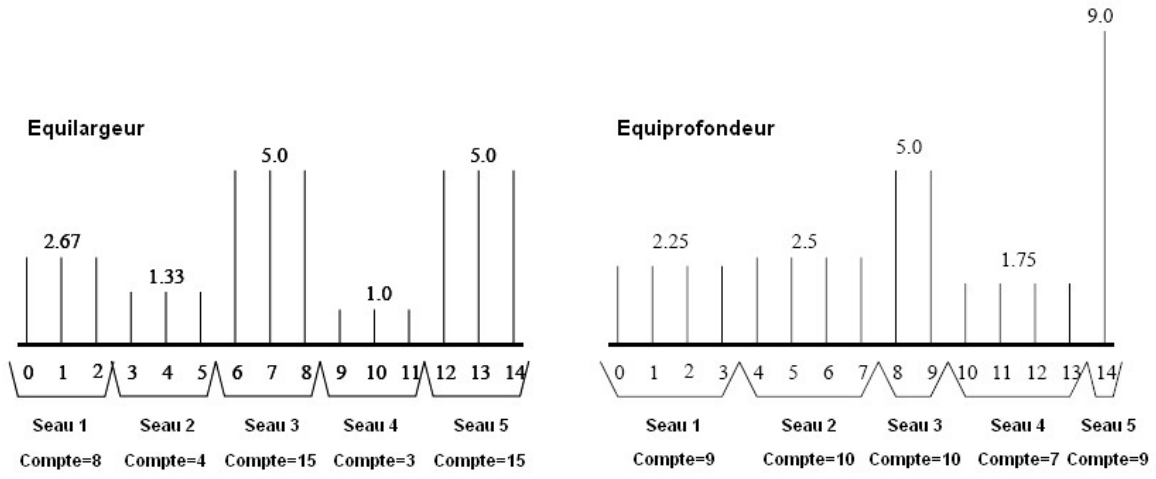


Figure 9 : Histogrammes Approximant la Distribution *D*.

Considérez la sélection *age>13* encore et le premier histogramme (équi-largeur). On peut estimer la taille du résultat à *5* car l'intervalle sélectionné inclut un tiers de l'intervalle du Seau 5. Puisque le *Seau 5* représente un total de *15* tuples, l'intervalle sélectionné correspond à *1/3*15=5* tuples. Comme le montre cet exemple, on suppose que la distribution à l'intérieur d'un seau de l'histogramme est uniforme. Ainsi, lorsqu'on maintient simplement les valeurs haute et basse de l'index *I*, on utilise effectivement un 'histogramme' avec un seul seau. Utiliser des histogrammes avec un petit nombre de seaux à la place, conduit à des estimations plus précises, au coût de quelques centaines d'octets par histogramme. (Comme toutes les statistiques dans un SGBD, les histogrammes sont mis à jour périodiquement, plutôt qu'à chaque fois qu'une donnée change).

Une question importante est la division de l'intervalle de valeurs en seaux. Dans un **histogramme équi-largeur**, on divise l'intervalle en sous-intervalle de taille égale (en termes d'intervalle de valeurs d'*age*). On peut aussi choisir des sous-intervalles tels que les nombres de tuples à l'intérieur de chaque sous intervalle (c-à-d, seau) soient égaux. Un tel histogramme est appelé **histogramme équi-profondeur** et est aussi illustré dans la Figure 9. Considérez encore une fois la sélection *age>13*. En utilisant l'histogramme équi-profondeur, on est mené au *Seau 5*, qui contient seulement la valeur d'*age 15*, et on arrive ainsi à la réponse exacte, *9*.

Tandis que le seau (ou les seaux) significatif contiendra généralement plus d'un tuple, les histogrammes équi-profondeur fournissent de meilleures estimations que les histogrammes équi-largeur. Intuitivement, les seaux avec des valeurs survenant très fréquemment contiennent moins de valeurs, et par conséquent la supposition de distribution uniforme est appliquée à un intervalle plus petit de valeurs, conduisant à des meilleures approximations. Réciproquement, les seaux avec des valeurs surtout non fréquentes sont approximés moins précisément dans un histogramme équi-profondeur, mais pour une bonne estimation, ce sont les valeurs fréquentes qui sont importantes.

Poursuivant davantage avec l'intuition sur l'importance des valeurs fréquentes, une autre alternative est de maintenir séparément les comptes pour un petit nombre de valeurs très fréquentes, disons les valeurs d'*age 7* et *14* dans notre exemple, et de maintenir un histogramme équi-profondeur (ou autre) pour couvrir les valeurs restantes. Un tel histogramme est appelé **histogramme compressé**. La plupart des SGBDs commerciaux utilisent actuellement des histogrammes équi-profondeur, et certains utilisent des histogrammes compressés.

6. Enumération des Plans Alternatifs

On arrive à présent au cœur de l'optimisateur, nommément, l'espace des plans alternatifs considérés pour une requête donnée. Pour cette dernière, l'optimisateur énumère un certain ensemble de plans et choisit le plan avec le plus petit coût estimé. Les équivalences algébriques forment la base pour générer les plans alternatifs, en conjonction avec le choix de la technique d'implémentation des opérateurs relationnels présents dans la requête. Cependant, seulement une partie des plans équivalents est considérée afin de réduire le coût de l'optimisation. Cette section décrit le sous-ensemble de plans considéré par un optimisateur typique.

Il y a deux cas importants à considérer : les requêtes dans lesquelles la clause FROM contient une seule relation et celles dont la clause FROM contient deux ou plusieurs relations.

6.1. Requêtes à une Relation

Si la requête contient une seule relation dans la clause FROM, seules les opérations de sélection, de projection, de groupage, et d'agrégation sont impliquées ; il n'y a pas de jointures. Si de plus, on a seulement une opération de sélection ou de projection ou d'agrégation, les techniques d'implémentation alternatives et les estimations de coût rendent l'optimisation plus simple. A présent, On s'intéresse à l'optimisation des requêtes qui impliquent une combinaison de plusieurs de ces opérations, en utilisant la requête suivante comme exemple :

Pour chaque taux plus grand que 5, imprimer le taux et le nombre de capitaines de 20 ans avec ce taux, sachant qu'il existe au moins deux tels capitaines avec des noms différents.

La version SQL de cette requête :

```
SELECT      S.rating, COUNT (*)
FROM        Sailors S
WHERE       S.rating > 5 AND S.age = 20
GROUP BY    S.rating
HAVING      COUNT DISTINCT (S.sname) > 2
```

En utilisant la notation d'algèbre étendu, on peut écrire cette requête comme :

$$\pi_{S.rating,COUNT(*)}($$
$$HAVING_{COUNT\ DISTINCT\ (S.name)>2}($$
$$GROUP\ BY_{S.rating}($$
$$\pi_{S.rating,S.name}($$
$$\sigma_{S.rating>5 \wedge S.age=20}($$
$$Sailors)))))$$

Notez que *S.sname* est ajouté à la liste de projection, même bien que qu'il n'est pas la clause SELECT, car il est requis pour tester la condition de la clause HAVING.

Nous sommes à présent prêts à discuter des plans qu'un optimisateur aurait à considérer. La décision principale à faire est de définir le chemin d'accès à utiliser dans la récupération des tuples de *Sailors*. Si on ne considère que les sélections, on aurait simplement à choisir le chemin d'accès le plus sélectif suivant les index disponibles concordant les conditions dans la clause WHERE. Pour les opérateurs additionnels dans cette requête, on doit prendre en compte les étapes de tri subséquentes et considérer si ces opérations peuvent être exécutées sans tri en exploitant des index. Deux cas de figures se présentent alors :

6.1.1. Plans Sans Index

L'approche basique dans l'absence d'un index approprié est de scanner la relation *Sailors* et d'appliquer les opérations de sélection et de projection (sans élimination des doublons) à chaque tuple récupéré, comme indiqué par l'expression suivante :

$$\pi_{S.rating,S.name}($$
$$\sigma_{S.rating>5 \wedge S.age=20}($$
$$Sailors))$$

Les tuples résultants sont ensuite triés suivant la clause GROUP BY (sur rating, dans l'exemple), et un seul tuple réponse est généré pour chaque groupe qui coïncide avec la condition dans la clause HAVING.

Le coût de cette approche est composé des coûts de chacune de ces étapes :

1. Réalisation d'un scannage de fichier afin de récupérer les tuples et leur appliquer les sélections et les projections.
2. Ecriture des tuples après ces opérations.
3. Tri de ces tuples afin d'implémenter la clause GROUP BY.

Notez que la clause HAVING ne cause pas d'E/S supplémentaires. Les calculs d'agrégation peuvent être réalisés à la volée lors de la génération des tuples dans chaque groupe à la fin de l'étape de tri pour la clause GROUP BY.

Dans l'exemple, le coût inclut le coût d'un scannage de fichier sur *Sailors* plus le coût de l'écriture des paires <*S.rating, S.sname*> plus le coût de tri par la clause GROUP BY. Le coût du scannage est *NPages(Sailors)*, qui est de 500 E/S, et le coût de l'écriture est de *NPages(Sailors)* fois le ratio de la taille d'une paire sur la taille d'un tuple de *Sailors* fois les facteurs de réduction des deux conditions de sélection. Le taux de la taille du tuple résultant est d'environ 0.8, la sélection *rating* a un facteur de réduction de 0.5 et on utilise le facteur par défaut de 0.1 pour la sélection *age*. Ainsi, le coût de cette étape est égal à 20 E/S. Le coût du tri de cette relation intermédiaire (appelée *Temp*) peut être estimé à *3*NPages(Temp)*, ou 60 E/S, si on suppose que suffisamment de pages sont disponibles dans la zone tampon pour la trier en deux passes, pour simplifier l'estimation des coûts du tri. Le coût total de la requête exemple est donc de 500 + 20 + 60 = 580 E/S.

6.1.2. Plans Utilisant un Index

Les index peuvent être utilisés de différentes manières et peuvent mener à des plans significativement plus rapides que n'importe quel plan sans index.

1. **Chemin d'accès à un seul index** : Si plusieurs index correspondent aux conditions de sélection dans la clause WHERE, chacun de ces index offre un chemin d'accès alternatif. Un optimisateur peut choisir le chemin d'accès qu'il estime permettre de récupérer le moins de pages, appliquer n'importe quelles projections et termes de sélection non primaire (c-à-d, les parties de la condition de sélection qui ne concordent

pas avec l'index), et calculer ensuite les opérations de groupage et d'agrégation (en triant les attributs du GROUP BY).

2. **Chemin d'accès à index multiple** : si plusieurs index utilisant les Alternatives (2) et (3) pour les entrées de données coïncident avec la condition de sélection, chacun de ses index peut être utilisé pour récupérer un ensemble de *rids*. On peut effectuer l'intersection de ces ensembles de *rids*, et trier ensuite le résultat par id de page (en supposant que la représentation du *rid* inclut l'*id* de page) et récupérer les tuples qui satisfont les termes de sélection primaire de tous les index correspondants. N'importe quelles projections et termes de sélection non primaires peuvent être appliqués, suivies par les opérations de groupage et d'agrégation.

3. **Chemin d'accès à index trié** : Si la liste des attributs de groupage est un préfix d'un index d'arbre, l'index peut être utilisé pour récupérer les tuples dans l'ordre requis par la clause GROUP BY. Toutes les conditions de sélection peuvent être appliquées sur chaque tuple récupéré, les champs non désirés peuvent être enlevés, et les opérations d'agrégation calculées pour chaque groupe. Cette stratégie fonctionne bien pour les index clusterés.

4. **Les chemins d'accès à index seul** : Si tous les attributs mentionnés dans la requête sont inclus dans la clé de recherche pour un certain index *dense* sur la relation dans la clause FROM, un **scannage à index seul** peut être utilisé pour calculer les réponses. Puisque les entrées de données dans l'index contiennent tous les attributs d'un tuple dont on a besoin pour cette requête, et il y a une entrée d'index par tuple, on n'aura jamais besoin de récupérer les tuples réels depuis la relation. Si l'index coïncide avec la sélection, on a juste besoin d'examiner un sous-ensemble des entrées de l'index ; autrement, on doit scanner toutes les entrées. Dans les deux cas, on peut éviter la récupération des enregistrements réels. De plus, si l'index est un index d'arbre et que la liste des attributs dans la clause GROUP BY forme un préfix de la clé d'index, on peut récupérer les entrées de données dans l'ordre nécessaire de cette clause et éviter ainsi le tri.

6.2. Requêtes à Relations Multiples

Les blocs de requêtes qui contiennent deux relations ou plus dans la clause FROM nécessitent des jointures (ou des produits cartésiens). Trouver un bon plan pour de telles requêtes est très important car ces requêtes peuvent être assez chères. Sans tenir compte du plan choisi, la taille du résultat final peut être estimée en effectuant le produit des tailles des relations dans la clause FROM et les facteurs de réduction pour les termes dans la clause WHERE. Mais suivant l'ordre dans lequel les relations sont jointes, des relations intermédiaires d'une grande variété de tailles peuvent être créées, produisant des plans à des coûts très différents.

6.2.1. Plans Profond-Gauche

Considérons une requête de la forme $A \otimes B \otimes C \otimes D$, autrement dit, la jointure naturelle de quatre relations. Deux arbres d'opérateurs algébriques qui sont équivalentes à cette requête sont montrés en Figure 10.

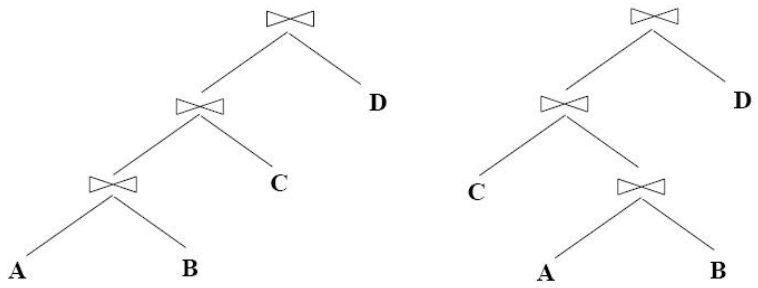


Figure 10 : Deux Arbres de Jointure Linéaire

On note que le fils gauche d'un nœud de jointure est la relation extérieure et le fils droit est la relation intérieure. En ajoutant des détails tels que la méthode de jointure pour chaque nœud de jointure, il est facile d'obtenir plusieurs plans d'évaluation de ces arbres.

Ces deux arbres sont appelés des arbres linéaires. Dans ce type d'arbres, au moins un fils d'un nœud de jointure est une relation de base. Le premier arbre est un exemple d'un arbre profond-gauche, le fils droit de chaque nœud de jointure est une relation de base. Un arbre de jointure qui n'est pas linéaire est dit confus.

Une heuristique fondamentale dans l'optimisateur du système R est d'examiner seulement les arbres profond-gauche dans la construction des plans alternatifs pour une requête de jointure. Bien sur, cette décision exclut beaucoup de plans alternatifs qui pourraient coûter moins cher que le plan optimal en utilisant l'arbre profond-gauche ; on doit se faire à l'idée que l'optimisateur ne trouvera jamais de tels plans. Il y a deux raisons principales pour cette décision :

1. Comme le nombre de jointures augmente, le nombre de plans alternatifs augmente rapidement et un élagage de l'espace des plans alternatifs devient nécessaire.

2. Les arbres profond-gauche permettent de générer des plans entièrement pipelinés, autrement dit, des plans dont les jointures sont toutes évaluées en utilisant le pipelining. Les relations intérieures doivent toujours être complètement matérialisées car on examine la relation intérieure entièrement pour chaque tuple de la relation extérieure. Ainsi, un plan dans lequel une relation intérieure est le résultat d'une jointure nous force à matérialiser son résultat. Bien sur, les plans issus d'arbres gauche-profond ne sont pas tous complètement pipelinés. Par exemple, un plan qui utilise une jointure par tri-fusion peut nécessiter que les tuples extérieurs soient récupérés dans un certain ordre trié, ce qui peut nous forcer à matérialiser la relation extérieure.

Les plans des requêtes à relations multiples sont générés en plusieurs passes. Dans la première passe, tous les plans à une seule relation les moins chers pour chaque ordre de sortie sont générés. La seconde passe génère les plans avec une jointure. Tous les plans générés durant la passe une sont considérés comme les relations extérieures et chaque autre relation comme intérieure. Les passes subséquentes procèdent de manière similaire et génère les plans avec plus de jointures. Ce processus génère finalement un plan qui contient toutes les relations de la requête.

7. Les Sous-Requêtes Emboîtées

L'unité d'optimisation dans un système typique est un **bloc de requête** ; et les requêtes emboîtées l'exploitent en utilisant une certaine forme d'évaluation par boucles imbriquées. Considérons la requête emboîtée suivante : *trouver les noms des capitaines avec le meilleur taux*.

```
SELECT      S.name
FROM        Sailors S
WHERE       S.rating = ( SELECT     MAX(S2.rating)
                         FROM       Sailors S2 )
```

Dans cette simple requête la requête emboîtée peut être évaluée seulement une fois, renvoyant une seule valeur. Cette valeur est incorporée dans la requête de niveau supérieur comme si elle a été une partie de la déclaration originale de la requête.

Cependant, la sous-requête peut renvoyer des fois une relation, ou plus exactement, une table dans le sens SQL. Considérons la requête suivante : *Trouver les noms des capitaines ayant réservé le bateau 103*.

```
SELECT      S.name
FROM        Sailors S
WHERE       S.sid IN ( SELECT       R.sid
                       FROM         Reserves R
                       WHERE        R.bid = 103 )
```

Encore une fois, la sous-requête emboîtée peut être évaluée juste une fois, renvoyant une collection de *sids*. Pour chaque tuple de *Sailors*, on doit maintenant vérifier si la valeur *sid* se trouve dans la collection de *sids* calculée ; cette vérification impose une jointure de *Sailors* et de la collection de *sids*, et on dispose en principe d'une multitude de méthodes de jointure pour le faire. Par exemple, si il y a un index sur le champ *sid* de *Sailors*, une jointure par boucles imbriquées à index avec la collection de *sids* comme relation extérieure et *Sailors* comme celle intérieure, peut être la méthode de jointure la plus efficace. Cependant, dans beaucoup de systèmes, l'optimisateur de requête n'est pas assez développé pour trouver cette

stratégie (une approche commune est de toujours faire une jointure par boucles imbriquées dans laquelle la relation intérieure est la collection de *sids* calculée de la sous-requête).

Cette approche est motivée par le fait qu'elle représente une variante simple de la technique utilisée pour traiter avec *les requêtes corrélées* comme la version suivante de la requête précédente :

```
SELECT      S.name
FROM        Sailors S
WHERE EXISTS ( SELECT   R.sid
               FROM     Reserves R
               WHERE    R.bid = 103
               AND      S.sid = R.sid )
```

Cette requête est *corrélée*: le tuple variable *S* de la requête de niveau supérieure apparaît dans sous-requête emboîtées. Par conséquent, on ne peut évaluer la sous-requête juste une fois. Dans ce cas, la stratégie typique d'évaluation est d'évaluer la sous-requête emboîtée pour chaque tuple de *Sailors*.

Un point important à noter par rapport aux requêtes emboîtées est qu'un optimisateur typique va effectuer un faible travail sur elles, à cause de l'approche limitée de l'optimisation des requêtes emboîtées :

- Dans une requête emboîtée avec corrélation, la méthode de jointure est par boucles imbriquées à index, avec comme relation intérieure une sous-requête (et donc potentiellement chère à calculer). Cette approche crée deux problèmes distincts. Premièrement, la sous-requête emboîtée est évaluée une fois par tuple extérieur ; si la même valeur apparaît dans le champ de corrélation (*S.sid* dans notre exemple) de plusieurs tuples extérieurs, la même sous-requête est évaluée plusieurs fois. Le deuxième problème est que l'approche aux sous-requêtes emboîtées n'est pas orientée ensemble. En effet, une jointure est vue comme un scannage de la relation extérieure avec une sélection sur la sous-requête intérieure pour chaque tuple extérieur. Ceci exclut la considération de méthodes de jointure alternatives telle la jointure par tri-fusion et par hachage, qui peuvent donner des plans supérieurs.

- Même si la jointure par boucles imbriquées à index est la méthode appropriée, l'évaluation de la requête emboîtée peut être inefficace. Par exemple, s'il y a un index sur le champ *sid* de *Reserves*, une bonne stratégie pourrait être d'effectuer une jointure par boucles imbriquées à index avec *Sailors* comme relation extérieure et *Reserves* comme la relation intérieure, et d'appliquer une sélection sur *bid* à la volée. Cependant, cette option n'est pas considérée lors de l'optimisation de la version de la requête utilisant IN car la sous-requête emboîtée est complètement évaluée comme première étape ; donc, les tuples de *Reserves* qui correspondent à la sélection de *bid* sont récupérés en premier.

- Les opportunités de trouver un bon plan d'évaluation peuvent être ratées aussi à cause l'ordonnancement implicite imposé par l'emboîtement. Par exemple, s'il y a un index sur le champ *sid* de *Sailors*, une jointure par boucles emboîtées à index avec *Reserves* comme relation extérieure et *Sailors* comme celle intérieure pourrait être le plan le plus efficace pour notre requête corrélée. Cependant, cette jointure n'est jamais considérée par un optimisateur.

Une requête emboîtée a souvent une requête sans emboîtement équivalente, et une requête corrélée possède souvent une requête équivalente sans corrélation. On a déjà vu des versions corrélées et non corrélée de la requête emboîtée en exemple. Voici aussi une requête équivalente sans emboîtement :

```
SELECT      S.name
FROM        Sailors S, Reserves R
WHERE       S.sid = R.sid AND R.bid = 103
```

Un optimisateur SQL typique trouvera probablement une meilleure stratégie d'évaluation si la version non emboîtée ou non corrélée est donnée au lieu d'une version emboîtée de la requête. Plusieurs optimisateurs actuels ne peuvent pas reconnaître l'équivalence de ces requêtes et transformer une des versions emboîtées à une forme non emboîtée. Ceci revient malheureusement à l'utilisateur.

On conclut notre discussion sur les requêtes emboîtées en observant qu'il pourrait y avoir plusieurs niveaux d'emboîtement. En général, l'approche qu'on a définie est étendue en évaluant de telles requêtes du niveau le plus intérieur au plus extérieur, en ordre et en l'absence de corrélation. Une sous-requête corrélée doit être évaluée pour chaque tuple candidat de la (sous) requête de niveau supérieur qui se réfère à lui. L'idée de base est ainsi similaire au cas des requêtes emboîtées d'un seul niveau.

8. Autres Approches de l'Optimisation de Requêtes

On a décrit l'optimisation de requêtes basée sur une recherche exhaustive d'un large espace de plan pour une requête donnée. L'espace des plans possibles grandit rapidement avec la taille de l'expression de la requête, en particulier par rapport au nombre de jointures. Par conséquent, des heuristiques sont utilisées afin de limiter l'espace considéré. Une heuristique largement utilisée est celle de ne considérer que les plans gauche-profond, laquelle fonctionne bien pour la plupart des requêtes. Cependant, une fois que le nombre de jointures dépasse 15 environ, le coût de l'optimisation utilisant cette approche exhaustive devient très prohibitif.

Pour de telles requêtes complexes, d'autres approches à l'optimisation de requêtes ont été proposées. Elles incluent des **optimisateurs basés sur règles** (*rule-based*), qui utilisent un ensemble de règles pour guider la génération de plans candidats, et ce de manière aléatoire en utilisant des algorithmes probabilistiques pour explorer rapidement un grand espace de plans, avec une probabilité raisonnable de trouver un bon plan.

Les recherche actuelles dans ce domaine impliquent aussi des techniques pour l'estimation de la taille des relations intermédiaires, plus précisément : **l'optimisation paramétrique**, qui recherche des bons plans d'une requête donnée pour chacune des différentes conditions pouvant être rencontrées au moment de l'exécution ; et une **optimisation de requêtes multiples**, dans laquelle l'optimisateur prend en compte l'exécution concurrente de plusieurs requêtes.

9. Optimisation des Requêtes Continues

9.1. Introduction

Les requêtes *continues* [10] sont similaires aux requêtes traditionnelles des bases de données, excepté le fait qu'elles sont émises une fois et exécutées par la suite 'continuellement'. Du fait que des modifications sont constamment apportées à la base de données, de nouveaux enregistrements peuvent concorder avec la requête ou d'autres qui étaient concordants peuvent ne plus l'être.

Un défi de base de la réoptimisation en exécution (*run-time optimization*) est de réordonner les opérateurs pipelinés du traitement de requête pendant qu'ils s'exécutent. Pour modifier un plan de requête à la volée (*on the fly*), une grande quantité d'état dans les différents opérateurs doit être considérée, et les changements arbitraires peuvent exiger un traitement conséquent et une complexité de code afin de garantir des résultats corrects. Par exemple, l'état maintenu par un opérateur comme la jointure à hachage hybride [2] peut devenir aussi grand que la taille d'une relation d'entrée, et exige une modification ou un recalcul si le plan est réordonné pendant que l'état est construit.

En limitant les scénarios dans lesquels on réordonne les opérateurs, on peut réduire ce travail à un minimum. L'étude de la gestion d'état des algorithmes de traitement de requête peut nous orienter sur les approches à adopter pour une réoptimisation à moindre coût et de façon continue.

Comme philosophie, on favorise l'adaptabilité sur la performance du meilleur cas (*adaptivity over best-case performance*). Dans un environnement très variable, le scénario du meilleur cas existe rarement pour une durée de temps significative. On sacrifiera alors les améliorations marginales dans les algorithmes idéalisés du traitement de la requête lorsqu'elles préviennent des réoptimisations fréquentes et inefficaces.

9.2. Opérateurs Unaires

On commence en étudiant le scénario simple du réordonnancement d'un opérateur unaire (une table) pendant l'exécution de la requête. Les opérateurs unaires standards dans un système objet-relationnel incluent des sélections, des projections et des expressions de transformation de sortie ("*Select list*"), tous traités d'une manière simple: ils extraient de façon répétitive un tuple de leur entrée, lui appliquent de la logique (incluant peut-être des fonctions coûteuses définies par l'utilisateur), et produisent éventuellement par la suite un tuple. Ce traitement "tuple à la fois" ne maintient aucun état à travers les tuples: les valeurs d'un tuple d'entrée ne sont pas considérées lors du traitement des tuples subséquents. Du moment que ces opérateurs sont indépendants de l'état (*stateless*), le réordonnancement en exécution (*run-time reordering*) peut être fait trivialement : chaque opérateur peut rester encapsulé, ignorant les changements dans l'ordonnancement. Les opérateurs ne peuvent pas tous commuter, bien sûr, mais les contraintes sur les ordonnancements légaux sont orthogonales à nos préoccupations dans cette section à propos de la gestion d'état. Les contraintes d'ordonnancement ne compliquent pas la tâche de permutation (*swapping*) de deux opérateurs qui sont d'office commutables.

Même les opérateurs unaires dépendant de l'état (*stateful*), sujets à contraintes sur les ordonnancements légaux, comme le tri (*sorting*) et le groupage (*grouping*) peuvent facilement être réordonnés. Les opérateurs de tri et de groupage consomment une relation d'entrée entière avant de produire n'importe quelle sortie. Alors qu'au milieu de la lecture de leurs entrées, ils sont insensibles aux changements des entrées subséquentes; aucun rapport n'est assumé entre les enregistrements d'entrée déjà traités et ceux qui le sont pas encore. Certains réordonnancements ne sont pas permis; par exemple, on ne peut en général, commuter un opérateur de groupage avec une sélection si l'opérateur de groupage calcule un agrégat.

En résumé, lorsque les opérateurs unaires peuvent être réordonnés légalement, la logique de le faire est triviale puisque aucun changement d'état n'est nécessaire. On poursuit en considérant le réordonnancement des jointures, lequel, on le verra, est significativement plus complexe.

9.3. Opérateurs Binaires : Barrières de Synchronisation

Les opérateurs binaires comme la jointure capturent souvent un état significatif. Une forme particulière d'états utilisée dans de tels opérateurs est en rapport avec l'imbrication des requêtes pour des tuples de différentes entrées. Certains algorithmes de jointure doivent exercer des contrôles importants sur l'ordre dans lequel ils consomment les tuples d'une entrée ou d'une autre, pendant que d'autres peuvent consommer des tuples de chaque entrée avec une imbrication arbitraire.

Comme exemple, considérez le cas d'une jointure par fusion (*merge join*) sur deux entrées triées. Pendant le traitement d'une jointure par fusion, le tuple suivant est toujours consommé de la relation dont le dernier tuple avait la valeur inférieure. Ceci contraint significativement l'ordre dans lequel les tuples peuvent être consommés: comme cas extrême, considérez une relation lentement délivrée *R1* avec beaucoup de valeurs basses dans sa colonne de jointure, et une relation à haut débit mais grande *R2* avec seulement des valeurs élevées dans sa colonne de jointure - le traitement de *R2* est différé pour une longue période en consommant beaucoup de tuples de *R1*. Utilisant la terminologie de la programmation parallèle, on décrit ce phénomène comme une **barrière de synchronisation**: un scannage de table doit attendre jusqu'à ce que l'autre scannage de table produise une valeur plus grande que celles vues auparavant.

En général, les barrières limitent la concurrence - et par conséquent la performance - quand deux tâches nécessitent différentes quantités de temps pour s'achever (c-à-d., "arriver" à la barrière). Rappelons que la concurrence apparaît même dans les moteurs de requête d'un site unique, qui peut effectuer simultanément des I/O réseau, I/O disque, et calcul. Il est ainsi souhaitable de minimiser la surcharge des barrières de synchronisation dans un environnement de performance dynamique (ou même statique mais hétérogène). Deux sujets affectent la surcharge des barrières dans un plan: la fréquence de barrières, et l'intervalle entre les temps d'arrivée des deux entrées à la barrière.

Nous verrons dans une discussion prochaine que les barrières peuvent souvent être évitées ou au moins minimisées en utilisant des algorithmes de jointure appropriés.

9.4. Opérateurs binaires : Moments de Symétrie

Notez que la barrière de synchronisation dans la jointure par fusion est déclarée de manière indépendante de l'ordre: elle ne distingue pas entre les entrées basées sur toute propriété autre que celle des données qu'elles délivrent. Ainsi, la jointure par fusion est souvent décrite comme un opérateur symétrique, puisque ses deux entrées sont traitées uniformément. Ce n'est pas le cas de beaucoup d'autres algorithmes de jointure. Considérez la jointure traditionnelle par boucles emboîtées, par exemple. La relation "extérieure" dans cette jointure est synchronisée avec la relation "intérieure", mais pas vice versa: après chaque consommation de tuple (ou de bloc de tuples) de la relation externe, une barrière est mise en place jusqu'à ce qu'un scannage complet de la relation intérieure soit effectué. Pour les opérateurs asymétriques comme la jointure par boucles emboîtées, des gains en performance peuvent souvent être obtenus en réordonnant les entrées.

Lorsqu'un algorithme de jointure atteint une barrière, il déclare la fin d'une dépendance de planification entre ses deux relations d'entrée. Dans de tels cas, l'ordre des entrées à la jointure peut souvent être changé sans la modification d'aucun état dans la jointure ; lorsque c'est le cas, on fait référence à la barrière comme un **moment de symétrie**.
Si on retourne à l'exemple d'une jointure par boucles emboîtées, avec comme relation extérieure *R* et celle intérieure *S*. A la barrière, la jointure a complété une boucle intérieure complète, ayant joint chaque tuple dans un sous-ensemble de *R* avec chaque tuple dans *S*. Le réordonnancement des entrées à ce point peut être fait sans affecter l'algorithme de jointure, aussi longtemps que l'itérateur produisant *R* note la position de son curseur courant c_R. Dans ce cas, la nouvelle boucle extérieure sur *S* démarre le rescannage en cherchant le premier tuple de *S*, et *R* est scannée de c_R à la fin. Ceci peut se répéter indéfiniment, en joignant les tuples de *S* avec tous les tuples de *R* de la position c_R à la fin. Alternativement, à la fin d'une certaine boucle sur *R* (durant un moment de symétrie), l'ordre des entrées peut être permuté encore en se rappelant de la position courante de *S*, et en joignant de manière répétitive le tuple suivant dans *R* (démarrant de c_R) avec les tuples de *S* entre c_S et la fin. La Figure 11 schématise ce scénario, avec deux changements d'ordonnancement. Certains opérateurs comme la jointure pipelinée par hachage de [3] n'a pas de barrière *whatsover*. Ces opérateurs sont dans une symétrie constante, du moment que le traitement des deux entrées est totalement découplé.

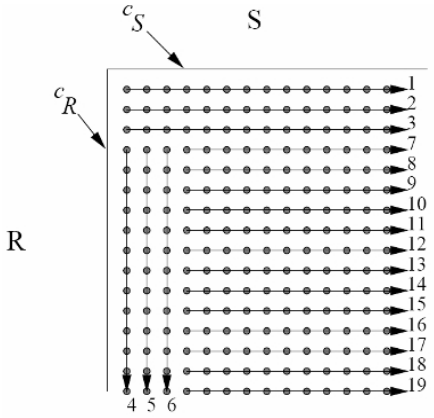


Figure 11 : Tuples produits par une jointure par boucles emboîtées, réordonnée à deux moments de symétrie. Chaque axe représente les tuples de la relation correspondante, dans l'ordre dans lequel ils sont délivrés par une méthode d'accès. Les points représentent les tuples générés par la jointure, dont quelques-uns peuvent être éliminé par le prédicat de jointure. Les nombres correspondent aux barrières atteintes, en ordre. Les c_R et c_S sont les positions de curseur maintenues par les entrées correspondantes lors des réordonnancements.

Les moments de symétrie permettent le réordonnancement des entrées à un opérateur binaire seul. Pour généraliser quelque peu ceci, notons que puisque les jointures commutent, un arbre de *n-1* jointures binaires peut être vu comme une seule jointure *n*-aire. On pourrait facilement implémenter un opérateur de jointure à double boucle emboîtée sur les relations *R*, *S* et *T*, et il y aurait des moments de symétrie complète à la fin de chaque boucle de *S*. À ce point, toutes les trois entrées pourraient être réordonnées (disons *T* ensuite *R* ensuite *S*) avec une extension simple à la discussion au-dessus: un curseur serait enregistré pour chaque entrée, et chaque boucle irait de la position du curseur enregistrée à la fin de l'entrée.

Le même effet peut être obtenu dans une implémentation binaire avec deux opérateurs, en commutant les positions des opérateurs binaires: effectivement, la transformation de l'arbre du plan doit se faire en étapes, de *(R X_1 S) X_2 T* à *(R. X_2 T) X_1 S* et ensuite à *(T X_2 R) X_1 S*. Cette approche traite un opérateur et une de ses entrées comme une unité (par exemple, l'unité *[X_2 T]*), et commute les unités; l'idée a été utilisée précédemment dans les plans statiques d'optimisation de requête [4, 9, 8]. En regardant la situation de cette manière, on peut naturellement considérer le réordonnancement de multiples jointures et de leurs entrées, même si les algorithmes de jointure sont des différents. Dans notre requête *(R X_1 S) X_2 T*, on a besoin que *[X_1 S]* et *[X_2 T]* soient mutuellement commutatifs, mais il n'est pas obligatoire qu'ils soient du même algorithme.

Notons que la combinaison de la commutativité et des moments de symétrie autorisent un ordonnancement très agressif d'un arbre de plan. Un opérateur du *n*-aire seul représentant un arbre de plan réordonnable est par conséquent une abstraction attirante, du moment qu'il encapsule tout ordonnancement pouvant être sujet à modification.

9.5. Réordonnabilité des Algorithmes de Jointure

Il y a plusieurs algorithmes de jointure dans la littérature, y compris ceux utilisés communément dans les systèmes actuels, ainsi que des alternatives pertinentes proposées pour le traitement "en ligne" [5] et celui adaptatif [6, 7]. Une partie des travaux antérieurs sur la mise en requête des tables externes s'est focalisée exclusivement sur la jointure pipelinée par hachage [3], et l'a trouvé très efficace; ceci a été attribué à son manque de barrières, sans influencer sa symétrie constante pour le réordonnancement. Cependant, la jointure par hachage n'est pas toujours appropriée, puisqu'elle n'exploite pas d'index et fonctionne seulement pour les attributs d'équi-jointure.

Dans cette section, on expose beaucoup d'algorithmes de jointure de la discussion au-dessus, en mettant en valeur une classe générale d'algorithmes autorisant un réordonnancement flexible en exécution, et des barrières adaptatives ou inexistantes :

9.5.1. Blocking Vs. Pipeling

On peut grossièrement diviser des méthodes standard du traitement de requête en algorithmes de *blocking* et de *pipelining*. Les algorithmes bloquants ne produisent aucune sortie jusqu'à ce qu'ils consomment une ou les deux relations d'entrée; l'exemple binaire le plus commun est la jointure par hachage hybride. Les algorithmes de pipelining peuvent produire une sortie en lisant en concurrence de leurs entrées. Les exemples de jointures par pipelining incluent aussi bien que les jointures par fusion, que la famille des jointures par onde "*Ripple Join*" [5] généralisant les jointures par boucles emboîtées, que les jointures pipelinées par hachage [3], et des variantes des deux. Les blocs dans un pipeline représentent les barrières entre un opérateur et tous les opérateurs qui le suivent dans le flot de données.

Cette barrière est un moment de symétrie (trivial) pour les opérateurs subséquents, puisqu'ils ne peuvent pas commencer jusqu'à ce que la barrière soit traversée. Du moment que les opérateurs après le bloc n'ont pas commencé leur traitement, la symétrie après un bloc est analogue à une optimisation statique d'une (sous) requête avant de l'exécuter; ce problème est traité dans [38, 7]. On se concentrera ici strictement sur les algorithmes à pipelining, qui peuvent être réordonnés agressivement sans de longues attentes pour l'achèvements des barrières (comme la barrière d'un opérateur de bloquant).

9.5.2. Jointures et Index :

Les jointures par boucles emboîtées peuvent exploiter les index sur la relation intérieure, produisant des algorithmes de jointure pipelinés nettement efficaces. Une jointure indexée par boucles emboîtées (désormais une "jointure par index") est fondamentalement asymétrique, car une relation d'entrée a été préindexée. Même lorsque les index existent sur les deux entrées, changer le choix de relation intérieure et extérieure "à la volée" est problématique. Ainsi, pour les besoins du réordonnancement, il est plus simple d'imaginer une jointure par index comme un genre d'opérateur de sélection unaire sur l'entrée non indexée. La seule distinction entre une jointure par index et une sélection est que -en respect de la relation non indexée- la sélectivité du nœud de jointure peut être plus grande que 1. Bien qu'on ne puisse pas commuter les entrées d'une jointure par boucles emboîtées seule, on peut réordonner une jointure à boucles emboîtées et sa relation indexée comme une unité parmi les autres opérateurs dans un arbre de plan. Notons que la logique pour les index peut être appliquée à des tables externes qui exigent que les liaisons soient passées; de telles tables peuvent être des passerelles à, par exemple, des pages de Web avec des formes, des systèmes d'index GIS, des serveurs LDAP et ainsi de suite.

9.5.3. Propriétés Physiques, Prédicats de Jointure et Commutativité

Clairement, un choix du pré-optimiseur d'un algorithme de jointure par index restreint les ordonnancements de jointure possibles. Dans la perspective d'une jointure n-aire, une contrainte d'ordonnancement doit être imposée afin que l'entrée non indexée de la jointure

soit ordonnée avant (mais pas nécessairement directement avant) l'entrée indexée. Cette contrainte apparaît à cause d'une propriété physique d'une relation de l'entrée: les index peuvent être sondés mais pas scanné, et par conséquent ne peuvent apparaître avant de leurs tables sondées. Des contraintes semblables mais plus étendues peuvent apparaître lorsqu'on conserve les entrées ordonnées pour une jointure à fusion.

L'applicabilité de certains algorithmes de jointure fait apparaître des contraintes supplémentaires. Beaucoup d'algorithmes de jointure fonctionnent seulement pour des équi-jointures, et pas avec d'autres jointures comme les produits cartésiens. De tels algorithmes restreignent aussi les réordonnancements sur l'arbre de plan, du moment qu'ils exigent que toutes les relations mentionnées dans leurs prédicats d'équi-jointure soient toujours exécutées avant eux sur chaque tuple. Dans ce papier, on considère que les contraintes d'ordonnancement comme étant un aspect inviolable d'un arbre de plan, et on s'assure qu'elles soient toujours sauvegardées.

9.5.4. Algorithmes de Jointure et Réordonnancement

Algorithme	**Moments de Symétrie**	**Barrières d'Entrée**	**Contraintes d'Ordre**
Jointure par Hachage hybride	Aucun	Aucune	Pas de produits-X
Jointure par Fusion	A la fin de chaque boucle emboîtée sur les doublons	Dépendantes des données	Entrées triées, Pas de produits-X
Boucles Emboîtées	A la fin de chaque boucle intérieure	A la fin de chaque boucle intérieure	Aucune
Ondulation à Bloc	A chaque coin	A chaque coin, mais coins choisis de manière adaptative	Aucune
Jointure par Index	A la fin de chaque parcours d'index	A la fin de chaque parcours d'index	Les extérieures précèdent les intérieures, Pas de produits-X
Ondulation à Hachage (+ Jointure par Hachage Pipeliné, Jointure-X)	Après chaque tuple	Aucune	Pas de produits-X

Table 1: Propriétés de Réordonnabilité des Divers Algorithmes de Jointure.

Dans le but d'améliorer l'efficacité, on favorise les algorithmes de jointure avec des moments de symétrie fréquents, des barrières adaptatives ou inexistantes, et un minimum de contraintes d'ordonnancement : ces algorithmes offrent les meilleures opportunités de réoptimisation. La Table 1 résume les propriétés saillantes d'une variété d'algorithmes de jointure. Notre souhait d'éviter les règles bloquantes exclue l'usage de la jointure par hachage hybride, et celui de minimiser les contraintes d'ordonnancement et les barrières exclue les jointures par fusion. Les jointures par boucles emboîtées ont des moments de symétrie peu fréquents et des barrières déséquilibrées, les rendant indésirables aussi.

Tous les algorithmes restants qu'on considère sont basés sur des versions fréquemment symétriques des plans par itération traditionnelle, par hachage et par indexation, ex. la Jointure par Ondulation [5]. Notons que la jointure par hachage pipeliné originale de [3] est une version restreinte de la jointure par ondulation à hachage. Les extensions de hachage externe de [6, 7] sont applicables à la jointure par ondulation à hachage à taux variable, et [5] capture les jointures par index comme un cas spécial aussi.

Pour les non équi-jointures, l'algorithme par ondulation à bloc est efficace, ayant des moments de symétrie assez fréquents, particulièrement au début du traitement [5]. La Figure 12 illustre les jointures par ondulation à bloc, à index et à hachage; le lecteur est reporté à [5, 7, 6] pour des discussions détaillées de ces algorithmes et de leurs variantes. Ces algorithmes sont adaptatifs sans pour autant sacrifier beaucoup de performance: [6] et [7] présentent des versions scalables de jointure par ondulation à hachage s'exécutant de manière compétitive avec la jointure par hachage hybride dans le cas statique; [5] montre que pendant que la jointure par ondulation à bloc peut être moins efficace que la jointure par boucles emboîtées, elle arrive aux moments de symétrie beaucoup plus fréquemment que la jointure par boucles emboîtées, surtout dans les premières étapes du traitement.

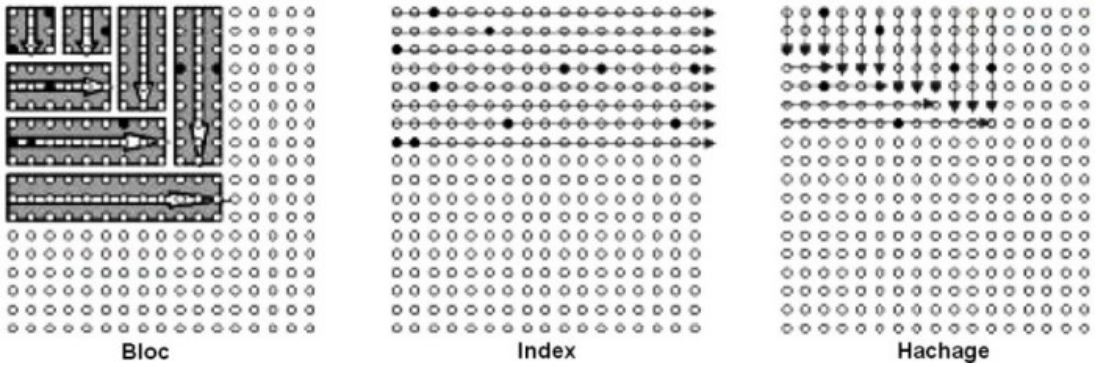


Figure 12 : Tuples générés par la jointure par ondulation à bloc, la jointure par index, et par ondulation à hachage. Dans le diagramme de l'ondulation par hachage, une relation arrive 3x plus rapidement que l'autre. Chaque axe représente les tuples d'une relation d'entrée, dans l'ordre dont ils sont délivrés par une méthode de l'accès. Dans la jointure par ondulation à bloc, tous les tuples sont produits par la jointure, mais certains peuvent être éliminés par le prédicat de jointure. Dans les jointures par index et par ondulation à hachage, les seuls tuples produits par la jointure sont ceux satisfaisant les prédicats de jointure; ceux-ci sont indiqués par les points noirs. Les flèches pour la jointure par index et par ondulation à hachage représentent la portion *logique* de l'espace du produit transversal vérifiée plus loin; l'index et les tables de hachage déploient le travail seulement sur les tuples qui satisfaisant les prédicats de jointure.

Les jointures par ondulation ont des moments de symétrie à chaque "coin" d'une ondulation rectangulaire dans la Figure 12, c.-à-d., à chaque fois qu'un préfix d'un flux entrant *R* a été joint avec tous les tuples dans d'un préfix de flux entrant *S* et vice versa. Pour les jointures par ondulation à hachage (comme la jointure pipélinée par hachage) et les jointures par index, ce scénario se produit entre chaque tuple consécutif consommé d'une entrée scannée. Ainsi, les jointures par ondulation offrent de très fréquents moments de symétrie.

Les jointures par ondulation sont attractives en ce qui concerne les barrières aussi. Les jointures par ondulation ont été conçues pour permettre des taux changeants pour chaque entrée; cela a été utilisé à l'origine pour déployer pro activement plus de traitement sur la relation d'entrée avec le plus d'influence statistique sur les résultats intermédiaires. Cependant, le même mécanisme autorise une adaptabilité réactive dans le scénario à large échelle: une barrière est atteinte à chaque coin, et le prochain coin peut refléter de manière adaptative les taux relatifs des deux entrées. Pour la jointure par ondulation à bloc, le coin suivant est choisi à l'atteinte du coin précédent; cela peut être fait de manière adaptative afin de refléter les taux relatifs des deux entrées avec le temps.

La famille de jointure par ondulation offre des caractéristiques d'adaptabilité attirantes à des coûts de surcharge modestes sur la performance et l'encombrement de la mémoire. Ils épousent par conséquent bien notre philosophie de sacrifice d'une vitesse marginale au profit de l'adaptabilité. Il est vrai que les exigences de mémoire supplémentaires peuvent être problématiques dans des cas extrêmes, ajoutant parfois des passes supplémentaires sur les données pour accommoder les débordements. On croit que les tendances technologiques

justifient la dépense de la mémoire pour maximiser la faculté d'adaptation, mais on peut mélanger des opérateurs bloquants avec des pipelines si nécessaire.

9.6. Conclusion

Les ressources des systèmes de gestion des bases de données peuvent présenter des caractéristiques très fluctuantes. Les suppositions faites à l'instant où une requête continue est soumise resteront valables rarement pendant la durée du traitement de la requête, ce qui rend les techniques traditionnelles d'optimisation statique et d'exécution inefficaces pour ce type de requêtes.

La réoptimisation en exécution des requêtes continues exploite le concept de réordonnabilité des opérateurs composant le plan de requête pour le garder optimal. Suivant les opérateurs impliqués et l'état en cours, des traitements, parfois assez conséquents, doivent être considérés afin de garantir des résultats corrects. Néanmoins, l'environnement spatio-temporel est très variable et le scénario du meilleur cas existe rarement pour une durée de temps significative. L'adaptabilité doit alors être favorisée à la performance. Les améliorations marginales dans les algorithmes idéalisés du traitement de la requête sont, par conséquent, sacrifiées lorsqu'elles préviennent des réoptimisations fréquentes et peu efficaces.

Trois propriétés peuvent varier durant le traitement d'une requête spatio-temporelle continue : le coût des opérateurs spatio-temporels, leurs sélectivités, et les taux avec lesquels les tuples arrivent à partir des entrées. Le premier et le troisième problème ont été bien étudiés dans la littérature pour les opérateurs standards [38, 7] et peuvent facilement être superposés aux opérateurs spatiaux. Cependant, les variations de la sélectivité en exécution n'ont pas été beaucoup abordées. Les sélectivités peuvent, en effet, changer durant l'exécution, et des techniques de réoptimisation des opérateurs pipelinés à l'exécution peuvent tirer profit de ce fait pour changer de plan vers le plus approprié durant l'exécution de la requête.

Le chapitre suivant va introduire l'estimation de cette sélectivité pour les opérateurs spatio-temporels et présentera les grandes deux approches utilisées pour la gérer.

Histogrammes Spatio-temporels

1. Estimation de la Sélectivité Spatio-temporelle

La connaissance de la sélectivité d'un prédicat, définie comme la fraction des lignes du jeu d'entrée du prédicat qui le satisfont, est primordiale pour l'optimisateur de requêtes. Ce dernier utilise des statistiques pour estimer la sélectivité des expressions, et donc la taille des résultats intermédiaires et finaux des requêtes. De bonnes statistiques permettent à l'optimiseur d'évaluer de manière précise le coût des différents plans de requête, et de choisir, en conséquence, un plan de qualité élevée.

A cet effet, plusieurs techniques ont été proposées pour calculer les estimations des tailles des résultats de requêtes dans les bases de données relationnelles. Les plus communes utilisent les *histogrammes* (présentés précédemment en section 4.4.2), *l'échantillonnage*, ou sont basées sur des *techniques paramétriques* modélisant les données par une distribution mathématique standard. De ces différentes techniques, les histogrammes, en particulier, ont démontré leur popularité dans les systèmes de bases de données car ils peuvent être calculés facilement, utilisent peu d'espace (de l'ordre de quelques centaines d'octets par relation), et n'imposent pas une connaissance préalable de la distribution de l'entrée.

On peut facilement superposer cette vision relative aux bases de données relationnelles, aux bases de données spatio-temporelles. Néanmoins, le problème de l'estimation de la sélectivité impliquant des données spatiales (et/ou temporelles) est très différent de ceux de l'estimation de la sélectivité dans les bases de données relationnelles, à cause notamment de la nature des données manipulées et de la dépendance aux structures de données utilisées.

Ainsi, beaucoup d'efforts de recherche se sont concentrés, récemment, sur l'adaptation de ces techniques d'estimation de la sélectivité aux différentes opérations spatio-temporelles :

Les *histogrammes spatio-temporels*, adaptés à partir des histogrammes standards, ont été proposés, au début, pour fournir l'estimation de la sélectivité pour les requêtes spatio-

temporelles prédictives [32]. Le principal centre d'intérêt étant les objets mouvants dans une dimension. L'estimation de la sélectivité des objets multidimensionnels est calculée en multipliant les estimations de sélectivité de chaque dimension. Une idée similaire dans le contexte de requêtes spatio-temporelles prédictives est introduite en [33]. Cependant, son contexte principal utilise des rectangles mouvants multidimensionnels. D'autres travaux sur l'estimation de la sélectivité des requêtes spatio-temporelles s'appuient sur la transformation de dualité [34], l'existence d'une structure d'index secondaire [34], ou des approches par *clustering* [35]. Les approches

La deuxième approche pour l'estimation de la sélectivité spatio-temporelle est l'échantillonnage. Le *Venn Sampling* [40] est une technique d'échantillonnage qui n'est pas basée sur les histogrammes et qui vise à réduire le nombre d'échantillons nécessaire à une estimation *parfaite*. L'idée est de permettre à chaque objet mouvant d'être au courant d'un ensemble de *requêtes pivots*. Les objets mouvants mettent à jour leurs positions et vitesses seulement lorsqu'ils commencent/cessent de satisfaire les requêtes pivots.

D'autres approches se sont focalisées sur le traitement de l'estimation de la sélectivité dans les diverses structures de données spatio-temporelles utilisées, ex. estimation de la sélectivité pour *QuadTree* [29], estimation de la sélectivité pour *R-Tree* [30], et l'estimation de la sélectivité pour les points de données [31].

2. Principe et Architecture

Les histogrammes traditionnels ont été largement utilisés comme un moyen d'estimation de la sélectivité dans les bases de données relationnelles. Actuellement, les histogrammes sont dirigés par requête (*query-driven*, [36, 37, 38]). L'idée principale est d'utiliser un *feedback* (retour) depuis le moteur d'exécution de requête afin d'estimer la distribution des données. Ainsi, le coût de la construction des histogrammes est réduit puisqu'ils sont construits durant le processus régulier d'exécution des requêtes.

Les *histogrammes spatio-temporels ou ST* démarrent par une estimation initiale de la *sélectivité spatio-temporelle* de l'espace en question. La précision de l'estimation initiale est continuellement augmentée en se basant sur la surveillance de l'exécution des requêtes spatio-temporelles en cours. Une des caractéristiques attractives d'un histogramme ST est que sa précision (et par conséquent l'efficacité de l'exécution des requêtes) s'accroît avec l'augmentation du nombre de requêtes continues en cours. De plus, l'histogramme ST peut utiliser certaines techniques de datamining pour la détection des périodicités afin de fournir une meilleure sélectivité spatio-temporelle avec moins de surcharge.

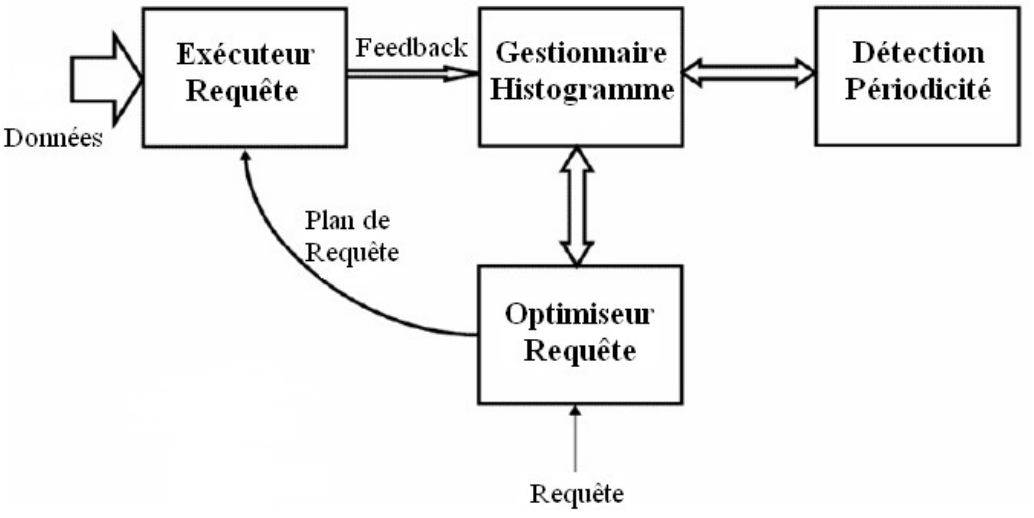


Figure 1 : Architecture

La Figure 1 donne un aperçu de l'architecture utilisée. Les requêtes spatio-temporelles sont soumises à l'optimisateur de requête pour générer les plans d'exécution adéquats. L'optimisateur de requête (ex. *System R*) sélectionne le meilleur plan d'exécution en se basant sur le coût total. Les histogrammes ST fournissent l'optimisateur de requête avec les estimations de sélectivité utilisées dans le calcul du coût total. Durant l'exécution d'une requête spatio-temporelle, les feedbacks sont envoyés au gestionnaire d'histogramme. Ces feedbacks représentent la sélectivité réelle de la requête qui vient d'être exécutée; c-à-d, la fraction des données d'entrée faisant partie de la réponse de la requête.

Le gestionnaire d'histogramme utilise ces feedbacks pour affiner en ligne les estimations de sélectivité. L'affinage en ligne sert aussi bien les nouvelles requêtes qui arrivent que les requêtes continues en cours. Les nouvelles requêtes trouvent un histogramme plus précis, tandis que les plans d'exécution des requêtes continues en cours peuvent être modifiés en conséquence et ce de manière adaptative lorsque l'environnement change.

3. Exécuteur de Requête

L'espace est mappé en une grille *N***N*. Lorsqu'un objet mobile s'enregistre dans une cellule de la grille, il envoie des mises à jour périodiques de sa position. Par conséquent, chaque cellule de grille est la seule sensible aux objets se trouvant à l'intérieur d'elle. Aussi, quand une requête s'enregistre avec le système, les cellules de grille se chevauchant avec elle sont notifiées. Le processus exécutant cette requête est notifié seulement lorsqu'un changement sur les objets en mouvement survient dans ces cellules. En d'autres termes, chaque cellule est sensible seulement aux objets se trouvant à l'intérieur d'elle et aux requêtes se chevauchant avec elle.

L'exécuteur de requête utilise le plan fourni par l'optimisateur pour exécuter la requête sur les données d'entrée (Figure 1). Chaque opérateur garde trace du ratio du nombre de ses tuples sortants sur le nombre de ses tuples entrants. Périodiquement, ce ratio est reporté au gestionnaire d'histogramme comme la sélectivité de l'opérateur en question.

4. Gestionnaire d'Histogramme

On ne peut aborder le stockage des données entrantes pour les applications basées sur des flux (*streaming applications*). Le modèle de requête continue ne permet pas de scanner la totalité des données dans le but de construire l'histogramme. En fait, un histogramme ST est construit et affiné progressivement. On utilise les feedbacks du résultat de requête pour mettre à jour en ligne l'histogramme spatio-temporel. Périodiquement, chaque opérateur reporte la sélectivité réelle de son étendue surveillée. Ces statistiques sont inhérentes au calcul dans les opérateurs. Ils n'imposent pas de surcharge additionnelle sur l'exécuteur de requête.

Définition 1. ***Cellule sombre :*** *est une cellule de grille correspondant à une région avec laquelle aucune requête ne se chevauche.*

Définition 2. ***Cellule éclairée :*** *est une cellule de grille correspondant à une région avec laquelle une ou plusieurs requêtes se chevauchent.*

Initialement, l'espace entier est supposé être sombre, où l'obscurité représente l'ignorance de la sélectivité. Les requêtes agissent comme des spots de lumière. Elles illuminent une région avec leur feedback sur sa sélectivité. On distribue cette sélectivité uniformément sur la région éclairée. Pour les cellules sombres restantes, on peut consulter la technique en ligne de mining de périodicité. Si une région présente une sorte de périodicité, sa sélectivité courante peut être estimée suivant une telle périodicité. En d'autres termes, le comportement périodique d'une région nuance celle-ci avec une petite lumière lorsque la cellule de grille correspondante est actuellement sombre. Les régions ne se trouvant pas à l'intérieur de régions de requête et ne présentant aucune périodicité resteront sombres. Les sélectivités des régions sombres restantes sont estimées de manière à complémenter les sélectivités de ces régions éclairées et nuancées. Cette estimation est uniformément distribuée sur les seaux de l'histogramme ST correspondants aux régions sombres.

L'histogramme ST est représenté par un tableau à deux dimensions. Chaque élément du tableau contient la sélectivité de la cellule correspondante de l'histogramme. On suppose qu'une variable maintient le nombre total des objets mobiles dans la base de données.

4.1. Construction de l'Histogramme

L'histogramme ST est d'une taille de *N*N* cellules de grille. La grille divise l'univers uniformément en un nombre de cellules disjointes. On dénote la vue courante de l'histogramme ST par *H*, où *H[r, c]* est l'estimation de la sélectivité de la cellule de grille *G[r, c]*. Démarrant avec toutes les cellules de la grille étant sombres, l'estimation de la sélectivité de chaque seau est initialisée uniformément suivant l'Equation 1. Avec les feedbacks successifs des opérateurs, de meilleures estimations de sélectivité sont obtenues dû à une vue plus claire de l'aire couverte.

$$H[r,c] = \frac{1}{N^2} \quad \forall r,c \in \{1,2,...,N\} \quad (1)$$

Une requête *q* est représentée par une région rectangulaire rectiligne R_q. Soit $F_q(r, c)$ une fonction scalaire renvoyant la fraction de la cellule de grille *G[r, c]* couverte par R_q. Par conséquent, l'estimation de la sélectivité d'une requête q peut être calculée par l'Equation 2.

$$EstSel(q) = \sum_{i=1}^{N} \sum_{j=1}^{N} H[i, j] * F_q(i, j) \quad (2)$$

De manière similaire, soit $F_{sombre}(i, j)$ la fraction sombre de *G[r, c]*. Ainsi, l'estimation de la sélectivité de l'aire sombre totale est calculée par l'Equation 3.

$$EstSel(sombre) = \sum_{i=1}^{N} \sum_{j=1}^{N} H[i, j] * F_{sombre}(i, j) \quad (3)$$

4.2. Affinage de l'Histogramme

Lorsque le gestionnaire d'histogramme reçoit un feedback du moteur de requête, il actualise l'histogramme pour refléter les statistiques nouvellement reportées. Les requêtes agissent comme des spots de lumière ; elles éliminent l'obscurité d'une région de l'histogramme. L'intensité du spot de lumière qu'une requête offre à un histogramme est proportionnelle à la fraction de région illuminée par la requête. Quand des requêtes se chevauchent, plusieurs spots de lumière sont dirigés sur la région chevauchée de

l'histogramme. Plus la lumière sur une région d'histogramme est intense, meilleure est la précision d'affinage de l'estimation de la sélectivité de cette région de l'histogramme.

Définition 3 : *Le taux normalisé d'une requête q pour une cellule de la grille G[r, c] est défini comme le ratio entre l'estimation de sélectivité de la partie de q qui chevauche G[r, c] et l'estimation de sélectivité de q. On dénote ce taux normalisé par* $R_q(r, c)$.

$$R_q(r,c) = \frac{F_q(r,c) * H[r,c]}{\sum_{i=1}^{N}\sum_{j=1}^{N} F_q(i,j) * H[i,j]} = \frac{F_q(r,c) * H[r,c]}{EstSel(q)} \quad (4)$$

$$R_q(r,c) = \frac{F_q(r,c)}{\sum_{i=1}^{N}\sum_{j=1}^{N} F_q(i,j)} \quad Lorsque\ EstSel(q) = 0 \quad (5)$$

La sélectivité réelle qu'une requête reporte est assumée être distribuée uniformément sur l'étendue de la requête. La Figure 2 donne la procédure pour affiner l'histogramme quand un feedback d'un moteur de requête reporte la sélectivité réelle *S* d'une requête *q*.

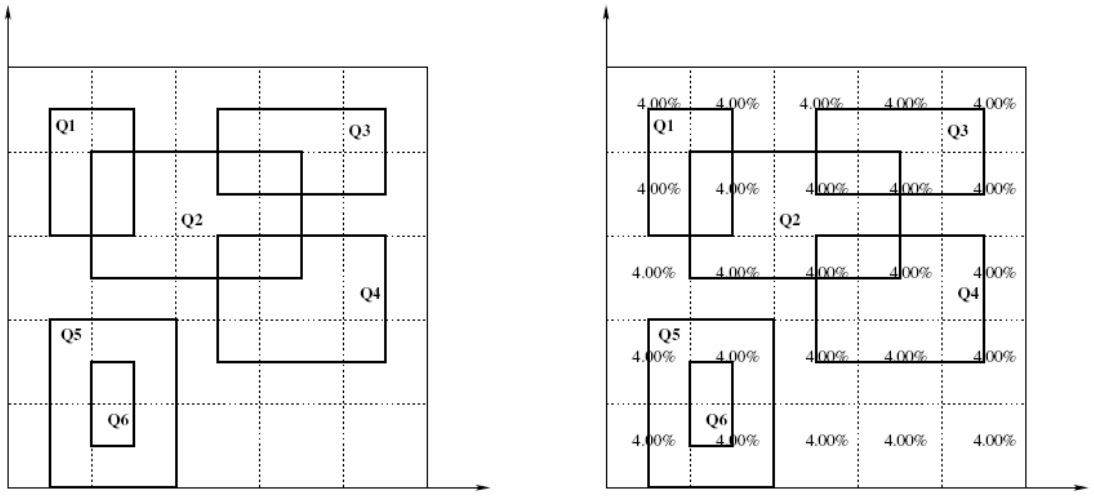
```
AffinerHistogramme(H, q, S) {
      Diff = S – EstSel(q) ;
      Si (Diff > 0)
            Diff = min(Diff, EstSel(sombre)) ;
      AjoutDiff(H, q, Diff) ;
      AjoutDiff(H, sombre, -Diff) ;
}

AjoutDiff(H, q, Diff) {
      Pour i=1 à N
            Pour j = 1 à N
                  H[i, j] = H[i, j] + Rq(i, j) * Diff;
}
```

Figure 2 : Procédure d'affinage de l'histogramme ST

En premier, les cellules de la grille chevauchées par *q* auront leurs valeurs changées suivant la différence entre *S* et l'estimation courante de la sélectivité de *q*. Typiquement, cette différence est distribuée selon le taux normalisé de q pour chacune de ces cellules. Ensuite, toutes les cellules ayant des portions sombres seront modifiées de manière analogue afin de se conformer avec l'invariance unitaire de *H* ($\sum_{i=1}^{N}\sum_{j=1}^{N} H[i, j] = 1$). Ainsi, l'estimation de la sélectivité des portions sombres est la limite supérieure de la différence lorsque celle-ci est positive.

Exemple : on illustre l'affinage d'histogramme par l'exemple donné en Figure 3(a). Dans cet exemple, on a six requêtes continues mappées sur une grille *5*5*. Chacune de ces requêtes renvoie les véhicules de type 'camion' dans sa région de couverture (Table 1). *Q2* est chevauchée par Q_1, Q_3 et Q_4. *Q6* est contenue dans Q_5. Chaque seau de la grille démarre avec une estimation de sélectivité égale à *4%* (Figure 3(b)). Le gestionnaire d'histogramme reçoit alors un feedback de l'exécuteur de requête indiquant que Q_1 déclare une sélectivité de *10%*.



(a) Six Requêtes Continues (b) Histogramme ST initial

Figure 3 : Exemple de six requêtes continues avec un histogramme initial

L'estimation de la sélectivité de Q_1 est de *6%* selon l'Equation 2. La différence *10 – 6 = 4%* est distribuée entre ces cellules chevauchées par Q_1. Considérez la cellule supérieure gauche C_{ul}. Le taux normalisé de Q1 pour C_{ul} est égal à *0,25 * 0,04 / 0,06 = 0,1667*. Ainsi, *H[1, 1] =*

*0,04 + 0,1667 * 0,04 = 0,0467 = 4,67%*. On a encore besoin de modifier l'histogramme afin d'atteindre l'invariance unitaire. L'augmentation (ou la diminution) de l'estimation de sélectivité dans une région éclairée doit être accompagnée d'une diminution (ou augmentation) de l'estimation de sélectivité dans la région sombre. Ainsi, on diminue l'estimation de sélectivité des régions sombres uniformément autant que l'augmentation dans les régions éclairées (*4%*). Par exemple, le seau inférieur droit C_{lr}, consiste de *1/13,25* de la région sombre. Le taux normalisé courant de la région sombre pour C_{lr} est de *0,0755*. La nouvelle valeur pour *H[5, 5]* sera égale à *0,04 – 0,0755 * 0,04 = 0,037 = 3,7%.* La Figure 4(a) montre l'histogramme après affinage. Le seau supérieur gauche a aussi une portion sombre qui résulte de la diminution de *H[1, 1]*.

Les Figures 4(b) et 6(f) montrent les mises à jour successives sur l'histogramme dues aux feedbacks subséquents que le gestionnaire d'histogramme reçoit comme suit : Q_2 rapporte *20%*, Q_3 rapporte *15%*, Q_4 rapporte 10%, Q_5 rapporte *10%*, et Q_6 rapporte *3%*.

Notez qu'après l'affinage de l'histogramme, on a seulement une meilleure estimation de la sélectivité pour la requête. La nouvelle estimation n'est pas la même que *S*. Aussi, de meilleures estimations de la sélectivité sont obtenues pour les régions sombres. Avec la succession des feedbacks qui rapportent (presque tous) la même sélectivité, l'estimation pour la requête converge au feedback.

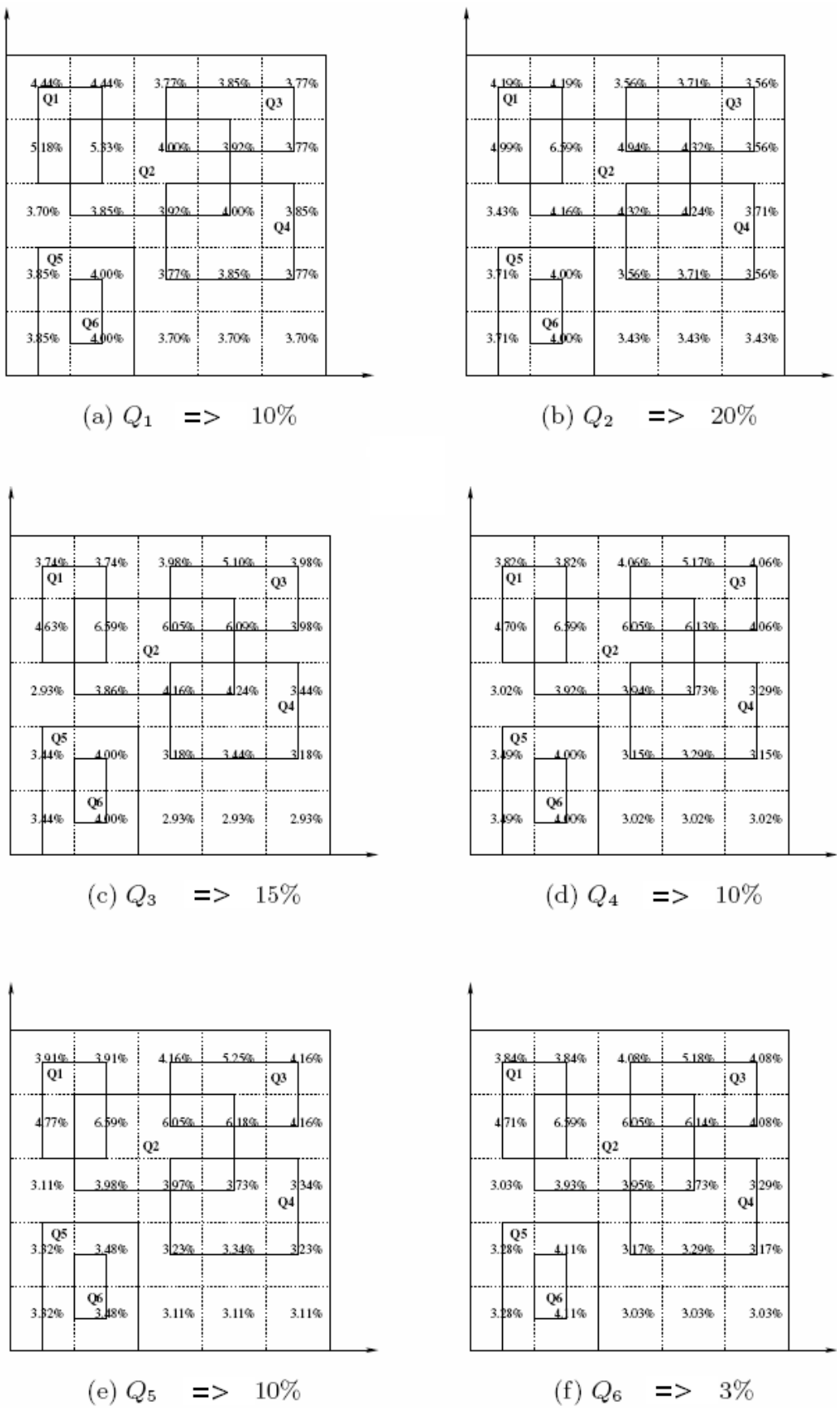


Figure 4 : Affinages Successifs de l'Histogramme ST après Réception du Feedback

5. Extensions des Histogrammes ST

Dans sa forme originale, chaque feedback déclenche un affinage de l'Histogramme ST, typiquement la sélectivité réelle d'un opérateur ST, afin d'accommoder la sélectivité réelle de l'opérateur de requête simple. Le prochain niveau est d'affiner les Histogrammes ST en utilisant de multiple feedbacks. En prenant en compte la liaison spatiale entre les différentes requêtes, on peut atteindre une meilleure précision d'affinage. Pour être plus spécifique, considérons les requêtes Q1 et Q2 de la Figure 5. Q2 est totalement contenue à l'intérieur de Q1. Supposons qu'on reçoit un feedback de Q1 « sélectivité réelle 20% », et un autre feedback de Q2 « sélectivité réelle 4% ». En utilisant ces feedbacks collectivement, on peut inférer plus d'information que si on les utilise chacun séparément et supposer l'uniformité pour les deux requêtes à chaque affinage. En fait, on peut déduire que la sélectivité de la partie droite de Q1 (qui est en dehors de Q2) est de 16%. Des inférences similaires peuvent être faites pour les requêtes imbriquées.

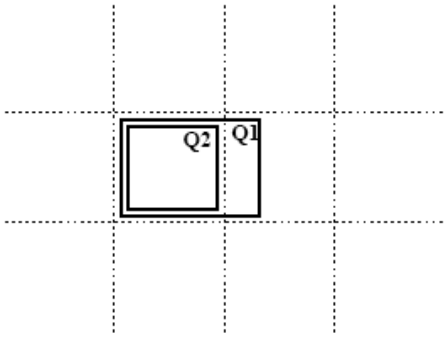


Figure 5 : Feedback Multi-Requête.

D'autre part, les histogrammes peuvent exploiter la technique de détection de périodicité en ligne pour vérifier si un motif périodique apparaît dans la sélectivité d'une région. A chaque fois qu'une telle périodicité est détectée, les histogrammes ST la prennent en compte pour avoir des estimations de sélectivité plus précises. Par exemple, les gens vont au travail chaque jour de semaine au matin et retournent à la maison vers 17h. Ces motifs périodiques ne se manifestent pas seulement dans la dimension temporelle mais aussi dans celle spatiale: ex. les gens empruntent souvent les mêmes routes pour aller au travail (autoroutes, trains, …). Les bus suivent les mêmes parcours pour ramasser et déposer leurs usagers.

6. Conclusion

Malgré l'importance de l'estimation de la sélectivité et la popularité croissante des bases de données spatio-temporelles, il n'y a eu que peu de travaux sur des techniques exactes et efficaces d'estimation de la sélectivité spatio-temporelle. Les données spatio-temporelles diffèrent tellement de celles relationnelles que les techniques relationnelles ne peuvent s'exécuter dans ce domaine. Sur ce point, les histogrammes ST semblent être les mieux adaptés pour estimer la sélectivité spatio-temporelle.

Les Histogrammes ST sont construits en surveillant la sélectivité réelle des requêtes continues en cours. L'idée principale est d'envoyer périodiquement des feedbacks, sous forme de statistiques, depuis l'exécuteur de requête vers le gestionnaire d'histogramme. Ces feedbacks sont typiquement la sélectivité des opérateurs spatio-temporels. Ces statistiques sont calculées de façon inhérente dans les opérateurs. Elles n'imposent pas de surcharge additionnelle sur l'exécuteur de requête.

Les Histogrammes ST sont basés sur des grilles (*grid-based*) où la grille divise l'univers uniformément en un nombre de cellules disjointes. Une cellule est soit claire ou sombre. Une cellule claire correspond à une région où aucune requête ne chevauche. Démarrant avec toutes les cellules étant sombres, l'estimation de la sélectivité de chaque seau (*bucket*) dans l'histogramme est initialisée uniformément. Avec les feedbacks successifs reçus des opérateurs, de meilleures estimations de la sélectivité sont obtenues dû à une vue plus claire de l'aire de couverture. La sélectivité réelle S qu'une requête q renvoie est supposée être distribuée uniformément sur l'étendue de la requête. Les cellules couvertes par q auront leurs valeurs changées suivant la différence entre S et l'estimation de la sélectivité courante de q (à partir de l'histogramme). Typiquement, cette différence est distribuée suivant l'intensité du spot de lumière offert par q. Par la suite, toutes les cellules de la grille ayant des portions sombres seront modifiées de manière similaire afin de les conformer avec l'invariance d'unité des Histogrammes ST.

Des résultats expérimentaux montrent que les histogrammes spatio-temporels fournissent 8,5% d'erreur pour les requêtes existantes impliquant 1% de l'espace. Une moyenne de 25% d'erreur pour les nouvelles requêtes couvrant 60% de l'espace.

Optimisation des Requêtes Spatio-Temporelles Continues

La plupart des travaux sur le traitement des requêtes spatio-temporelles continues se sont focalisés sur le développement d'algorithmes 'hors-boite', autrement dit, construits au dessus des SGBD-ST. Ce type d'algorithmes court-circuite complètement le rôle de l'optimisateur de requêtes dans les SGBD-ST, et limite, ainsi, sévèrement les performances des requêtes ST.

1. Motivations

Dans cette section, on présente plus en détails les défis apparus dans l'optimisation des requêtes ST dans les SGBD-ST, par les requêtes multi-prédicats et leurs sélectivités changeantes.

1.1. Défi temporel

En général, la distribution des objets mouvants change avec le temps. Par exemple, un grand nombre de voitures se déplacent vers le centre ville de 9h00 à 17h00 laissant la périphérie avec moins de voitures. Durant la nuit, la plupart des voitures se garent et par conséquent se dé-enregistrent du SGBD-ST. Par conséquent, le nombre de voitures dans le SGBD-ST est moindre.

Considérons la requête *Q* renvoyant l'*ID* de n'importe quel camion dont la position est à l'intérieur d'une aire *A* :

```
SELECT      M.ID
FROM        VEHICULE V
WHERE       V.Type = 'Truck'
INSIDE      Area A
```

L'opérateur de requête INSIDE est proposé dans [26] pour vérifier si un objet mouvant est dans une certaine étendue.

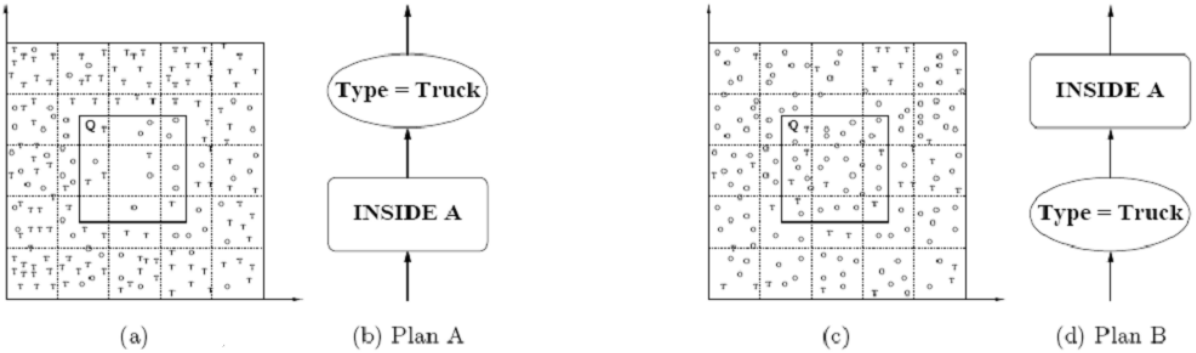


Figure 1 : Véhicules en mouvement (T=Truck, O=Other) sur un espace et les plans d'exécution correspondants.

Initialement, à l'instant t_1, l'optimisateur de requête constate que la sélectivité de l'opérateur INSIDE est moindre que la sélectivité du prédicat dans la clause WHERE (Figure 1(a)). Ainsi, l'optimisateur choisit d'utiliser le plan d'exécution de requête de la Figure 1(b) pour répondre à la requête (Plan A). A l'instant t_2, plusieurs véhicules entrent dans l'aire A conduisant à l'augmentation de la sélectivité de l'opérateur INSIDE tandis que le nombre de camions diminue (Figure 1(c)). Par conséquent, le prédicat dans la clause WHERE est plus sélectif que le prédicat INSIDE. Dans ce cas, le plan A devient sous-optimal et le plan optimal d'évaluation de la requête est le plan B (voir Figure 1(d)). De l'exemple au-dessus, on peut conclure qu'on ne peut construire un plan *statique* d'évaluation de requête pour répondre efficacement à une requête ST continue sur une période de temps étendue.

1.2. Défi spatial

Considérons la même requête dans la Table 1, mais avec deux aires différentes comme exposée dans la Figure 2(a) et (c). La distribution des objets sur l'espace affecte grandement le choix d'un plan d'exécution optimal. Dans la Figure 2(a), le prédicat INSIDE est plus sélectif que le prédicat dans la clause WHERE et par conséquent l'opérateur INSIDE vient en premier dans le plan ; Plan A (Figure 2(b)). D'un autre coté, pour la même distribution de données, et au même instant, mais pour un prédicat INSIDE différent (Figure 2(c)), Plan B (Figure 2(b)) est le plan d'évaluation optimal.

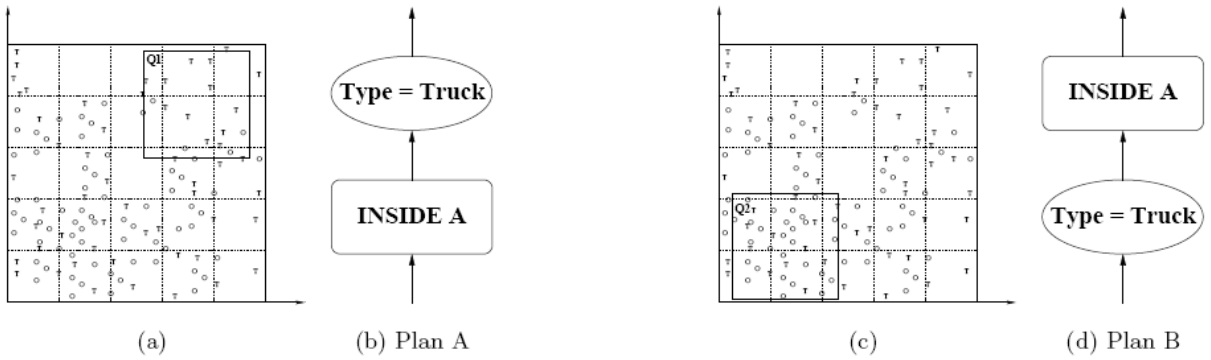


Figure 2 : Véhicules en mouvement (T=Truck, O=Other) sur un espace et les plans d'exécution correspondants.

1.3. Défi spatio-temporel

Avoir des requêtes mouvantes avec des objets mouvants rend le problème plus complexe. La sélectivité du plan d'évaluation de la requête ne dépend plus seulement de la distribution des objets en mouvement dans l'espace, mais aussi de la position de l'objet focal de la requête. Lorsqu'une requête a plus d'un objet focal et qui sont tous en mouvement, le plan d'évaluation de la requête continue devient rapidement sous-optimal.

Les défis présentés dans cette section ont plusieurs implications :

- La sélectivité des différents opérateurs de la requête change avec le temps et l'espace. Ainsi, le coût de ces opérateurs et par conséquent le plan optimal change aussi à travers le temps et l'espace.

- Il est crucial de considérer des techniques d'optimisation adaptative de requête lorsqu'on traite avec des SGBD-STs.

2. Optimisation Spatio-Temporelle Adaptative

Puisque le coût d'une requête change avec l'espace et le temps, comme montré dans la section précédente, on propose d'avoir une fonction de coût pour une requête ST continue. Cette fonction de coût sera affectée par les facteurs suivants :

1. **Durée de vie moyenne d'une mise à jour d'entrée dans le pipeline de la requête** : c'est le temps d'exécution nécessaire pour une mise à jour de position à se propager à travers le pipeline jusqu'à ce qu'elle soit reportée à l'utilisateur, qu'elle soit expulsée du pipeline, ou qu'elle soit stockée dans un état intermédiaire de l'opérateur ST.

2. **Stockage moyen requis comme états internes par une mise à jour d'entrée** : une mise à jour de position qui arrive au pipeline, peut rester dans un état interne pour quelque temps avant qu'elle ne soit reportée à l'utilisateur. Ce facteur de coût se rapporte à la probabilité d'être stocké dans un opérateur et la quantité de stockage nécessaire.

3. **Sélectivité moyenne des mises à jour d'entrée** : c'est le ratio du nombre total des objets mouvants dont l'actualisation de position sont acheminées au pipeline de la requête.

Capturer une fonction de coût représentative des requêtes ST continues permettra de fournir un cadre de travail pour le traitement et l'optimisation adaptatifs de ces requêtes.

2.1. Proposition

En exploitant l'estimation de la sélectivité, nous nous intéresserons à deux fonctionnalités fondamentales de l'optimisation de l'exécution des requêtes spatio-temporelles continues à savoir :

1. La construction d'un nouveau plan de requête *optimal* pour chaque requête continue nouvellement soumise, et

2. La surveillance continuelle de la performance des requêtes continues afin d'être sûr que le plan de requête optimal original maintient son optimalité.

Une fois que l'optimisateur de requête détecte que le plan de requête original est devenu sous-optimal, il substitue le plan sous-optimal par un autre optimal.

2.1.1. Moyens

Histogrammes ST

Afin de supporter ces fonctionnalités, on propose de construire et de constamment maintenir des Histogrammes Spatio-Temporels. Ces histogrammes ST vont maintenir continuellement les estimations de la sélectivité spatio-temporelle, utilisées par l'optimisateur de requête pour décider de l'optimalité des divers plans d'exécution candidats. La précision de l'estimation initiale est continuellement augmentée et affinée en se basant sur la surveillance de l'exécution des requêtes spatio-temporelles en cours.

Grille ST

Aussi, on utilise une grille ST données-à-plan (*data-to-plan grid*) pour éviter qu'un plan de requête ne soit exécuté sur toutes les mises à jour arrivant au SGBD-ST. Cette grille données-à-plan servira de couche pour router les actualisations de position vers les plans appropriés disponibles. Par conséquent, une actualisation de position entrante sera acheminée seulement aux requêtes pour lesquelles elle peut en affecter les réponses.

Définition des Eléments de l'histogramme ST

***G* :** grille $N*N$ divisant l'espace uniformément en cellules disjointes.
***G[i, j]* :** cellule correspondant à l'intersection de la $i^{ème}$ ligne et de $j^{ème}$ colonne de la grille G.
***H* :** vue courante de l'histogramme ST.
***H[i, j]* :** estimation de la sélectivité de la cellule $G[i, j]$.
$\boldsymbol{F_q(i, j)}$ **:** fraction de la cellule $G[i, j]$ couverte par la requête q.
Initialisation : $H[i, j] = 1/N$ pour tout $1 \leq i, j \leq N$.

2.1.1. Principe

Ainsi, on implémentera un mécanisme de formation dynamique des plans pour adapter les paramètres changeants du système. L'adaptation implique l'addition, le retrait ou la réorganisation des opérateurs dans le plan d'évaluation de la requête. Lorsqu'une requête ST continue est enregistrée, un plan d'évaluation initial est construit. Ce plan peut couvrir de multiples cellules de la grille. Dans ce cas, un sous-plan est crée pour chaque cellule et un opérateur d'union est utilisé pour collecter les réponses des différents sous-plans. Quand le coût du plan change dû au mouvement de la requête et/ou des objets, trois événements peuvent se produire :

1- **une réorganisation des opérateurs dans le plan,**
2- **une décomposition du plan en sous-plans, ou**
3- **une reconsolidation des sous-plans afin de former un seul plan.**

2.2. Etapes d'Optimisation de l'Exécution d'une Requête ST

2.2.1. Identification des cellules couvertes par les prédicats ST et de leurs sélectivités relatives respectives :

Dans cette étape, pour chaque prédicat ST, on identifie séparément les cellules chevauchées. On calcule ensuite pour chaque cellule couverte, l'estimation relative de la sélectivité du prédicat par rapport à sa couverture de la cellule. Si une cellule *G[i, j]* est complètement couverte par le prédicat, alors l'estimation de sélectivité de ce dernier sera égale à celle de la cellule, autrement, elle sera égale à l'estimation de la cellule multipliée par la fraction de sa couverture par le prédicat.

2.2.2. Définition du ou des plans initiaux :

Ici, un plan d'exécution spécifique est établi pour chaque cellule couverte. D'abord, on utilise le catalogue système pour récupérer les estimations de sélectivité des prédicats standards. Par la suite, on compare toutes les sélectivités (que ce soit celles des prédicats standards ou celles

des prédicats ST) afin de définir un ordre d'exécution : Les prédicats ayant une grande sélectivité sont placés en premier dans le sous-plan tandis que ceux ayant une plus faible sélectivité sont placés en dernier.

2.2.3. Exécution continue des sous-plans :

Les sous plans correspondants aux cellules couvertes par la requête ST sont exécutés de manière continue et synchronisée. Leurs résultats sont fusionnés pour produire le résultat complet et instantané de la requête ST. Ce résultat sera utilisé, en autres, pour affiner les estimations de sélectivité des cellules couvertes par la requête ST ainsi que celles non couvertes par le principe de complémentarité : opération d'affinage de l'histogramme ST.

- **Maintenance de l'optimalité des sous-plans :** Si l'estimation de la sélectivité d'une cellule couverte par la requête ST change de valeur, par exemple à cause de sa mise à jour (affinage) par le retour de résultat d'une autre requête ST qui était en cours d'exécution, une nouvelle analyse et comparaison des estimations de sélectivité des différents prédicats composant le plan correspond à cette cellule sont effectuées. Si on juge que les estimations ont suffisamment changé pour affecter l'optimalité du sous-plan original, un nouveau sous-plan est construit suivant les nouvelles estimations.

2.3. Implémentation

```
Var    Q : Requête
       R : Résultat de Requête = Φ;

// Fonction d'évaluation principale
//++++++++++++++++++++++++++++++++++++++
Fonction Evaluation_RequêteST() {

// Q représente la requête après son analyse syntaxique et parsing
       Pour chaque cellule G[i, j] couverte par Q {
              SP = Creation_Sous_Plan(i, j) ;
              Execution_Sous_Plan(SP, i, j) ;
       }
}
```

```
//+++++++++++++++++++++++++++++++++++++++
Fonction Creation_Sous_Plan(Q, i, j) {

      SP = nouveau Sous_Plan();
      Pour chaque prédicat P de Q {
            Si P est un prédicat ST {

                  /* relativisation spatiale du prédicat P à la cellule G[i, j] */
                  P = Reajustement_Predicat(P, i, j);

                  /* Taux de couverture de la cellule G[i, j] par P */
                  F_P[i, j] = Couverture(P, i, j);

                  /*  EstSel_P */
                  EstSel_P[i, j] = H[i, j] * F_P[i, j];
            }
            Sinon {
                  /* Utilisation catalogue système pour EstSel_P */
                  EstSel_P[i, j] = EstSel_Cat(type(P));
            }
            Inserer_Predicat_Plan(SP, P, EstSel_P[i, j]);
      }
      Return SP;
}


//+++++++++++++++++++++++++++++++++++++++
Fonction Execution_Sous_Plan(SP, i, j) {

      /* Exécution continue */
      Tant Que (vrai) {

            /* Exécution du SP avec sauvegarde de la sélectivité réelle S */
            SR = Execution(SP, S);

            /* Ajout du résultat aux résultats des autres sous-plans */
            R = R ∪ SR;

            /* Affinage de l'histogramme ST : H */
            Diff = S - EstSel_P[i, j];
            Si (Diff > 0) {
                  Diff = min (Diff, EstSel_sombre());
            }
            H[i, j] = H[i, j] + H[i, j] * F_P[i, j];
      }
}
```

```
// Evènement lancé lors du mouvement de l'émetteur de requête
//++++++++++++++++++++++++++++++++++++++
Evénement Mouvement_Requête() {

        Si (Q est dépendante de sa position) {

                Terminer tous les processus Execution_Sous_Plan() correspondants à Q ;

                Recalculer les cellules couvertes par Q après son déplacement ;

                Evaluation_RequeteST();
        }
}


// Evènement lancé lors du changement de la sélectivité d'une cellule G[i, j]
//++++++++++++++++++++++++++++++++++++++
Evènement Changement_EstSel(i, j) {     // au dessus d'un certain seuil ε


        Si (|H_ancien[i, j]- H_nouveau[i, j]|> ε) {

                Relancer tous les processus Execution_Sous_Plan() correspondants à la
                cellule G[i, j];
        }
}
```

Remarques

- En général, le changement de l'estimation de la sélectivité d'une cellule donnée s'accompagne du changement de l'estimation de la sélectivité d'une ou de plusieurs cellules adjacentes, du fait que les objets ayant provoqué ce changement ne peuvent se trouver, après leur déplacement dans l'intervalle de temps d'une exécution de la requête, que dans les cellules avoisinantes.

- Puisque un prédicat ST *P* est initialement relatif à l'espace entier et non à une cellule particulière, une fonction `Reajustement_Predicat(P, i, j)` est nécessaire afin de permettre un ajustement spatial de *P* à une cellule *G[i, j]* donnée.

- Si la une requête Q contient des prédicats ST dépendants de sa propre position (i.e. de son émetteur), alors la fonction évènement `Mouvement_Requête()` est automatiquement lancée dès le mouvement de Q. Dans ce cas, les processus exécutant les sous-plans de Q sont terminés, et une nouvelle évaluation de la requête est effectuée après la redéfinition des cellules nouvellement couvertes par Q.

- De la même manière, la fonction évènement `Changement_EstSet(i, j)` est lancée si une modification supérieure à un seuil ε de l'estimation de la sélectivité de la cellule G[i, j] est constatée. Dans ce cas, une relance des processus exécutant les sous-plans relatifs à la cellule impliquée, et ceci pour toutes les requêtes, est nécessaire.

Exemples

Comme premier exemple, considérons le cas d'une requête continue Q constituée d'un prédicat d'étendue et d'une sélection. A l'instant t_1, le prédicat d'étendue est exclusivement couvert par une seule cellule de la grille données-à-plan et est plus sélectif que la clause WHERE. Le plan d'évaluation correspondant est schématisé dans la Figure 3(a).

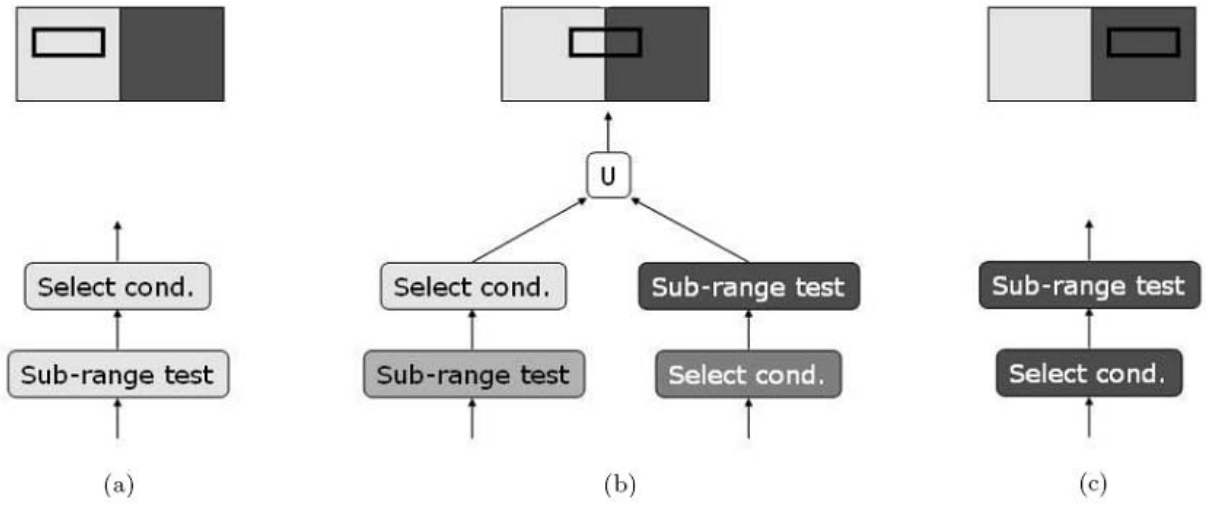


Figure 3 : Formation Dynamique de Plan (Scénario 1).

A l'instant t_2, la requête Q se déplace et couvre deux cellules du plan. Ainsi, deux sous-plans sont composés pour Q, un par cellule. Les résultats de ces deux sous-plans sont fusionnés ensemble pour former la réponse de la requête (Figure 3(b)). A l'instant t_3, Q se déplace et est

couverte exclusivement par une seule cellule mais dispose d'une plus grande sélectivité que la clause WHERE. Les deux sous-plans reconsolidés forment le plan d'évaluation courant (Figure 3(c)).

Même sur la granularité de sous-plan, la sous-optimalité peut exister. Considérons l'exemple dans la Figure 4(a)-4(b), durant le mouvement de la requête, le coût d'un plan d'évaluation peut changer à cause de la modification de la sélectivité des opérateurs. Dans ce cas, on pourra avoir besoin de réorganiser les opérateurs d'un ou plusieurs sous-plans afin de maintenir l'optimalité du plan courant.

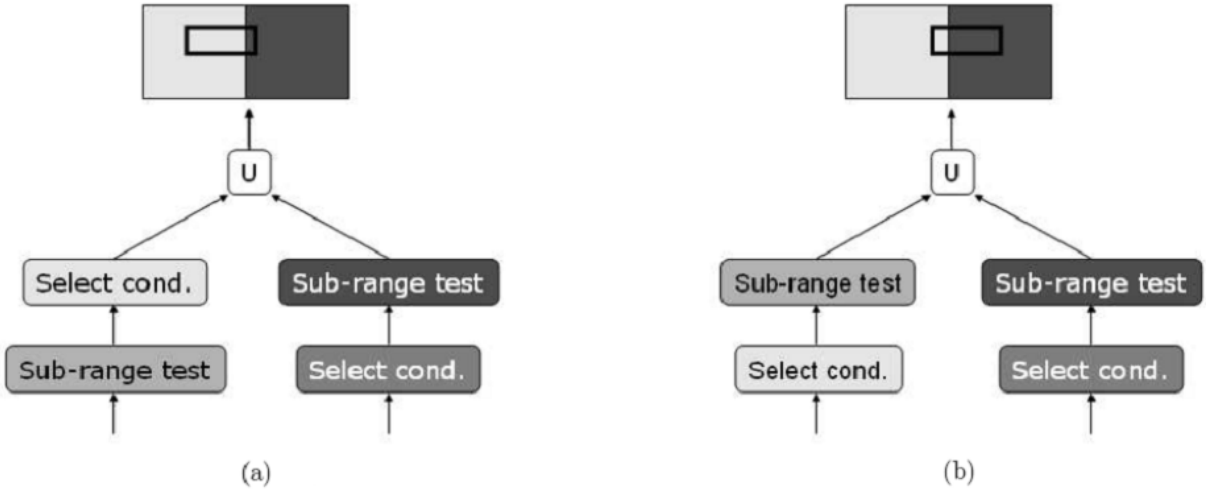


Figure 4 : Formation Dynamique de Plan (Scénario 2).

Essayer de maintenir un plan optimal tout le temps peut ne pas être toujours un bon choix. Premièrement, les ré-optimisations fréquentes de requête peuvent affecter négativement les performances du système et causer un retard dans la délivrance des réponses après qu'elles soient invalidées. Traquer l'optimalité à chaque fois qu'une sous-optimalité est détectée peut conduire à de fréquentes commutations de ré-optimisation entre quelques plans d'évaluations. Par conséquent, les cycles CPU seront plus perdus sur les ré-optimisations que sur le traitement réel de requête. En contraste, un plan d'évaluation de requête ST qui est robuste pour une certaine de temps raisonnable pourra aider à éviter de telles situations. Durant l'énumération des différents plans de requête, l'optimisateur des requêtes ST doit sélectionner un plan d'évaluation résistera aux petites variations de la sélectivité des opérateurs et au coût total du plan de requête.

Défis

1. Manque d'opérateurs ST pipelinés

La plupart des algorithmes existants dans la littérature s'occupent de répondre aux requêtes ST à un prédicat sur des objets en mouvement et aucun d'eux n'est pipeliné. Ce dont on a besoin au contraire, c'est d'être capable de composer un pipeline d'opérateurs de requête afin de répondre aux requêtes ST multi-prédicats.

Les requêtes ST multi-prédicats nécessitent une manipulation spéciale et ne peuvent être répondues efficacement en empilant simplement les opérateurs ST les uns au-dessus des autres pour former un pipeline.

Les approches existantes fonctionnent comme suit : l'entrée des opérateurs existants est un flux d'actualisation des positions des objets en mouvement. Les sorties de ces opérateurs sont les objets mouvants satisfaisant la requête. La réponse à la requête peut être reportée soit périodiquement (ou sur changement) soit progressivement. Les réponses périodiques sont les identifiants de tous les objets formant l'ensemble de réponse. Dans le cas progressif, l'ensemble de réponse est reporté progressivement comme une séquence d'actualisations positives et négatives. Une mise à jour positive indique un nouveau objet devant être ajouté à l'ensemble de réponse tandis qu'une négative indique un objet devant être retiré de cet ensemble. Du moment que l'entrée des opérateurs courants est différente de leur sortie dans le format et les sémantiques, il est clair qu'on ne peut se contenter de juste brancher les opérateurs existants les uns au sommet des autres pour former un pipeline afin de répondre aux requêtes multi-prédicats.

2. Optimisation ST des Multiples Requêtes

Il est souvent le cas que plusieurs requêtes ST concurrentes possèdent des opérateurs communs. Cette situation pourra permettre le partage des opérateurs et parfois des sous-plans complets en incorporant la charge de travail courante dans l'évaluation du coût de n'importe quel plan d'évaluation énuméré. Plus spécifiquement, à chaque fois qu'une nouvelle requête ST est soumise, elle est décomposée en fragments de requête. Ensuite, si un fragment similaire existe déjà, il peut potentiellement être réutilisé. Dans ce cas, le coût d'exécution de ce fragment de requête peut être ignoré lors du calcul du coût total de la requête. De plus, les requêtes continues existantes peuvent avoir leur plan d'évaluation changé avec l'arrivée d'un nouveau plan si ce changement permettra plus de partage des opérateurs et ainsi moins de surcharge sur le système que dans le cas du non-partage.

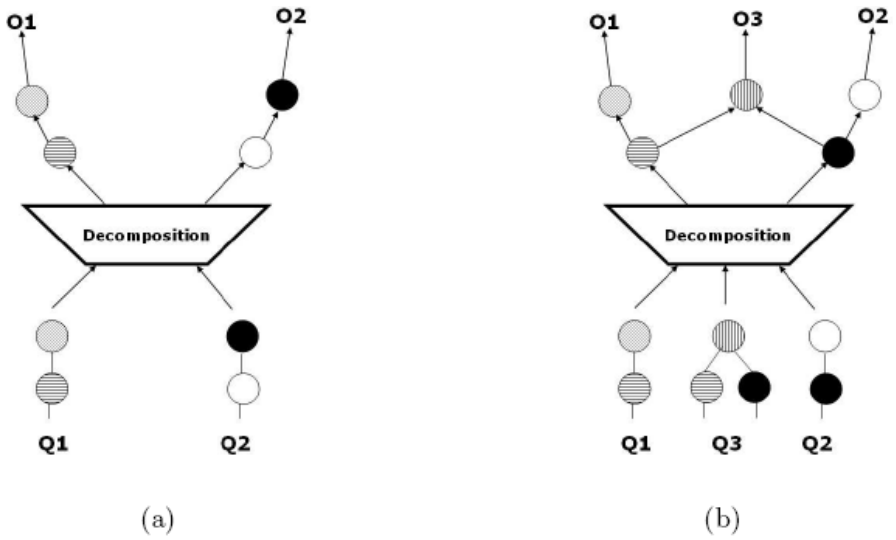


Figure 5 : Optimisation de Multiple Requêtes Spatio-Temporelles.

Considérons les deux requêtes ST continues *Q1* et *Q2* dont les plans d'évaluations sont montrés dans la Figure 5(a). Supposons que *Q2* consiste en une sélection (opérateur noir) et une requête d'étendue (opérateur blanc). Dans cet exemple, la requête d'étendue est plus sélective que la sélection, et ainsi l'opérateur blanc est exécuté en premier. Dans la figure 5(b), on montre l'arrivée d'une nouvelle requête *Q3* consistant en une jointure de fenêtre de l'opérateur noir (sélection) et d'un autre opérateur. *Q2* aura son plan d'évaluation changé pour permettre le partage de la sélection entre les deux requêtes. Le coût total des tous les opérateurs dans le système est moins moindre avec le partage bien que le plan d'évaluation individuel de *Q2* est sous-optimal.

Conclusion

Nous devions étudier l'environnement entourant le traitement des requêtes spatio-temporelles continues et répondre à la problématique de leur l'optimisation relative à l'impact du temps, de l'espace, et de leur combinaison sur l'optimalité des plans de requête sous différentes circonstances de requêtes et de distribution des objets.

Durant notre étude, nous avons exploré :

- Les différents aspects des bases de données spatio-temporelles ainsi que les requêtes ST continues et leurs caractéristiques.

- Nous avons aussi passé en revue tous les aspects relatifs à l'optimisation des requêtes standards et celles continues.

- Ensuite, nous avons présenté le concept de l'estimation de la sélectivité ST ainsi qu'un outil permettant de la gérer de manière simple et efficace, à savoir les histogrammes spatio-temporels.

Pour une requête ST donnée, nous avons constaté que la sélectivité des opérateurs ST qui la composent, et par conséquent leurs coûts, changent naturellement avec le temps et l'espace. Les plans originaux à priori optimaux peuvent très rapidement devenir sous optimaux et dégrader les performances du système.

Ainsi, en exploitant l'estimation des sélectivités des opérateurs ST et en utilisant les histogrammes ST, nous avons essayé d'implémenter formellement un mécanisme d'adaptation dynamique et continue des plans d'exécution des requêtes ST.

Nous en avons conclu que cette technique d'optimisation adaptative des plans d'exécution, améliore dans beaucoup de cas les performances des requêtes ST continues et qu'elle pourrait facilement être intégrée aux modules d'optimisation de bas niveau existants.

Perspectives

On croit que ce modeste cadre de travail permettra d'aborder et de développer une solution globale plus poussée pour le traitement et l'optimisation des requêtes spatio-temporelles.

Capturer une fonction de coût représentative des requêtes ST continues permettra de fournir un outil pour le traitement et l'optimisation adaptatifs des requêtes, et qui inclura entre autres : l'estimation de la sélectivité pour des données spatio-temporelles élaborées prenant en considération leurs propriétés comme la périodicité et la corrélation, et une extension de l'optimisation de requête dans les SGBD-ST afin de couvrir de multiples requêtes spatio-temporelles multi-prédicats.

Enfin, il serait très intéressant d'essayer d'intégrer ces techniques dans des SGBD-ST ou des applications ST grand public et de les appliquer à un environnement réel afin de connaître leurs performances, identifier leurs avantages et corriger leurs éventuelles faiblesses.

Bibliographie

[1]. Database Management Systems. Editions Mcgraw Hill. 2ème edition

Optimisation des Requêtes Continues

[2]. D. DeWitt, R. Katz, F. Olken, L. Shapiro, Michael R. Stonebraker, D. Wood. Implementation Techniques for Main Memory Database Systems. In Proc. ACM-SIGMOD International Conference on Management of Data, pages 1-8, Boston, June 1984.

[3] A. N. Wilschut, P. M. G. Apers. Data flow Query Execution in a Parallel Main-Memory Environment. In Proc. First International Conference on Parallel and Distributed Info. Sys. (PDIS), pages 68-77, 1991.

[4] T. Ibaraki, T. Kameda. Optimal Nesting for Computing N-relational Joins. ACM Transactions on Database Systems, 9(3):482{502, October 1984.

[5] P. Haas, J. Hellerstein. Ripple Joins for Online Aggregation. In Proc. ACM- SIGMOD International Conference on Management of Data, pages 287-298, Philadelphia, 1999.

[6] T. Urhan, M. Franklin. XJoin: Getting Fast Answers From Slow and Bursty Networks. Technical Report CS-TR-3994, University of Maryland, February 1999.

[7] Z. Ives, D. Florescu, M. Fiedman, A. Levy, D. Weld. An Adaptative Query Execution System for Data Integration. In Proc. ACM-SIGMOD International Conference on Management of Data, Philadelphia, 1999.

[8] J. Hellerstein. Optimization Techniques for Queries with Expensive Methods. ACM Transactions on Database Systems, 23(2):113-157, 1998.

[9] R. Krishnamurthy, H. Boral, C. Zaniolo. Optimization of Nonrecursive Queries. In Proc. 12th International Conference on Very Large Databases (VLDB), pages 128-137, August 1986.

[10]. D. Terry, D. Goldberg. Continuous queries over append-only databases. In Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data, pages 321–330, June

Bases de Données Spatiales

[11] R.H. Güting. An Introduction to Spatial Database Systems. The VLDB Journal, 3(4):357–399, 1994.

[12] P. Rigaux, M. Scholl, and A. Voisard. Spatial Databases. Morgan Kaufmann, 2000.

[13] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM), pages 111–122, 1998.

[14] V. Gaede and O. Guenther. Multidimensional Access Methods. ACM Computing Surveys, 30(2), 1998.

[15] J. Tayeb, O. Ulusoy, and O.Wolfson. A Quadtree Based Dynamic Attribute Indexing Method. Computer Journal, 41:185–200, 1998.

[16] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In Proc. IEEE Intl. Conf. on Data Engineering (ICDE), pages 422–433, 1997.

[17] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In Proc. ACM SIGMOD Symp. on the Management of Data, 2000.

[18] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*tree : An Efficient and Robust Access Method for Points and Rectangles. In Proc. ACM SIGMOD Intl. Symp. on the Management of Data, pages 322–331, 1990.

Evaluation Requetes ST continues

[19] B. Zheng, D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In SSTD, 2001.

[20] J. Zhang, M. Zhu, D. Papadias, Y. Tao, D. L. Lee. Location-based Spatial Queries. In SIGMOD, 2003.

[21] B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In EDBT, 2004.

[22] H. Hu, J. Xu, and D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In SIGMOD, 2005.

[23] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic Queries over Mobile Objects. In EDBT, 2002.

[24] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In SSTD, 2001.

[25] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In VLDB, 2002.

[26] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: scalable incremental processing of continuous queries in spatio-temporal databases. In SIGMOD, 2004.

[27] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In ICDE, 2005.

Estimations de la sélectivité – Histogrammes ST - Echantillonnage

[28] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. "Access Path Selection in a Relational Database Management System". In Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, pages 23–34, Boston, Massachusetts, June 1979.

[29]. W. G. Aref and H. Samet. Estimating Selectivity Factors of Spatial Operations. In Proceedings, Optimization in Databases - 5th Int'l Workshop on Foundations of Models and Languages for Data and Objects, pages 31–43, Sep 1993.

[30]. W. G. Aref and H. Samet. A Cost Model for Query Optimization using R-trees. In ACM-GIS '94: Proceedings of the ACM Workshop on Advances in Geographic Information Systems, Dec 1994.

[31]. A. Belussi and C. Faloutsos. Estimating the Selectivity of Spatial Queries Using the 'Correlation' Fractal Dimension. In VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases, pages 299–310. Morgan Kaufmann Publishers Inc., 1995.

[32]. Y.-J. Choi and C.-W. Chung. Selectivity Estimation for Spatio-temporal Queries to Moving Objects. In SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pages 440–451. ACM Press, 2002.

[33]. Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In VLDB '03: Proceedings of the 29th International Conference on Very Large Data Bases, Sep 2003.

[34]. M. Hadjieleftheriou, G. Kollios, and V. J. Tsotras. Performance Evaluation of Spatio-temporal Selectivity Estimation Techniques. In SSDBM '03: Proceedings of the 15th International Conference on Scientific and Statistical Database Management, pages 202–211, 2003.

[35]. Q. Zhang and X. Lin. Clustering Moving Objects for Spatio-temporal Selectivity Estimation. In CRPIT '27: Proceedings of the fifteenth conference on Australasian database, pages 123–130. Australian Computer Society, Inc., 2004.

[36]. A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms without Looking at Data. In SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data, pages 181–192. ACM Press, 1999.

[37]. I. F. Ilyas, H. G. Elmongui, and W. G. Aref. Adaptive Processing of Ranking Queries. Technical Report CSD-TR-05-002, Purdue University, 2005.

[38]. N. Kabra and D. J. DeWitt. Efficient Mid-query Re-optimization of Sub-optimal Query execution Plans. In SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data, pages 106–117. ACM Press, 1998.

[39]. H. G. Elmongui, M. F. Mokbel, and W. G. Aref. Spatio-temporal Histograms. In SSTD, 2005.

[40]. Y. Tao, D. Papadias, J. Zhai, Q. Li. Venn Sampling: A Novel Prediction Technique for Moving Objects. In ICDE, 2005.

[41]. S. Chaudhuri, G. Das, Narasayya. A Robust Optimization-Based Approach for Approximate Answering of Aggregate Queries. *SIGMOD*, 2001.

[42]. C. Jermaine, Making Sampling Robust with APA.VLDB, 2003.

Bases de Données Mobiles

[43]. M. Özsu, P. Valduriez, "Principles of Distributed Databases", International Edition, 2ª. Edition 1999.

[44]. H. Kottkamp and O. Zukunft "Location-Aware Query Processing in Mobile Database Systems", In Proc. of the ACM Symposium on Applied Computing, Feb. 1998.

[45]. T. Antila, J. Jokela and L. Kenttälä, "A Survey on Data Management in Mobile Computing", 2001

[46]. T. Imielinski and B. R.Badrinath, "Mobile wireless computing: Challenges in data management", Communications of the ACM,Vol. 37, No. 10, October 1994, p. 19-28.

[47]. R. Alonso and H. F. Korth. "Database System Issues in Nomadic Computing", In Proceedings of the 1993 SIGMOD Conference, Washington, May 1993.

[48]. A. Bouguettaya, "On the Construction of Mobile Database Management Systems",

[49]. H. Gök; "Processing of Continuous Queries from Moving Objects in Mobile Computing Systems", 1999,

[50]. S. Helal, J. Hammer, J. Zhang, A. Khusharaj, "A Three-tier Architecture for Ubiquitous Data Access", ACS/IEEE International Conference on Computer Systems and Applications, Beirut, Lebanon, June 2001.

[51]. [Noble 1998] B. D. Noble, “Mobile Data Access”, 1998

[52]. Y. Saito and M. Saphiro, “Replication: Optimistic Approaches”, 2002

[53]. M. H. Dunham, A. Helal, S. Balakrishnan; “A Mobile Transaction Model that Captures Both the Data and Movement Behaviour”, 1997, ACM/Baltzer Journal on Special Topics in Mobile Networks and Applications (MONET), 1997

[54]. A. Elmagarmid, J. Jing, O. Bukhres, “An Efficient and Reliable Reservation Algorithm for Mobile Transactions”, 1995, Department of Computer Sciences, Purdue University, in Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM'95).

[55]. E. Pitoura, G. Samaras, “Data Management for Mobile Computing”, Kluwer Academic Publishers, 1998

[56]. S. Phatak, B. R. Badrinath, “Data Partitioning for Disconnected Client Server Databases”, 1998.

[57]. J. Jing, A. Helal, A. Elmagarmid, “Client-Server Computing in Mobile Environments”, 1999.

[58]. G. Samaras, P. Evripidou, E. Pitoura, “A Mobile-Agent Based Infrastructure for eWork and eBussiness Applications”, EMMSEC 2000.

[59]. A. Heuer and A. Lubinski. “Database Access in Mobile Environments” (extended version). In Preprint of the Dept. of CS, University of Rostock, CS-04-96, 1996.