Traitement de requêtes top-k multicritères et application à la recherche par le contenu dans les bases de données multimédia

Mehdi Badr

To cite this version:

Mehdi Badr. Traitement de requêtes top-k multicritères et application à la recherche par le contenu dans les bases de données multimédia. Autre. Université de Cergy Pontoise, 2013. Français. NNT : 2013CERG0666 . tel-00978770

HAL Id: tel-00978770
https://theses.hal.science/tel-00978770

Submitted on 27 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Cergy-Pontoise - Ecole doctorale Sciences et Ingénierie

THÈSE

présentée pour obtenir le titre de DOCTEUR en Informatique

TRAITEMENT DE REQUÊTES *top-k* MULTICRITÈRES ET APPLICATION À LA RECHERCHE PAR LE CONTENU DANS LES BASES DE DONNÉES MULTIMÉDIA

---

par

Mehdi BADR

ETIS - ENSEA / Université de Cergy-Pontoise / CNRS UMR 8051

6 avenue du Ponceau, 95014 Cergy-Pontoise Cedex, France

Soutenue le 07 Octobre 2013 devant le jury composé de :

B. AMANN,	LIP6, UPMC	Rapporteur
M. CRUCIANU ,	CEDRIC, CNAM	Rapporteur
K. ZEITOUNI,	PRISM, UVSQ	Examinateur
D. LAURENT,	ETIS, UCP	Examinateur
P. GOSSELIN,	ETIS, ENSEA	Examinateur
D. VODISLAV,	ETIS, UCP	Directeur de thèse

Remerciements

Je souhaite sincèrement remercier toutes les personnes qui ont contribué à l'aboutissement de cette thèse. Elle est le fruit de quelques années passées en compagnie de nombreuses personnes qui ont, chacune, apporté leur pierre à l'édifice. Il sera difficile de toutes les citer ici, mais j'espère que celles que j'aurais oubliées se reconnaîtront.

Mes premiers remerciements iront à mon directeur de thèse, *Dan Vodislav*, qui m'a guidé tout au long de ce périple. Je le remercie très sincèrement pour la confiance qu'il m'a accordée en acceptant de diriger ma thèse, mais également pour son soutien, sa disponibilité, ses précieux conseils et sa ténacité qui m'ont aidé à me surpasser. J'ai pu partager sa curiosité et son intérêt pour la recherche, j'ai également pu profiter de sa rigueur scientifique et de sa pédagogie pour présenter son travail.

Je remercie également les membres de mon jury qui ont pris le temps de lire, revoir et commenter mes travaux, ainsi que de se déplacer à Cergy, dans la lointaine banlieue parisienne, pour assister à ma soutenance. Merci, donc, à *Bernd Amann* et *Michel Crucianu* pour leur rôle de rapporteur et à *Karine Zeitouni*, *Dominique Laurent* et *Phlippe-Henri Gosselin* pour leur temps passé à évaluer mes travaux.

De manière générale, je voudrais dire merci à l'équipe *MIDI* et à la direction du laboratoire *ETIS* de m'avoir accueilli chaleureusement durant cette thèse. L'excellente ambiance de travail sera probablement un des meilleurs souvenirs que je garderai de cette période de ma vie. Je tiens à préciser que j'ai eu du plaisir à travailler aux côtés de personnes très agréables, je remercie particulièrement *Abdelhak Chatty* pour sa bonne humeur, son soutien et ses informations précieuses sur l'histoire de deux villes anciennes en Tunisie, Kairouan et M'saken, j'ai adoré ces discussions, mais je préfère toujours Tunis. Je remercie également *Claudia Marinica* pour la bonne ambiance qu'elle a amené dans l'équipe et pour son soutien surtout à des moments difficiles, parfois il faut qu'elle soit présente pour débloquer une situation difficile. Je pense également à nos chères secrétaires, *Astrid Cebron* et *Nelly Bonard* pour la bonne ambiance qu'elles maintiennent dans le laboratoire. Je tiens aussi à remercier *Abdulhafiz Alkhouli* avec qui j'ai eu le plaisir de partager mon bureau et de discuter longuement des révolutions dans les pays arabes; je profite de cette occasion pour souhaiter la réussite et le bonheur au peuple Syrien et que la paix sera rétablie. Je pense aussi à *Ali Karaouzene*, *Antoine Rolland de Rengervé*, *Boris Borzic*, *Shaoyi Yin*, *David Picard*, *Sylvie Philipp-Foliguet*, *Tuyêt Trâm Dang Ngoc*, *Mathias Quoy*, *Michel Jordan*, *Hedi Tabia* et à toutes les personnes qui m'ont supporté durant ces années au laboratoire *ETIS*.

De manière plus personnelle, je tiens à remercier chaleureusement mes parents pour leur encouragement sans fin. Ils ont sacrifié beaucoup pour moi sans rien attendre en retour. Je leur dois beaucoup pour ce qu'ils ont fait pour moi, et j'espère que je pourrai être toujours une source de bonheur avec mes deux très chers frères *Oussama* et *Mohamed Hamza*. Je remercie également mon cher ami *Chiheb* pour tout ce qu'il fait pour moi et ma famille, ainsi que *Abderrahman Matoussi* et tous mes amis.

Cette thèse n'aurait pas été possible sans le soutien de ma jolie femme, *Maryouma*, qui a été là pour moi à chaque étape de cette expérience. Son amour et son soutien ont beaucoup contribué à mon succès. J'adore partager chaque moment heureux dans ma vie avec elle, et elle a toujours été mon premier recours à chaque fois que je suis fatigué ou frustré. Merci beaucoup, et bon courage pour ta thèse (ma promesse tient toujours).

Résumé

Le développement des techniques de traitement des requêtes de classement est un axe de recherche très actif dans le domaine de la recherche d'information. Plusieurs applications nécessitent le traitement des requêtes de classement multicritères, telles que les méta-moteurs de recherche sur le web, la recherche dans les réseaux sociaux, la recherche dans les bases de documents multimédia, etc. Contrairement aux requêtes booléennes traditionnelles, dans lesquelles le filtrage est basé sur des prédicats qui retournent vrai ou faux, les requêtes de classement utilisent des prédicats de similarité retournant un score de pertinence. Ces requêtes spécifient une fonction d'agrégation qui combine les scores individuels produits par les prédicats de similarité, permettant de calculer un score global pour chaque objet. Les k objets avec les meilleurs scores globaux sont retournés dans le résultat final.

Dans cette thèse, nous étudions dans un premier temps les techniques et algorithmes proposés dans la littérature, conçus pour le traitement des requêtes top-k multicritères dans des contextes spécifiques de type et de coût d'accès aux scores, et nous proposons un cadre générique capable d'exprimer tous ces algorithmes. Ensuite, nous proposons une nouvelle stratégie en largeur «breadth-first», qui maintient l'ensemble courant des k meilleurs objets comme un tout, à la différence des stratégies en profondeur habituelles qui se focalisent sur le meilleur candidat. Nous présentons un nouvel algorithme «Breadth-Refine» (*BR*), basé sur cette stratégie et adaptable à n'importe quelle configuration de types et de coûts d'accès aux scores. Nous montrons expérimentalement la supériorité de l'algorithme *BR* sur les algorithmes existants.

Dans un deuxième temps, nous proposons une adaptation des algorithmes *top-k* à la recherche approximative, dont l'objectif est de trouver un compromis entre le temps de recherche et la qualité du résultat retourné. Nous explorons l'approximation par arrêt prématuré de l'exécution et proposons une première étude expérimentale du potentiel d'approximation des algorithmes *top-k*.

Dans la dernière partie de la thèse, nous nous intéressons à l'application des techniques top-k multicritères à la recherche par le contenu dans les grandes bases de données multimédia. Dans ce contexte, un objet multimédia (une image par exemple) est représenté par un ou plusieurs descripteurs, en général sous forme de vecteurs numériques qui peuvent être vus comme des points dans un espace multidimensionnel. Nous explorons la recherche des k plus proches voisins (*k-ppv*) dans ces espaces et proposons une nouvelle technique de recherche *k-ppv* approximative «Multi-criteria Search Algorithm » (*MSA*) basée sur les principes des algorithmes *top-k*. Nous comparons *MSA* à des méthodes de l'état de l'art dans le contexte des grandes bases multimédia où les données ainsi que les structures d'index sont stockées sur disque, et montrons qu'il produit rapidement un très bon résultat approximatif.

Abstract

Efficient processing of ranking queries is an important issue in today information retrieval applications such as meta-search engines on the web, information retrieval in social networks, similarity search in multimedia databases, etc. We address the problem of *top-k* multi-criteria query processing, where queries are composed of a set of ranking predicates, each one expressing a measure of similarity between data objects on some specific criteria. Unlike traditional Boolean predicates returning true or false, similarity predicates return a relevance score in a given interval. The query also specifies an aggregation function that combines the scores produced by the similarity predicates. Query results are ranked following the global score and only the best *k* ones are returned.

In this thesis, we first study the state of the art techniques and algorithms designed for *top-k* multi-criteria query processing in specific conditions for the type of access to the scores and cost settings, and propose a generic framework able to express any top-k algorithm. Then we propose a new breadth-first strategy that maintains the current best k objects as a whole instead of focusing only on the best one such as in all the state of the art techniques. We present Breadth-Refine (*BR*), a new top-k algorithm based on this strategy and able to adapt to any combination of source access types and to any cost settings. Experiments clearly indicate that *BR* successfully adapts to various settings, with better results than state of the art algorithms.

Secondly, we propose an adaptation of *top-k* algorithms to approximate search aiming to a compromise between execution time and result quality. We explore approximation by early stopping of the execution and propose a first experimental study of the approximation potential of *top-k* algorithms.

Finally, we focus on the application of multi-criteria top-k techniques to Large Scale Content-Based Image Retrieval. In this context an image is represented by one or several descriptors, usually numeric vectors that can be seen as points in a multidimensional space. We explore the k-Nearest Neighbors search in such spaces and propose “Multi-criteria Search Algorithm” (*MSA*), a new technique for approximate *k-NN* based on multi-criteria top-k techniques. We compare *MSA* with state of the art methods in the context of large multimedia databases, where the database and the index structure are stored on disk, and show that *MSA* quickly produces very good approximate results.

Table des matières

Introduction 13

1 Etat de l'art 21

1.1 Traitement de requêtes *top-k* multicritères . . . 21

1.1.1 Types de requêtes *top-k* . . . 22

1.1.2 Types d'accès aux scores . . . 23

1.1.3 Niveau d'exécution . . . 25

1.1.4 Certitude des données . . . 26

1.1.5 Fonction de classement . . . 27

1.2 Les algorithmes *top-k* . . . 28

1.2.1 Algorithmes *top-k* pour *S-sources* . . . 28

1.2.2 Algorithmes *top-k* pour *SR-sources* . . . 30

1.2.3 Algorithmes *top-k* avec des accès directs contrôlés . . . 32

1.2.4 Le cas générique . . . 33

1.2.5 Optimalité . . . 35

1.3 La recherche *top-k* approximative . . . 35

1.4 Recherche des k *p*lus *p*roches *v*oisins dans les données multimédia . . . 37

1.4.1 Définition et méthodes . . . 37

1.4.2 Recherche approximative des k-plus proches voisins . . . 39

1.4.3 Méthodes de hachage pour la recherche des k-ppv . . . 40

1.5 Conclusion . . . 44

2 *Breadth-Refine* : une nouvelle stratégie pour le traitement de requêtes *top-k* 47

2.1 Modèle de données . . . 47

2.1.1 Requêtes . . . 48

2.1.2 Sources . . . 48

2.1.3 Candidats . . . 49

2.2 Cadre Générique pour le traitement des requêtes *top-k* . . . 49

2.2.1 Exemples d'algorithmes *top-k* exprimés dans *GF* . . . 51

2.3 La stratégie *Breadth-Refine* . . . 53

2.3.1 L'algorithme *Breadth-Refine (BR)* . . . 53

2.3.2 Les variantes de *Breadth-Refine* . . . 55

2.4 Algorithmes Génériques . . . 59

2.4.1 Necessary Choices 60
2.4.2 Combined Algorithm 62
2.5 Évaluations expérimentales 63
2.5.1 Algorithmes *top-k* spécifiques 64
2.5.2 Évaluations des méthodes génériques 69
2.6 Conclusion 73

3 Les algorithmes *top-k* dans un contexte de recherche approximative par arrêt prématuré 75
3.1 Recherche approximative par arrêt prématuré : motivation 76
3.2 Recherche approximative dans le cadre *GF* 76
3.3 Évaluation de la qualité du résultat approximatif 78
3.4 Évaluation expérimentale 80
3.4.1 Choix des paramètres 80
3.4.2 Potentiel d'approximation 80
3.4.3 Résultats expérimentaux 81
3.5 Conclusion 89

4 Application de la recherche *top-k* multicritères au contenu multimédia 91
4.1 Recherche approximative ϵ-exclusive des k-plus proches voisins 92
4.2 La structure d'index *MSA* 93
4.3 L'algorithme de recherche multicritères : MSA 95
4.4 Évaluations expérimentales 97
4.4.1 Algorithmes testés 97
4.4.2 Paramètres 99
4.4.3 Résultats expérimentaux 99
4.5 Dynamic Multi-Probe LSH : recherche efficace des k-ppv sur disque 104
4.5.1 DMLSH : idée générale 104
4.5.2 Construction de l'index 106
4.5.3 Recherche des k-plus proches voisins 109
4.5.4 Évaluation expérimentale 111
4.6 Conclusion 115

Conclusion 117

Publications 123

Table des figures

1 Exemple de requête *top-k* 13
2 Exemple de sources de données et de traitement de la requête *top-k* de la figure 1 14

1.1 Requête *top-k* de sélection 22
1.2 Requête top-k de jointure 23
1.3 Exemple de requête top-k d'agrégation 24
1.4 Premières étapes de l'algorithme NRA 29
1.5 Premières étapes de l'algorithme TA 31
1.6 Recherche des k plus proches voisins avec ϵ-approximation, $k = 2$ 40
1.7 Construction de l'index LSH : initialisation des cellules (buckets) des L tables de hachage [Gorisse, 2010]. 41
1.8 Distance entre une requête q et sa cellule voisine 43

2.1 Les scores dans une source triée S_j 58
2.2 SR-sources + S-sources : variation de la valeur de k 66
2.3 SR-sources + S-sources : variation de la distribution et de la taille des données 67
2.4 SR-sources + S-sources : variation du nombre de sources en variant le ratio $|SR|/|S|$ . . 67
2.5 SR-sources + R-sources : variation de la distribution et de la taille de la base de données 68
2.6 SR-sources + R-sources : variation de la valeur de k et du nombre de sources $|SR|/|R|$. 69
2.7 SR-sources + S-sources + R-sources : variation de la valeur de k et de la taille de la base ($C_s < C_r$) 70
2.8 SR-sources + S-sources + R-sources : variation de la valeur de k et de la taille de la base ($C_s = C_r$) 70
2.9 SR-sources + S-sources + R-sources : variation de la valeur de k et de la taille de la base ($C_s > C_r$) 71
2.10 Comparaison des stratégies génériques : variation de la distribution 72
2.11 Évaluation des performances des algorithmes génériques dans deux contextes spécifiques 72

3.1 Approximation avec $\mathcal{U}_k$, cadre générique : distribution uniforme 82
3.2 Approximation avec $\mathcal{U}_k$, cadre générique : distribution mixte 82
3.3 Approximation avec $\mathcal{U}_k$, 6 S-sources + 6 SR-sources : distribution uniforme 83
3.4 Approximation avec $\mathcal{U}_k$: 6 S-sources + 6 SR-sources : distribution mixte 83
3.5 Approximation avec $\mathcal{U}_k$, 6 R-sources + 6 SR-sources : distribution uniforme 84
3.6 Approximation avec $\mathcal{U}_k$: 6 R-sources + 6 SR-sources : distribution mixte 85

3.7 Approximation avec $\mathcal{L}_k$, 6 S-sources + 6 R-sources + 6 SR-sources : distribution uniforme 85
3.8 Approximation avec $\mathcal{L}_k$, 6 S-sources + 6 R-sources + 6 SR-sources : distribution mixte . 86
3.9 Approximation avec $\mathcal{L}_k$, 6 S-sources + 6 SR-sources : distribution uniforme 87
3.10 Approximation avec $\mathcal{L}_k$, 6 S-sources + 6 SR-sources : distribution mixte 88
3.11 Approximation avec $\mathcal{L}_k$, 6 R-sources + 6 SR-sources : distribution uniforme 88
3.12 Approximation avec $\mathcal{L}_k$, 6 R-sources + 6 SR-sources : distribution mixte 89

4.1 Exemple de recherche des k plus proches voisins de type ϵ-exclusive 93
4.2 Structure d'index utilisée dans MSA 94
4.3 Exemple d'images de la base de données Holidays 98
4.4 Comparaison des algorithmes *MSA* avec la méthode brute (MB) : temps d'exécution en fonction de la valeur de ϵ 100
4.5 Comparaison des algorithmes *MSA* avec la méthode brute (MB) et la technique *MLSH* : *mAP* en fonction du temps d'exécution 101
4.6 Évaluation de la qualité du résultat approximatif (*mAP*) en fonction du nombre d'accès pour *MLSH* 102
4.7 Évaluation de la qualité du résultat approximatif (*mAP*) en fonction du nombre de table pour *MLSH* 103
4.8 Utilisation des fonctions de hachage pour la création des cellules 105
4.9 Structure d'une entrée dans la feuille de l'arbre *DMLSH* 106
4.10 Exemple d'index *DMLSH* 110
4.11 Effet de la variation de M sur le coût d'entrée/sorties 113
4.12 Effet de la variation de M sur le rappel 114
4.13 Effet de la variation de L sur le coût d'entrée/sorties 114
4.14 Effet de la variation de L sur le rappel 115
4.15 Effet de la variation de T sur le coût d'entrée/sorties 115
4.16 Effet de la variation de T sur le rappel 116

La souris est un animal qui, tué en quantité suffisante et dans des conditions contrôlées, produit une thèse de doctorat.

– Woody Allen

Introduction

Le développement de techniques de traitement des requêtes de classement est un axe de recherche très actif dans le domaine de la recherche d'information. Contrairement aux requêtes booléennes traditionnelles, dans lesquelles le filtrage est basé sur des prédicats qui retournent vrai ou faux, les requêtes de classement utilisent des prédicats de similarité retournant un score de pertinence. Ces requêtes spécifient une fonction d'agrégation qui combine les scores individuels produits par les prédicats de similarité permettant de calculer un score global pour chaque objet. Le résultat des requêtes de classement est donc un ensemble d'objets (n-uplets dans les bases de données relationnelles) triés par score. Chaque objet est représenté par un identifiant et un score permettant de mesurer sa pertinence et sa similarité par rapport à la requête. Le résultat d'une requête de classement est généralement l'ensemble des k meilleurs objets, le plus souvent ceux avec les scores les plus élevés, cet ensemble est appelé *top-k*, et la requête est dite simplement *requête top-k*.

Considérons l'exemple d'un touriste à Paris qui prend en photo un détail particulier d'un monument, et cherche le meilleur monument selon trois critères : *(i)* proche de l'endroit où il se trouve, *(ii)* reproduisant un détail similaire à celui qu'il vient de photographier, et *(iii)* exposant une sculpture de la Renaissance. Les objets sont ici les monuments dans Paris, et la requête est composée de trois prédicats de classement :

- p_1 : proximité spatiale de l'emplacement actuel du touriste.
- p_2 : similarité à une image prise d'un autre monument.
- p_3 : description textuelle contient l'expression "sculpture de la Renaissance".

select * **from** Monument m
order by *near*(m.address, here()) +
similar(m.photo, myPhoto) +
ftcontains(m.descr, 'Renaissance sculpture')
limit 1

FIGURE 1 – Exemple de requête *top-k*

La figure 1 montre un exemple d'une requête *top-k* permettant d'exprimer cette recherche en utilisant la syntaxe *Rank SQL* [Li et al., 2005]. Dans cette requête, nous considérons $k = 1$, et une fonction d'agrégation simple, basée sur la somme des scores.
Comme dans cet exemple, les prédicats de classement coûteux proviennent souvent de l'évaluation de

S_1 (S)	S_2 (SR)	S_3 (R)
$(o_2, 0.4)$	$(o_3, 0.9)$	$(o_1, 0.9)$
$(o_1, 0.3)$	$(o_1, 0.2)$	$(o_2, 0.7)$
$(o_4, 0.25)$	$(o_4, 0.15)$	$(o_3, 0.8)$
$(o_3, 0.2)$	$(o_2, 0.1)$	$(o_4, 0.6)$

Source/Accès	Objets trouvés	*candidats*	U_{unseen}
		$\emptyset$	3.0
S_1/S	$(o_2, 0.4)$	$\{(o_2, [0.4, 2.4])\}$	2.4
S_2/S	$(o_3, 0.9)$	$\{(o_2, [0.4, 2.3]), (o_3, [0.9, 2.3])\}$	2.3
S_2/R	$(o_2, 0.1)$	$\{(o_2, [0.5, 1.5]), (o_3, [0.9, 2.3])\}$	2.3
S_3/R	$(o_3, 0.8)$	$\{(o_2, [0.5, 1.5]), (o_3, [1.7, 2.1])\}$	2.3
S_2/S	$(o_1, 0.2)$	$\{(o_2, [0.5, 1.5]), (o_3, [1.7, 2.1]), (o_1, [0.2, 1.6])\}$	1.6

FIGURE 2 – Exemple de sources de données et de traitement de la requête *top-k* de la figure 1

la similarité entre des images, des textes, des positions géographiques et autres types de données multimédia, dont le contenu est décrit par des valeurs numériques. Cela implique une recherche coûteuse dans des espaces avec plusieurs dimensions, souvent basée sur des structures d'index multidimensionnel spécifiques [Böhm et al., 2001].

Dans de nombreux cas, les prédicats sont évalués par des sites spécialisés et/ou distants qui fournissent des services web spécifiques, tel que les cartes géographiques ou géoservice évaluant la proximité spatiale, les sites de partage de photos permettant une recherche des images similaires, les sites web spécialisés proposant un classement des hôtels, des restaurants, etc. L'accès internet à ce genre de services implique une évaluation coûteuse des prédicats par des sites indépendants et distants. Cela s'ajoute à un contrôle sur l'évaluation du prédicat souvent minime, et se réduit à appeler le service web fourni.

Pour chaque requête, un prédicat de classement peut produire un score de pertinence pour chaque objet. Dans le reste du manuscrit, nous appelons *source* la liste de scores des objets, produite par le prédicat de classement. La liste de scores peut être produite, par exemple, en se basant sur une structure d'index local qui retourne les résultats par ordre de pertinence. Nous considérons ici le cas général, où l'accès aux scores de la source est limité à des accès séquentiels (triés) et/ou directs. Ces deux types d'accès permettent d'avoir trois types possibles pour une source *S* :

- *S-source* : une source avec uniquement des accès séquentiels (triés), en utilisant l'opérateur *getNext(S)* qui permet de retourner à chaque itération le prochain meilleur score s et l'identifiant de l'objet o possédant ce score.
- *R-source* : avec uniquement des accès directs, en utilisant l'opération *getScore(S,o)* qui retourne le score de l'objet o dans la source S.
- *SR-source* : une source avec les deux types d'accès, séquentiel et direct.

La figure 2 présente un exemple de configuration des sources et de traitement possible de la requête *top-k* de la figure 1.

La première partie de la figure présente les trois sources à consulter : S_1 est une source triée, accessible uniquement par des accès séquentiels (*S-source*) permettant de retourner à chaque fois le meilleur objet suivant de la liste dans l'ordre décroissant des scores ; S_2 est une *SR-source* accessible par des accès séquentiels et directs, et S_3 est une source non triée (*R-source*) accessible par des accès directs. Chaque source présente un classement différent des mêmes objets. Dans la figure 2, les deux premières sources sont présentées par des listes triées dans l'ordre décroissant des scores. La troisième source classe les objets simplement par identifiant et non par score. Le score d'un objet dans une source est dit score local et l'agrégation de tous les scores d'un objet permet de calculer le score final (global) de l'objet. Ici nous supposons que les scores locaux des objets sont dans l'intervalle $[0, 1]$, donc l'intervalle de score global possible pour n'importe quel objet est initialement $[0, 3]$.

En bas de la figure est illustré un traitement possible de la requête *top-k* de la figure 1. Nous supposons que les objets ne sont pas connus à l'avance, mais qu'ils sont découverts suite aux accès séquentiels. La première colonne montre l'ordre des sources interrogées, et à chaque fois le type d'accès réalisé. Pour chaque accès à une source, la deuxième colonne précise l'identifiant et le score de l'objet retourné. *candidats* est l'ensemble des candidats trouvés et U_{unseen} est le score maximum que peut avoir un objet qui n'est pas encore découvert. Initialement l'ensemble *candidats* est vide et $U_{unseen} = 3$ puisque la fonction d'agrégation est ici la somme et le score maximum dans chaque source est 1. Grâce à la monotonie de la fonction d'agrégation, chaque accès trié permet de baisser la valeur de U_{unseen}. En effet, le score maximum possible pour les accès suivants baisse dans une source triée après chaque accès. Cette baisse monotone de la valeur de U_{unseen} permet d'avoir des informations plus précises sur les scores des objets qui ne sont pas encore retournés.

L'exemple d'exécution de la figure 2 permet de trouver le *top-1* ($k = 1$) en suivant les étapes suivantes :

- Un premier accès séquentiel à S_1 permet de retourner l'objet o_2 avec un score local de 0.4. Alors l'intervalle de score de o_2 passe de $[0, 3]$ à $[0.4, 2.4]$. En même temps, la valeur de U_{unseen} devient 2.4 puisque le score maximum possible dans S_1 ne dépassera pas désormais 0.4.
- Un accès séquentiel à S_2 permet d'abord de découvrir $(o_3, 0.9)$. Un nouveau candidat o_3 est alors ajouté à l'ensemble *candidats* avec l'intervalle de score $[0.9, 2.3]$. Ensuite il permet de baisser la valeur de U_{unseen} qui devient 2.3 (la somme de 0.4 score maximum dans S_1, 0.9 score maximum dans S_2 et 1 score maximum dans S_3). Enfin, la valeur maximale de l'intervalle de score de o_2 baisse à 2.3 car cet objet n'est pas encore découvert dans S_2 et donc son score local ne peut pas dépasser 0.9 dans cette source.
- Un accès direct à S_2, pour retourner le score local de o_2 (0.1), permet de mettre à jour son intervalle de score. Ce dernier devient $[0.5, 1.5]$.
- Un accès direct à S_3 pour découvrir le score local de o_3 (0.8), met à jour son intervalle de score : $[1.7, 2.1]$.
- Un accès séquentiel à S_2 retourne $(o_1, 0.2)$ et baisse la valeur de U_{unseen} à 1.6. Les intervalles de scores de o_3 et o_2 ne changent pas, car leurs scores locaux sont déjà connus pour cette source.

- Maintenant, en analysant la liste des candidats, nous remarquons que le score maximum possible pour un objet non découvert (exprimé par U_{unseen}) est inférieur au score minimum de o_3. En même temps le score minimum de ce dernier est supérieur à tous les scores maximums des autres candidats. Dans ce cas, nous pouvons déjà conclure que o_3 est le meilleur objet (*top-1*).

Plusieurs techniques pour le traitement de requêtes *top-k* ont été proposées dans ce contexte. Elles considèrent généralement des conditions spécifiques pour le type d'accès aux sources (par exemple, seulement des accès séquentiels, ou les deux types d'accès dans toutes les sources, etc) et pour le coût des accès (elles considèrent généralement qu'un accès séquentiel est moins coûteux qu'un accès direct). Ces méthodes proposent différentes heuristiques pour choisir la prochaine source à interroger en se basant généralement sur le classement actuel des candidats dans l'ensemble *candidats* et/ou le score du meilleur candidat afin d'affiner son score. Nous montrons dans un premier temps que les différents algorithmes proposés dans notre contexte, peuvent être exprimés dans un cadre générique commun. Nous proposons dans ce sens un cadre générique *GF* (Generic Framework), permettant d'exprimer n'importe quel algorithme *top-k* indépendamment des types et des coûts d'accès considérés. Nous nous servons du cadre *GF* pour comparer les stratégies utilisées dans les différents algorithmes et techniques proposés pour le traitement des requêtes *top-k*.

Dans un deuxième temps, nous proposons une nouvelle stratégie pour le traitement des requêtes *top-k* appelée *Breadth-Refine* (*BR*). *BR* est un algorithme *top-k* générique, fonctionnant pour toute combinaison de types d'accès aux sources et de coûts d'accès. Il utilise une heuristique *breadth-first*, visant à affiner le score de tous les candidats du *top-k* courant, afin de maintenir à chaque étape le meilleur *top-k* courant possible et de s'approcher rapidement du *top-k* final. Pour confirmer l'efficacité de notre stratégie, nous proposons un large panel d'expériences dans lequel nous comparons *BR* et ses variantes avec les techniques de l'état de l'art. Ces expériences, montrent la capacité d'adaptation de *BR* à des contextes spécifiques concernant le type et le coût d'accès aux sources. Très peu d'algorithmes génériques (considérant n'importe quelle configuration de type et de coût d'accès aux sources) ont été proposés, et sont difficilement comparables avec *BR* à cause de différentes hypothèses supplémentaires considérées. Pour comparer avec *BR*, nous proposons des variantes génériques adaptés à notre contexte, et nous montrons l'efficacité de notre stratégie par rapport à celles proposées par les méthodes de l'état de l'art.

Les algorithmes *top-k* restent coûteux surtout dans un contexte où les sources sont distantes et le résultat final doit être exact, c'est à dire avoir les garanties suffisantes que les objets retournés dans le résultat final ont les meilleurs scores parmi tous ceux qui sont dans les sources. Pour baisser le coût d'exécution, nous explorons la recherche *top-k* approximative. Dans certaines approches, la recherche approximative se base sur des modèles probabilistes ou sur de l'échantillonnage, mais ces techniques sont plus difficiles à mettre en œuvre. Nous nous intéressons à une technique d'approximation très simple, l'approximation par arrêt prématuré de l'exécution, en considérant les k meilleurs candidats à ce moment là.
Nous proposons, dans ce sens, une variante de *BR* pour le traitement de requêtes *top-k* multicritères, et nous généralisons cette technique à l'ensemble des algorithmes du cadre générique *GF*. Le choix du moment de l'arrêt prématuré de l'exécution peut correspondre à un coût maximal pour le traitement d'une requête *top-k*.
Nous présentons des évaluations expérimentales dans lesquelles nous mesurons la qualité des résultats retournés suite à un arrêt prématuré de l'exécution. Nous montrons que l'algorithme BR présente constamment un *top-k* courant proche du résultat exact, et s'adapte parfaitement à une recherche approximative

utilisant une stratégie basée sur l'arrêt prématuré de l'exécution (*early stopping*).

Dans la deuxième partie de la thèse, nous nous plaçons dans le contexte de l'application de la recherche multicritères à la recherche des k-plus proches voisins dans des bases de données multimédia volumineuses, et plus particulièrement les bases de données d'images. Dans ce contexte, la recherche est basée sur le contenu des images, et nous parlons de *CBIR* (Content-Based Image Retrieval). Nous montrons que la recherche des k-ppv peut être réduite à un problème *top-k* multicritères, et nous proposons une nouvelle structure d'index et un nouvel algorithme pour la recherche des k images les plus similaires à une image requête (recherche par similarité du contenu).
Dans le contexte *CBIR*, chaque image est représentée par un ou plusieurs descripteurs représentés par exemple sous forme d'un vecteur numérique multidimensionnel . Dans ce cadre, la recherche par similarité du contenu consiste à trouver les vecteurs les plus proches à un vecteur requête : k-plus proche voisins (k-ppv). Les scores des images sont ici remplacés par les distances calculées entre la requête et les objets de la base. Nous réduisons la recherche des k-ppv à un problème *top-k* en considérant chaque dimension comme un critère indépendant produisant des scores qui représentent la distance sur cette dimension. Nous proposons un nouvel algorithme *MSA* (Multcriteria Search Algorithm) basé sur une structure d'index permettant de trouver rapidement les objets les plus proches à la requête pour chaque dimension. L'algorithme *MSA* utilise les techniques *top-k* de l'état de l'art pour trouver les k-plus proches voisins. Nous considérons de grands volumes de données, nécessitant le stockage de l'index et des vecteurs sur disque.

Un des problèmes majeurs de la recherche k-ppv est la *malédiction de la dimensionnalité* : pour des vecteurs de grande dimension, les structures d'index classiques (hiérarchiques de division de l'espace ou des données) sont inefficaces, car chaque zone a trop de zones voisines, et le nombre de vecteurs proches est généralement trop important. Dans ce cas, un parcours séquentiel de la base est plus efficace. Dans un tel contexte, la recherche approximative est une solution très prometteuse, d'autant plus que les vecteurs utilisés comme descripteurs des objets de la base sont intrinsèquement imprécis, car ils sont basés sur des caractéristiques de bas niveau du contenu multimédia. Nous parlons alors de fossé sémantique (semantic gap).

Dans ce contexte, nous considérons les techniques de recherche *top-k* approximative par arrêt prématuré de l'exécution appliquées à la méthode *MSA*.

MSA introduit un nouveau type de recherche approximative des k-ppv, permettant de garantir que tous les images non trouvées pendant la recherche sont à une distance minimale de la requête ϵ. Nous appelons cette recherche ϵ-exclusive. Des évaluations expérimentales sur une vraie base de données d'un million d'images, montrent que *MSA* peut être, au moins, trois fois plus rapide que la méthode brute, qui consiste à parcourir tous les vecteurs de la base pour les comparer à la requête et retourner les k images les plus similaires. Nous proposons aussi une comparaison de *MSA* avec la méthode de référence dans la recherche approximative des k-plus proches voisins, *LSH*, et plus particulièrement la variante *Multi-Probe LSH*.

Les méthodes de recherche approximative des k-plus proches voisins les plus utilisées dans ce contexte, sont basées sur le hachage sensible à la localisation. *LSH* (Locality Sensitive Hashing) est considérée comme la méthode de référence pour ce type de recherche. Elle utilise des fonctions de hachage permettant de grouper les objets (images) similaires dans les mêmes cellules avec une forte probabilité. Un des

problèmes de *LSH* est que dans chaque table de hachage nous consultons une seule cellule, et que pour augmenter le rappel, le nombre de tables nécessaires est très grand. *MultiProbe LSH* (*MLSH*) est une variante de *LSH* permettant d'accéder à plusieurs cellules dans la même table de hachage dans le but de réduire le nombre de tables.

Dans le contexte de *MSA*, une recherche par similarité du contenu dans une base d'images de grande taille nécessite l'utilisation du disque. Dans ce cadre, la technique utilisée dans *MultiProbe LSH* risque de perdre de son efficacité puisque les objets de cellules voisines peuvent être dans des emplacements aléatoires sur différentes pages disque.

Nous proposons dans la dernière partie de cette thèse, une optimisation de la méthode *MultiProbe LSH* (*MLSH*) avec comme objectif la réduction des entrées/sorties disque. Notre nouvelle méthode *Dynamic MultiProbe LSH* (*DMLSH*), permet de regrouper des cellules voisines sur une seule page disque pour limiter le nombre de lecture de pages. Nous évaluons expérimentalement notre méthode, et nous montrons que le nombre d'entrées/sorties ne dépasse pas celui réalisé par la méthode *MultiProbe LSH*, tout en obtenant un meilleur rappel.

En bref, les principales contributions de cette thèse sont les suivantes :

- Un cadre générique permettant d'exprimer n'importe quel algorithme *top-k* dans notre contexte, qui servira de base d'analyse et de comparaison des différentes stratégies utilisées dans les algorithmes.
- Une nouvelle stratégie de recherche *top-k Breadth-Refine* (*BR*), considérant l'ensemble des candidats du *top-k* comme un tout au lieu de se focaliser sur le meilleur candidat seulement. *BR* est un algorithme générique qui s'adapte à n'importe quelle configuration de types et de coûts d'accès aux sources.
- Une comparaison expérimentale de *BR* avec les algorithmes spécifiques de référence de l'état de l'art.
- Une première comparaison expérimentale entre des techniques génériques, montrant l'efficacité de notre stratégie par rapport aux méthodes de l'état de l'art adaptées à notre contexte.
- Une adaptation des algorithmes *BR* pour une recherche approximative *top-k* centralisée au contexte *GF*, et une première étude expérimentale permettant d'évaluer et de comparer le potentiel d'approximation des algorithmes *top-k*.
- Une application des algorithmes *top-k* dans le cadre de la recherche par similarité du contenu dans les bases de données multimédia, et la proposition d'un algorithme (*MSA*) pour la recherche approximative des k-plus proches voisins dans une base de descripteurs. Des évaluations expérimentales sur une base d'un million d'images montrant la supériorité de *MSA* par rapport à la *méthode brute* et *MLSH*.
- Une amélioration de la méthode *MultiProbe LSH* pour un contexte où l'utilisation d'une mémoire secondaire est nécessaire (grands volumes de données).

Le manuscrit est organisé en quatre chapitres ; dans le premier nous passons en revue les différentes techniques de la recherche *top-k* dans les deux cas de recherche : exacte et approximative. Nous abordons

également dans ce chapitre, les différentes techniques de recherche par similarité du contenu dans une base de données multimédia, en particulier les bases d'images.
Le deuxième chapitre aborde la première contribution de cette thèse, en présentant le Cadre Générique *GF* et l'algorithme *Breadth-Refine* avec ses différentes variantes, ainsi qu'une évaluation expérimentale de *BR* et une comparaison avec les algorithmes de l'état de l'art.
Dans le chapitre trois, nous abordons la notion d'approximation dans les algorithmes *top-k*, et nous montrons, grâce à des évaluations expérimentale, l'efficacité de *BR* pour une approximation par arrêt prématuré de l'exécution. Le chapitre quatre concerne l'étude des algorithmes *top-k* dans la recherche par similarité du contenu dans les bases de données multimédia, et l'optimisation de la méthode *MultiProbe LSH* pour une recherche k-ppv dans le cas de grands volumes de données sur disque.
Nous finirons ce manuscrit par une conclusion générale et des perspectives.

L'information est un flux illimité dans un espace limité .

– Joêl Dicker

CHAPITRE 1

Etat de l'art

Dans un premier temps, ce chapitre passe en revue les techniques de traitement de requêtes multicritères *top-k* dans un cadre général. Nous commençons par présenter les différents paramètres qui distinguent les diverses catégories de recherche *top-k* : type de requête, fonction d'agrégation, niveau d'exécution, accès aux données. Ensuite, nous parlons des techniques proposées pour le traitement de requêtes *top-k*. Dans un deuxième temps, nous abordons la notion d'approximation dans la recherche *top-k*. Dans la troisième partie de ce chapitre, nous considérons le cas de la recherche dans les bases de données multimédia, en particulier la recherche par similarité de contenu dans les bases d'images, et nous présentons les techniques les plus significatives dans ce contexte.

Les techniques de traitement de requêtes de classement *top-k* ont été largement étudiées ces dernières années et à plusieurs niveaux : modèle de requêtes, types d'accès aux données, niveau d'exécution, fonction d'agrégation, certitude des données, etc. L'étude réalisée dans [Ilyas et al., 2008] présente un riche aperçu de ces différentes techniques, que nous passons en revue dans ce chapitre.

1.1 Traitement de requêtes *top-k* multicritères

Dans cette section, nous étudions les différents paramètres liés au traitement de requêtes *top-k* multicritères. Ces paramètres définissent le cadre dans lequel s'effectue la recherche *top-k* : type de requête, type d'accès aux scores, etc. Nous commençons par présenter les différents types des requêtes *top-k*, ensuite, nous abordons le problème d'accès aux scores et des différents types d'accès considérés dans la littérature. Enfin, nous terminons par définir les différents niveaux d'exécution des requêtes, et le choix des fonctions de classement qui permettent d'établir un classement final des objets afin de dégager les k meilleures solutions.

Select *
From $R\ r$
Where Sélection(r)
Order by $\mathcal{F}(p_1(r), ..., p_m(r))$
Limit k;

FIGURE 1.1 – Requête *top-k* de sélection

1.1.1 Types de requêtes *top-k*

Plusieurs façons d'exprimer une requête de classement sont possibles ; dans la littérature trois types principaux de requêtes *top-k* ont été distingués :

- Requêtes *top-k* de sélection ;
- Requêtes *top-k* de jointure ;
- Requêtes *top-k* d'agrégation.

Requêtes *top-k* de sélection

La requête *top-k* de sélection est la plus simple et la plus utilisée dans la littérature [Fagin et al., 2001] [Güntzer et al., 2000] [Fagin et al., 2002] [Güntzer et al., 2001]. Ce type de requête spécifie les critères de classement, la valeur de k, éventuellement une fonction de filtrage (sélection) des objets, et la fonction d'agrégation qui permet d'établir un classement des objets selon l'ensemble de critères. Chaque critère de recherche est évalué par un prédicat permettant d'avoir un score local pour chaque objet. Une fonction de fusion (agrégation), généralement monotone, est utilisée pour calculer le score global de chaque objet. Les objets avec les k meilleurs scores sont retournés dans le résultat. La figure 1.1 montre un modèle de requête *top-k* de sélection. Dans cet exemple, m est le nombre de critères de recherche et $\mathcal{F}$ est la fonction de fusion. Cette dernière permet l'agrégation des scores locaux des objets afin de calculer un seul score global par objet. Le score final X d'un objet o est calculé de la manière suivante :

$$X = \mathcal{F}(x_1, ..., x_m) \tag{1.1}$$

x_i est le score local de o retourné par le prédicat p_i, et m est le nombre de critères de la recherche.

Le traitement de requêtes *top-k* de sélection est le sujet de nombreux travaux récents qui ont donné lieu à plusieurs algorithmes pour traiter ce type de requêtes, parmi lesquels, nous pouvons citer les algorithmes *NRA* et *TA* [Fagin et al., 2002], l'algorithme Upper [Bruno et al., 2002], etc ; nous passons en revue les algorithmes *top-k* dans la section 1.2.

Dans cette thèse, nous proposons une nouvelle stratégie pour le traitement de requêtes top-k de sélection, que nous comparons avec les différentes techniques déjà proposées dans la littérature.

Autres types de requêtes *top-k*

Deux autres types de requêtes ont été distingués dans la littérature, la requête *top-k* de jointure et la requête *top-k* d'agrégation. Dans les requêtes *top-k* de jointure [Natsev et al., 2001] [Ilyas et al., 2004]

[Zhang et al., 2006], les scores sont attribués aux objets suite à une ou plusieurs opérations de jointure. Comme les requêtes de sélection, les requêtes *top-k* de jointure retournent les k meilleurs objets en se basant toujours sur une fonction de fusion généralement *monotone* permettant d'établir un classement final des objets.

Select *
From $R_1, ..., R_m$
Where Jointure $(R_1, ..., R_n)$
Order by $\mathcal{F}(p_1, ..., p_m)$
limit k;

FIGURE 1.2 – Requête top-k de jointure

La figure 1.2 présente un modèle de requête *top-k* de jointure utilisée pour la recherche dans les bases de données relationnelles, où les opérations de jointure sont très fréquentes. La requête *top-k* de jointure est exécutée à l'intérieur du moteur de la base de données, puisqu'elle peut débuter par des opérations de jointure. Ici, l'objectif est d'optimiser l'exécution des requêtes *top-k* en élaborant un plan d'exécution efficace.

Dans les requêtes *top-k* d'agrégation [Li et al., 2006] [Ré et al., 2007], l'objectif est d'attribuer un score à un groupe d'objets et non individuellement. Une requête *top-k* d'agrégation retourne les k meilleurs groupes d'objets. La figure 1.3 montre un exemple d'utilisation d'une requête top-k d'agrégation dont l'objectif est de retourner les 3 meilleurs départements de France en termes de services proposés aux étudiants et où le nombre d'habitants dépasse les 48.000 personnes. Dans [Li et al., 2006], les auteurs introduisent un nouvel opérateur de *ranking* pour permettre une exécution efficace des requêtes *top-k* agrégation, qui traditionnellement consiste à évaluer tous les objets des différents groupes, ensuite trier les groupes d'objets pour dégager à la fin un top-k. Les auteurs proposent deux nouveaux principes dans l'exécution d'une requête top-k, le classement de groupe et le classement d'objets (n-uplet). Le premier consiste à évaluer les groupes d'objets selon un ordre établi, avec une priorité aux groupes où les objets ont les meilleurs scores. Ce qui permet de ne pas évaluer tous les objets des groupes qui ne sont pas dans le *top-k* courant. Le deuxième principe consiste à établir dans chaque groupe un ordre d'évaluation des objets, ces derniers peuvent êtres triés par ordre croissant et/ou décroissant des scores. Ce principe permet d'avoir des informations supplémentaire sur les objets non-évalués comme le score maximum (minimum) possible dans le cas d'une fonction d'agrégation monotone.

Dans cette thèse, nous nous limitons au cas le plus général, celui du traitement de requêtes *top-k* de sélection. Les deux autres modèles de requêtes peuvent êtres considérés comme des extensions du modèle général dans lequel des opérations d'agrégation ou de jointure sont rajoutées au traitement de base. Nous proposons une nouvelle technique de traitement de requêtes *top-k* de sélection, et un cadre générique pour exprimer tous les algorithmes proposés pour répondre à ce type de requêtes.

1.1.2 Types d'accès aux scores

Le traitement des requêtes de classement *top-k* multicritères nécessite l'évaluation de plusieurs critères de recherche, produisant des scores de pertinence individuel. Nous appelons *source* la liste des scores

Select code-Postal, AVG (nb-Residence-Etud + 2*nb-Cinema + 2*nb-Biblio) as score
From $Ville$
Where $Ville.population \geq 48.000$
Group By département
Order by $score$
limit 3;

FIGURE 1.3 – Exemple de requête top-k d'agrégation

produits par un critère pour tous les des objets de la base. Pour une requête *top-k* composée de m critères de recherche, le traitement consiste à interroger m listes de scores permettant différents classements du même ensemble d'objets, jusqu'à retourner le *top-k* final. L'accès à la liste de scores peut se faire en suivant l'ordre décroissant des scores, ou non. Pour consulter les sources, deux types d'accès sont distingués :

- Séquentiel (sorted) : permettant de retourner à chaque fois l'objet suivant dans la liste, suivant l'ordre décroissant des scores. Ce type d'accès est réalisé par une opération *getNext*(S). Cette opération permet de retourner l'objet suivant dans la source S ainsi que son score.
- Direct (random) : permettant de retourner le score d'un objet donné. L'accès direct est effectué par une opération *getScore*(o, S). Cette opération permet de retourner le score local de l'objet o dans la source S.

En se basant sur ces deux types d'accès aux scores, trois catégories de sources sont distingués dans la littérature :

- *S-source* : pour *Sorted-Source*, représente une source triée par ordre décroissant des scores, accessible uniquement par des accès séquentiels.
- *R-source* : pour *Random-Source*, représente une source non triée, où le seul type d'accès possible est l'accès direct permettant de retourner le score d'un objet donné.
- *SR-source* : pour *Sorted Random-Source*, représente une source accessible via les deux types d'accès (séquentiel et direct).

Les accès aux sources ont généralement des coûts différents en fonction du type d'accès. Un accès séquentiel est généralement considéré comme moins coûteux, puisqu'il se base le plus souvent sur une structure d'index permettant de retourner directement l'objet suivant dans la liste. Inversement, l'accès direct consiste souvent à effectuer une recherche séquentielle pour trouver l'objet concerné par l'accès et retourner son score local dans la liste.

Par rapport au type d'accès aux sources et aux différences de coût entre les types d'accès, les algorithmes *top-k* proposés jusqu'ici peuvent êtres divisés en quatre catégories :

- *Accès direct et séquentiel*. Dans cette catégorie, les sources sont accessibles via des accès directs et séquentiels (*SR-sources*). Parmi les algorithmes proposés, le plus connu est l'algorithme *TA* (Threshold Algorithm) [Fagin et al., 2001], qui considère l'accès à tour de rôle à toutes les sources.

Dans la même catégorie, l'algorithme *Quick Combine* [Güntzer et al., 2000] diffère de *TA* en faisant un choix de la prochaine source à chaque pas.

- *Sans accès direct*. Nous considérons dans cette catégorie des sources accessibles uniquement par accès séquentiel (*S-sources*). L'algorithme *NRA* [Fagin et al., 2002] (No Random Access), est le plus connu dans cette catégorie. Comme *TA*, il accède à tour de rôle à toutes les sources. L'algorithme *Stream-Combine* [Güntzer et al., 2001] est une variante de *NRA* qui, comme *Quick Combine*, fait un choix de la prochaine source à consulter suivant un calcul de bénéfice de chaque source.

- *Accès séquentiel avec des accès directs contrôlés*. Cette troisième catégorie des techniques de traitement de requêtes *top-k* considère généralement une source avec accès séquentiel (*S-source*) permettant de découvrir les objets, et plusieurs sources non triées (*R-sources*) interrogées par des accès directs pour calculer les scores finaux des objets découverts dans la première source. Le coût d'un accès direct est souvent considéré comme beaucoup plus important que celui d'un accès séquentiel, pour cela différents paramètres sont utilisés pour contrôler et limiter si possible le nombre d'accès directs. Les algorithmes *top-k* les plus connus dans cette catégorie sont *Upper* et *Pick* [Bruno et al., 2002] [Marian et al., 2004], *MPro* (pour Minimal Probing) [Chang and Hwang, 2002], Rank-Join [Ilyas et al., 2004], *RankSQL* [Li et al., 2005], etc.

- *Tous types d'accès*. Le dernier cas est celui qui a été le moins traité dans la littérature. Cette catégorie représente le cas générique, avec des algorithmes capables de traiter n'importe quelle configuration de types de sources et de coût d'accès. A notre connaissance, le seul travail réalisé dans cette catégorie est l'algorithme *Necessary Choices* [Hwang and Chang, 2007].

Dans la section 1.2, nous présenterons en détail ces algorithmes, groupés selon ce même critère, le mode d'accès aux scores.

1.1.3 Niveau d'exécution

Le traitement des requêtes *top-k* sur des données stockées dans un *SGBD* peut être intégré dans le moteur du *SGBD* ou gardé à l'extérieur, au niveau applicatif. La première approche (traitement à l'extérieur du moteur de la base de données) est la plus courante, mais quelques travaux ont proposé des solutions d'intégration dans le *SGBD*. Dans ce cas, il faut intégrer le classement des données (un score et un ordre pour chaque objet) dans le modèle de données, dans les opérateurs et dans l'optimisation des requêtes. Dans cette approche, le plan d'exécution de la requête *top-k* doit considérer l'opération de classement comme tout autre opérateur classique (sélection, projection, jointure, etc).

Dans ce contexte, nous distinguons deux niveaux d'intégration du traitement des requêtes *top-k* avec le moteur de la base de données :

- *Haut niveau* (application). Le traitement des requêtes de classement *top-k* est considéré ici comme une application sur une couche de haut niveau à l'extérieur du moteur de la base de données. Une ou plusieurs structures particulières peuvent être nécessaires pour accéder aux scores des critères, mais le noyau et le fonctionnement du moteur de la base restent inchangés. La grande majorité des techniques de traitement des requêtes *top-k* considère la propriété de classement à l'extérieur du moteur de la base de données, et propose des algorithmes nécessitant des structures particulières pour accéder aux scores.

- *Bas niveau* (moteur de la base). Ce niveau concerne toutes les techniques qui nécessitent des adaptations et des modifications dans le moteur de la base de données pour supporter la propriété de classement et la considérer au niveau des autres opérateurs dans le modèle relationnel. Dans [Li et al., 2005], les auteurs proposent le système *RankSQL*, le premier cadre algébrique permettant une évaluation efficace des requêtes de classement *top-k* dans le moteur de base de données relationnelles. il s'agit de la première formalisation de l'interaction entre l'opérateur de classement et les opérateurs relationnels et permet d'intégrer l'opérateur de classement à la génération du plan d'exécution de la requête. Plus précisément *RankSQL* modifie le modèle de données pour attacher un score aux n-uplets et un ordre aux collections de n-uplets et propose un nouvel opérateur de classement des données μ. Les autres opérateurs classiques (jointure, sélection, projection, etc) sont adaptés à ce modèle pour interagir avec ce nouvel opérateur.

Dans cette thèse, nous nous limitons au cas général du traitement des requêtes *top-k* en dehors du moteur de la base de données.

1.1.4 Certitude des données

Un résultat final *top-k* peut être soit approximatif soit exact, selon le contexte de la recherche et la technique utilisée. Dans le cas de grands volumes de données, plusieurs techniques favorisent l'efficacité et la rapidité d'exécution par rapport à la précision des résultats, la recherche approximative sera la solution adoptée dans ce genre de situation. Dans un autre sens le résultat approximatif peut être dû simplement à la nature même des données. Dans des modèles probabilistes, les données sont incertaines et par conséquent un résultat approximatif est la conséquence directe de ce type de données. Dans un contexte plus classique, une base de données relationnelle par exemple, la recherche dans des données certaines ou exactes peut avoir deux types de résultat, approximatif ou exact, selon la technique utilisée. Tout cela nous permet de classer les techniques de traitement de requêtes *top-k* selon la certitude des données et des résultats retournés :

- *Données certaines et résultat exact* : cette configuration est adoptée par la grande majorité des travaux réalisés pour le traitement de requêtes *top-k*. Ces techniques s'appuient sur des données certaines pour retourner le résultat *top-k* de la meilleur qualité. Dans ce cas, l'exécution s'arrête quand toutes les garanties d'avoir un résultat exact sont obtenues. Les techniques utilisées dans cette catégorie seront détaillées dans la section 1.2.

- *Données certaines et résultat approximatif* : la recherche d'un résultat approximatif d'une requête *top-k* à partir de données certaines est très pratiquée dans la recherche multicritères, et a pour objectif de réduire le coût d'exécution des requêtes *top-k*. Plusieurs techniques ont été proposées dans cette catégorie, généralement pour des applications de recherche dans les données multimédia [Philipp-Foliguet et al., 2006] [Amato et al., 2003a] [Theobald et al., 2005] [Fagin et al., 2003].

- *Données incertaines* : dans cette catégorie, nous considérons toutes les techniques de traitement de requêtes *top-k* se basant sur des données probabilistes [Zhan et al., 2012] [Considine et al., 2004] [Cheng et al., 2004] [de Almeida and Güting, 2005], etc. Plusieurs modèles probabilistes ont été proposés pour traiter des requêtes *top-k* sur des données incertaines [Friedman et al., 1999] [Fuhr, 1990] [Lakshmanan et al., 1997] [Abiteboul et al., 1987] [Barbará et al., 1992], etc. Nous ne considérons pas les données incertaines dans cette thèse.

Dans la première partie de la thèse, nous nous situons dans le cadre général, des données certaines avec un résultat exact. Dans un deuxième temps, nous proposons une adaptation de ces algorithmes *top-k* à une recherche approximative, toujours sur des données certaines.

1.1.5 Fonction de classement

Comme l'indique son nom, la fonction de classement doit permettre d'établir un classement des objets afin de pouvoir retourner les k meilleurs parmi eux. Elle permet d'agréger les scores produits par chaque critère en score global pour chaque objet. La fonction de classement dépend de l'application dans laquelle la recherche est réalisée, la grande majorité des techniques de traitement des requêtes *top-k* considèrent des fonctions de classement *monotones* simples (somme, min, etc). D'autres techniques utilisent des fonctions spécifiques plus complexes, pour répondre à des problèmes plus complexes, d'aide à la décision par exemple. Certaines applications ne nécessitent pas un classement des objets par score de pertinence (nous parlons généralement de critères de recherches conflictuels), et dans ce cas, l'utilisation d'une fonction de classement n'est pas nécessaire. Pour répondre à ce genre de problématique, la technique dite *skyline* [Börzsönyi et al., 2001] remplace les algorithmes *top-k*, et permet le traitement d'une requête multicritères *top-k* sans se baser sur une fonction de classement.

Les techniques de traitement des requêtes de *top-k* multicritères peuvent donc être classées selon la fonction de classement en deux catégories :

- Avec fonction de classement. La grande majorité des techniques de traitement des requêtes *top-k* considèrent une fonction de classement *monotone*. La monotonie de cette fonction permet d'optimiser le coût de la recherche en fournissant des garanties sur l'évolution des scores des candidats. Nous nous plaçons dans ce travail dans le cadre le plus courant de l'utilisation d'une fonction de classement monotone.

- Sans fonction de classement : cette catégorie de techniques est principalement utilisée dans le cadre des requêtes *skyline* [Börzsönyi et al., 2001] [Yuan et al., 2005] dont le résultat est l'ensemble des objets non dominés par d'autres objets selon des critères de recherche conflictuels.

Dans cette partie, nous avons présenté les différents paramètres liés au traitement de requêtes *top-k* multicritères. Ces paramètres permettent de définir le cadre dans lequel s'effectue la recherche. Nous avons commencé par présenté les différents types de requêtes, ensuite nous avons parlé des types d'accès aux scores et du niveau d'exécution de la requête *top-k*. Nous avons après discuter de la certitude de données, avant de finir par un classement des techniques de recherche *top-k* selon la fonction de classement.

Dans cette thèse, nous nous plaçons dans le cadre suivant pour le traitement de requêtes *top-k* multicritères :

- Requêtes *top-k* de sélection, alors des algorithmes comme *J** [Natsev et al., 2001] et *Rank-Join* [Ilyas et al., 2004] permettant de traiter des requêtes de jointure ne sont pas considérés.

- Le coût d'accès aux sources est le plus élevé, alors l'optimisation de la gestion de la liste des candidats proposée par l'algorithme *LARA* [Mamoulis et al., 2007], n'est pas pertinente dans notre contexte.

- Un accès correspond à la consultation d'une seule source, et un accès permet de retourner un seul objet avec son score si l'accès est séquentiel, et uniquement un score si l'accès est direct. Les techniques comme *TPUT* [Cao and Wang, 2004] ou KLEE [Michel et al., 2005], qui utilisent des méthodes permettant respectivement de retourner plusieurs objets dans un seul accès et d'effectuer des accès parallèles aux sources ou encore disposant d'informations sur les scores des objets dans les sources sont aussi exclu de cette étude.

- Les seules informations disponibles dans les sources sont les scores des objets, donc les techniques utilisant des informations supplémentaires, comme le classement (rang) dans *BPA* [Akbarinia et al., 2007] ne sont pas directement comparables aux algorithmes dans notre contexte qui considèrent des informations limitées sur les données.

- Les différents types d'accès aux données présentés dans 1.1.2 sont tous considérés. Même si la grande majorité des algorithmes *top-k* ont été proposés pour répondre à des configurations spécifiques de types d'accès aux données, notre objectif est de proposer une nouvelle stratégie de recherche *top-k* dans un cadre générique où les différents types d'accès sont considérés.

1.2 Les algorithmes *top-k*

Dans cette section, nous étudions les algorithmes proposés pour le traitement des requêtes *top-k* multi-critères dans le contexte décrit ci-dessus. Nous présentons ces algorithmes suivant la classification par type d'accès présentée dans la section 1.1.2. Même si nos travaux se placent dans la dernière catégorie (algorithmes génériques), nous comparons les performances de nos algorithmes à celles des méthodes des autres catégories, afin de montrer que notre approche s'adapte très bien aux configurations particulières de ces algorithmes avec de meilleurs résultats.

1.2.1 Algorithmes *top-k* pour *S-sources*

L'algorithme de référence dans cette catégorie est *No Random Access* (*NRA*) [Fagin et al., 2002], qui considère uniquement des sources triées, donc aucun accès direct n'est possible. *NRA* interroge les sources à tour de rôle, en suivant une simple stratégie *round-robin*. Chaque accès à une source permet de retourner un seul objet qui sera considéré comme un candidat potentiel pour intégrer le *top-k* final. Comme illustré dans l'exemple de l'introduction, le score global d'un candidat est représenté sous forme d'un intervalle de scores indiquant la valeur minimale et la valeur maximale possible. La valeur minimale (maximale) du score est calculée en agrégeant le score local de l'objet avec les valeurs minimales (maximales) possibles dans les autres sources. Les scores globaux des candidats du *top-k* final ne sont pas forcément calculés mais l'intervalle de score peut être suffisant pour garantir l'appartenance d'un candidat au résultat final.

La figure 1.4 présente un exemple de traitement d'une requête *top-k* avec l'algorithme *NRA*. Nous considérons deux sources représentés sous forme de listes d'objets (L_1, L_2) dont chacune présente un classement différent des mêmes objets. Les scores des objets sont dans l'intervalle $[0, 1]$. La fonction

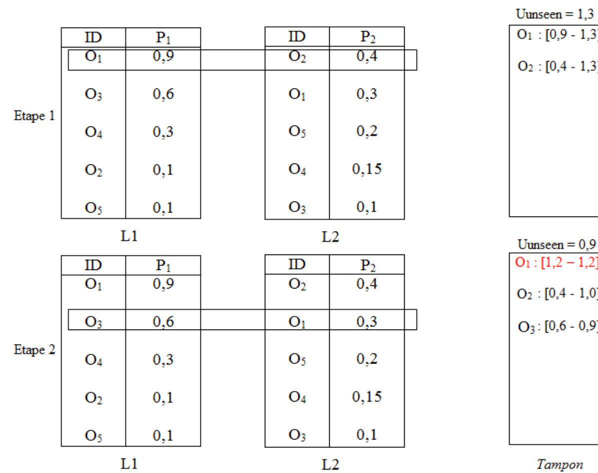


FIGURE 1.4 – Premières étapes de l'algorithme NRA

d'agrégation est une simple somme, donc le score global d'un objet O est la somme de ses scores locaux dans L_1 et L_2. L'exécution de l'algorithme *NRA* se déroule de la manière suivante :

Une première étape, illustrée en haut de la figure, consiste à retourner le premier objet de chaque liste, par exemple le candidat O_1 a un intervalle de score de $[0.9, 1.3]$, puisque son score dans la liste L_1 est de 0.9 et le score qu'il peut avoir dans la liste $L2$ ne dépasse pas 0.4. Le score maximum possible pour le score global de O_1 est $0.9 + 0.4 = 1.3$. La liste des candidats retournés, suite à des accès séquentiels, est maintenue de la même manière que celle présentée dans l'introduction : un intervalle de score pour chaque candidat, et une mise à jour suite à chaque accès dans une liste de scores. Une valeur U_{unseen} est maintenue pour calculer à chaque étape le score maximum possible que peut avoir un objet qui n'a pas été retourné précédemment (c'est la somme des valeurs maximales que peut avoir un objet dans chaque liste). Les objets retournés sont placés dans un tampon dans l'ordre du score minimum, afin de pouvoir comparer R_k, le score minimum du $k^{\text{ème}}$ candidat, d'abord, avec la valeur de U_{unseen} et ensuite, avec les scores maximums des candidats ayant un score minimum inférieur à R_k. L'algorithme s'arrête quand $R_k \geq U_{unseen}$ et R_k dépasse les valeurs maximum des scores des candidats. La deuxième étape dans la figure 1.4 permet de retourner les prochains objets dans les listes, et de mettre à jour les intervalles de scores des candidats. A ce moment, le score de O_1 est plus grand que les valeurs maximales que peuvent avoir les autres candidats, et aucun objet non retourné ne peut avoir un score supérieur à 1.2 puisque la valeur de U_{unseen} est égale à 0.9 (le score maximum possible dans L_1 est 0.6, et dans L_2 0.3). Pour ces raisons, nous pouvons arrêter l'exécution à cet instant si $k = 1$ pour retourner O_1 comme *top-1*, sinon, l'exécution continue de la même manière que dans les deux premières étapes.

La deuxième technique populaire pour ce type d'accès est l'algorithme *Stream-Combine* [Güntzer et al., 2001] qui utilise le même principe que *NRA* mais établit un ordre d'accès aux sources basé sur un bénéfice calculé pour chaque source. Deux objectifs sont derrière le choix de calculer à chaque étape un bénéfice pour chaque source. Le premier est de réduire rapidement la taille des intervalles de scores des candidats, ce qui permettra de s'approcher rapidement du score exact si la baisse enregistrée est conséquente. Le deuxième objectif, est de baisser rapidement la valeur de U_{unseen} afin de s'assurer que tous les objets qui ne sont pas retrouvés dans les sources ne peuvent pas intégrer le *top-k*. Par exemple,

si une source S_j permet d'avoir la plus grande baisse dans les intervalles de scores des candidats et la valeur de U_{unseen}, alors S_j aura le plus grand bénéfice et sera choisie pour le prochain accès. Une méthode simple est proposée pour calculer le bénéfice de chaque source en se basant sur quatre facteurs : *(i)* l'importance de la source dans le score global (par exemple : le coefficient de la source dans une fonction d'agrégation de type somme pondérée), *(ii)* la baisse du score maximum possible dans la source (une grande baisse de score dans les sources favorise un arrêt rapide de l'exécution, puisqu'elle permet de baisser la valeur de U_{unseen}), *(iii)* le nombre de candidats du *top-k* courant qui n'ont pas encore été retrouvés dans la source, l'accès à cette dernière permettra de baisser les valeurs maximales des scores de ces candidats, et *(iv)* le coût de l'accès séquentiel (trié) à la source () le bénéfice est inversement proportionnel au coû d'accè). *Stream-Combine* combine ces critères pour calculer à chaque étape et pour chaque source de données S_j un bénéfice B_j, en utilisant la formule suivante :

$$B_j = \frac{N_j * \frac{\partial F}{\partial S_j} * \delta_j}{C_s(S_j)}. \tag{1.2}$$

Dans cette formule, N_j représente le nombre de candidats qui seront affectés par l'accès, c'est à dire qu'ils ne sont pas encore retrouvés dans la source S_j, et par conséquent un accès à cette source permettra de mettre à jours leurs intervalles de scores. Le terme $\frac{\partial F}{\partial S_j}$, où F est la fonction d'agrégation, permet de mesurer le poids de la source dans le calcul du score global d'un candidat (par exemple le coefficient du critère évalué par la source de donnée dans le cas d'une fonction de score de type somme pondérée). En fin la valeur δ_j est la baisse de score attendue par cet accès. $C_s(S_j)$ est le coût d'un accès séquentiel à S_j.

La méthode proposée par *Stream-Combine* pour mesurer le bénéfice de chaque source triée, afin de pourvoir choisir la source proposant le meilleur bénéfice, considère dans un premier temps, que l'information sur la baisse de score attendue dans chaque source est connue, ce qui n'est pas le cas de l'algorithme *NRA*. Ensuite, elle considère aussi une mesure de l'importance d'une source dans le calcul du score global, le coefficient dans une fonction de type somme pondérée, ce qui n'est pas généralisable pour toute les fonctions d'agrégations monotones, par exemple le minimum ou le maximum.

1.2.2 Algorithmes *top-k* pour *SR-sources*

Les techniques proposées dans cette catégorie, considèrent que toutes les sources peuvent être interrogées avec les deux types d'accès, direct et séquentiel. L'algorithme le plus connu dans cette catégorie est celui proposé par Fagin et al. : *TA* (Threshold Algorithm) [Fagin et al., 2001]. La stratégie adoptée pour interroger les sources est similaire à celle utilisée dans *NRA*, c'est à dire considérer les sources l'une après l'autre dans un ordre prédéfini (*round-robin*).
A chaque étape, *TA* commence par effectuer un accès séquentiel dans la prochaine source pour retourner un seul objet, ensuite, des accès directs sont effectués dans toutes les autres sources pour retourner à chaque fois le score local du candidat retourné suite à l'accès séquentiel. Cela permet de calculer directement le score final du candidat. Si on note m le nombre de sources, pour chaque accès séquentiel, si l'objet trouvé n'est pas déjà candidat, $m - 1$ accès directs sont effectués.
Comme dans *NRA*, l'algorithme *TA* maintient une liste de candidats triée dans l'ordre décroissant des scores. La notion d'intervalle de score n'est pas utile dans *TA*, puisqu'il calcule directement le score final pour chaque nouvel objet découvert. La condition d'arrêt dans *TA* est similaire à celle utilisée dans *NRA*, et se base sur la comparaison du score du $k^{\text{ème}}$ candidat et la valeur de U_{unseen} qui comme dans *NRA*

représente le score maximum que peut avoir un objet qui n'a pas encore été retourné. Si R_k, le score du $k^{\text{ème}}$ candidat est supérieur à U_{unseen}, alors l'algorithme s'arrête pour retourner les k premiers candidats.

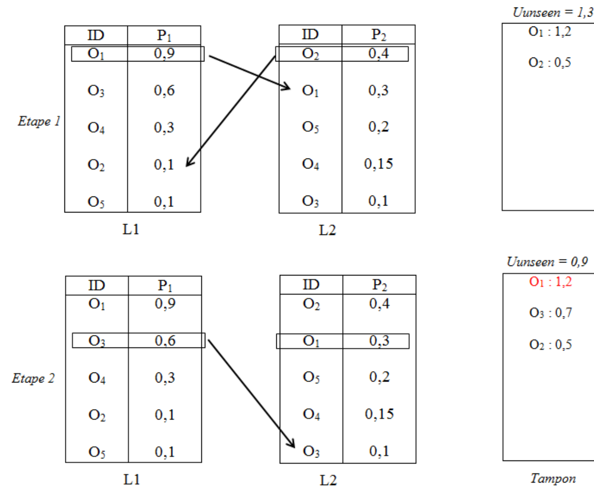


FIGURE 1.5 – Premières étapes de l'algorithme TA

La figure 1.5 présente un exemple de traitement d'une requête top-k avec l'algorithme *TA*. Nous considérons deux *SR-sources* (pour deux critères de recherche) qui présentent deux classements différents des mêmes objets. Comme dans NRA, nous considérons ici une simple somme comme fonction d'agrégation et une stratégie "round-robin" pour interroger les sources. La première étape de TA consiste à retourner le premier objet de chaque source, et pour chaque objet retourné, l'autre liste est accédée pour calculer son score final. Par exemple l'objet O_1 est retourné suite à un accès séquentiel dans L_1 avec un score 0.9, et en interrogeant L_2 via un accès direct, le score final de O_1 est connu et égal à 1.2. A ce moment aucun résultat ne peut être donné tant que tous les candidats ont des scores inférieures au seuil U_{unseen}.L'accès séquentiel permet de baisser à chaque fois la valeur du seuil. En réalisant la même opération dans la deuxième étape, un nouveau candidat est découvert, et surtout le seuil baisse pour atteindre une valeur de 0.9. A cet instant le candidat O_1 qui a un score final de 1.2 est confirmé en tant que top-1 et l'algorithme peut s'arrêter si $k = 1$ ou continuer de la même façon sinon.

Dans [Bruno et al., 2002], les auteurs proposent *TAz*, une extension de *TA* qui considère un ensemble de *R-sources* en plus de l'ensemble des *SR-sources* initial. L'algorithme *TAz* effectue les accès séquentiels de la même manière que *TA*, les sources sont considérées une à une (stratégie round-robin), et chaque accès séquentiel à un source permet de retourner un seul objet. Pour les accès directs, *TAz* inclut l'ensemble des *R-sources* afin de calculer le score final de chaque candidat découvert suite à un accès séquentiel.

Un deuxième algorithme appelé *Quick-Combine* (*QC*) [Güntzer et al., 2000] est proposé dans ce contexte et peut être considéré comme une variante de *TA* qui utilise la même stratégie que *Stream-Combine* pour sélectionner à chaque étape la source à interroger. Le bénéfice calculé dans *QC* considère seulement les deux premiers facteurs utilisés dans *Stream-Combine* pour choisir la meilleure source à interroger :

1. L'importance de la source dans le score global.

2. La baisse du score maximum possible dans la source.

Dans [Fagin et al., 2001] les auteurs proposent une autre variante de *TA* appelée *Combined-Algorithm* (*CA*) et qui considère les accès directs h fois plus couteux que les accès séquentiels. *CA* combine les algorithmes *NRA* et *TA* dans le but de réduire le nombre d'accès directs. Il commence de la même manière que *NRA* et effectue h accès séquentiels dans chaque source. Les candidats sont alors classés dans l'ordre décroissant du score maximum, ensuite le score final du meilleur candidat est calculé en effectuant des accès directs dans les sources où le candidat en question n'est pas encore découvert. L'algorithme s'arrête quand tous les candidats du top-k courant sont assurés d'appartenir au résultat final. La condition d'arrêt est la même que pour *NRA*. Nous remarquons que le nombre d'accès séquentiels dans chaque cycle est $h * m$ pour au plus $m - 1$ accès directs, donc un rapport minimum de $\frac{h*m}{(m-1)} > h$. Les accès séquentiels sont donc favorisés, avec au moins h accès séquentiels pour un accès direct.

Plusieurs algorithmes top-k ont considéré une différence de coût entre l'accès séquentiel et l'accès direct, et se sont basés sur ce principe pour proposer une stratégie spécifique à ce modèle de coût (voir la sous section suivante). Dans nos travaux nous avons également pris en considération cet aspect et nous avons proposé une solution inspirée de celle proposée dans *CA*.

1.2.3 Algorithmes *top-k* avec des accès directs contrôlés

Dans cette catégorie les algorithmes top-k proposés ont généralement considéré une seule S-source permettant de découvrir les objets, et plusieurs R-sources. A la différence des algorithmes déjà présentés, ici les accès directs aux R-sources n'ont pas toujours le même coût, et un ordre est généralement établi pour commencer à interroger les sources les moins coûteuses. Plusieurs algorithmes ont été proposés dans cette catégorie, nous présentons ici les algorithmes *Upper* [Bruno et al., 2002] [Marian et al., 2004] et *MPro* [Chang and Hwang, 2002].

L'algorithme $Upper$ maintient les candidats selon une stratégie différente de celle des autres algorithmes, dans l'ordre de la valeur du *score attendu* ou estimé. Pour chaque candidat, un score attendu est calculé en remplaçant chaque score local inconnu par la moyenne des scores basée sur une éventuelle connaissance de la distribution des scores dans la source (moyenne simple des scores minimum et maximum sinon).
A chaque étape, *Upper* choisit le type du prochain accès à effectuer : séquentiel ou direct. Pour cela, un candidat o est sélectionné ayant le score maximum U le plus élevé parmi tous les candidats. Si U_{unseen} est la valeur maximale que peut avoir le score d'un objet inconnu, et $U < U_{unseen}$ alors un accès séquentiel est décidé afin de baisser la valeur de U_{unseen}. L'intuition est de trouver le candidat avec le meilleur score maximum, mais quand $U < U_{unseen}$, alors un objet non découvert peut avoir un score maximum plus grand que U. Si $U \geq U_{unseen}$, un accès direct pour affiner le score de o sera effectué. Dans ce cas, un bénéfice est calculé pour chaque R-source et la source permettant le meilleur bénéfice est sélectionnée. Deux cas sont possibles pour calculer le bénéfice. *(i)* Si le candidat choisi o appartient au top-k courant (par rapport au score attendu), alors le bénéfice de chaque source S_i est le rapport entre la baisse attendue δ de U et le coût d'accès à la source : $\frac{\delta}{C_r(S_i)}$. *(ii)* Dans le cas contraire, le candidat o a plus de chances d'être à l'extérieur du *top-k*, alors la baisse nécessaire Δ pour prouver que o est en dehors du *top-k* est calculée. Le bénéfice de chaque R-source dans ce cas est le rapport du minimum entre δ et Δ, et le coût d'accès à la source : $\frac{\min(\delta,\Delta)}{C_r(S_i)}$.

Dans [Marian et al., 2004], les auteurs proposent une nouvelle variante de $Upper$ qui s'adapte à une

configuration de sources composée de plusieurs SR-sources et plusieurs R-sources. Ici les accès séquentiels aux *SR-sources* sont effectués en appliquant la stratégie utilisée par *NRA* (*round robin*). Pour les accès directs, les SR-sources où les scores locaux du candidat choisi ne sont pas connus, sont considérées comme des *R-sources*, et un bénéfice est calculé pour chacune des sources pour définir la meilleure source à interroger.

De la même manière que *Upper*, l'algorithme *MPro* sélectionne le meilleur candidat par rapport au score maximum U, et effectue un accès trié si le $U_{unseen} > U$, sinon un accès direct est décidé. Mais, contrairement à $Upper$, l'algorithme $MPro$ préétablit l'ordre des R-sources à interroger, cet ordre sera respecté pour n'importe quel candidat. Le meilleur ordre à suivre pour interroger les sources et qui permet de réduire le nombre d'accès, peut être déterminé par différentes méthodes. Les auteurs proposent une méthode d'optimisation par échantillonnage basée sur la simulation de l'exécution sur un échantillon de la base.

1.2.4 Le cas générique

A notre connaissance, un seul travail aborde la généricité (n'importe quel type de source). L'algorithme *Necessary Choices* (*NC*) [Hwang and Chang, 2007] est le résultat de ce travail. *NC* considère n'importe quelle configuration de types et de coûts d'accès aux sources et définit ce que les auteurs ont appelé les *choix nécessaires* d'accès pour obtenir un *top-k* final avec des scores exacts. *NC* maintient les candidats dans l'ordre des scores maximum, et se base sur ce classement pour identifier les accès nécessaires à effectuer. Pour limiter le nombre d'accès aux sources, les algorithmes exprimés dans le cadre *NC* sélectionnent à chaque étape un candidat avec un score incomplet (intervalle de score), pour effectuer un accès permettant de raffiner son intervalle de score. L'algorithme s'arrête si tous les candidats du *top-k* ont des scores complets.

Un accès considéré nécessaire dans l'algorithme *NC* est tout accès séquentiel sa_j ou direct $ra_j(o_i)$ qui n'a pas été encore réalisé dans toute source S_j pour un objet o_i du *top-k* courant avec un score incomplet. *NC* considère un cadre dans lequel les objets du *top-k* final doivent avoir des scores exacts (complets). Intuitivement si un candidat appartient au *top-k* courant, il faut soit calculer son score exact s'il est dans le *top-k* final, soit faire baisser son score pour le faire sortir du *top-k* dans le cas contraire. Dans les deux cas, les accès permettant de raffiner le score de ces objets du *top-k* courant sont nécessaires.

L'algorithme 1 présente le cadre général *NC*, qui prend en entrée la requête q et l'ensemble de sources S. La fonction d'agrégation F et la valeur de k sont définies dans la requête elle-même. La première étape consiste à effectuer des accès séquentiels jusqu'à avoir au moins k candidats dans la liste des candidats. La liste de candidats n'est pas représentée explicitement, mais seulement $\mathcal{K}_p$, la liste des k meilleurs candidats selon le score maximum. $\mathcal{P}$ garde tous les accès effectués. Ensuite, à chaque pas nous récupérons dans O les candidats de $\mathcal{K}_p$ ayant des scores incomplets. Si O est vide (tous les candidats ont des scores complets), la recherche s'arrête pour retourner l'ensemble $\mathcal{K}_p$. Si l'ensemble O n'est pas vide, un candidat de O est sélectionné. Pour raffiner l'intervalle de score du candidat choisi, l'ensemble N_j des choix d'accès nécessaires est identifié, ensuite, un accès est choisi. L'intervalle de score du candidat choisi est maintenant raffiné suite au dernier accès, et les ensembles $\mathcal{K}_p$ et $\mathcal{P}$ sont mis à jour. L'algorithme s'arrête quand les k candidats de $\mathcal{K}_p$ ont des scores complets.

Toujours dans le cadre générique *NC*, les auteurs proposent une stratégie *SR* (Sorted then Random)

Algorithm 1 Le cadre générique *NC*

Require: q and $\mathcal{S}$
  //Initialisations
  $\mathcal{P} \leftarrow \emptyset$ *//les accès effectués*
  $\mathcal{K}_p \leftarrow \{o_1, ..., o_k\}$*//top-k courant trié par score maximum*
  while $O \neq \{\}$ **do**
    $O \leftarrow \{o_1, ..., o_j\}$, $o \in \mathcal{K}_p$ *//Tous les candidats avec des scores incomplets*
    $o_i \leftarrow$ *OneCandidate*(O) *//Choix d'un candidat avec score incomplet*
    $N_j \leftarrow \{sa_j, ra_j(o_i) |\ p_j[o_i]$*, le score de* o_i *pour la source* j *n'est pas connu par* $\mathcal{P}\}$
    $access \leftarrow chooseaccess(N_j)$; *// Choisir un accès*
    $perform(access)$; *// réaliser l'accès*
    Update$(\mathcal{K}_p)$;
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{access\}$;
  end while
  return $\mathcal{K}_p$

qui favorise, pour tous les candidats, les accès séquentiels aux accès directs. Un algorithme *SR/G* qui implémente cette stratégie est proposé. Dans tous les algorithmes proposés dans le framework *NC*, le choix du candidat (*OneCandidate*) se porte toujours sur celui ayant le score maximum le plus élevé. *SR/G* est guidé principalement par deux paramètres :

- Le premier paramètre concerne la profondeur des accès séquentiels dans chaque *S-source* ou *SR-source* : $D = \{d_1, ..., d_m\}$, D indique l'ensemble des seuils fixés dans toutes les sources triées. Ces seuils indiquent une profondeur pour chaque (S/SR)-source, et tant que ces profondeurs ne sont pas atteintes, la priorité est donnée aux accès séquentiels. Plus précisément, pour chaque source triée S_j, si $crtmax_j$ est le dernier score retourné, tant que $crtmax_j \geq d_j$, les accès séquentiels sont préférés aux accès directs. En d'autres termes, les accès directs ne sont possibles pour un candidat c que lorsque pour toute source triée S_j dont le score local de c n'est pas connu, nous avons $crtmax_j < d_j$.

- Le deuxième paramètre concerne l'ordre des accès directs. H indique un ordre préfixé des accès directs dans les cas des *R-sources* et *SR-sources*. Pour tous les candidats, un même ordre est établi entre les accès directs, de la même manière que dans l'algorithme *MPro* [Chang and Hwang, 2002].

Parmi toutes les possibilités des valeurs (D,H), l'algorithme *SR/G* sélectionne le couple optimal en se basant sur une méthode d'échantillonnage. Plus précisément, l'exécution de *SR/G* est simulée sur un échantillon de données en variant les valeurs des paramètres afin de trouver la meilleure configuration avec le coût d'exécution le plus bas. Le processus d'optimisation est itératif, d'abord, à partir d'un ordre initial H, l'ensemble D est déterminé. Ensuite, pour l'ensemble D choisi, un ordre H meilleur peut être généré, et ainsi de suite jusqu'à réussir à fixer un couple (D,H).

Malgré sa généricité, la stratégie de *NC* est difficilement comparable avec des algorithmes qui considèrent un contexte où les scores des objets du *top-k* ne sont pas nécessairement complets, contexte considéré par plusieurs techniques : *NRA*, *RankSQL*, *Stream-Combine*, etc. Dans ce cas plus général, la notion d'accès nécessaire n'est plus valable.

Nous proposons dans cette thèse un cadre générique *GF* plus général que *NC*, pour n'importe quel algorithme *top-k* permettant le traitement de requêtes de sélection. *GF* est adapté à n'importe quelle configuration des types de sources et coûts d'accès. Nous proposons aussi un algorithme générique *BR* dans ce contexte, et nous le comparons avec les autres algorithmes *top-k* de l'état de l'art, génériques ou spécifiques à une configuration particulière. Pour rendre possible la comparaison de *BR* avec *NC*, nous proposons une adaptation de *NC* à notre contexte, tel que défini précédemment.

1.2.5 Optimalité

Dans [Fagin et al., 2002], il est montré que les algorithmes *NRA* et *TA* sont optimaux pour n'importe quelle instance de base de données. Pour cela, l'article définit une notion d'optimalité dite *instance-optimalité*, basée sur le coût total d'exécution de la manière suivante : pour une classe d'algorithmes A et une classe de bases de données D, un algorithme $\mathcal{B} \in A$ est dit *instance-optimal* si pour tout algorithme $\mathcal{A} \in A$ et toute base de données $\mathcal{D} \in D$, nous avons : $cost(\mathcal{B}, \mathcal{D}) = O(cost(\mathcal{A}, \mathcal{D}))$, donc il existe deux constantes c et $c' > 0$, tel que $cost(\mathcal{B}, \mathcal{D}) \leq c * cost(\mathcal{A}, \mathcal{D}) + c'$ pour tout $\mathcal{A} \in A$ et tout $\mathcal{D} \in D$. La valeur de c est appelée ratio d'optimalité.

Les auteurs montrent aussi que les algorithmes qui se basent sur des choix dynamiques concernant la prochaine source à interroger ne sont pas instance-optimaux, même s'ils peuvent avoir de meilleurs coûts d'exécution en pratique. Intuitivement, le choix dynamique de la prochaine source de données risque de mener à un choix déséquilibré des candidats, par exemple des candidats à partir d'une seule source, ce qui peut "casser" la garantie d'optimalité. Les auteurs montrent que si nous garantissons que chaque source est interrogée au moins tous les p pas, l'instance-optimalité est restaurée.
Nos travaux se placent dans une approche de choix de source, donc sans garantie d'optimalité, mais la même technique permettant d'interroger périodiquement chaque source peut être incluse ou rajoutée à nos algorithmes.

Dans cette section, nous avons passé en revue les algorithmes proposés pour le traitement de requêtes *top-k* multicritères. Nous avons présenté aussi les différents types d'algorithmes *top-k* de l'état de l'art qui correspondent à notre contexte. Notre objectif est de proposer une approche générique (pour n'importe quelle configuration de types et de coûts d'accès aux sources), mais aussi une comparaison expérimentale avec tout ces algorithmes dans leur cadre spécifique ou adapté à notre contexte.

1.3 La recherche *top-k* approximative

Dans le souci de réduire le coût d'exécution des requêtes *top-k*, généralement très élevé pour des grands volumes de données, plusieurs travaux ont été menés pour proposer des solutions approximatives. Dans le même contexte considéré dans TA [Fagin et al., 2001], les auteurs ont proposé un nouvel algorithme appelé TA_θ, considéré comme une variante de *TA* permettant de réduire le temps d'exécution du premier algorithme en retournant un résultat approximatif. TA_θ définit un paramètre d'approximation $\theta > 1$ qui permettra de mesurer le degré d'approximation en retournant une θ-approximation du résultat exact retourné par la variante originale TA. Le résultat approximatif de TA_θ (θ-approximation) est un ensemble

K composé de k objets, tel que :

$$\forall x \in K, \forall y \notin K, \theta score(x) \geq score(y) \quad (1.3)$$

Cela signifie que la qualité de la solution approximative est contrôlée par la valeur de θ. Quel que soit l'objet y "raté" par TA_θ dans le *top-k* (faux négatif), le score d'un éventuel mauvais candidat (faux positif) respecte la condition $score(x) \geq score(y)/\theta$. Il est simple d'adapter l'algorithme original *TA* afin d'obtenir une θ-approximation du résultat final. Les auteurs montrent que pour obtenir une θ-approximation, il suffit de modifier la condition d'arrêt. La nouvelle condition d'arrêt considérée par l'algorithme TA_θ est la suivante :

$$\forall x \in K, score(x) \geq \frac{U_{unseen}}{\theta} \quad (1.4)$$

U_{unseen} est la valeur maximale possible pour les scores des objets pas encore découverts dans les sources. Si $\theta = 1$, la nouvelle condition d'arrêt correspond exactement à celle utilisée dans *TA* et par conséquent le résultat obtenu est un *top-k* exact. En relaxant la condition d'arrêt pour obtenir un résultat approximatif, TA_θ est équivalent à un arrêt précoce de l'algorithme *TA*, puisque la condition 1.4 est satisfaite avant la condition d'arrêt originale. L'inconvénient de la θ-approximation est la difficulté de trouver la "bonne" valeur de θ en fonction de l'application, et l'absence d'un modèle générique pour identifier la valeur de θ dans différents contextes.

L'approximation par arrêt prématuré est aussi utilisé dans le cadre du traitement des requêtes *top-k* de jointure avec l'algorithme *J** [Natsev et al., 2001], qui applique aussi la θ-approximation.

D'autres algorithmes approximatifs pour le traitement des requêtes *top-k* sélection ont été proposés dans [Theobald et al., 2004], le cadre considéré ici est le même que dans *NRA* : toutes les sources sont triées par ordre décroissant des scores, uniquement des accès séquentiels sont utilisés pour calculer les scores des objets. Ici, les auteurs proposent une famille d'algorithmes approximatifs appelée *Prob-sorted*, dans un contexte où des informations sur les distributions des scores sont disponibles et exploitables. Cette famille d'algorithmes se base sur deux points essentiels : *(i)* une prédiction probabiliste des scores, permettant une réduction drastique du nombre d'accès séquentiels avec une qualité de résultat légèrement moins bonne, et *(ii)* une gestion intelligente des priorités d'interrogation des listes d'objets.

Dans le même contexte, plusieurs techniques utilisent des modèles probabilistes pour calculer les scores des objets et limiter le nombre d'accès aux sources de données [Amato et al., 2003b] [Beyer et al., 1999] [Ciaccia and Patella, 2000] [Donjerkovic and Ramakrishnan, 1999]. Dans [Theobald et al., 2004], Theobald et al. proposent une variante de l'algorithme *TA* retournant un *top-k* approximatif avec un modèle probabiliste offrant des garanties sur le résultat retourné. Dans cette variante, pour chaque nouvel objet c découvert suite à un accès séquentiel, un test probabiliste permettant de mesurer la probabilité pour que c soit dans le *top-k* final ou pas est effectué. Pour intégrer le *top-k* final, la probabilité suivante doit être estimée :

$$Pr(\sum_{i \in E(c)} p_i(c) + \sum_{j \notin E(c)} pe_j(c) > L_k) \quad (1.5)$$

$E(c)$ représente l'ensemble des sources où le score de c est déjà connu, $p_i(c)$ est le score de c dans la source i, $pe_j(c)$ est l'estimation du score de c dans la source j où il n'est pas encore découvert, et L_k est le $k^{\text{ème}}$ score minimum dans le *top-k* courant. Une valeur ϵ est utilisée comme seuil permettant d'identifier

les bons candidats (qui ont une grande probabilité de pouvoir intégrer le *top-k* final). Si la probabilité est inférieure à la valeur de ϵ, le candidat c est considéré comme non pertinent et sera supprimé de la liste des candidats. Il faut savoir que : $\epsilon = 0$, correspond à la version originale de l'algorithme *TA*, permettant de retourner un résultat exact.

Dans le contexte que nous considérons, plusieurs sources de grande taille avec des accès très coûteux, la recherche approximative est une solution prometteuse pour réduire le coût total d'exécution et le temps de réponse. Nous utilisons la même technique que dans TA_θ et *J**, et nous proposons une adaptation des algorithmes *top-k*, en relaxant les conditions d'arrêts afin d'obtenir un résultat approximatif. Nous proposons dans cette thèse une première étude expérimentale permettant une comparaison des comportements des algorithmes *top-k* dans le cadre d'une recherche approximative basée sur la technique d'arrêt précoce.

1.4 Recherche des k *p*lus *p*roches *v*oisins dans les données multimédia

La recherche des k plus proches voisins dans les bases de données multimédia est un problème central dans la recherche par le contenu. Sans perte de généralité, nous nous intéressons ici à la recherche par le contenu dans les bases d'images (Content-Based Image Retrieval). Les autres objets multimédia utilisent des techniques similaires. Dans ce contexte, la première étape consiste à définir une description de toutes les images de la base, en commençant par l'extraction de primitives visuelles, qui peuvent être définies par des régions, des points d'intérêts, des formes, etc. Ce travail permet de d'avoir une description du contenu des images. Plusieurs caractéristiques visuelles sont utilisées dans la littérature. Dans [Smeulders et al., 2000], les auteurs proposent un état de l'art des principaux descripteurs considérés dans le cadre de la recherche par similarité du contenu (CBIR). Ces descripteurs peuvent être de couleur [Carrilero, 1999] [Ma and Manjunath, 1997] [Burghouts and Geusebroek, 2009] [Oliva and Torralba, 2006], de texture [Heinrichs et al., 1959] [Fournier et al., 2001] [de Sande et al., 2010], de points d'intérêt [Lowe, 2004] [Sivic and Zisserman, 2003] [de Sande et al., 2008], etc.

1.4.1 Définition et méthodes

La recherche par le contenu des images les plus similaires à une image requête se réduit donc à une recherche dans l'espace des signatures, vues comme des points dans un espace multidimensionnel. Plus précisément, la recherche des k-ppv signatures s'apparente à la recherche des k points les plus proches du point requête. Nous considérons ici une recherche textitk-ppv dans un espace vectoriel.

***Définition* 1.** *(Recherche des $k - ppv$) l'espace normé l_p^m représentant l'espace Euclidien de m dimensions R^m avec la norme l_p et la distance $d(.,.)$ induite par cette norme, P étant l'ensemble de points dans cet espace, P étant l'ensemble de points de la base de données dans cet espace. Pour une requête q, tel que $q \in R^m$, la recherche des k-plus proches voisins consiste à traiter l'ensemble P pour trouver l'ensemble $K = \{p_1, ...p_k\}$, $K \subseteq P$, tel que :*

$$\forall p_i \in K, \forall p_j \in P - K, d(q, p_i) \leq d(q, p_j) \tag{1.6}$$

La solution de base pour retourner les k-ppv est une recherche séquentielle dans laquelle la distance de chaque point dans l'ensemble P au point requête est calculée, et les k plus proches voisins sont retournés dans le résultat. Cette stratégie appelée *méthode brute* (*Brute-force*) peut être utilisée dans de petites bases de données. Malheureusement, le temps de recherche évolue linéairement avec la taille de la base et le nombre de dimensions. Par conséquent, le coût de la recherche devient prohibitif pour des grandes bases de données, dans le contexte que nous considérons.
Pour remédier à ce problème, plusieurs structures d'index ont été proposées pour accélérer la recherche des plus proches voisins dans un espace multidimensionnel, en commençant par des index basées sur des structures d'arbres tel que le *R-tree* [Guttman, 1984], *KD-tree* [Bentley, 1975], *SR-tree* [Katayama and Satoh, 1997], *X-tree* [Berchtold et al., 1996] et *M-tree* [Ciaccia et al., 1997], et menant à des résultats exacts. D'autres types de méthodes ont été proposés par la suite. Dans [Valle, 2008], l'auteur propose une classification des méthodes d'indexation de la manière suivante :

- Division spatiale : cette méthode divise l'espace en plusieurs cellules, en prenant en considération la distribution des données, ce qui permet de diviser plus finement les régions très peuplées. L'objectif de cette approche est de limiter l'exploration de l'espace aux cellules proches de la requête. La méthode de référence dans cette approche est *Best-Bin-First* [Beis and Lowe, 1997].

- Division des données : dans cette approche, les données sont divisées en cellules, et l'objectif est aussi de limiter la recherche aux cellules les plus probables pour contenir les objets les plus proches. Une des méthodes les plus connues dans cette approche, est *R-tree* [Guttman, 1984]. La difficulté de ce genre de méthode est de choisir la meilleure heuristique pour la division des données. En effet, si nous essayons d'optimiser au maximum la division, le coût de construction de l'index risque de devenir prohibitif.

- Clustering : l'objectif de cette méthode est de regrouper les objets similaires dans les mêmes cellules. Une des méthodes les plus connues dans ce sens est *Clindex* [C et al., 2002].

- Métrique : dans cette approche, les méthodes proposées utilisent seulement les distances sans se limiter aux espaces vectoriels. Elles séparent les données dans des groupes de points proches et utilisent l'inégalité triangulaire pour ne pas considérer les groupes de points non pertinents pour une requête donnée. Une des méthode populaires dans cette approche est *M-tree* [Ciaccia et al., 1997].

- Approximation des données : cette approche quantifie les données pour considérer uniquement quelques bits par dimension. Cette approche est utilisée dans la méthode VA-file [Weber et al., 1998].

- Projection et hachage : cette approche consiste à projeter les données dans des espaces de plus petites dimensions. Les méthodes utilisant cette approche se basent sur les espaces projetés, et les résultats partiels sont agrégés pour calculer le résultat final. Plusieurs façons sont possibles pour projeter les données : sur une droite [Shustek et al., 2006] [Fagin et al., 2003], en utilisant des fonctions de hachage [Gionis et al., 1999] [Lv et al., 2007], ou en utilisant les *Spaces-Filling Curves* [Griebel and Zumbusch, 2002].

En raison du phénomène de la *malédiction de la dimension* [Bellman and Kalaba, 1959] [Böhm et al., 2001], lorsque les images sont décrites par des vecteurs denses (contenant peu de valeurs nulles), le temps

de recherche dans les structures d'index grandit exponentiellement avec la dimension du descripteur. Les performances de ces structures d'index (permettant une recherche exacte) deviennent moins bonnes que celles de la méthode brute (recherche exhaustive) dès lors que l'espace de représentation dépasse 10 dimensions [Böhm et al., 2001]. Nous signalons aussi que la dégradation des performances de ces méthodes, dépend aussi de la distribution des données. Pour surmonter cette difficulté, des méthodes permettant une recherche approximative sont proposées, visant à trouver le meilleur compromis entre la diminution de la complexité de la recherche et la dégradation de la qualité des résultats.

1.4.2 Recherche approximative des k-plus proches voisins

La recherche approximative dans les données multimédia est justifiée principalement par deux phénomènes, le premier est celui de la *malédiction de la dimension*, et le deuxième est le *fossé sémantique/numérique* (semantic gap) [Smeulders et al., 2000]. En effet, les descripteurs des images sont des vecteurs exprimant les propriétés bas-niveau utilisés souvent pour répondre à des requêtes d'utilisateur de haut-niveau. Le deuxième phénomène (fossé sémantique) correspond à l'inadéquation entre la similarité calculées numériquement et la similarité entre les images attendues par l'utilisateur. Le premier type de similarité tente de modéliser des caractéristiques perceptuelles de l'image, alors que le deuxième est principalement conceptuel. Plus précisément, deux niveaux d'analyse sont distingués :

- Le bas-niveau (numérique) : fait référence au contenu signal de l'image.
- Le haut-niveau (sémantique) : fait référence au contenu interprétable de l'image, comme l'ensemble des objets et leurs significations.

Il existe souvent une grande différence entre la description bas-niveau et la description haut-niveau de l'image. Par exemple pour identifier le type sémantique recherché, il faut au moins conntaitre le contexte. Cela rend une recherche approximative des *k-ppv* plus adapté à ce contexte qu'une recherche exacte.

Plusieurs techniques ont été conçues pour une recherche approximative [Shustek et al., 2006] [Fagin et al., 2003], etc. Parmi les méthodes proposées, nous nous intéressons particulièrement à la recherche approximative ϵ-approximation. Plusieurs méthodes ont été proposées dans ce contexte, les plus connues sont les méthodes *LSH* (Locality Sensitive Hashing) [Gionis et al., 1999] [Indyk and Motwani, 1998] et ses variantes [Bawa et al., 2005] [Lv et al., 2007].

L'objectif de la recherche approximative du plus proche voisin avec ϵ-approximation est de trouver les points voisins qui ont une distance inférieure ou égale à $\epsilon * R$ par rapport au point requête, où R est la distance entre la requête et son plus proche voisin.

Définition 2. *(ϵ-approximation) Nous considérons l'espace normé l_p^m représentant l'espace Euclidien de m dimensions R^m avec la norme l_p et la distance $d(.,.)$ induite par cette norme, P étant l'ensemble de points de la base de données dans cet espace. Pour une requête q, la recherche du plus proche voisin avec ϵ-approximation consiste à retourner un point $p \in P$ tel que $d(q,p) \leq (1+\epsilon)d(q,p)$, où $d(q,p)$ est la distance entre le point requête q et son point le plus proche dans P.*

Cette définition est généralisable pour la recherche des k plus proches voisins avec $k > 1$, où la recherche consiste à trouver k points $\{p_1, ...p_k\}$ tel que la distance entre la requête q et tout point p_i est au

plus $(1+\epsilon)$ fois la distance entre q et son $k^{\text{ème}}$ point le plus proche.

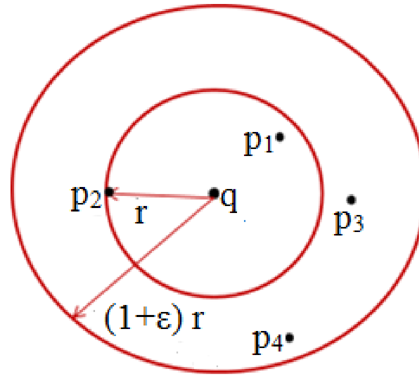


FIGURE 1.6 – Recherche des k plus proches voisins avec ϵ-approximation, $k = 2$

L'exemple de la figure 1.6 illustre le problème de la recherche des k plus proches voisins avec ϵ-approximation. Ici nous considérons $k = 2$. Une solution possible de la recherche des 2 plus proches voisins du point requête q est l'ensemble des points $\{p_3, p_4\}$ puisque nous avons : $d(q, p_3) < (1 + \epsilon)d(p, p_1)$ et $d(q, p_4) < (1 + \epsilon)d(q, p_2)$, avec p_1 et p_2 les vrais deux plus proches voisins de la requête q.

Dans la suite, nous présenterons le principe de la recherche des k-ppv avec ϵ-approximation, et nous expliquerons les stratégies *LSH* et ses variantes en particulier *MultiProbe LSH* [Lv et al., 2007].

1.4.3 Méthodes de hachage pour la recherche des k-ppv

Locality Sensitive Hashing (LSH)

L'idée générale de *LSH* [Indyk and Motwani, 1998] est d'utiliser un ensemble de fonctions de hachage qui permettent de diviser l'espace en de petites cellules, et de placer les objets similaires dans la même cellule avec une forte probabilité. Si l'on considère, pour simplifier, une seule table de hachage, pour une requête q, la recherche des plus proches voisins dans un index *LSH* est divisée en deux étapes. *(i)* Sélectionner les candidats susceptibles d'êtres similaires à la requête en utilisant la fonction de hachage pour calculer le code ou l'empreinte de la requête. Ce code identifie une cellule de l'index, ensuite les points contenus dans la cellule (*bucket*) sélectionnée composent l'ensemble des points candidats. *(ii)* Ensuite, les candidats sélectionnés seront triés par rapport à leur distance de la requête q.

Une fonction de hachage pour *LSH* doit hacher les points proches avec le même code et les points éloignés avec des codes différents. *Indyk* et *Motwani* ont défini les propriétés que doivent respecter les fonctions de hachage pour être considérées sensibles à la localité ou *Locality-Sensitive*.
Une fonction est dite sensible à la localité ou *Locality-Sensitive* si deux points proches obtiennent le même code avec une forte probabilité et deux points éloignés obtiennent le même code avec une faible probabilité :

Définition 3. *(Fonctions de hachage sensibles à la localité) Une famille de fonctions* $H = \{h : S \rightarrow U\}$ *est dit sensible à* (r, ϵ, p_1, p_2)*, avec* $p_1 > p_2 > 0$ *et* $\epsilon > 0$ *si pour tout* $q, p \in S$ *les propriétés suivantes sont vraies :*

- *Si* $d(p, q) \leq r$ *alors* $Pr_{h \in H}[h(p) = h(q)] \geq p_1$
- *Si* $d(p, q) > r$ *alors* $Pr_{h \in H}[h(p) = h(q)] \leq p_2$

Nous considérons ici, S un ensemble de points de R^m et $d(.,.)$ est la distance entre deux éléments de S. Les familles de fonctions *LSH* peuvent êtres utilisées pour différentes distances, tel que la distance de *Jaccard*, la distance de *Hamming*, la distance l_1 et l_2. La famille de fonctions la plus utilisée est celle proposée par Datar et al. [Datar et al., 2004] adaptée aux distances l_1 et l_2, et où chaque fonction est définie sur un espace Euclidien R^m comme suit :

$$h_{a,b}(p) = \left\lfloor \frac{(\bar{a}.\bar{p}) + b}{W} \right\rfloor \tag{1.7}$$

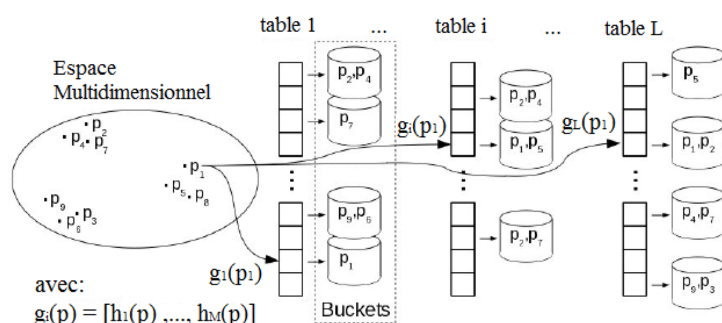


FIGURE 1.7 – Construction de l'index LSH : initialisation des cellules (buckets) des L tables de hachage [Gorisse, 2010].

$\bar{a}$ est un vecteur de m dimensions choisi aléatoirement, b est un nombre réel choisi aléatoirement dans l'intervalle $[0, W]$. Chaque fonction de hachage permet de réduire un vecteur de m dimension à une valeur entière. Comme la figure 1.7 le montre, la construction d'un index *LSH* est réalisée de la manière suivante :

1. Définir L et M respectivement le nombre de tables de hachage et le nombre de fonctions de hachage par table, $M > 0$.
2. Définir un ensemble de fonctions de hachage sensibles à la localité (*Locality-Sensitive*) : $H = \{h : S \rightarrow U\}$.
3. Définir une famille de fonctions $G = \{g : S \rightarrow U^M\}$, où pour chaque $g_i \in G$ et pour un point p dans S : $g_i(p) = (h_1(p), ..., h_M(p))$, avec $h_j \in H$ et $1 \leq j \leq M$. Chaque fonction g_i $(1 \leq i \leq L)$, est utilisée pour construire une table de hachage.
4. Insérer chaque vecteur p dans la cellule correspondante de chaque table de hachage en se basant sur les valeurs calculées par les fonctions $g_i(p)$ pour $i \in \{1, ..., L\}$

Étape de recherche : pour une requête q de recherche des k-plus proches voisins, le traitement s'effectue sur deux étapes :

1. Calculer la valeur de hachage $g_i(q)$ et récupérer tous les candidats dans la cellule correspondante. Répéter cette étape pour chaque $g_i(q)$, pour $i = \{1, ..., L\}$.
2. Établir un classement des candidats trouvés par ordre croissant de leurs distances par rapport à la requête q, et retourner, ensuite, les k meilleurs candidats.

La combinaison de plusieurs fonctions de hachage permet de réduire la probabilité que deux vecteurs distants se trouvent dans la même cellule, mais augmente le risque que deux vecteurs proches soient séparés dans différentes cellules. Fusionner des candidats de différentes tables de hachage, réduit le risque de rater des objets similaires (proches). Le principal inconvénient de la structure d'index proposée dans *LSH*, est la nécessité d'utiliser plusieurs tables de hachage pour couvrir et trouver les plus proches voisins. Par exemple, pas moins de 100 tables de hachage sont nécessaires pour parvenir à une 1-approximation dans [Gionis et al., 1999], et 583 tables de hachage sont utilisées dans [Buhler, 2001]. Ainsi, les données et toute la structure d'index prennent trop d'espace. Afin de réduire cet espace, et dans l'objectif de réduire le nombre de tables de hachage tout en gardant une bonne approximation du résultat final, la variante *MultiProbe LSH* est proposée [Lv et al., 2007].

MultiProbe LSH

MultiProbe LSH (*MLSH*) [Lv et al., 2007] a pour objectif de réduire le nombre de tables de hachage et par conséquent réduire la forte consommation de mémoire de *LSH* due au grand nombre de tables nécessaires pour une bonne approximation du résultat. L'idée principale de *MPLSH* est d'utiliser très peu de tables de hachage (au plus une dizaine), et d'effectuer intelligemment plusieurs accès dans chaque table afin de récupérer les objets, d'abord dans la cellule qui correspond à la clé de la requête, ensuite, dans les cellules voisines. Pour cela, *MPLSH* utilise une séquence d'accès permettant d'effectuer les accès dans les meilleurs cellules susceptibles de contenir les plus proches voisins de la requête. Et puisque chaque table de hachage va permettre de retrouver beaucoup plus de candidats que dans *LSH*, le nombre de tables nécessaire est beaucoup moins important que pour la méthode *LSH*.
MLSH est basée sur la famille des fonctions sensibles à la localité (*Locality-SLensitive*) proposée par Data et al. [Datar et al., 2004] présentée précédemment. Pour définir une séquence d'accès dans chaque table de hachage, Lv et al. définissent d'abord, un vecteur de perturbation $\Delta = \{\delta_1, ..., \delta_M\}$, avec $\delta_i \in \{-1, 0, 1\}$, où M est le nombre de fonctions de hachage pour chaque table. Ensuite, pour une requête q, au lieu d'interroger uniquement la cellule identifiée par $g(q) = [h_1(p), ..., h_M(q)]$ comme dans *LSH*, la méthode *MPLSH* interroge aussi les cellules (*buckets*) $g(q) + \Delta_1, g(q) + \Delta_2, ..., g(q) + \Delta_T$. Ces cellules sont interrogées dans l'ordre suivant leurs *probabilités de succès* qui sont des scores calculés de la manière suivante :

$$score(\Delta) = \sum_{(i=1)}^{M} \mathcal{X}_i(\delta_i)^2 \tag{1.8}$$

$\mathcal{X}_i(\delta_i)$ est la distance sur la dimension i entre l'image du point requête q et la frontière de la cellule $h_i(q) + \delta_i$. Par exemple, dans la figure 1.8, $\mathcal{X}_i(1)$ est la distance de q jusqu'à la frontière de la cellule $h_i(q) + 1$, où $h_i(q) = \lfloor[(a_i.q + b_i)/W]\rfloor$ et $\mathcal{F}_i(q) = a_i.q + b_i$.

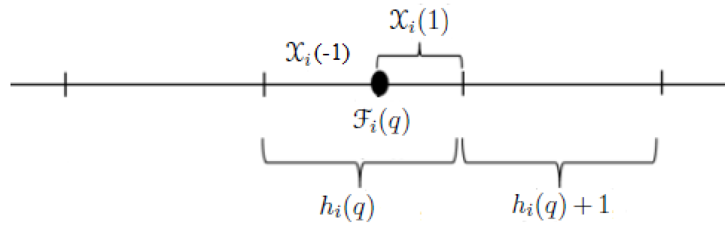


FIGURE 1.8 – Distance entre une requête q et sa cellule voisine

La nouvelle structure d'index *MLSH* est comme *LSH* proposée pour des techniques de recherche des k-plus proches voisins en considérant des données placées en mémoire principale. Dans notre travail, nous considérons le cas de grands volumes de données, et nous supposons que les vecteurs multidimensionnels sont stockés en mémoire secondaire, sur disque par exemple. Le principal inconvénient de *MLSH* dans ce nouveau contexte, est justement son point fort actuel (plusieurs accès par table de hachage). Ici, les cellules (*buckets*) sont stockées aléatoirement sur disque, alors en moyenne une entrée/sortie (I/Os) est nécessaire pour chaque cellule. Notre objectif dans la dernière partie de cette thèse, est de réduire le nombre d'entrées sorties pour une recherche des k-plus proches voisins.

Autres méthodes

Dans [Tao et al., 2009], les auteurs proposent *Locality Sensitive B-tree* (*LSB-tree*) avec comme objectif d'avoir une méthode rapide pour la recherche des k-ppv dans un espace multidimensionnel avec une très bonne qualité de résultat. Pour cela, ils se basent sur une structure d'index construite en deux étapes :

1. Pour un espace multidimensionnel R, chaque objet $o \in R$ est converti en un vecteur de m dimensions en utilisant m fonctions de hachage : $G(o) = [h_1(o), ..., h_m(o)]$.
2. Pour les codes des cellules, obtenus dans la première étape, on considère le Z-ordre des codes, qui permet d'obtenir une valeur de Z-ordre pour chaque code. Ces codes Z-ordre de cellules sont gardés dans un index $B+$ (LSB-tree).

Dans l'algorithme *LSB-tree*, les auteurs considèrent comme cellules voisines celles étant proche selon le Z-ordre. La distance entre deux codes Z-ordre est mesurée par la longueur du plus long préfixe commun (Length of the Longest Common Prefix -LLCP), par exemple, pour $Z(o) = 100101$ et $Z(q) = 100001$, le $LLCP(Z(o), Z(q)) = 3$. La recherche des k-plus proches voisins dans *LSB-tree* s'effectue en trois étapes : *(i)* calculer le Z-ordre d'un point requête q : $Z(q)$. *(ii)* Utiliser $Z(q)$ pour chercher dans le *LSB-tree* : les objets sont accédés dans l'ordre décroissant des valeurs du *LLCP*. Cela correspond à interroger plusieurs cellules dans une table de hachage selon le Z-ordre. Comme dans *MPLSH*, plusieurs arbres peuvent êtres considérés dans *LSB-tree* pour former ce que les auteurs ont appelé *LSB-forest*.

Dans [Gan et al., 2012], les auteurs proposent une nouvelle méthode permettant de réduire le nombre de tables de hachage. Cette méthode appelée *C2LSH*, utilise un ensemble de M fonctions de hachage sans les concaténer. Pour une requête q, si m est le nombre de fonctions de hachage utilisées permettant à un objet o de "rentrer en collision" avec q, et $m > l$, alors o peut être considéré comme un bon candidat dans la recherche des k-ppv de q. Cette méthode permet d'avoir une garantie sur le résultat obtenu, mais comme *MLSH*, elle n'est pas adaptée à notre contexte (données et index sur disque).

1.5 Conclusion

Dans ce chapitre, nous avons commencé par présenter le contexte que nous considérons pour le traitement des requêtes *top-k* et dans lequel nous avons distingué plusieurs paramètres permettant de classifier les algorithmes *top-k* :

- Le type de requête : nous avons mis en évidence trois types de requêtes *top-k* distingués dans la littérature, requêtes de sélection, de jointure et d'agrégation.
- Le types d'accès aux scores : par rapport aux deux types d'accès aux scores locaux (séquentiel et direct), les algorithmes *top-k* proposés peuvent êtres divisés en quatre catégories, *(i)* Accès direct et séquentiel, *(ii)* Sans accès direct, *(iii)* Accès séquentiel avec des accès directs contrôlés, et *(iiii)* Tous types d'accès.
- Le niveau d'exécution des requêtes : nous avons distingué deux niveaux, en dehors ou à l'intérieur du *SGBD*.Le second concerne toutes les techniques qui nécessitent des adaptations et des modifications dans le moteur de la base de données pour supporter la propriété de classement.
- La certitude des données : nous avons mis en évidence trois classes d'algorithmes *top-k* suivant ce critère. La première regroupe les algorithmes considérant des données certaines et un résultat exact, la deuxième concerne les techniques qui utilisent des données certaines et retournant un résultat approximatif et la troisième classe considère des données incertaines.
- La fonction de classement : les techniques de recherche du *top-k* peuvent utiliser ou non une fonction de classement qui agrège les scores de chaque critère. Dans le dernier cas, les techniques basées uniquement sur les scores locaux entrent dans la catégorie "*skyline*".

Dans cette thèse, nous nous plaçons dans le cadre suivant pour le traitement de requêtes *top-k* multicritères :

- Requêtes top-k de sélection.
- Un accès correspond à la consultation d'une seule source, et un accès permet de retourner un seul objet avec son score si l'accès est séquentiel, et uniquement un score si l'accès est direct.
- Les seules informations disponibles dans les sources sont les scores des objets
- Les différents types d'accès aux données sont considérés, et les sources peuvent être des *S-sources*, des *R-sources* ou des *SR-sources*.

Dans la suite de ce chapitre, nous avons passé en revue les principales techniques proposées pour le traitement des requêtes de classement *top-k* à travers plusieurs catégorie d'algorithmes *top-k*, avec lesquels nous souhaitons comparer les performances de nos algorithmes. Ensuite, nous avons présenté les méthodes utilisées dans la littérature pour adapter ces techniques à une recherche approximative dans le but de réduire le coût total du traitement d'une requête *top-k* à travers un compromis entre le coût d'exécution et la qualité du *top-k* retourné. Nous abordons une de ces techniques, l'approximation par arrêt prématuré de l'exécution, dans le contexte général des algorithmes *top-k* et plus particulièrement de nos algorithmes *Breadth-Refine*.

Dans la deuxième partie de ce chapitre, nous avons considéré le contexte particulier de la recherche k-ppv dans les bases de donnés multimédia. Cette recherche est basée sur le contenu multimédia, caractérisé par des descripteurs de contenu représentés par des vecteurs numériques (signatures). Dans ce contexte la recherche par similarité du contenu (CBIR) consiste à trouver les k-plus proches voisins dans l'espace multidimensionnel des signatures. Nous nous intéressons en particulier à la recherche approximative des k-ppv et pour cela, nous avons présenté les techniques les plus connues comme *LSH* [Gionis et al., 1999] et ses variantes [Lv et al., 2007]. Dans ce contexte, nous proposons une nouvelle technique pour la recherche des k-plus proches voisins, inspirée des algorithmes *top-k*, que nous comparons avec les techniques de l'état de l'art et particulièrement la méthode *Multi-Probe LSH*. Comme nous considérons un contexte dans lequel les données sont stockées sur disque (grandes masses de données), nous proposons également une adaptation de *MultiProbe LSH* pour optimiser la recherche approximative des k-ppv avec des données sur disque.

Dans le chapitre suivant, nous présentons la première partie de notre contribution, un cadre générique permettant d'exprimer tous les algorithmes *top-k* dans notre contexte, ce qui nous servira de base de comparaison avec les techniques *top-k* de l'état de l'art. Dans un deuxième temps, nous proposons un nouvel algorithme *top-k* générique permettant le traitement de requêtes top-k de sélection avec n'importe quelle configuration de type et de coût d'accès aux scores. Cet algorithme se base sur une nouvelle stratégie permettant de considérer le *top-k* comme un tout et essayant de maintenir à chaque étape de l'exécution le meilleur *top-k* possible.

La Nature n'utilise que les plus longs fils pour tisser ses motifs, de sorte que la plus petite pièce révèle la structure de la tapisserie toute entière.

– Richard Feynman

CHAPITRE 2

Breadth-Refine : une nouvelle stratégie pour le traitement de requêtes *top-k*

Ce chapitre présente notre première contribution, *Breadth-Refine* (*BR*), une nouvelle stratégie pour le traitement de requêtes *top-k* multicritères, dans laquelle l'ensemble des candidats du *top-k* est considéré comme un tout dans le processus de raffinement des scores. Intuitivement, l'objectif principal de l'heuristique *BR* est d'avoir à chaque instant le meilleur top-k possible, en procédant à un raffinement homogène de tous les candidats du *top-k* courant, au lieu de se focaliser seulement sur le meilleur candidat, comme le font les autres stratégies proposées dans la littérature. Nous commençons ce chapitre par présenter un cadre générique (*GF* pour Generic Framework), permettant d'exprimer n'importe quel algorithme *top-k* dans le contexte général que nous considérons. Ensuite nous présentons l'algorithme *BR* et ses variantes, et avant de conclure, nous finirons le chapitre avec des évaluations expérimentales peermettant de confirmer l'efficacité de la stratégie utilisée dans *BR*.

Contrairement aux algorithmes présentés dans le chapitre 1, section 1.2, à l'exception de *NC*, l'algorithme *Breadth-Refine* (*BR*) couvre toutes les configurations possibles de types d'accès aux sources et s'adapte aux différents paramètres liés aux coûts d'accès aux scores. Nous commençons par présenter le modèle de requête et les données considérées dans le contexte de *BR*, ensuite nous présentons le cadre générique *GF* et plusieurs variantes de l'algorithme dans ce cadre.

2.1 Modèle de données

Nous considérons un ensemble d'objets $\mathcal{O} = \{o_1, ..., o_n\}$, une requête *top-k* multicritères q sur ces objets, et un ensemble de sources $\mathcal{S} = \{S_1, ..., S_m\}$, donnant accès aux scores pour les critères de la requête.

2.1.1 Requêtes

Une requête top-k multicritères q est définie par :

- Un nombre k d'objets à retourner.
- Un ensemble de prédicats de classement (critères) $\mathcal{P} = \{p_1, ..., p_m\}$ qui dépendent de la requête.
- Une fonction d'agrégation monotone $\mathcal{F}$.

Un prédicat $p_j : \mathcal{O} \rightarrow [min_j, max_j]$ retourne pour tout objet un score dans un intervalle donné, et la fonction $\mathcal{F} : \mathbb{R}^m \rightarrow \mathbb{R}$ permet l'agrégation des scores des prédicats en un score global d'objet. Chaque prédicat p_j est évalué indépendamment par une source S_j.

2.1.2 Sources

Une source S_j est caractérisée par *(i)* le type d'accès aux données (*S* pour accès trié (Sorted), *R* pour accès direct (Random), *SR* pour les deux à la fois (Sorted-Random)), et *(ii)* un coût par accès, noté $C_s(S_j)$ pour un accès trié et $C_r(S_j)$ pour un accès direct. Les scores minimum et maximum dans la source S_j sont notés respectivement min_j et max_j. L'ensemble des sources S peut être divisé suivant le type d'accès en trois ensembles disjoints :

$$\mathcal{S} = \mathcal{S}_S \cup \mathcal{S}_R \cup \mathcal{S}_{SR}. \tag{2.1}$$

- $\mathcal{S}_S$ représente l'ensemble des sources triées accessibles uniquement via des accès séquentiels.
- $\mathcal{S}_R$ représente l'ensemble des sources non triées accessibles via des accès directs.
- $\mathcal{S}_{SR}$ représente l'ensemble des sources triées accessibles à la fois avec des accès directs et séquentiels.

Une source triée (type S ou SR) offre une fonction d'accès :

$$\textit{getNext} : \mathcal{S}_S \cup \mathcal{S}_{SR} \rightarrow \mathcal{O} \times \mathbb{R} \cup \{\textit{nil}\}.$$

Cette fonction retourne le prochain couple (o, s), où o, s'il existe, est le prochain objet dans la source en ordre décroissant des scores et s est son score, sinon si o n'existe pas (tous les objets de la source sont déjà retournés) elle retourne la valeur *nil*.

Une source de type R ou SR offre une autre fonction d'accès :

$$\textit{getScore} : (\mathcal{S}_R \cup \mathcal{S}_{SR}) \times \mathcal{O} \rightarrow \mathbb{R}$$

La fonction *getScore* retourne le score d'un objet donné dans la source. Nous appelons *score*(o, S_j) le score retourné pour un objet o dans la source S_j, et *crtmax*$_j$ le plus grand score que la source S_j pourrait retourner. Pour une source non triée $S_j \in \mathcal{S}_R$, $crtmax_j = max_j$ (comme S_j n'est pas triée, le score maximum possible est toujours la valeur maximale dans l'intervalle de scores de la source). Par contre, si S_j est une source triée : $S_j \in \mathcal{S}_S \cup \mathcal{S}_{SR}$, $crtmax_j$ est le score retourné lors du dernier accès séquentiel à la source S_j (initialement $crtmax_j = max_j$).

2.1.3 Candidats

L'algorithme *Breadth-Refine* considère tout objet retourné suite à un accès séquentiel comme un candidat. Le candidat est représenté par un couple *(identifiant, intervalle de score)*. Pour un candidat c, l'intervalle de score $[L(c), U(c)]$, est maintenu tant que son score final n'est pas connu. Les valeurs $L(c)$ et $U(c)$ sont respectivement le score minimum possible et le score maximum possible que peut avoir le candidat c. Ces deux valeurs (bornes) sont calculées en agrégeant les scores du candidat dans les sources où il a été déjà retourné, avec les valeurs minimales et maximales des autres sources.
$L(c) = \mathcal{F}(l_1, ..., l_m)$, l_j =*score*(c, S_j) si c a été retourné dans S_j, sinon $l_j = min_j$
$U(c) = \mathcal{F}(u_1, ..., u_m)$, u_j = *score*(c, S_j) si c a été retourné dans S_j, sinon $u_j = crtmax_j$.

Dans la liste des candidats, nous notons L_k, respectivement U_k la $k^{\text{ème}}$ valeur dans l'ordre décroissant de $L(c)$, respectivement $U(c)$ parmi les candidats. Si le nombre de candidats est inférieur à k, L_k et U_k prennent la valeur *nil*.

Un candidat est appelé *viable* s'il a des chances d'intégrer le *top-k* final. La *condition de viabilité* pour un candidat c est $U(c) \geq L_k$. Avec la monotonicité il est simple de prouver qu'un candidat *non viable* restera ainsi jusqu'à la fin de l'exécution de l'algorithme et peut être supprimé. Nous appelons U_{unseen} la valeur maximale possible du score global d'un objet qui n'est pas encore un candidat (n'a pas été retourné par un accès trié). Initialement $U_{unseen} = \mathcal{F}(max_1, ..., max_m)$.

Nous considérons deux ordres différents pour les candidats. Le premier permet d'avoir la liste des candidats triée dans l'ordre décroissant des scores maximum, nous appelons dans le reste du manuscrit l'ensemble des k meilleurs candidats dans cet ordre : $\mathcal{U}_k$. Le deuxième classement permet d'avoir la liste des candidats triée dans l'ordre décroissant des scores minimum, nous appelons l'ensemble des k meilleurs candidats dans cet ordre $\mathcal{L}_k$.

2.2 Cadre Générique pour le traitement des requêtes *top-k*

Nous proposons tout d'abord un cadre permettant d'exprimer tous les algorithmes *top-k* de la même manière, afin de pouvoir comparer les différentes stratégies utilisées. Le *Cadre Générique* (Generic Framework-*GF*) pour le traitement de requêtes top-k de sélection, considère un contexte où l'accès aux scores est coûteux, et chaque accès à une source permet de retourner un seul objet. Dans *GF*, nous considérons un algorithme *top-k* comme une suite d'accès aux sources, où chaque accès permet de découvrir une information concernant un seul objet. Ce que nous appelons ici information est simplement le score local d'un objet de la base permettant de mesurer la pertinence de l'objet pour un critère donné. En découvrant progressivement ces informations, chaque étape de l'algorithme doit être un pas de plus vers le résultat final *top-k*.

L'algorithme 2 présente *GF*, qui prend comme paramètres d'entrée une requête de sélection q et un ensemble de sources S (toutes les configurations de types d'accès sont possibles). La requête q spécifie la valeur de k (nombre d'objets à retourner dans le résultat), et la fonction d'agrégation monotone $\mathcal{F}$, alors que l'ensemble des sources S matérialise les scores retournés par les prédicats de classement de la requête.
Dans GF, les algorithmes *top-k* maintiennent un ensemble de candidats avec leurs intervalles de scores, cet ensemble est initialement vide. Tous les algorithmes maintiennent aussi un seuil U_{unseen}, sa valeur

Algorithm 2 *GF* : cadre générique pour le traitement de requêtes *top-k*

Require: q and $\mathcal{S}$
  //Initialisations
  candidates $\leftarrow \emptyset$
  $U_{unseen} \leftarrow \mathcal{F}(max_1, ..., max_m)$
  ... *//Autres variables locales*
  repeat
    //Choix du type d'accès : séquentiel ou direct
    if *SortedAccessCondition()* **then**
      //Accès séquentiel
      $S_j \leftarrow$ **BestSortedSource()** *//Choisir une source triée*
      (o, s) $\leftarrow$ getNext(S_j) *//Accès séquentiel dans la source choisie*
      Mise à jour *candidates*, U_{unseen} et les variables locales
    else
      //Accès direct
      $c \leftarrow$ **ChooseCandidate()** *//Choisir un candidat*
      $S_j \leftarrow$ **BestRandomSource(**c**)** *//Choisir une source non triée pour le candidat c*
      $s \leftarrow$ getScore(S_j, c) *//Accès direct dans la source choisie*
      Mise à jour *candidates* et les variables locales
    end if
  until *StopCondition()*
  return *candidates*

est initialisée en agrégeant les scores les plus élevés max_i dans les sources. Les algorithmes peuvent maintenir aussi d'autres variables locales spécifiques à leurs stratégies. La fonction monotone $\mathcal{F}$ garantit que le seuil U_{unseen} et les scores maximum des candidats n'augmentent jamais, elle garantit aussi que les scores minimum des candidats ne diminuent jamais durant l'exécution de l'algorithme.

A chaque étape, un type d'accès est d'abord choisi en utilisant la fonction générique $SortedAccessCondition$ qui indiquera si le prochain accès est séquentiel ou direct. Ensuite une source permettant ce type d'accès est sélectionnée pour effectuer un seul accès.

Si un accès séquentiel est décidé, une source S_j est sélectionnée en utilisant la fonction $BestSortedSource$, pour l'interroger et retourner un objet suite à l'appel de la fonction $getNext$. L'objet retourné sous forme d'un couple identifiant-score est utilisé pour mettre à jour la valeur du seuil U_{unseen}, l'ensemble des candidats et éventuellement les variables locales. Cet objet est ajouté à la liste des candidats s'il est rencontré pour la première fois, ou simplement mis à jour en considérant le score local retourné. En même temps, l'intervalle de score de chaque candidat dont le score local dans la source S_j est encore inconnu est mis à jour. Cette mise à jour permet aussi d'éliminer les candidats *non-viables*. Un candidat c avec un score maximum $U(c) < L_k$ est considéré comme *non-viable* car il n'intègrera jamais le top-k final puisqu'il existe au moins k candidats avec de meilleurs scores et qui le resteront.

Dans le cas d'un accès direct, la fonction $ChooseCandidate$ choisit un candidat c, ensuite, la fonc-

tion $BestRandomSource$ sélectionne une *R-source* ou une *SR-source* à interroger pour retourner le score local de c. L'accès à la source se fait avec la fonction $getScore$ qui permet de retourner le score local du candidat, ensuite, la liste des candidats et éventuellement les variables locales sont mises à jour. Dans la liste des candidats, la mise à jour ne concerne que l'intervalle de score du candidat en question c, et la vérification de l'existence de nouveaux candidats *non-viables*.

La condition d'arrêt des algorithmes *top-k* est contrôlée par la fonction générique $StopCondition$ qui dépend du type de résultat final souhaité par l'algorithme, par exemple des candidats avec des scores exacts ou avec des intervalles de scores. L'arrêt le plus rapide pour un résultat final est obtenu par la condition suivante :

$$StopCondition \equiv (|candidates| = k \wedge L_k \geq U_{unseen}) \tag{2.2}$$

Cette condition signifie que le nombre de candidats viables est k, donc *candidates* $= \mathcal{U}_k = \mathcal{L}_k$, et qu'aucun objet inconnu ne peut intégrer le *top-k*. Il est simple de montrer que cette condition est nécessaire et suffisante pour garantir que l'ensemble des k candidats est un résultat correct.

En effet, si la condition soit remplie, nous disposons de k candidats viables qui ne peuvent pas être dépassés par des objets non vus ($U_{unseen} \leq L_k$), donc les k candidats forment un *top-k* correct.
Au contraire, si la condition n'est pas remplie, alors soit $|candidats| \neq k$ et donc *candidates* ne peut pas constituer le *top-k*, soit $U_{unseen} > L_k$, donc il est possible qu'un objet non vu dépasse le candidat ayant le score minimum L_k.

D'autres conditions sont nécessaires pour assurer des propriétés supplémentaires du résultat final, tel que l'ordre dans le résultat ou les scores exacts des candidats.

Les caractéristiques du résultat exact obtenu par cette condition sont les suivantes :

- Les scores exacts des k candidats retournés ne sont pas forcément calculés, on peut ne disposer que d'intervalles de score.
- L'ensemble des candidats retournés n'est pas trié par score. Un ordre peut être établi entre deux candidats c_1 et c_2 seulement si $L(c_1) \geq U(c_2)$ ou $L(c_2) \geq U(c_1)$, ce qui n'est pas garanti par la condition d'arrêt.

2.2.1 Exemples d'algorithmes *top-k* exprimés dans *GF*

Il est simple de montrer que tout algorithme *top-k* dans le contexte présenté précédemment peut s'exprimer dans *GF*. En effet, pour une requête top-k et un ensemble de sources, chaque algorithme peut être vu comme une séquence d'accès à ces sources, suivis de la mise à jour des structures internes (*candidates*, U_{unseen}, autres variables). Cette séquence peut être obtenue grâce à une suite de décisions concernant le type à effectuer, ainsi que la source à interroger et le candidat à évaluer dans le cas d'un accès direct, ce que fait *GF*. Intuitivement, pour toute séquence, nous pouvons définir de façon appropriée les fonctions *SortedAccessCondition*, **BestSortedSource**, **ChooseCandidate**, **BestRandomSource**, et *StopCondition* pour générer cette séquence avec *GF*.

Nous signalons que ceci n'est pas réalisable avec l'algorithme *NC* [Hwang and Chang, 2007], dans lequel à chaque étape de l'exécution, la première opération consiste à choisir un candidat parmi les k candidats de $\mathcal{U}_k$, ensuite une source est choisie parmi celles où le candidat en question n'a pas été retrouvé.

Cette stratégie appliquée dans le cadre *NC* n'est pas compatible avec des algorithmes qui fixent un ordre des sources à interroger, comme dans *NRA*, car une source S ne sera jamais sélectionnée si les scores des candidats dans $\mathcal{U}_k$ sont déjà connus dans S.

Contrairement à *NC*, le cadre *GF* que nous proposons est compatible avec tous les algorithmes *top-k* dans notre contexte. Nous présentons dans la suite trois exemples d'algorithmes *top-k* exprimés avec *GF* : *NRA* pour les algorithmes sans accès direct, *TA* pour ceux avec *SR-sources* et *MPro* pour les algorithmes avec accès directs contrôlés.

L'algorithme *NRA* exprimé dans *GF*

L'algorithme *NRA* peut être exprimé dans *GF* de la manière suivante :

- *SortedAccessCondition* : retourne à chaque fois *true*, puisque *NRA* considère uniquement des sources triées.
- **BestSortedSource** : retourne simplement la source suivante (stratégie *round-robin*). Une variable locale est nécessaire pour indiquer le numéro d'ordre de la prochaine source à consulter.
- **ChooseCandidate** et **BestRandomSource** ne seront pas considérées puisque la fonction *SortedAccessCondition* retourne toujours *true*.
- *StopCondition* : utilise la condition 2.2.

L'algorithme *TA* exprimé dans *GF*

Nous montrons ici une façon pour exprimer l'algorithme *TA* dans *GF*. *TA* appartient aux algorithmes *top-k* considérant uniquement des sources de données de types *SR-sources*. Les fonctions génériques de *GF* peuvent être instancier de la manière suivante. A noter que des variables locales sont nécessaires pour indiquer la prochaine source pour un accès séquentiel (comme dans *NRA*) et pour indiquer le nombre d'accès directs nécessaires pour l'évaluation complète du candidat découvert lors du dernier accès séquentiel.

- *SortedAccessCondition* : retourne *true* si le nombre d'accès directs restants est 0.
- **BestSortedSource** : Comme dans *NRA*, cette fonction retourne à chaque fois la source suivante. A noter que suite à l'accès séquentiel, si l'objet trouvé est déjà candidat alors le nombre d'accès directs restants reste 0, ce qui produira un nouvel accès séquentiel la prochaine fois. Sinon, le cycle d'accès directs est enclenché en initialisant le nombre d'accès directs à $m - 1$.
- **ChooseCandidate** : le candidat choisi est celui découvert lors du dernier accès séquentiel.
- **BestRandomSource** : pour effectuer les accès directs, *TA* utilise la même méthode que **BestSortedSource** pour le parcours successif des sources en évitant la source où le dernier accès séquentiel a été réalisé.
- *StopCondition* : retourne true si le score du $k^{\text{ème}}$ candidat est supérieur à U_{unseen}, ce qui est en fait équivalent à 2.2.

A noter que les algorithmes *top-k* pour *SR-sources*, de type *TA* peuvent éviter de maintenir les candidats avec des intervalles de score, puisqu'ils calculent immédiatement le score final de chaque nouveau candidat retourné suite à un accès séquentiel. Néanmoins, nous nous plaçons dans un contexte où l'opération la plus coûteuse est l'accès aux sources et nous pouvons donc laisser de côté l'amélioration de la gestion des candidats obtenue par ce type d'algorithme. La version de *TA* obtenue par instanciation de *GF* est donc tout à fait acceptable ici.

L'algorithme *MPro* exprimé dans *GF*

Comme exemple d'algorithme *top-k* avec accès directs contrôlés, nous présentons ici l'instanciation de *MPro* dans *GF* :

- *SortedAccessCondition* : cette fonction retourne $true$, si un objet non vu peut devenir le meilleur de $\mathcal{U}_k$, l'ensemble des k candidats avec les meilleurs scores maximum. Plus précisément, un accès séquentiel est décidé quand $U_1 < U_{unseen}$.
- **BestSortedSource** : *MPro* choisit la seule source triée disponible.
- **ChooseCandidate** : retourne le meilleur candidat de $\mathcal{U}_k$.
- **BestRandomSource** : *MPro* établit un ordre fixe de parcours des sources non triées pour les accès directs, **BestRandomSource** retourne la source suivante selon cet ordre.
- *StopCondition* : retourne *true* si les candidats dans $\mathcal{U}_k$ ont des scores exacts (pas des intervalles de scores), et le score du $k^{\text{ème}}$ candidat est supérieur à U_{unseen} ($U_k \geq U_{unseen}$). Une variante avec des scores incomplets peut aussi être obtenue en utilisant la condition 2.2.

Avec cette capacité d'exprimer tous les algorithmes *top-k*, le cadre *GF* simplifie la comparaison entre les différentes stratégies utilisées dans les algorithmes *top-k*, car elle met en évidence les paramètres qui font la différence entre ces algorithmes.

Dans la suite de ce chapitre, nous présentons la stratégie *BR* et les variantes d'algorithmes que nous proposons pour le traitement des requêtes *top-k* de sélection.

2.3 La stratégie *Breadth-Refine*

Contrairement aux algorithmes *top-k* de l'état de l'art présentés dans le chapitre 1, l'algorithme *Breadth-Refine* (BR), considère n'importe quelle configuration de types d'accès aux sources et de coûts d'accès. Alors que tous les algorithmes *top-k* proposés essayent de raffiner l'intervalle de score du meilleur candidat de $\mathcal{U}_k$, *BR* propose une nouvelle approche heuristique avec l'objectif de garder à chaque étape le meilleur $\mathcal{U}_k$ possible et de l'améliorer à chaque nouvelle itération.

2.3.1 L'algorithme *Breadth-Refine (BR)*

Nous proposons ici un nouvel algorithme pour le traitement des requêtes *top-k* de sélection dont l'idée principale est de maintenir l'ensemble des candidats du *top-k* comme un tout et de ne pas considérer à

Algorithm 3 L'algorithme général *Breadth-Refine*

Require: q and $\mathcal{S}$
  //Initialisations
  candidates $\leftarrow \emptyset$
  $U_{unseen} \leftarrow \mathcal{F}(max_1, ..., max_m)$
  repeat
    //Choisir entre un accès direct ou un accès séquentiel
    if $|candidates| < k$ **or** $U_k < U_{unseen}$ **or** ***CostCondition()*** **then**
      //Accès séquentiel
      $S_j \leftarrow$ **BestSortedSource(**$\mathcal{S}$**)** *//Choisir une source triée*
      $(o, s) \leftarrow$ getNext(S_j) *//Accès séquentiel dans la source choisie*
      Mise à jour *candidates* et U_{unseen}
    else
      //Accès direct
      $c \leftarrow$ **ChooseCandidate(*candidates*,** k**)** *//Choisir un candidat dans le Top-k courant*
      $S_j \leftarrow$ **BestRandomSource(**$\mathcal{S}$**,**c**)** *//Choisir une source non triée*
      $s \leftarrow$ getScore(S_j, c) *//Accès direct dans la source choisie*
      Mise à jour *candidates*
    end if
  until $|candidates| = k$ and $L_k \geq U_{unseen}$
  return *candidates*

chaque fois le meilleur candidat comme une priorité dans le calcul des scores. L'algorithme 3 présente le cadre général de *BR* à partir duquel plusieurs variantes peuvent êtres développées.

Nous présentons ici *BR* directement comme une instantiation de *GF*. Pour le choix de type d'accès avec *SortedAccessCondition*, l'accès trié est préféré dans les cas suivants :

- Si le nombre de candidats est inférieur à *k* : priorité est donnée à la constitution d'un *top-k* courant.
- Si le score maximum possible pour un objet non candidat dépasse le score maximum des candidats du top-k courant de $\mathcal{U}_k$: $U_k < U_{unseen}$. Un accès trié baisse la valeur de U_{unseen} et permet de garder dans le *top-k* courant uniquement des candidats (objets connus).
- Si la condition $CostCondition()$ retourne $true$. Cette condition permet à *BR* de s'adapter à des paramètres de coût d'accès aux scores très différents, typiquement pour préférer les accès triés dans le cas d'accès directs très coûteux.

Si un accès séquentiel est préféré, une source triée est choisie par la fonction $BestSortedSource()$, et l'accès séquentiel est réalisé dans cette source. Par conséquent, la valeur de U_{unseen} ainsi que la liste des candidats sont mises à jour, la mise à jour de cette dernière est composée de trois étapes :

- Ajouter l'objet retourné à la liste des candidats s'il n'est pas déjà parmi les candidats (découvert pour la première fois). Si l'objet est déjà un candidat, alors, son intervalle de score est mis à jour.
- Mettre à jour les intervalles de score des candidats qui ne sont pas encore retrouvés dans la source triée choisie.

- Éliminer les candidats non viables.

Dans le cas d'un accès direct, la fonction $ChooseCandidate()$ retourne un candidat du *top-k* courant pour lequel un accès direct sera effectué. Le choix porte sur le candidat le moins précis (plusieurs variantes sont possibles), afin de faire évoluer le *top-k* courant dans son ensemble, et pas seulement le meilleur candidat. Ensuite, parmi les sources non évaluées pour le candidat choisi, une source de type R ou SR est sélectionnée par la fonction $BestRandomSource()$. L'intervalle de score du candidat choisi est mis à jour pour inclure le score du candidat dans la source sélectionnée.

La condition d'arrêt de l'algorithme est celle de 2.2. Pour rappel, elle signifie que deux conditions sont remplies :

1. Le nombre de candidats viables est égal à k ($|candidates| = k$).

2. Les objets inconnus (non trouvés) ne pourront jamais intégrer le top-k final ($L_k \geq U_{unseen}$).

Le résultat final est l'ensemble des k candidats avec en principe des scores incomplets (intervalles de scores).

2.3.2 Les variantes de *Breadth-Refine*

Plusieurs variantes peuvent êtres obtenues à partir du cadre *BR*, suivant l'instanciation des fonctions suivantes :

- $CostCondition$
- $BestSortedSource$
- $ChooseCandidate$
- $BestRandomSource$

Nous proposons dans un premier temps, trois variantes de *BR* : *(i) BR-Cost*, *(ii) BR-Basic*, et *(iii) BR-First*, ensuite nous présenterons une amélioration de l'algorithme *BR-Cost* appelée *BR-Cost**, qui permet une meilleur estimation du bénéfice de chaque type d'accès. Nous présentons tous ces algorithmes en détail dans le reste de ce chapitre.

BR-Cost

BR-Cost est la variante de référence de l'algorithme général *BR* que nous avons proposé. L'algorithme *BR-Cost* permet d'affiner l'ensemble des candidats du *top-k* courant de la manière *breadth-first* ou *en largeur d'abord*, et s'adapte à plusieurs configurations de coûts d'accès aux sources de données. *BR-Cost* considère une simple somme pondérée comme fonction d'agrégation monotone. La raison est que *BR-Cost* emploie une méthode similaire à celle utilisée dans *StreamCombine* [Güntzer et al., 2001] pour sélectionner la prochaine source triée à interroger, qui a besoin de pouvoir évaluer $\partial\mathcal{F}/\partial S_j$ (l'importance) d'une source. ce qui n'est pas possible pour toute fonction monotone.

Dans *BR-Cost*, les fonctions génériques sont instanciées de la manière suivante :

- $CostCondition$: l'objectif de cette fonction est de contrôler à chaque étape de l'exécution le type d'accès à effectuer, en se basant uniquement sur le critère coût d'accès. Plus précisément, la fonction $CostCondition$ permet de limiter les accès coûteux, afin de réduire le coût total de l'exécution. Dans le cas général, l'utilisation de cette condition permet de réaliser plus d'accès séquentiels que d'accès directs, généralement plus coûteux. La méthode que nous utilisons ici est similaire à celle utilisée dans l'algorithme *CA* [Fagin et al., 2001], et elle consiste à calculer le ratio r entre le coût d'un accès direct et le coût d'un accès séquentiel. Ensuite, elle permet de contrôler le nombre d'accès coûteux, en autorisant un accès direct pour chaque r accès séquentiels réalisés (où l'inverse si l'accès séquentiel est plus coûteux).

- $BestSortedSource$: cette fonction permet de sélectionner la meilleure source triée S_j en se basant sur une notion de bénéfice apporté par chaque source. Le bénéfice est ici mesuré par l'impact d'un accès séquentiel à une source triée sur les intervalles de scores des candidats du *top-k* courant ($\mathcal{U}_k$). L'objectif est de réduire la taille des intervalles de scores de ces candidats, afin d'avoir des informations plus précises sur les scores, et de s'approcher plus de la valeur du score final de chaque candidat. Par exemple, si les scores locaux de tous les candidats du *top-k* courant ne sont pas connus dans une source triée S_j, cette dernière est probablement la meilleure source à interroger dans cette étape, puisqu'elle va permettre de réduire la taille des intervalles de scores de tous les candidats du *top-k* courant. L'idée d'utiliser la notion de bénéfice pour choisir la meilleur source à interroger a été proposé dans l'algorithme *StreamCombine* [Güntzer et al., 2001]. Dans notre cas, le bénéfice B_j d'une source triée S_j est calculé de la manière suivante :

$$B_j = coef_j \times N_j \times \delta_j / C_s(S_j) \tag{2.3}$$

  La variable $coef_j$ est le coefficient de la source S_j dans la fonction d'agrégation, N_j est le nombre de candidats du *top-k* courant dont les scores locaux dans la source S_j sont encore inconnus, δ_j est la baisse attendue du score maximum dans la source S_j, et $C_s(S_j)$ est le coût d'un accès séquentiel dans S_j. La variable N_j mesure ici le nombre d'objets du *top-k* qui sont concernés par la baisse du score résultant de l'accès à S_j puisqu'ils n'ont pas été retrouvés dans cette source, et $coef_j$ et δ_j permettent de connaitre la valeur de cette baisse. La valeur de δ_j est très importante pour le choix de la meilleure source triée à interroger, puisqu'elle mesure la baisse résultante dans les intervalles de scores des candidats concernés, mais aussi la baisse de la valeur de U_{unseen}.

- $ChooseCandidate$: sélectionne le candidat le moins précis dans le *top-k* courant en comparant simplement le nombre d'accès directs réalisés pour chaque candidat. Le top-k courant considéré ici est toujours composé des k meilleurs candidats triés par ordre décroissant des scores maximums ($\mathcal{U}_k$).

- $BestRandomSource$: c'est la fonction qui sélectionne une *SR-source* ou une *R-source* pour un accès direct concernant un candidat déjà choisi. Le choix de la meilleure source S_j pour un accès direct est basé sur une notion de bénéfice similaire à celle utilisée pour choisir la meilleure source triée si un accès séquentiel est préféré. La différence ici, est que l'accès direct ne change pas la valeur de U_{unseen} mais juste l'intervalle de score du candidat concerné par l'accès. La taille de l'intervalle de score venant de la source S_j est $crtmax_j - min_j$, où $crtmax_j$ rest fixe à max_j pour une *R-source* et baisse avec les accès triés dans une *SR-source*. En obtenant un score exact

dans S_j après un accès direct, la taille de l'intervalle de scores du candidat baisse de $coef_j \times (crtmax_j - min_j)$.

Dans notre cas, le bénéfice B_j est calculé comme suit :

$$B_j = coef_j \times (crtmax_j - min_j)/C_r(S_j) \quad (2.4)$$

$C_r(S_j)$ est le coût d'un accès direct pour la source S_j. Le bénéfice correspond donc à la baisse de la taille de l'intervalle de score, rapportée au coût de l'accès.

BR-Basic

BR-Basic est la version de base de l'algorithme général *BR*, exprimant simplement la stratégie *Breadth-First* sans prendre en considération les contraintes liées aux coûts d'accès. L'algorithme *BR-Basic* peut être considéré comme une variante de *BR-Cost* qui ne s'adapte pas à toutes les configurations de coût d'accès aux sources de données, mais reste plutôt adapté à un contexte dans lequel les accès directs et séquentiels ont le même coût. Dans *BR-Basic*, les fonctions génériques $BestSortedSource$, $ChooseCandidate$ et $BestRandomSource$ sont instanciées de la même manière que dans *BR-Cost*, mais, la fonction $CostCondition$ retourne systématiquement *false* pour ne pas favoriser un accès par rapport à un autre. Dans ce cas la fonction générique $SortedAccessCondition$ du cadre *GF* retourne $true$ et favorise un accès séquentiel si une des deux conditions suivantes est vérifiée : $|candidates| < k$, ou $U_k < U_{unseen}$. En comparant les algorithmes *BR-Basic* et *BR-Cost*, nous pourrons mesurer la pertinence de la condition du coût utilisée dans *BR-Cost* permettant une limitation du nombre des accès coûteux.

BR-First

L'algorithme *BR-First* suit la même logique utilisée dans tous les algorithmes de l'état de l'art qui doivent choisir un candidat à raffiner par accès directs, en sélectionnant le meilleur candidat selon le score maximum. Il peut être considéré comme une variante de l'algorithme *BR-Basic*, avec une différence dans l'approche suivie pour choisir le candidat à évaluer par la fonction *ChooseCandidate*. Cette utilisation de la fonction générique $ChooseCandidate$ est un cas particulier pour la même fonction ayant servi dans le cadre BR. La comparaison entre les algorithmes *BR-First* et *BR-Basic* nous permettra d'évaluer notre nouvelle heuristique et de mesurer le bénéfice résultant de l'utilisation de l'approche *breadth-First* pour le choix du candidat à raffiner.

*BR-Cost** : une alternative à l'algorithme *BR-Cost*

Dans un deuxième temps, nous proposons une amélioration de l'algorithme *BR-Cost* en proposant une nouvelle méthode permettant de mieux estimer le ratio r entre les coûts d'accès directs et séquentiels. Dans *BR-Cost**, r est calculé comme un rapport de bénéfices et non plus un rapport de coûts d'accès. Plus précisément, un bénéfice est calculé pour chaque accès à une source, ensuite nous calculons un bénéfice aggrégé pour les accès triés/directs, dont le rapport donnera la valeur de r. Intuitivement, le bénéfice d'une source est donné par le rapport entre la baisse de la taille des intervalles de score des candidats et le coût d'accès à la source.

L'exemple de la figure 2.1 montre le calcul du bénéfice pour une source triée S_j de type *S-source*. La valeur maximale courante des scores dans cette source est $crtmax_j$, et le nombre d'objets qui ne sont

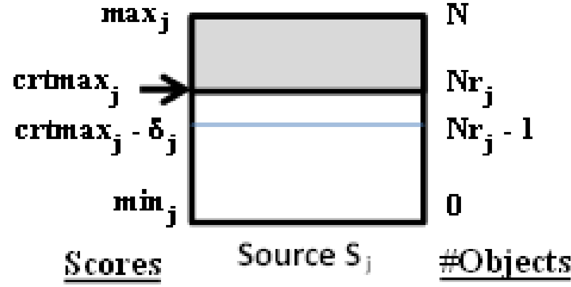


FIGURE 2.1 – Les scores dans une source triée S_j

pas encore retrouvés est Nr_j. Un accès séquentiel dans la source S_j permet dans un premier temps, de raffiner l'intervalle de score de l'objet retourné, et dans un deuxième temps, de produire une baisse de δ_j de la valeur de $crtmax_j$. Cette baisse affecte aussi les scores maximum des $Nr_j - 1$ objets qui ne sont pas encore retournés dans la source S_j . Concernant l'objet retourné suite à cet accès, la taille de son intervalle de score baisse de $coef_j \times (crtmax_j - min_j)$, $coef_j$ étant le coefficient de la source S_j dans la fonction d'agrégation. Pour les $Nr_j - 1$ objets non retournés par la source S_j, les scores maximum de leurs intervalles de scores baissent de δ_j. Par conséquent, le bénéfice d'un accès trié à la source S_j est :

$$B_s(S_j) = coef_j \times (crtmax_j - min_j + (Nr_j - 1) \times \delta_j)/C_s(S_j) \tag{2.5}$$

Il faut noter que les bénéfices des sources varient dans le temps. Si δ_j ne varie pas beaucoup, le bénéfice diminue globalement car les valeurs de $crtmax_j$ et Nr_j baissent. Nous proposons une approximation du bénéfice moyen en considérant :

- $\delta_j \approx (max_j - min_j)/N$,
- $crtmax_j \approx (max_j + min_j)/2$,
- $Nr_j \approx N/2$.

Le bénéfice moyen pour un accès à une source triée de type *S-source* ou *SR-source* sera donc, en considérant les valeurs moyennes :

$$\begin{aligned}\overline{B_s}(S_j) &\approx coef_j \times (\tfrac{(max_j - min_j)}{2} + (\tfrac{N}{2} - 1) \times \tfrac{(max_j - min_j)}{N})/C_s(S_j) \\ &\approx coef_j \times (\tfrac{(max_j - min_j)}{2} + \tfrac{N}{2} \times \tfrac{(max_j - min_j)}{N})/C_s(S_j)\end{aligned}$$

$$\overline{B_s}(S_j) \approx coef_j \times (max_j - min_j)/C_s(S_j) \tag{2.6}$$

Pour les accès directs, le bénéfice est calculé sur les mêmes principes, et correspond à la discussion ayant mené à la formule 2.4. Donc si S_j est une *SR-source*, le bénéfice d'un accès direct sera :

$$B_{rs}(S_j) = coef_j \times (crtmax_j - min_j)/C_r(S_j) \tag{2.7}$$

Le bénéfice moyen d'un accès direct à une source de type *SR*, en considérant les même suppositions que pour le calcul du bénéfice d'un accès séquentiel et en remplaçant $crtmax_j$ par sa valeur moyenne $\frac{max_j + min_j}{2}$, est le suivant :

$$\overline{B_{rs}}(S_j) \approx coef_j \times (max_j - min_j)/2C_r(S_j) \quad (2.8)$$

Pour une *R-source*, le score maximum possible pour un objet ne varie pas durant l'exécution, et aucune baisse de la valeur de $crtmax_j$ n'est mesurable. Dans ce cas $crtmax_j = max_j$ tout au long de l'exécution. Le bénéfice d'un accès direct dans une *R-source* est donc calculé de la manière suivante :

$$B_r(S_j) = \overline{B_r}(S_j) = coef_j \times (max_j - min_j)/C_r(S_j) \quad (2.9)$$

Pour calculer le ratio des accès, un bénéfice global est calculé pour chaque type d'accès. Le bénéfice global SB d'un accès séquentiel est la somme des bénéfices des accès triés dans les sources triées de types *S* ou *SR* :

$$SB = \sum_{S_j \in \mathcal{S}_S \cup \mathcal{S}_{SR}} \overline{B_s}(S_j) \quad (2.10)$$

De la même manière, le bénéfice global RB pour un accès direct, est la somme des bénéfices des accès directs dans les sources de types *R-source* et *SR-source* :

$$RB = \sum_{S_j \in \mathcal{S}_R} \overline{B_r}(S_j) + \sum_{S_j \in \mathcal{S}_{SR}} \overline{B_{rs}}(S_j) \quad (2.11)$$

$\mathcal{S}_S$, $\mathcal{S}_R$ et $\mathcal{S}_S R$ sont respectivement les ensembles des sources de type *S-source*, *R-source* et *SR-source*.

La nouvelle valeur du ratio des bénéfices utilisée dans l'algorithme *BR-Cost** est alors calculée de la manière suivante :

$$r = SB/RB = \frac{\sum_{S_j \in \mathcal{S}_S \cup \mathcal{S}_{SR}} \frac{A_j}{C_s(S_j)}}{\sum_{S_j \in \mathcal{S}_R} \frac{A_j}{C_r(S_j)} + \sum_{S_j \in \mathcal{S}_{SR}} \frac{A_j}{2C_r(S_j)}} \quad (2.12)$$

Nous avons noté A_j l'amplitude de l'intervalle de scores produit par la source S_j : $coef_j \times (max_j - min_j)$.

Dans cette section, nous avons présenté l'algorithme *Breadth-Refine*, un nouvel algorithme *top-k* permettant le traitement des requêtes de sélection. *BR* est un algorithme générique, permettant de s'adapter à n'importe quelle configuration de types et de coûts d'accès aux sources. Il se base sur une nouvelle stratégie *breadth-first* permettant de considérer l'ensemble des candidats du *top-k* comme un tout, en essayant de maintenir à chaque étape de l'exécution le meilleur top-k possible. Initialement, nous avons présenté trois variantes de *BR* : *BR-Cost*, *BR-Basic*, *BR-First*, ensuite, nous avons présenté une variante de *BR-Cost*, l'algorithme *BR-Cost**, dans lequel nous avons proposé une amélioration de l'estimation du ratio r. Tous ces algorithmes seront comparés expérimentalement dans la section 2.5

2.4 Algorithmes Génériques

Comme nous l'avons mentionné, la grande majorité des algorithmes *top-k* proposés dans la littérature considère des configurations spécifiques de types et de coûts d'accès aux sources. *NC* est le seul algorithme qui vise la généricité, mais dans un cadre différent que celui de *BR* : calculer les scores exacts des k meilleurs objets et les retourner dans le résultat.

Avant *NC*, l'algorithme *CA* a proposé un premier pas vers la généricité, mais limité au coût d'accès.

Toutefois, *NC* n'est pas comparable à *BR*. Au niveau théorique, la notion d'accès nécessaire de *NC* ne s'applique pas dans le contexte de score finaux incomplets de *BR*. Au niveau pratique, l'algorithme de référence de *NC*, *SR/G*, utilise des informations supplémentaires (échantillonnage) et se place dans un contexte différent de *BR*.

Pour pouvoir comparer *BR* avec d'autres stratégies génériques, nous proposons de nouvelles variantes des algorithmes *NC* et *CA* adaptées à notre contexte.

2.4.1 Necessary Choices

Necessary Choices [Hwang and Chang, 2007] est le premier algorithme *top-k* qui vise la généricité en s'adaptant à n'importe quelle configuration de types et de coûts d'accès aux données, dans un contexte où les scores exacts des candidats du *top-k* final doivent êtres calculés. Dans ce contexte, à chaque étape de la recherche du *top-k*, *NC* identifie des accès nécessaires à effectuer pour les candidats de l'ensemble $\mathcal{U}_k$ (*top-k* courant trié selon les valeurs des scores maximums). L'objectif de *NC* est de proposer un cadre générique permettant d'instancier des algorithmes *top-k* en limitant les accès aux scores, aux seuls accès nécessaires. Les auteurs montrent que dans le contexte *NC*, les accès des candidats de $\mathcal{U}_k$ sont nécessaires. Dans ce cadre, un algorithme de type *NC* effectue des accès nécessaires aux sources en suivant la procédure suivante :

- A chaque étape, sélectionner un candidat avec un score incomplet parmi les candidats de l'ensemble $\mathcal{U}_k$, généralement le meilleur candidat (avec le score maximum le plus élevé).
- Effectuer un accès non réalisé pour le candidat choisi. Un accès trié dans une source S est réalisé pour un candidat c lorsque c est a été trouvé dans S.

Dans le cadre générique de *NC* [Hwang and Chang, 2007], les auteurs proposent une stratégie *SR* (Sorted then Random) qui favorise systématiquement les accès séquentiels aux accès directs, pour chaque candidat. Cette stratégie est utilisée dans l'algorithme *SR/G* (Sorted then Random/ Global scheduling) proposé par les auteurs comme l'algorithme de référence du cadre *NC*. Comme nous l'avons déjà expliqué dans le chapitre précédent, l'algorithme *SR/G* est guidé principalement par les deux paramètres D et H, qui indiquent respectivement la profondeur jusqu'à laquelle les accès séquentiels seront toujours favorisés, et l'ordre à suivre pour effectuer les accès directs pour tous les candidats. *SR/G* utilise une technique d'échantillonnage pour sélectionner les valeurs optimales du couple (D, H). Nous rappelons que D est l'ensemble des seuils fixés dans toutes les sources triées de type *S* ou *SR* : $D = \{d_1, ..., d_m\}$, et H indique pour tous les candidats l'ordre à suivre pour les accès directs dans les sources de type *R* ou *SR*.

Malgré sa généricité, *NC* est difficilement comparable avec *BR*. Dans un contexte où les scores finaux exacts ne sont pas nécessairement calculés pour les objets du *top-k* final, l'identification des accès nécessaires effectuée par *NC* n'est plus pertinente. Une stratégie basée sur l'échantillonnage comme celle utilisée dans l'algorithme *SR/G* n'est pas toujours possible, et ne garantit pas systématiquement une distribution de scores similaire aux échantillons étudiés.

Nous proposons ici une variante de l'algorithme de référence *SR/G*, adaptée au contexte de *BR* en considérant des objets avec des scores incomplets dans le *top-k*. Nous proposons aussi une stratégie heuristique pour l'approximation du meilleur couple (D, H) inspirée de *BR-Cost**, sans échantillonnage. Notre objectif est de comparer les stratégies proposées dans les algorithmes *BR-Cost** et *SR/G* dans un contexte

similaire ou très proche. Pour cela, la variante de *SR/G* que nous proposons est exprimée dans le cadre *GF* de la manière suivante :

- En dehors des paramètres D et H, une variable locale garde le *meilleur candidat* avec un score incomplet dans l'ensemble $\mathcal{U}_k$ (candidat avec le meilleur score maximum). *SR/G* commence par réaliser un accès séquentiel dans une source triée, et le *meilleur candidat* est initialisé avec le premier objet retourné, ensuite il sera mis à jour à chaque étape de l'exécution. Il faut noter que tant que la condition d'arrêt n'est pas vérifiée, il y a au moins un candidat avec un score incomplet dans $\mathcal{U}_k$.

- *SortedAccessCondition() :* après avoir sélectionné le *meilleur candidat* c, un accès séquentiel est décidé après le calcul de l'ensemble suivant $S = \{S_j \in S_S \cup S_{SR} | c$ n'est pas consulté dans $S_j \wedge d_j \leq crtmax_j\}$. S représente l'ensemble des sources où c n'a pas été trouvé et où la position courante est au dessus du seuil. *SortedAccessCondition* retourne *true* si S n'est pas vide.

- **BestSortedSource() :** cette fonction retourne une source parmi les sources de S défini précédemment.

- **ChooseCandidate() :** le candidat choisi est le meilleur candidat défini précédemment.

- **BestRandomSource() :** en respectant l'ordre des sources exprimé dans H, cette fonction retourne la première source de type *R-source* ou *SR-source* où le candidat sélectionné n'est pas encore retourné.

- *StopCondition() :* pour pouvoir retourner un top-k sans calculer les scores finaux exacts de tous les candidats, cette fonction utilise la formule 2.2

La stratégie que nous utilisons pour identifier le meilleur couple (D, H) est basée sur la notion de bénéfice des sources utilisée dans *BR-Cost** :

- H : comme dans *BR-Cost**, nous calculons pour chaque *SR-source* ou *R-source* un bénéfice en utilisant les formules 2.8 et 2.9. Ensuite H prend l'ensemble des sources dans l'ordre décroissant des bénéfices.

- D : l'estimation des valeurs de l'ensemble D se base sur trois hypothèses concernant l'optimisation des accès :

  1. Le nombre d'accès séquentiels à une source triée doit être proportionnel au bénéfice de la source calculé en utilisant la formule 2.6.
  2. Les accès séquentiels effectués dans les sources triées jusqu'à atteindre les seuils de D doivent produire une baisse du seuil U_{unseen} pour pouvoir distinguer le *top-k* final. Cette baisse doit permettre au moins d'avoir $U_{unseen} = R_k$, avec R_k le score réel du $k^{\text{ème}}$ objet du *top-k* final.
  3. Si nous considérons n_j le nombre d'accès séquentiels nécessaire pour atteindre le seuil d_j dans la source S_j ($n_j = N - Nr_j$), la relation entre n_j et le seuil d_j dépend de la distribution des scores dans les sources. Comme la distribution des scores est généralement non connue, nous proposons ici une approximation basée sur une distribution uniforme.

En notant Δ_j la baisse du score nécessaire pour atteindre le seuil d_j ($\Delta_j = max_j - d_j$), les trois hypothèses précédentes donnent :

1. $\forall j, n_j = C \times \overline{B_s}(S_j)$, où C est une constante.
2. $U_{max} - R_k = \sum coef_j \times \Delta_j$, où U_{max} est le score maximum possible pour un objet calculé en utilisant la fonction d'agrégation $\mathcal{F}$: $U_{max} = \mathcal{F}(max_1, ..., max_m)$.
3. $\forall j, n_j/N = (max_j - d_j)/(max_j - min_j)$.

La résolution de ce système d'équations produit l'estimation suivante pour le seuil d_j :

$$d_j = max_j - \frac{A_j^2}{coef_j \times C_s(S_j)} \times \frac{U_{max} - R_k}{\sum_j A_j^2 / C_s(S_j)} \tag{2.13}$$

Nous rappelons que $A_j = coef_j \times (max_j - min_j)$ représente l'impact de la source S_j sur l'intervalle de score d'un objet, calculé selon une fonction d'agrégation de type somme pondérée dans laquelle $coef_j$ est le coefficient du critère évalué par la source S_j.

Plusieurs méthodes sont possibles pour estimer R_k, le score final du $k^{\text{ème}}$ objet dans le *top-k* , par exemple, en considérant une distribution Gaussienne des scores globaux avec des paramètres qui dépendent des distributions de scores dans les sources. Dans notre évaluation expérimentale de l'algorithme *NC*, la méthode d'estimation de la valeur R_k n'est pas importante, puisque nous pré-calculons la vraie valeur de R_k, et ainsi nous considérons le cas le plus favorable de la variante *SR/G*.

2.4.2 Combined Algorithm

Initialement, l'algorithme *CA* ne vise pas la généricité, car il considère uniquement des sources de type *SR-source*, où les accès directs sont plus coûteux que les accès séquentiels. Cependant, en le comparant aux algorithmes *NRA* et *TA*, l'algorithme *CA* peut être considéré comme une première tentative vers la généricité, en considérant différents coûts pour les accès directs et séquentiels, et en proposant une combinaison des stratégies de *NRA* et *TA* pour s'adapter au contexte des coûts d'accès aux sources différents.

Nous proposons ici *CA-gen*, une variante générique de *CA* adaptée à n'importe quelle configuration de types d'accès aux sources. Comme dans l'algorithme *CA*, nous utilisons dans *CA-gen* le même principe pour calculer r le ratio entre le coût d'un accès direct et le coût d'un accès séquentiel. Par contre, comme nous considérons une configuration générique concernant les types d'accès aux sources, nous adaptons la stratégie *CA* à ce contexte. Comme *CA*, l'algorithme *CA-gen* commence par effectuer r accès séquentiels dans chaque source triée (de type *S-source* ou *SR-source*). Les candidats retournés suite à ces accès sont maintenus dans une liste de candidats triée par ordre décroissant du score maximum. Ensuite, l'algorithme sélectionne c, le candidat possédant le meilleur score maximum, qui n'a pas été évalué pour toutes les sources permettant un accès direct. Tous les accès directs restants pour c sont ensuite effectués, et la liste des candidats est mise à jour. L'algorithme s'arrête quand au moins k candidats possèdent des scores minimum supérieurs à la valeur de U_{unseen}.

La différence entre les algorithmes *CA* et *CA-gen* vient en partie du nouveau contexte considéré dans le second algorithme. Comme *CA* considère uniquement des sources de type *SR-source*, les accès directs effectués pour un candidat sélectionné, permettent de calculer son score exact. A la différence de *CA*, l'algorithme *CA-gen* ne peut pas garantir cela, puisqu'il suffit de ne pas connaitre le score local du

candidat sélectionné c dans une *S-source* pour maintenir ce dernier avec un intervalle de score et non un score exact. En cela, *CA-gen* se place dans le même contexte que *BR*, où nous considérons un *top-k* final composé de candidats avec des scores incomplets (intervalles de score).

La stratégie utilisée dans l'algorithme *CA-gen* est exprimée dans le cadre générique *GF* de la manière suivante :

- D'abord, deux variables locales sont utilisées pour permettre la gestion du cycle des accès séquentiels : *(i)* ns indique la prochaine source triée à interroger, la valeur initiale est la première source triée. *(ii)* na indique le nombre d'accès effectués dans la dernière source consultée lors du cycle courant. Le cycle se termine quand toutes les sources triées sont interrogées r fois, c'est à dire quand la variable ns pointe de nouveau vers la première source triée, et quand la variable $na = r$.

- *SortedAccessCondition() :* tant que le cycle des accès séquentiels n'est pas terminé, cette fonction retourne *true*.

- **BestSortedSource()** : cette fonction retourne toujours la source pointée par la variable ns.

- **ChooseCandidate()** : pour effectuer les accès directs, cette fonction retourne le meilleur candidat de $\mathcal{U}_k$, tel que décrit ci-dessus (ayant encore des accès directs à effectuer)

- **BestRandomSource()** : cette fonction retourne la première source de type *R-source* ou *SR-source*, dans laquelle le meilleur candidat choisi n'est pas encore retourné. Si aucune source ne correspond à ces critères, les accès directs seront reportés après le prochain cycle des accès séquentiels, et na est réinitialisée.

- *StopCondition()* : la condition d'arrêt est donnée par la formule 2.2 qui permet d'avoir des candidats avec des scores incomplets dans le résultat final.

Dans la section suivante, nous proposons une évaluation expérimentale des différents techniques de traitement des requêtes *top-k* de sélection, afin de comparer l'algorithme *BR* avec les algorithmes de l'état de l'art. Dans un premier temps, nous évaluons les algorithmes de type *BR* en considérant les cadres spécifiques aux algorithmes de l'état de l'art, par exemple un cadre où les sources sont uniquement de type *S-source* ou *SR-source* pour NRA, etc. Ensuite, nous proposons une comparaison des algorithmes génériques et montrons l'efficacité de la stratégie *BR* comparée aux stratégies proposées par *NC* ou *CA*.

2.5 Évaluations expérimentales

Dans un premier temps, nous comparons les algorithmes *BR* aux algorithmes considérant des configurations spécifiques pour le type et le coût d'accès aux scores, et nous proposons une première évaluation des algorithmes *BR* dans un cadre générique. Dans un deuxième temps, nous nous plaçons dans un contexte générique et nous proposons une première comparaison expérimentale des algorithmes génériques. Comme dans *NC*, l'algorithme *BR-Cost** utilise la notion du bénéfice pour choisir la source à interroger ainsi que calculer le ratio *r*, c'est pour cette raison que nous utilisons *BR-Cost** dans ces expériences et non *BR-Cost*.

2.5.1 Algorithmes *top-k* spécifiques

Nous présentons tout d'abord une comparaison expérimentale des algorithmes *BR* et des algorithmes de l'état de l'art dans le contexte spécifique à ces derniers (en termes de type et coût d'accès). Nous mesurons aussi l'impact de la nouvelle heuristique proposée dans *BR* et la prise en compte des différentes configurations de coûts d'accès, en comparant les variantes de l'algorithme *BR* : *BR-Cost*, *BR-Basic*, *BR-First*.

Les données

Pour les évaluations expérimentales, nous utilisons une base de donnée synthétique. Nous générons une liste de scores pour chaque source de données, où chaque liste représente les scores de tous les objets de la base pour un prédicat de la requête. Les valeurs des scores sont dans l'intervalle $[0, 1]$. Les sources de données sont indépendantes et ont des distributions de données similaires. Le coût d'accès séquentiel C_s ou direct C_r est le même dans toutes les sources triées, respectivement avec accès direct. Nous considérons ici trois distributions de scores :

- Distribution Uniforme : les valeurs sont uniformément distribuées.
- Distribution Gaussienne : les valeurs sont générées à l'aide d'un mélange de trois gaussiennes.
- Distribution Zipfian : les valeurs des scores sont générées en utilisant une fonction de *Zipf* avec1000 valeurs distinctes et un paramètre $z = 1$.

Le choix des paramètres

Dans toutes les expériences, nous mesurons le coût total d'exécution de chaque algorithme, qui est la somme des coûts d'accès aux sources effectués durant l'exécution. Plus précisément, si Ns_j est le nombre d'accès séquentiels et Nr_j est le nombre d'accès directs à une source S_j, alors le coût total d'exécution d'un algorithme *top-k* est mesuré de la manière suivante :

$$cost = \sum_{S_j \in \mathcal{S}_S \cup \mathcal{S}_{SR}} Ns_j C_s(S_j) + \sum_{S_j \in \mathcal{S}_R \cup \mathcal{S}_{SR}} Nr_j C_r(S_j) \tag{2.14}$$

Chaque résultat que nous présenterons sur les figures, est la moyenne de 8 mesures réalisées sur différentes sources générées aléatoirement. Les paramètres considérés dans les expériences sont les suivants :

- La **taille de la base** : la valeur par défaut de la taille des données est 10000, mais pour évaluer le comportement de nos méthodes sur des volumes plus importants, la taille de la base sera variée et nous considèrerons les valeurs suivantes : 20000, 40000, 60000, 80000 et 100000.
- Le **nombre d'objets dans le résultat** : la valeur de k par défaut est 50, mais nous considèrerons aussi d'autres valeurs pour évaluer l'impact de k sur le coût d'exécution final. Les valeurs considérées sont alors : 20, 40, 50, 60, 80, 100.
- Le **nombre de sources** : nous considérons par défaut le même nombre de sources pour chaque type : $Ns = 3$ S-sources, $Nr = 3$ R-sources et $Nsr = 3$ SR-sources. Nous avons aussi varié le nombre de sources pour évaluer l'adaptabilité des algorithmes *BR* dans différentes configurations.

- Les **coûts d'accès aux sources** : la valeur par défaut considérée est $Cr(=5) > Cs(=1)$, mais nous testons d'autres configurations avec $Cr = Cs(=1)$ et $Cr(=1) < Cs(=5)$.
- **Distribution des données** : nous considérons trois types de distributions, uniforme (distribution par défaut), gaussienne et de zipf.

Les algorithmes testés

Les expériences que nous présentons sont regroupées par type d'accès aux sources, deux catégories seront considérées et pour chacune de ces catégories nous comparons les algorithmes *BR* avec des algorithmes adaptés à la configuration correspondante. Les deux catégories sont les suivantes :

1. Seulement des *S-sources* et des *SR-sources* : ici nous comparons nos méthodes avec les algorithmes *NRA* et *MPro*.
2. Seulement des *R-sources* et des *SR-sources* : les algorithmes considérés sont *Upper*, *TAz* et *MPro*.

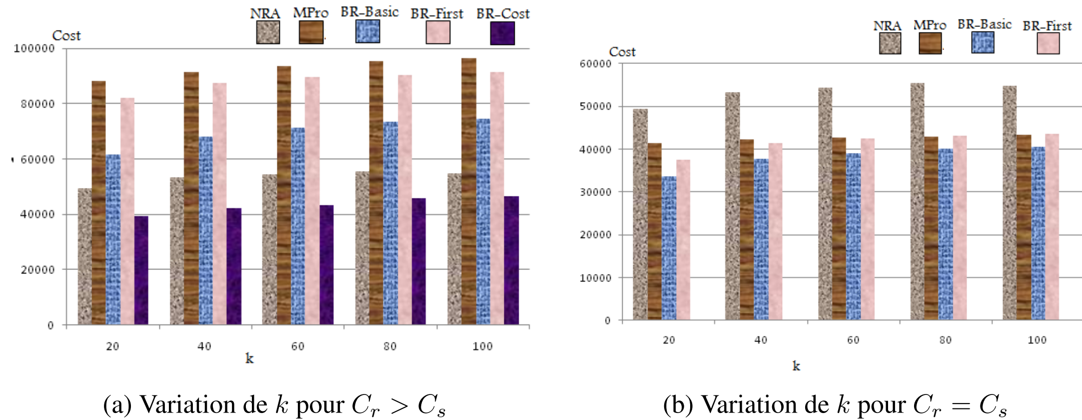
Dans ces expériences, nous avons adapté l'algorithme *MPro* pour n'importe quelle configuration de type d'accès. En principe, dans *MPro*, une seule source triée est prévue. En cas de plusieurs sources triées, nous considérons que ces sources sont combinées avec un algorithme *NRA* pour produire en sortie une seule source triée les combinant. Les *SR-sources* sont considérées différemment dans *MPro* selon le contexte : s'il n'existe pas de *S-sources*, elles sont considérées comme sources triées, sinon s'il n'existe pas de *R-sources*, elles sont considérées comme des sources non triées avec des accès directs. Pour le choix des accès directs, nous fixons un ordre non pas par échantillonnage, mais en utilisant le même ordre des bénéfices utilisé par *BR*.

Résultats expérimentaux

Comme nous l'avons déjà signalé, les résultats des expériences sont regroupés par configuration de types d'accès aux sources.

SR-sources + S-sources :
Nous comparons ici les algorithmes *BR-Cost*, *BR-Basic* et *BR-First* avec *NRA* et *MPro*. D'abord, nous commençons par varier la valeur de k dans deux cas différents : $C_r > C_s$ et $C_r = C_s$. Dans le cas où $C_r < C_s$, l'algorithme *NRA* est largement défavorisé, pour cette raison nous ne considérons pas cette configuration ici.

La figure 2.2(a) présente le coût d'exécution des différents algorithmes en variant la valeur de k dans le cas où l'accès direct est plus coûteux que l'accès séquentiel. Pour tous les algorithmes, le coût d'exécution augmente avec la valeur de k, mais l'algorithme *BR-Cost* est moins coûteux que *MPro* et même que *NRA* qui n'effectue pas d'accès directs. L'importance de la prise en compte des paramètres de coût d'accès dans *BR* est évidente quand nous comparons les performances des algorithmes *BR-Cost* et *BR-Basic*. Pour toutes les valeurs de k, *BR-Cost* est beaucoup moins coûteux que *BR-Basic* qui ne tient pas compte des différences de coût d'accès entre les accès directs et les accès séquentiels. De la même manière, la comparaison entre les algorithmes *BR-Basic* et *BR-First* permet d'évaluer le bénéfice de l'utilisation de la nouvelle heuristique dans *BR* qui permet de gérer le *top-k* comme un tout et ne pas se focaliser sur le meilleur candidat du top-k courant. Cette dernière comparaison montre que pour toutes



(a) Variation de k pour $C_r > C_s$ (b) Variation de k pour $C_r = C_s$

FIGURE 2.2 – SR-sources + S-sources : variation de la valeur de k

les valeurs de k l'algorithme *BR-Basic* est moins coûteux que *BR-First* qui donne priorité au meilleur candidat.

La figure 2.2(b) présente la même variation de la valeur de k mais en considérant le même coût pour les accès séquentiels et directs. Dans ce nouveau contexte, *BR-Cost* est exactement le même algorithme que *BR-Basic*. Ce dernier est le meilleur en terme de coût d'exécution mais sans être beaucoup moins coûteux que *MPro* par exemple.

Comme conclusion de ces expériences, nous remarquons que le coût total d'exécution augmente avec la valeur de k et que pour tous les algorithmes et pour toutes les valeurs considérées de k l'algorithme *BR-Cost* (ensuite *BR-Basic*) est le moins coûteux. La comparaison entre les deux variantes *BR-Basic* et *BR-First*, nous a permis de confirmer la supériorité de l'heuristique *BR*.

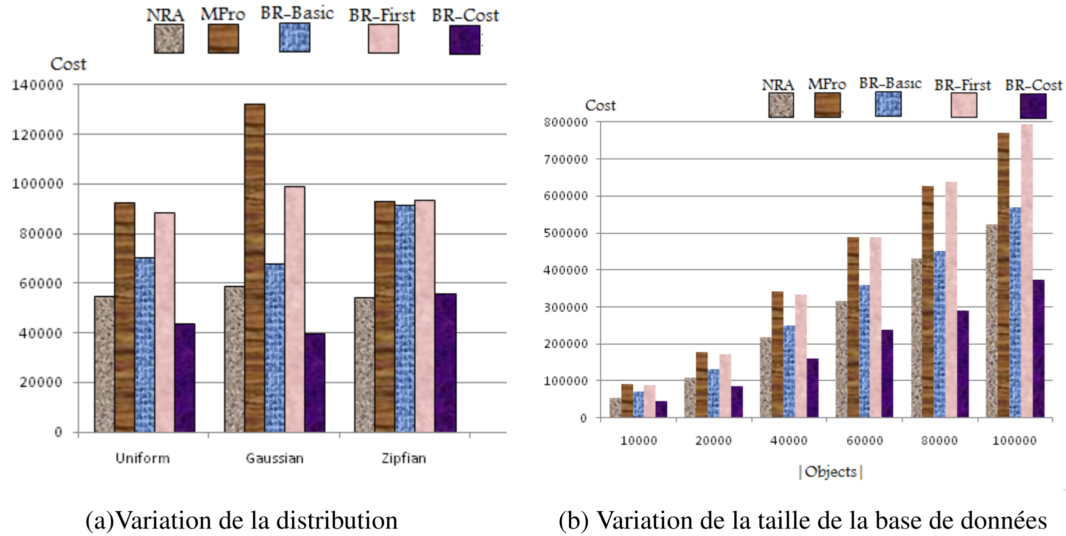
Nous évaluons maintenant les performances des algorithmes *BR*, *MPro* et *NRA* en variant la taille de la base et en considérant différentes distributions des données.

La figure 2.3(a) illustre le résultat de la variation de la distribution de données. Pour les deux distributions uniforme et gaussienne, *BR-Cost* est le meilleur algorithme en coût d'exécution, et *BR-Basic* est moins coûteux que *BR-First*. Pour la distribution qui utilise la loi de Zipf, les coûts d'exécution sont assez proches, mais *BR-Cost* reste avec une bonne performance, il a le même coût d'exécution que *NRA*. Même avec cette distribution, avec beaucoup de scores identiques qui baissent de façon discontinue et qui est défavorable à *BR*, ses algorithmes gardent de bonnes performances comparés à *MPro* et *NRA*.

La figure 2.3(b) montre le résultat de la variation de la taille des données pour une distribution uniforme et une valeur de $k = 50$. Comme dans les premières expériences, nous remarquons que le coût total d'exécution augmente en augmentant la taille des données. Mais nous remarquons toujours une très bonne performance pour l'algorithme *BR-Cost*, et une différence de plus en plus grande entre *BR-Basic* et *BR-First*.

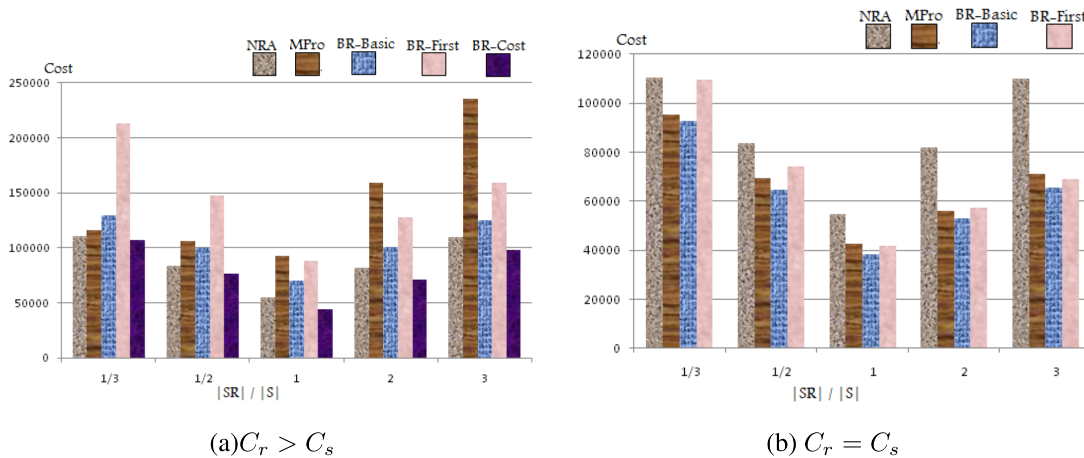
Les dernières expériences dans cette configuration (*SR-sources* + *S-sources*) permettent d'étudier l'impact de la variation du ratio entre le nombre de *S-sources* et le nombre de *SR-sources*.

La figure 2.4 présente les résultats des expériences concernant la variation du nombre de sources : (a) pour le rapport de coûts par défaut, et (b) pour le cas où $C_r = C_s$. À noter que les performances des algorithmes entre les différentes valeurs de $|SR|/|S|$ ne sont pas comparables, puisque le nombre de sources total n'est pas le même. En effet, le nombre de sources est de 3 pour le type le moins nombreux et



(a)Variation de la distribution (b) Variation de la taille de la base de données

FIGURE 2.3 – SR-sources + S-sources : variation de la distribution et de la taille des données



(a)$C_r > C_s$ (b) $C_r = C_s$

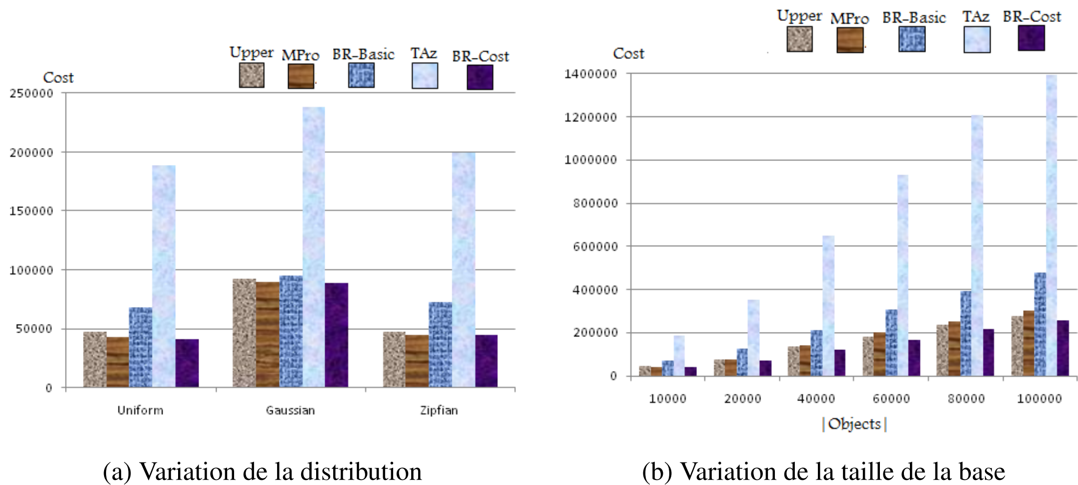
FIGURE 2.4 – SR-sources + S-sources : variation du nombre de sources en variant le ratio $|SR|/|S|$

de 3, 6 ou 9 pour l'autre. Dans la configuration par défaut à gauche de la figure, dans tous les cas, l'algorithme *BR-Cost* présente le meilleur coût d'exécution, mais il surpasse nettement les autres algorithmes quand $|SR| > |S|$. L'algorithme *BR-Basic* est toujours moins coûteux que *BR-First* , mais la différence devient importante quand $|SR| < |S|$. Dans le cas où $C_r = C_s$, l'algorithme *BR-Basic* qui est le même que *BR-Cost*, *BR-First* et *MPro* ont relativement les mêmes performances, alors que les performances de l'algorithme *NRA* se dégradent beaucoup quand $|SR| > |S|$.

SR-sources + R-sources :

Les algorithmes de référence dans ce contexte sont *Upper*, *TAz* et *MPro*. Pour la lisibilité des figures, et vu le nombre d'algorithmes testés, nous présenterons uniquement les variantes *BR-Cost* et *BR-Basic*

sachant que les mesures montrent la même conclusion pour l'algorithme *BR-First* : ses performances moins bonnes que *BR-Basic*, confirmant l'efficacité de l'heuristique *BR*. La figure 2.5 présente les résultats de la comparaison entre les algorithmes *BR*, *Upper*, *TAz* et *MPro* pour différentes distributions des données et pour des tailles différentes de la base des données. Pour la première expérience, l'algorithme *TAz* est toujours très coûteux par rapport aux autres, alors que *MPro* présente avec *BR-Cost* les meilleurs performances suivis de près par *Upper*. Comme dans les premières expériences, les coûts d'exécution de tous les algorithmes testés augmentent avec la taille de la base. *BR-Cost* est meilleur quand N augmente, et *MPro* évolue moins bien que *Upper*.



(a) Variation de la distribution

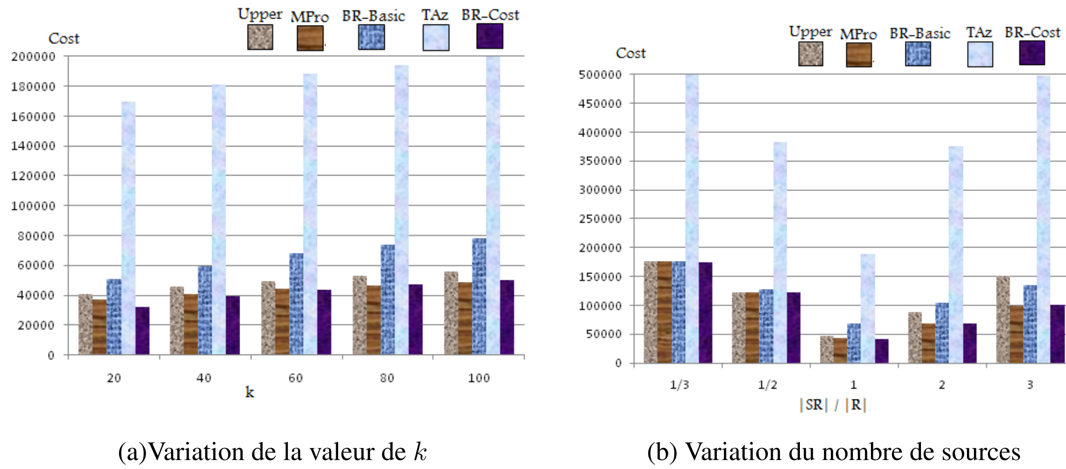
(b) Variation de la taille de la base

FIGURE 2.5 – SR-sources + R-sources : variation de la distribution et de la taille de la base de données

La figure 2.6 présente une comparaison de ces algorithmes en variant dans un premier temps la valeur de k dans l'intervalle $[20, 100]$. Nous remarquons que les coûts d'exécution augmentent aussi avec la valeur de k, et que l'algorithme *TAz* est très coûteux par rapport à tous les autres algorithmes. *BR-Cost* présente à chaque fois la meilleure performance, légèrement mieux que celle obtenue par l'algorithme *MPro*. Les remarques sont applicables aussi pour les résultats présentés sur la figure de droite qui montre les différents coûts d'exécution pour différentes valeurs de $|SR|/|R|$.

Dans cette partie, nous avons présenté les résultats des évaluations expérimentales permettant de comparer les performances des algorithmes *BR* avec celles des algorithmes *top-k* spécifiques. Dans un premier temps, nous avons considéré un cadre dans lequel les sources sont uniquement de type *S* ou *SR*, et la comparaison est faite avec les algorithmes *MPro* et *NRA*. Dans ces expériences, nous avons varié plusieurs paramètres (k, la taille de la base, la distribution des données, etc), et nous avons montré que *BR-Cost* présente à chaque fois la meilleure performance avec un coût total d'exécution inférieur aux autres algorithmes. La comparaison des variantes de *BR*, nous a permis d'abord de montrer l'efficacité de la nouvelle stratégie qui considère le *top-k* comme un tout et ne donne pas une priorité au meilleur candidat, puisque les performances de *BR-Basic* sont supérieures à celles de *BR-First*. Ensuite, il faut noter l'efficacité de la condition de coût utilisée dans *BR-Cost* qui est moins coûteux que *BR-Basic* quand $Cr > C_s$.

Dans un deuxième temps, nous avons comparé les algorithmes *BR* avec *TAz*, *MPro* et *Upper*, dans un cadre où les sources sont de type *R* ou *SR*. En effectuant les mêmes expériences, et en mesurant le coût



(a)Variation de la valeur de k (b) Variation du nombre de sources

FIGURE 2.6 – SR-sources + R-sources : variation de la valeur de k et du nombre de sources $|SR|/|R|$

d'exécution de ces algorithmes en variant plusieurs paramètres (distribution des données, taille de la base, valeur de k ainsi que le nombre de sources), nous avons montré la supériorité de *BR-Cost* aussi dans ce cadre spécifique.

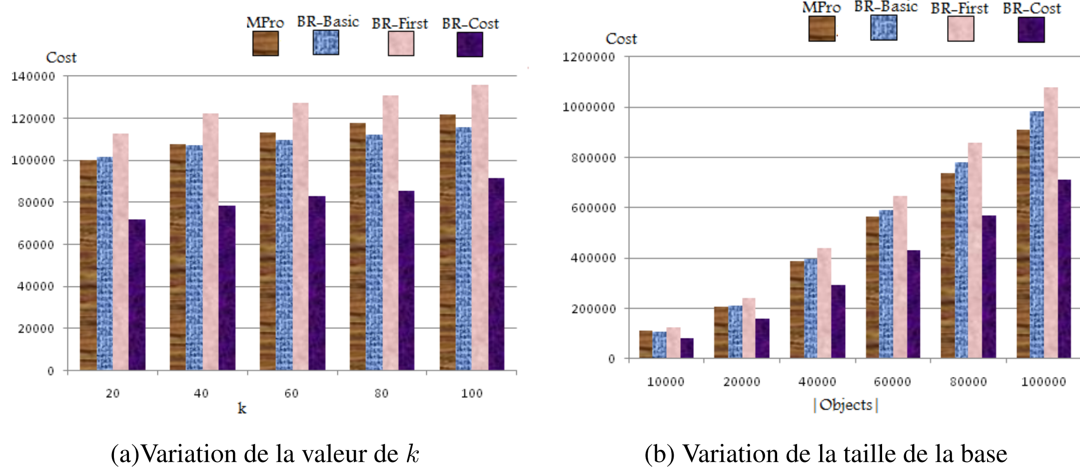
2.5.2 Évaluations des méthodes génériques

Nous considérons ici le cas où tous les types de sources existent. Dans un premier temps, nous conservons les mêmes paramètres présentés précédemment pour comparer les variantes de *BR* : *BR-Basic*, *BR-Cost* et *BR-First*. Nous rajoutons à la comparaison l'adaptation de l'algorithme *MPro* présentée auparavant. Dans ces expériences, nous considérons dans *MPro* deux cas pour les sources de type *SR*. Dans le premier cas, les *SR-sources* sont considérées comme des *S-sources*, dans le deuxième cas, elles sont considérées comme des *R-sources*. Le résultat de l'exécution de *MPro* est la moyenne des deux mesures.

La figure 2.7 présente une comparaison des algorithmes *BR* avec *MPro*. Ici, nous considérons une configuration de coût d'accès classique, c'est à dire un coût d'accès séquentiel inférieur au coût d'accès direct. Pour les deux mesures, nous retrouvons les mêmes conclusions que pour les cas spécifiques : un coût d'exécution qui augmente avec la taille de la base et/ou la valeur de k, et les performances de la stratégie *BR-Cost* permettant à chaque fois d'avoir le coût le plus bas. L'algorithme *MPro* présente une performance proche de celle obtenue par *BR-Basic*, qui reste toujours plus efficace que la stratégie *BR-First*. Les résultats dans la figure 2.7(b) montrent l'évolution des coûts d'exécution pour des tailles de données différentes, et même si le coût d'exécution de tous les algorithmes augmente avec la taille des données, l'algorithme *BR-Cost* reste toujours le moins coûteux.

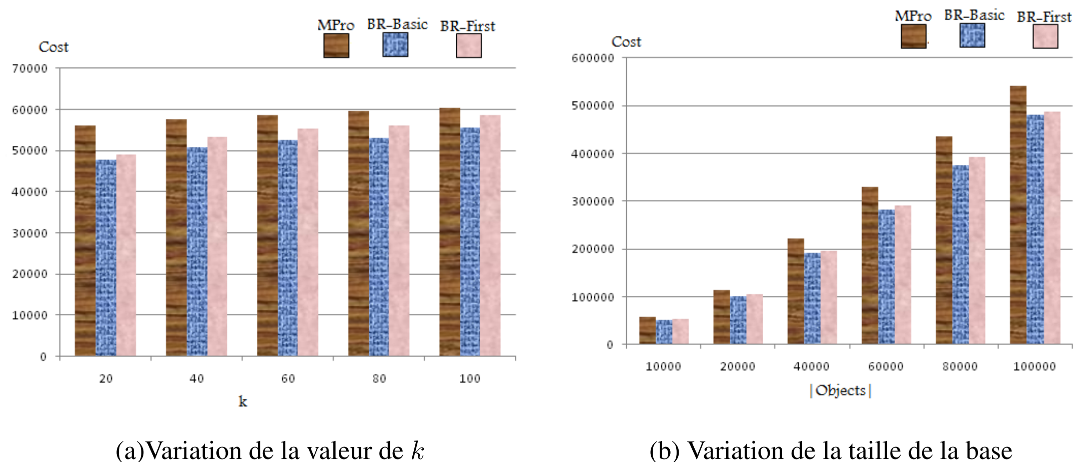
Nous présentons maintenant les mêmes expériences en considérant dans la figure 2.8 et 2.9 respectivement $C_s = C_r$ et $C_s > C_r$. La figure 2.8 présente une comparaison entre *BR-First*, *BR-Basic* et *MPro*. Dans un contexte où le coût de l'accès séquentiel est le même que l'accès direct, l'algorithme *BR-Cost* correspond exactement à *BR-Basic*. L'algorithme *BR-Basic* présente dans ce cas aussi la meilleure performance comparé à *BR-First* et *MPro*.

Dans la figure 2.9, nous considérons un rapport différent entre les coûts d'accès $C_s > C_r$, avec $C_s/C_r = 5$. Avec un accès séquentiel coûteux, nous considérons que l'algorithme *BR-Cost* correspond à



(a)Variation de la valeur de k (b) Variation de la taille de la base

FIGURE 2.7 – SR-sources + S-sources + R-sources : variation de la valeur de k et de la taille de la base ($C_s < C_r$)

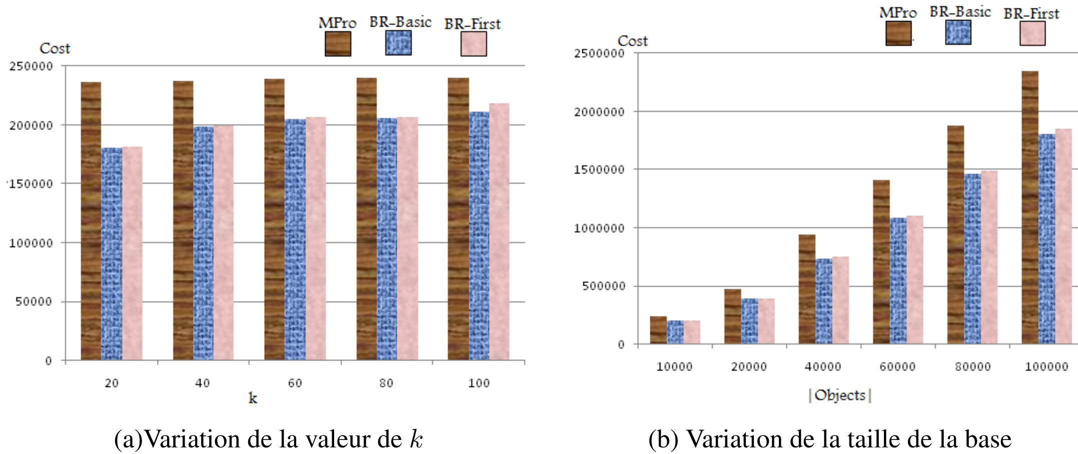


(a)Variation de la valeur de k (b) Variation de la taille de la base

FIGURE 2.8 – SR-sources + S-sources + R-sources : variation de la valeur de k et de la taille de la base ($C_s = C_r$)

BR-Basic et nous le comparons avec *MPro* et *BR-First*. Dans la partie (a) de la figure, en variant la valeur de k dans l'intervalle $[20, 100]$, *BR-First* et *BR-Basic* ont des performances très proches, avec une légère avance pour le deuxième. *MPro* reste un peu plus coûteux pour toutes les valeurs de k considérés. Pour la partie (b) de la figure, les mêmes conclusions sont obtenues pour *BR-Basic* et *BR-First*. La différence entre *MPro* et ces algorithmes s'accentue en augmentant la taille de la base, et *BR-Basic* représente un coût d'exécution inférieur de 35% du coût total de MPro dans le cas où la taille de la base est de 100.000.

Dans la suite, nous comparons la stratégie générique de *BR* avec celle des autres algorithmes à vocation générique, à travers les variantes de *NC* et *CA* que nous avons proposées. La comparaison utilise la variante *BR-Cost**, la plus proche du contexte commun de ces variantes, toutes basées sur des heuristiques utilisant les mêmes notion de bénéfice.



(a)Variation de la valeur de k (b) Variation de la taille de la base

FIGURE 2.9 – SR-sources + S-sources + R-sources : variation de la valeur de k et de la taille de la base ($C_s > C_r$)

Comparaison des stratégies génériques

Nous comparons les algorithmes *BR-Cost**, *CA-gen* et la variante de *NC* que nous avons proposé, dans un cadre expérimental un peu différent de celui utilisé précédemment. Dans ces expériences, nous utilisons des sources synthétiques, générées indépendamment. Ces sources sont représentées par des listes de scores dans l'intervalle $[0, 1]$, nous considérons les trois types de sources. Nous considérons deux types de distribution des scores dans les sources : uniforme ou exponentielle. La loi de distribution exponentielle utilisée est la suivante :

$$p(x) = \lambda e^{-\lambda x} \tag{2.15}$$

Ici, nous considérons $\lambda = 1$, et l'intervalle $[0, 1]$ pour les scores. La distribution exponentielle permet d'avoir des sources triées avec des scores qui baissent rapidement au début, ce qui est potentiellement plus discriminant que la distribution uniforme par exemple. Comme pour les expériences précédentes, nous mesurons le coût d'exécution de chaque algorithme qui est la somme des coûts d'accès à toutes les sources pour trouver le *top-k* final. Les résultats que nous présentons sont la moyenne de 10 mesures sur des sources générées aléatoirement. Concrètement, nous considérons les paramètres suivants pour ces expériences :

- Le nombre d'objets dans la base : $N = 10.000$.
- Le résultat souhaité est un *top-k* avec $k = 50$.
- Nous considérons 6 *S-sources*, 6 *SR-sources* et 6 *R-sources*.
- Nous considérons la configuration la plus utilisée pour les coûts d'accès, avec un coût d'accès séquentiel plus bas : $C_s = 1$, $C_r = 10$.
- Deux configuration sont considérées pour la distribution des scores dans les sources : uniforme ou mixte. Pour la distribution mixte, la moitié des sources triées ont une distribution exponentielle.

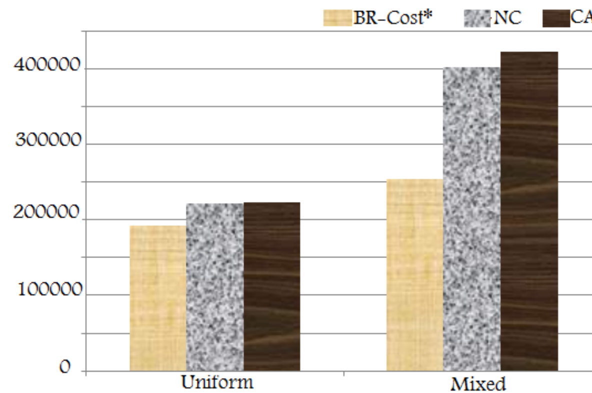
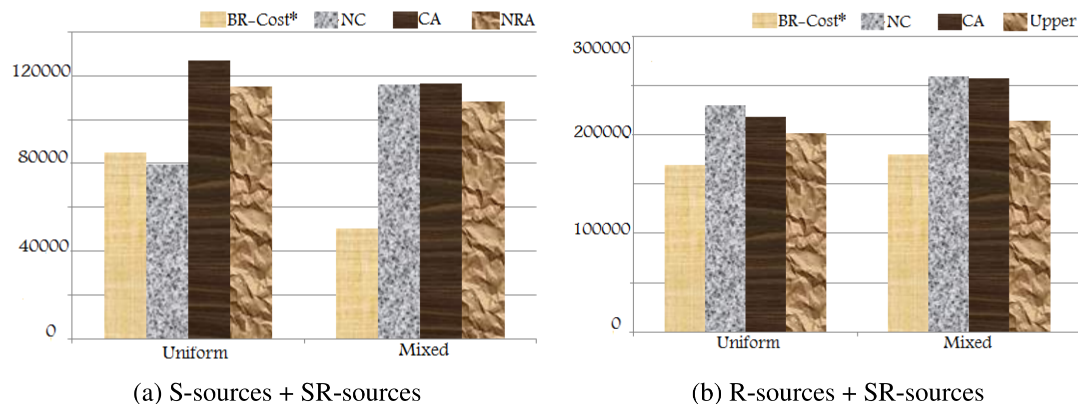


FIGURE 2.10 – Comparaison des stratégies génériques : variation de la distribution

Nous présentons maintenant la première comparaison d'algorithmes *top-k* génériques, et nous montrons l'efficacité de la stratégie *BR*. La figure 2.10 présente une comparaison des algorithmes *BR-Cost**, *CA-gen* et *NC*, respectivement pour une distribution uniforme et mixte. Dans la première expérience (distribution uniforme), *BR-Cost** est moins coûteux (10%) que *NC* et *CA*, qui ont la même performance. Pour la distribution mixte, les performances de *CA* ainsi que *NC* se dégradent clairement, et la différence de coût d'exécution avec *BR-Cost** devient importante : presque 37% de moins que *NC* et 40% par rapport à *CA-gen*. En conclusion, la stratégie *BR* est globalement plus efficace que *NC* et *CA*, et s'adapte mieux à des distributions hétérogènes des scores.



(a) S-sources + SR-sources

(b) R-sources + SR-sources

FIGURE 2.11 – Évaluation des performances des algorithmes génériques dans deux contextes spécifiques

La figure 2.11 présente une évaluation des algorithmes génériques dans des cadres spécifiques de types de sources. La figure 2.11 (a) considère uniquement des *S-sources* et des *SR-sources*. Cette situation, potentiellement favorable à l'algorithme *NRA*, dont nous comparons les performances avec les algorithmes génériques, montre que les algorithmes *BR-Cost** et *NC* s'adaptent beaucoup mieux que *CA*, très coûteux dans ce contexte. Même si *NC* présente une meilleure performance pour une distribution uniforme, il est largement dépassé par *BR-Cost** dans une distribution mixte (plus que 2 fois le coût d'exécution). Dans la figure 2.11 (b), nous remarquons que l'algorithme *BR-Cost** s'adapte très bien dans un contexte où aucune *S-source* n'est considérée. L'algorithme Upper, conçu pour ce cadre spécifique est plus per-

formant que les stratégies *NC* et *CA* (*CA-gen*). Nous signalons aussi qu'en considérant une distribution mixte des données (exponentielle + uniforme), les coûts d'exécution de tous les algorithmes augmentent, mais *BR-Cost** reste toujours le meilleur avec le coût d'exécution le plus bas, largement devant *NC* et *CA*.

2.6 Conclusion

Dans ce chapitre, nous avons commencé par proposer un nouveau cadre générique *GF* permettant d'exprimer n'importe quel algorithme de traitement des requêtes *top-k* de sélection. *GF* considère un algorithme *top-k* comme une suite d'accès aux sources permettant à chaque fois de découvrir un score local d'un seul objet. Nous nous sommes basés sur *GF* pour exprimer les algorithmes de l'état de l'art, et nous avons donné plusieurs exemples d'algorithmes *top-k* exprimés dans *GF*, tel que *NRA* qui considère uniquement des *S-sources*, TA qui ne considère que des *SR-sources*, etc. Ensuite, nous avons présenté une nouvelle technique, *Breadth-Refine*, pour le traitement des requêtes *top-k* se basant sur une nouvelle stratégie de type *breadth-first* (en largeur d'abord). Cette stratégie considère le *top-k* comme un tout, et essaye à chaque étape de maintenir le meilleur *top-k* possible en évaluant le candidat ayant l'intervalle de score le moins précis plutôt que le meilleur candidat. Nous avons présenté, dans un premier temps, trois variantes de *BR* : *(i) BR-Basic* qui considère uniquement la stratégie *BR* sans prendre en considération le paramètre de coût d'accès aux scores, *(ii) BR-First* qui considère la stratégie classique utilisée par tous les algorithmes *top-k* de l'état de l'art qui doivent choisir un candidat à raffiner, en choisissant toujours le meilleur candidat dans l'ordre des scores maximum. La comparaison de ces deux algorithmes nous a permis de comparer concrètement notre stratégie à celle utilisée habituellement. *(iii) BR-Cost* est la troisième variante proposée, elle inclut une approche qui ressemble à celle utilisée dans l'algorithme *CA* pour réduire le nombre d'accès directs, souvent très coûteux.
Vue l'absence d'algorithmes génériques comparables, nous avons proposé dans un premier temps, une variante de *SR/G*, l'algorithme de référence de *NC*, dans le contexte *BR*, où nous considérons des objets avec des scores incomplets dans le *top-k* final. Cette variante est exprimée dans le cadre *GF*. Dans un deuxième temps, nous avons adapté l'algorithme *CA* initialement conçu pour un contexte où les sources sont uniquement de type *SR-source* et proposé *CA-gen* une variante générique de CA adaptée à n'importe quelle configuration de type d'accès aux sources. Enfin, nous avons proposé *BR-Cost**, une alternative de l'algorithme de *BR-Cost*, dans laquelle nous avons adopté une nouvelle stratégie pour calculer le ratio r qui est considéré comme un rapport des bénéfices et non plus un rapport des coûts d'accès.

A la fin du chapitre, nous avons présenté une évaluation expérimentale très riche, dans laquelle nous avons considéré initialement des cadres spécifiques favorables aux algorithmes d'état de l'art. Ces expériences ont montré la capacité des algorithmes *BR* à s'adapter à différentes configurations de type et de coût d'accès aux sources. Nous avons montré que *BR-Cost* présente à chaque fois la meilleure performance dans des cas favorables aux algorithmes de l'état de l'art. Cette comparaison a montré aussi la supériorité de la stratégie *breadth-first* en comparant les algorithmes *BR-First* et *BR-Basic*, et l'impact de la prise en considération de la différence de coût d'exécution en montrant que *BR-Cost* est plus performant que *BR-Basic* quand l'accès direct est plus coûteux que l'accès séquentiel. Ensuite, nous avons considéré un cadre générique, dans lequel nous avons proposé une première comparaison des algorithmes *top-k* génériques, où l'algorithme *BR-Cost** a montré sa supériorité par rapport aux variantes de *NC* et *CA*.

Dans le chapitre suivant, nous explorons la recherche approximative *top-k*, afin de réduire le coût d'exécution, souvent élevé pour un résultat exact. Nous considérons l'approximation par arrêt prématuré ou *early stopping*, et nous comparons le potentiel d'approximation des différents algorithmes génériques ou spécifiques de l'état de l'art avec l'algorithme *BR-Cost**.

On résout certains problèmes par approximation, en négligeant de petites quantités.
– Jean le Rond d' Alembert

CHAPITRE 3

Les algorithmes *top-k* dans un contexte de recherche approximative par arrêt prématuré

Ce chapitre présente une adaptation des algorithmes *top-k* vus dans le chapitre 1 et des algorithmes *BR* présentés dans le chapitre précédent, au contexte de la recherche *top-k* approximative. La technique considérée ici est l'approximation par arrêt prématuré de l'exécution. L'avantage de ces techniques est de caler sur un fonctionnement normal en choisissant le moment de l'arrêt de l'exécution, ce qui permet si nécessaire de poursuivre jusqu'à l'obtention d'un résultat exact. Au moment de l'arrêt, le *top-k* courant sera retourné comme résultat approximatif. Nous commençons le chapitre par présenter les motivations de ce type de recherche. Ensuite, nous montrons comment adapter le cadre *GF* à une recherche approximative par arrêt prématuré de l'exécution. Nous poursuivons en proposant une évaluation du résultat approximatif retourné, et avant de conclure, nous présentons une évaluation expérimentale des variantes approximatives des algorithmes *BR* et de plusieurs algorithmes *top-k* proposés dans la littérature.

Le traitement des requêtes de classement top-k est généralement très coûteux à cause du coût de l'évaluation des prédicats de classement, combiné avec de grands volumes de données et souvent avec un nombre important de prédicats/critères (multidimensionnalité), qui ralentit la convergence des algorithmes. La réduction du coût d'exécution est l'objectif commun de la grande majorité des techniques et des algorithmes proposés dans l'état de l'art. La recherche approximative est l'une des solutions proposées pour la réduction du coût d'exécution, avec pour objectif principal d'obtenir un bon compromis entre coût et qualité d'approximation. Nous commençons ce chapitre par présenter le contexte de la recherche approximative que nous considérons, ensuite nous présentons une adaptation à la recherche approximative des algorithmes *top-k* dans le cadre général *GF*, et en fin du chapitre, nous proposons une première évaluation du potentiel d'approximation des algorithmes *top-k*.

3.1 Recherche approximative par arrêt prématuré : motivation

Comme nous l'avons indiqué dans le chapitre 1, plusieurs techniques sont proposées pour une recherche *top-k* approximative. Nous nous intéressons ici aux méthodes qui permettent, si besoin, de calculer le résultat exact. Plus précisément, nous considérons l'approximation par arrêt prématuré de l'exécution, avec retour d'un ensemble de k candidats susceptibles de former le *top-k*. Cette approche a l'avantage de la flexibilité, car elle permet d'allouer des durées d'exécution variables en fonction de la précision souhaitée, voire continuer jusqu'à la solution exacte.

Cette technique est utilisée dans l'algorithme TA_θ [Fagin et al., 2002], proposée comme une variante de *TA* permettant une recherche *top-k* approximative. La différence entre les algorithmes *TA* et TA_θ se situe au niveau de la condition d'arrêt. Cette dernière est relaxée avec la valeur $\theta > 1$, et l'algorithme s'arrête quand, au moins k candidats ont des scores supérieurs à U_{unseen}/θ. Si $\theta = 1$, nous obtenons l'algorithme *TA*. Par la baisse du seuil d'un facteur θ, TA_θ produit la même exécution que *TA*, à part l'arrêt, qui est réalisé plus tôt. Cette méthode produit une θ-approximation du *top-k* exact calculé dans *TA* : si nous considérons K_a l'ensemble des objets retournés par l'algorithme TA_θ, nous avons : $\forall x \in K_a$, $\forall y \notin K_a, \theta \times score(x) \geq score(y)$, où les scores sont considérés dans l'intervalle $[0, 1]$.

***Définition* 4.** *($\theta - approximation$) Étant donnée une requête top-k q sur une Base de Données D et une valeur $\theta > 1$, le résultat $P_{k,\theta}(q)$ d'une recherche approximative du top-k est appelé θ-approximation, si, pour tout $x \in P_{k,\theta}(q)$ et pour tout $y \notin P_{k,\theta}(q)$, nous avons $\theta \times score(x) \geq score(y)$.*

Notre objectif est de généraliser l'approche utilisée dans l'algorithme TA_θ dans un contexte de *top-k* avec des scores incomplets, et dans le cadre général de *GF* afin de pouvoir appliquer cette méthode à n'importe quel algorithme *top-k*.

3.2 Recherche approximative dans le cadre *GF*

Dans cette section, nous proposons une adaptation de la recherche approximative par arrêt prématuré au cadre général *GF*. L'objectif est de profiter de la capacité de *GF* à exprimer tous les algorithmes *top-k* pour généraliser cette technique à n'importe quel algorithme.

L'algorithme 4, présente le cadre GF dans un contexte θ-approximation. Dans les paramètres d'entrée, θ est ajouté à la requête q et à l'ensemble des sources $\mathcal{S}$. Aussi, les scores de chaque source S_j sont dans l'intervalle $[0, max_j]$ afin de pouvoir garder la même définition de la θ-approximation. Seule la condition d'arrêt change dans ce cas.

Nous avons vu dans le chapitre 2, que la condition d'arrêt permettant un résultat *top-k* exact avec des scores incomplets est donnée par la formule 2.2. Si nous considérons une solution approximative K_a composée de k candidats à un instant t de l'exécution d'un algorithme *top-k* dans le cadre *GF*, alors leurs scores sont éventuellement incomplets. Dans ce cas, la condition d'arrêt permettant de dire avec certitude que l'ensemble K_a est une θ-approximation du résultat exact, quel que soit le score exact de ces candidats, est donnée par le théorème suivant :

***Théorème* 1.** *Une solution approximative K_a composée de k candidats à un moment de l'exécution d'un algorithme top-k, ayant des scores éventuellement incomplets, est sûrement une θ-approximation du top-k exact, **ssi** :*

Algorithm 4 Recherche *top-k* approximative (θ-approximation) dans le cadre général *GF*

Require: q, $\mathcal{S}$ and θ
  //Initialisations
  candidates $\leftarrow \emptyset$
  $U_{unseen} \leftarrow \mathcal{F}(max_1, ..., max_m)$
  ... *//Autres variables locales*
  repeat
    //Choix du type d'accès : séquentiel ou direct
    if *SortedAccessCondition()* **then**
      //Accès séquentiel
      $S_j \leftarrow$ **BestSortedSource()** *//Choisir une source triée*
      $(o, s) \leftarrow$ getNext(S_j) *//Accès séquentiel dans la source choisie*
      Mise à jour de *candidates*, U_{unseen} et des variables locales
    else
      //Accès direct
      $c \leftarrow$ **ChooseCandidate()** *//Choisir un candidat*
      $S_j \leftarrow$ **BestRandomSource(**c**)** *//Choisir une source non triée*
      $s \leftarrow$ getScore(S_j, c) *//Accès direct dans la source choisie*
      Mise à jour de *candidates* et des variables locales
    end if
  until θ-*StopCondition()*
  return *candidates*

$$\theta \times min_{c \in K_a}(L(c)) \geq max_{c \notin K_a}(U(c)) \tag{3.1}$$

Démonstration. Ici, $min_{c \in K_a}(L(c))$ est le score minimum possible d'un candidat dans K_a, et $max_{c \notin K_a}(U(c))$ est le score maximum possible d'un objet n'appartenant pas à K_a.

Nous montrons tout d'abord que si la condition 3.1 est vraie, alors K_a est une θ-approximation de K.

Pour tout candidat c, nous avons, $L(c) \leq score(c) \leq U(c)$. Alors : $\forall x \in K_a, score(x) \geq L(x) \geq min_{c \in K_a}(L(c))$, et $\forall y \notin K_a, max_{c \notin K_a}(U(c)) \geq U(y) \geq score(y)$. Si la formule 3.1 est vraie, alors $\forall x \in K_a$ et $y \notin K_a,$: $\theta \times score(x) \geq score(y)$, donc K_a est une θ-approximation du résultat exact K.

Nous montrons l'implication inverse par réduction à l'absurde.

Nous montrons que si la condition 3.1 n'est pas vraie, alors K_a n'est pas sûrement une θ-approximation de K, en montrant que la condition de θ-approximation peut ne pas être satisfaite dans ce cas.

Nous considérons maintenant, deux candidats x et y, respectivement le $k^{ème}$ candidat dans l'ensemble K_a dans l'ordre décroissant des scores minimums, et le candidat ayant le meilleur score maximum en dehors de K_a : $x = argmin_{c \in K_a}(L(c))$ et $y = argmax_{c \notin K_a}(U(c))$. Si la condition 3.1 n'est pas respectée dans K_a, alors $\theta \times L(x) < U(y)$ est vraie. Par conséquent, il est possible d'avoir $\theta \times score(x) < score(y)$, dans ce cas, la solution calculée dans K_a n'est pas une θ-approximation du résultat exact K. □

Adaptée au contexte *GF*, où les algorithmes n'ont pas une connaissance a priori sur les données,

mais découvrent les objets uniquement suite aux accès séquentiels effectués dans les sources triées, et en considérant que l'ensemble K_a n'est composé que d'objets déjà découverts ($K_a \subset candidates$), la condition d'arrêt devient :

$$\theta - StopCondition \equiv (\theta \times min_{c \in K_a}(L(c)) \geq max(U_{unseen}, max_{c \in candidates - K_a}(U(c)))) \quad (3.2)$$

La différence avec la condition 3.1 est que ici U_{unseen} donne le score maximum des objets qui ne sont pas encore découverts (donc ne sont pas membres de *candidates*). Dans le cadre *GF*, les candidats qui ne peuvent plus intégrer le top-k, et que nous avons appelé *non-viables* dans le chapitre 2, sont automatiquement éliminés. Dans le nouveau cadre, avec θ-approximation, nous montrons que cette opération ne nécessite pas une modification de la condition d'arrêt.

Théorème 2. *L'élimination d'un candidat non-viable dans GF n'affecte pas la condition d'arrêt donnée par la condition 3.2.*

Démonstration. Supposons que, à un instant t de l'exécution d'un algorithme dans le cadre *GF*, un candidat *non-viable* x affecte la condition d'arrêt. Puisque $x \notin K_a$, alors le candidat x ne peut influencer que la partie droite de l'inégalité 3.2, et seulement si *(i)* $U(x) > U_{unseen}$ et *(ii)* $U(x) > U(y), \forall y \in candidates - K_a$. Mais nous savons déjà que *(iii)* $U(x) < L_k$, puisque x est un candidat *non-viable*. Par conséquent *(ii)* et *(iii)* permettent d'affirmer que tous les candidats dans $(candidates - K_a)$ sont aussi *non-viables*, et *(i)* et *(iii)* que $L_k > U_{unseen}$. Cela correspond à la condition d'arrêt originale donnée par 2.2 et donc au moment t l'exécution est déjà arrêtée car le *top-k* final est déjà trouvé (K_a). Par conséquent, un candidat *non viable* ne peut pas influencer la condition d'arrêt pendant l'exécution.

□

Dans cette section, nous avons montré qu'en plus des techniques de traitement des requêtes *top-k* de sélection, les algorithmes qui proposent une θ-approximation de la solution exacte sont aussi exprimables dans le cadre *GF* en respectant la même approche. Nous avons en plus donné la condition d'arrêt qui permet d'obtenir une θ-approximation dans *GF*.

Dans la section suivante, nous proposons une nouvelle mesure pour la qualité de la solution approximative et nous montrons la relation entre cette mesure et les θ-approximations.

3.3 Évaluation de la qualité du résultat approximatif

Dans le contexte de *GF*, pour estimer la qualité d'une solution approximative retournée par un algorithme *top-k*, nous proposons une mesure de distance basée sur deux principes :

- Dans le top-k, l'ordre et les scores finaux des candidats ne sont pas importants.
- Parmi les k candidats retournés dans la solution approximative, uniquement les faux-positifs influencent et affectent la qualité. En d'autres termes, nous considérons que la qualité du résultat est donnée par la qualité des faux-positifs.

La distance entre une solution approximative K_a et la solution exacte K est mesurée par la différence entre les vrais scores des faux-positif et la valeur de R_k, le score du $k^{\text{ème}}$ élément dans K. Cette distance exprime ce qui manque aux faux-positifs pour entrer dans le *top-k*. Elle est normalisée pour être dans l'intervalle $[0, 1]$ en la divisant par R_k la distance maximale possible, puisque 0 est la valeur minimale du score global d'un candidat. La distance entre un élément $o \in K_a$ et le *top-k* exact K est définie de la manière suivante :

$$\text{dist}(o, K) = \begin{cases} (R_k - score(o))/R_k, & \text{if } o \notin K \\ 0, & \text{if } o \in K \end{cases} \tag{3.3}$$

Nous définissions la distance globale entre une solution *top-k* approximative K_a et un résultat exact K comme la moyenne des distances individuelles entre les faux-positifs dans K_a et K :

$$\text{dist}(K_a, K) = \frac{1}{k} \sum_{o \in K_a} \text{dist}(o, K) \tag{3.4}$$

La qualité d'un résultat approximatif est mesurée par la fonction quality, que nous définissons simplement :

$$quality(K_a) = 1 - dist(K_a, K) \tag{3.5}$$

Dans le cadre GF, nous considérons un cas particulier d'approximation que nous avons présenté précédemment : la θ-approximation. La relation entre la distance que nous proposons et la θ-approximation est donnée par le théorème suivant :

***Théorème* 3.** *Si K_a est une θ-approximation du résultat exact K, alors :*

$$dist(K_a, K) \leq \theta - 1 \tag{3.6}$$

Démonstration. Si nous considérons $K_a = K$, alors $dist(K_a, K) = 0$ et l'inégalité est valide. Sinon,pour tout candidat x tel que $x \in K - K_a$, alors $score(x) \geq R_k$. Si K_a est une θ-approximation de K, alors $\forall o \in K_a, \theta \times score(o) \geq score(x) \geq R_k$, et par conséquent $R_k - score(o) \leq (\theta - 1)score(o)$.

En conclusion :

$dist(K_a, K) = \frac{1}{k} \sum_{o \in K_a} dist(o, K) = \frac{1}{k} \sum_{o \in K_a - K} \frac{R_k - score(o)}{R_k} \leq \frac{1}{k} \sum_{o \in K_a - K} \frac{(\theta-1)score(o)}{R_k} = \frac{\theta-1}{kR_k} \sum_{o \in K_a - K} score(o) \leq \frac{\theta-1}{kR_k} kR_k = \theta - 1$ □

Nous pouvons noter que l'on ne peut pas garantir une distance plus petite que $\theta - 1$. En effet, si K est composé de k objets de même score s et K_a de k objets de même score s/θ, alors K_a est une θ-approximation de K à distance exactement $\theta - 1$.

Nous proposons dans la suite une étude comparative du potentiel d'approximation des algorithmes *top-k*. Le potentiel d'approximation d'un algorithme *top-k* est sa capacité, à un instant t, de retourner un résultat approximatif de bonne qualité. Il est mesuré en calculant la qualité du résultat K_a retourné à l'instant t, et qui correspond au *top-k* courant maintenu dans la liste des candidats. Nous étudions le comportement des algorithmes tout au long de l'exécution, en mesurant la distance entre la solution exacte et le *top-k* courant qui correspondrait à la solution K_a en cas d'arrêt prématuré de l'exécution.

3.4 Évaluation expérimentale

Pour comparer le potentiel d'approximation des algorithmes top-k, nous traçons les courbes *coût-distance* pour toutes ces techniques, en considérant les différentes configurations de types de sources. Nous étudions ensuite le comportement des algorithmes en comparant les formes des courbes obtenues. Un point sur la courbe *coût-distance* indique la qualité du résultat approximatif à cet instant (coût). L'approximation par arrêt prématuré ne donne pas de garantie pour la qualité du résultat approximatif. Nous mesurons aussi le coût de la θ-approximation (qui garantit la précision) pour tous les algorithmes *top-k*, et nous comparons le coût d'exécution et la précision (qualité) mesurée.

3.4.1 Choix des paramètres

Dans ces expériences, nous utilisons les paramètres que nous avons présenté dans le chapitre précédent 2.5.2. Nous les résumons ici en six points :

- Le nombre d'objets dans la base de données est : $N = 10.000$.
- $k = 50$.
- Les sources sont divisées en : 6 S-sources, 6 R-sources et 6 SR-sources.
- La configuration du coût d'accès considère l'accès direct plus cher que l'accès trié : $C_r = 10$, $C_s = 1$.
- Deux configurations sont considérées pour la distribution des scores dans les sources : uniforme ou mixte. Pour la distribution mixte, nous considérons la moitié des sources triées avec une distribution exponentielle des scores, et le reste avec une distribution uniforme.
- Le nombre d'exécution est 10. Le résultat est la moyenne des 10 exécutions.

3.4.2 Potentiel d'approximation

Dans toutes les expériences, nous mesurons le potentiel d'approximation des algorithmes *top-k*, par arrêt prématuré en traçant des courbes *coût-distance*, où la distance entre la solution approximative et le résultat exact est donnée par les formules 3.3 et 3.4. Nous mesurons cette distance dans plusieurs points durant l'exécution des algorithmes (chaque 2000 ou 1000 unités de coût d'exécution). Chaque point de la courbe représente la distance entre la solution approximative et le top-k final, si l'algorithme s'arrêtait à cet instant. L'algorithme qui possède la courbe la plus basse présente le meilleur potentiel d'approximation. La forme de la courbe indique aussi la *stabilité de l'approximation*, plus précisément, une courbe avec une descente monotone présente une approximation stable, qui s'améliore durant l'exécution. Inversement, une courbe avec une variation non-monotone montre que l'algorithme n'est pas adapté à l'approximation par arrêt prématuré.

Pour chaque courbe coût-distance, nous mesurons aussi la θ-approximation obtenue, en utilisant la formule 3.2. Nous considérons deux valeurs d'approximation : $\theta = 1.05$ et $\theta = 1.01$, ces valeurs garantissent des distances respectives de 0.05 et 0.01, c'est à dire une précision respective de 95%et 99%. Nous comparons dans un premier temps, les positions de ces points pour tous les algorithmes pour identifier les meilleurs algorithmes en termes de coût d'exécution pour les $(1.05/1.01)$-approximations.

Dans un deuxième temps, nous comparons les positions de ces points correspondant aux θ-approximations avec celles de l'intersection entre les courbes et les distances correspondantes. Plus précisément, nous comparons le point pour une θ-approximation avec l'intersection de la courbe avec l'horizontale de distance $\theta - 1$. Cette comparaison permet de mesurer la différence entre le coût d'approximation avec une garantie de précision (qualité) et le coût minimal potentiel pour une solution approximative avec la même qualité.

Types d'approximation

Nous considérons deux cas pour la solution approximative. Le premier est l'ensemble des meilleurs k candidats triés par ordre décroissant des scores maximums : $\mathcal{U}_k$. Ce choix pour la solution approximative peut être considéré comme naturel, vue que $\mathcal{U}_k$ est l'ensemble sur lequel un algorithme *top-k* se base durant l'exécution. Plus précisément, tous les algorithmes *top-k* proposés jusqu'ici fondent leurs stratégies sur $\mathcal{U}_k$, par exemple, pour choisir un type d'accès ou pour sélectionner les candidats pour effectuer des accès directs. Intuitivement, les candidats avec des scores maximums élevés, doivent êtres évalués pour potentiellement les éliminer si leurs scores baissent, ou s'assurer qu'ils appartiennent vraiment au *top-k* final. Un algorithme top-k s'arrête quand il s'assure que les candidats du *top-k* courant présentent des scores minimaux supérieures aux scores maximums des autres candidats. Par exemple, l'algorithme *NC* se base sur cette analyse pour définir les choix nécessaires, et considère $\mathcal{U}_k$ comme l'ensemble des candidats intéressants.
Le deuxième cas pour la solution approximative est l'ensemble des k meilleurs candidats triés par ordre décroissant des scores minimums : $\mathcal{L}_k$. Un candidat appartenant à l'ensemble $\mathcal{L}_k$, est un élément qui a été évalué dans une ou plusieurs sources avec de bons scores. Ceci peut être une bonne indication du fait que le candidat peut appartenir au *top-k* final, probablement meilleure que l'appartenance à $\mathcal{U}_k$ où les scores maximums élevés peuvent être le résultat d'une faible évaluation des candidats, donc avec plus d'incertitude sur leur qualité.

3.4.3 Résultats expérimentaux

Nous présentons tout d'abord la comparaison expérimentale des algorithmes *top-k* dans le cas d'une approximation avec $\mathcal{U}_k$, ensuite avec $\mathcal{L}_k$. Nous considérons comme dans les expériences précédentes, trois configurations de types de sources : tous les types, pas de *R-sources*, pas de *S-sources*. Les algorithmes comparés sont les algorithmes génériques, *BR-Cost**, *CA-gen* et la variante de *NC*, dans tous les cas, à côté d'autres algorithmes spécifiques dans les configurations qui leur correspondent.

Approximation avec $\mathcal{U}_k$

S-sources + R-sources + SR-sources :

La figure 3.1 présente les courbes coût-distance dans un contexte générique où tous les types des sources sont considérés (6 sources de chaque type). Les coûts finaux des algorithmes sont les mêmes que dans les expériences réalisées dans le chapitre précédent(2.5.2). Dans cette figure, nous considérons une distribution uniforme des scores dans les sources. L'algorithme *BR-Cost** présente une distance d'approximation qui baisse rapidement et de façon monotone, il possède clairement un meilleur potentiel d'approximation que l'algorithme *CA-gen* qui garde une grande distance entre le *top-k* courant maintenu

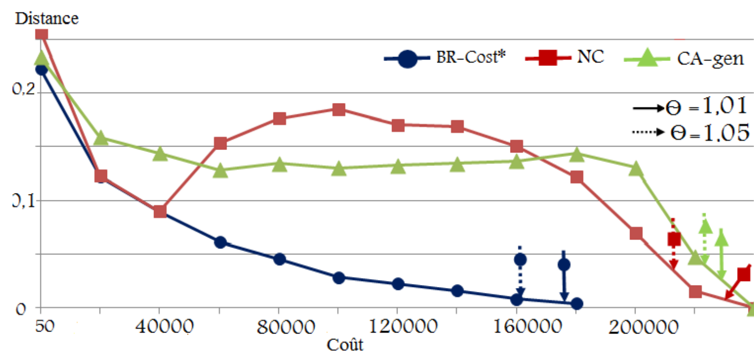


FIGURE 3.1 – Approximation avec $\mathcal{U}_k$, cadre générique : distribution uniforme

et le résultat exact, cette distance ne baisse qu'à la fin de l'exécution. *BR-Cost** est aussi beaucoup plus performant que *NC*, qui est très instable principalement dans la première partie de la courbe.

La figure 3.2 présente les mêmes expériences, dans le même cadre générique, mais avec une distribution mixte des scores dans les sources. Cette distribution accentue les problèmes des algorithmes *NC* et *CA-gen*, qui devient lui même instable, alors que *BR-Cost** maintient un bon comportement et une distance qui baisse rapidement.

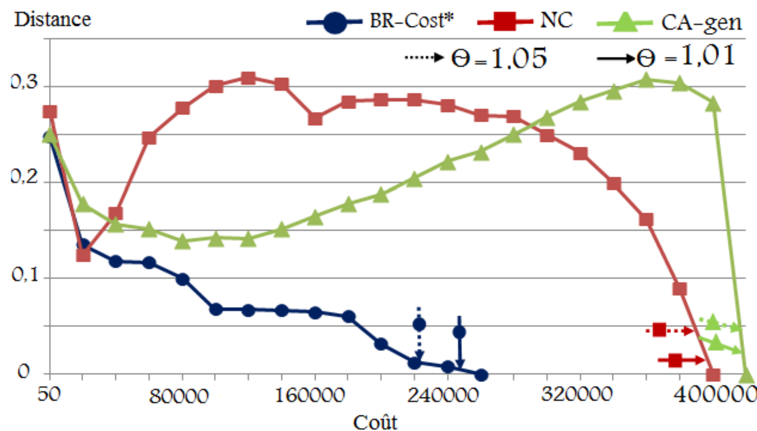


FIGURE 3.2 – Approximation avec $\mathcal{U}_k$, cadre générique : distribution mixte

Pour la θ-approximation, le point où la θ-approximation est garantie est indiqué par une flèche (en pointillé pour $\theta = 1.05$, et en continu pour $\theta = 1.01$). Le gain en coût d'exécution de l'algorithme *BR-Cost** est significativement réduit. Par exemple, pour la distribution uniforme (figure 3.1), pour $\theta = 1.05$, l'algorithme s'arrête à un coût de 160.000 unités, alors que la distance correspondante (0.05) est atteinte à un coût de 70.000 unités. La réduction du gain pour *NC* et *CA-gen* avec une θ-approximation n'est pas significative car leur gain potentiel est généralement faible, ceci est dû aux mauvaises formes de leurs courbes ce qui donnera des coûts élevés pour une distance de 0.05.

S-sources + SR-sources :

Nous considérons dans ces expériences l'algorithme NRA en plus des autres algorithmes génériques.

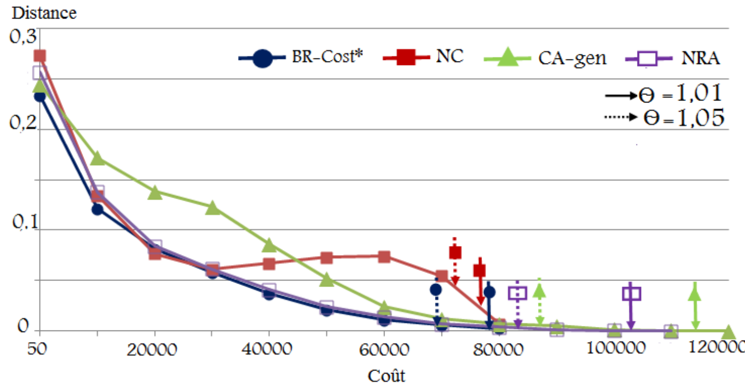


FIGURE 3.3 – Approximation avec $\mathcal{U}_k$, 6 S-sources + 6 SR-sources : distribution uniforme

La figure 3.3 présente les courbes coût-distance dans un contexte où toutes les sources sont triées, et pour une distribution uniforme. A l'exception de NC, tous les algorithmes testés produisent une approximation stable. Les algorithmes *BR-Cost** et *NRA* ont des courbes similaires, et donc le même potentiel d'approximation, même si *BR-Cost** produit des θ-approximations avec des meilleurs coûts (69.000 contre 85.000 unités pour *NRA*). L'algorithme *CA-gen* a aussi un bon potentiel d'approximation surtout dans la deuxième moitié de la courbe, mais présente une θ-approximation plus coûteuse que *NRA*. Dans ce cadre d'exécution, *NC* est plus stable que dans le cas précédent, et son coût d'exécution bas lui permet d'avoir une bonne θ-approximation.

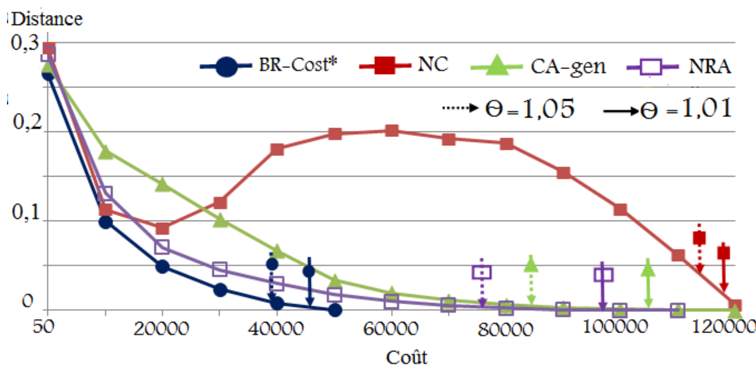


FIGURE 3.4 – Approximation avec $\mathcal{U}_k$: 6 S-sources + 6 SR-sources : distribution mixte

La figure 3.4 présente les courbes coût-distance dans la même configuration de types d'accès aux sources, mais avec une distribution mixte des scores dans les sources. Les résultats de ces expériences montrent que l'algorithme *NC* devient totalement instable, alors que tous les autres produisent une approximation stable. L'algorithme *BR-Cost** améliore ici son potentiel d'approximation par rapport à *NRA*, par exemple, pour $\theta = 1.05$, le premier algorithme présente un coût de 40.000 unités contre environ 77.000 unités pour le deuxième.

R-sources + SR-sources :

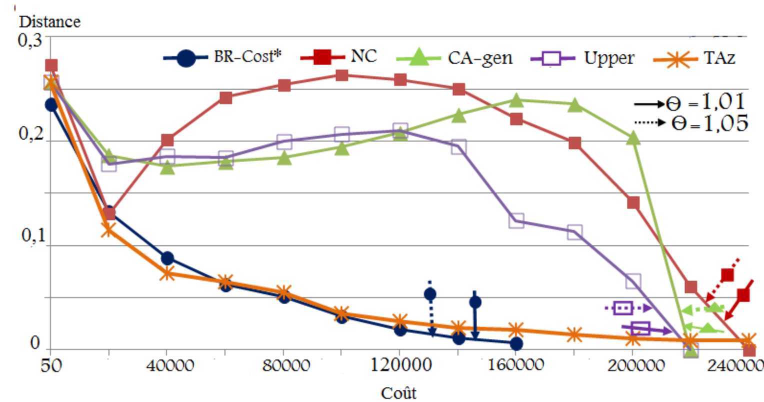


FIGURE 3.5 – Approximation avec $\mathcal{U}_k$, 6 R-sources + 6 SR-sources : distribution uniforme

En plus des algorithmes génériques *(BR-Cost**, *NC*, *CA-gen*), nous évaluons ici *Upper* et surtout l'algorithme *TAz* qui est très coûteux (environ 6 fois le coût de *BR-Cost**), mais avec une très bon potentiel d'approximation. Pour les deux distributions de scores, uniforme (figure 3.5) et mixte (figure 3.6), les comportements des algorithmes sont très similaires. *CA-gen*, *NC* et *Upper* sont très instables, alors que *BR-Cost** et *TAz* ont des courbes très similaires avec un bon potentiel d'approximation. Cependant, avec un coût d'exécution très élevé, la θ-approximation de *TAz* est beaucoup plus coûteuse que celle de *BR-Cost**, elle n'est pas visible sur la figure car l'exécution s'arrête très tard.

En conclusion, *BR-Cost** a clairement le meilleur potentiel d'approximation avec $\mathcal{U}_k$ parmi tous les algorithmes testés, et dans les différentes configurations de types de sources et de distribution des scores. Les autres algorithmes génériques ne sont pas adaptés à une approximation par arrêt prématuré avec $\mathcal{U}_k$, d'abord *NC* est systématiquement instable, alors que *CA-gen* présente un bon comportement uniquement quand les sources interrogées sont des *S-sources* et *SR-sources*. Parmi les algorithmes non-génériques, *NRA* et *TAz* ont un bon potentiel d'approximation, mais leurs coûts d'exécution élevés mènent à un coût élevé pour la θ-approximation.

Le prix à payer pour garantir la bonne qualité (précision) de la θ-approximation est important pour les algorithmes qui présentent des bonnes courbes d'approximation. Une différence significative est à signaler entre le coût possible et le coût potentiel pour le même degré de précision. Le coût de la θ-approximation dépend visiblement du coût total d'exécution de l'algorithme, et pour des algorithmes avec des potentiels d'approximation similaires, un coût total d'exécution plus élevé implique systématiquement une θ-approximation plus coûteuse.

Nous pensons que les bons résultats d'approximation que présente l'algorithme *BR-Cost** sont obtenues grâce à la stratégie *breadth-first* utilisée. Considérer le *top-k* courant ($\mathcal{U}_k$) comme un tout, au lieu de se focaliser sur le meilleur candidat seulement, permet de produire une évolution plus stable de $\mathcal{U}_k$ vers le résultat final exact.

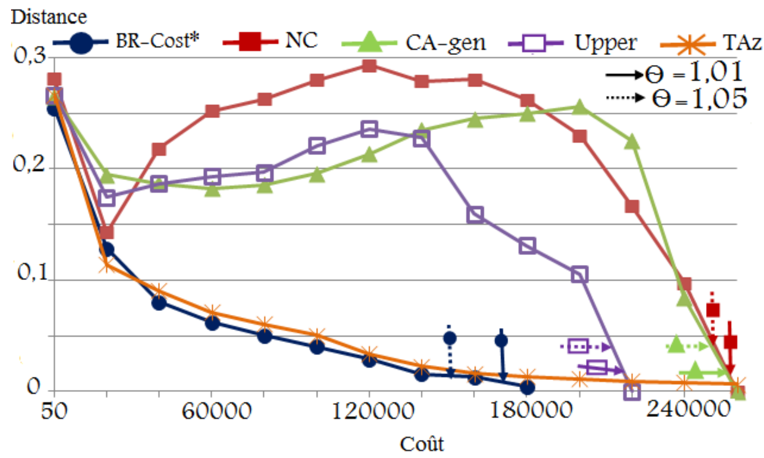


FIGURE 3.6 – Approximation avec $\mathcal{U}_k$: 6 R-sources + 6 SR-sources : distribution mixte

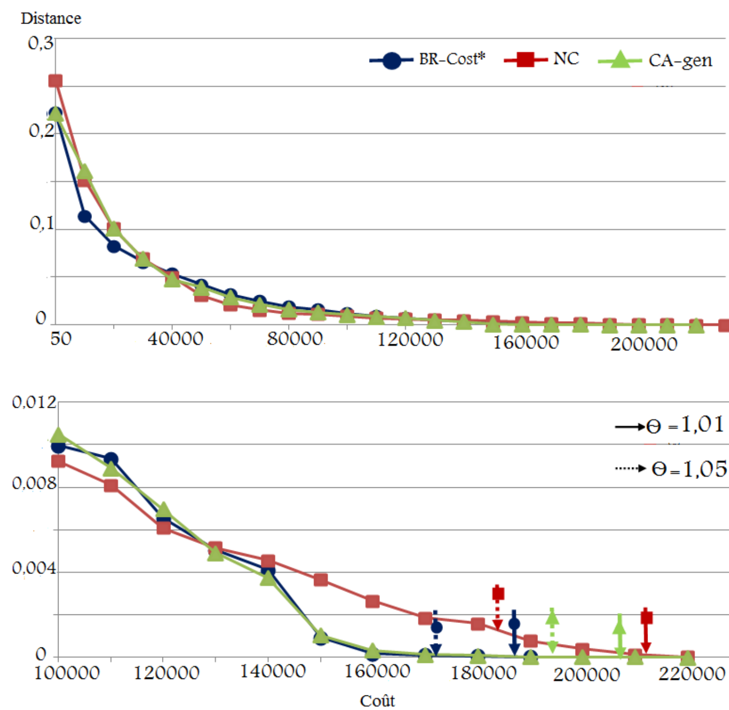


FIGURE 3.7 – Approximation avec $\mathcal{L}_k$, 6 S-sources + 6 R-sources + 6 SR-sources : distribution uniforme

Approximation avec $\mathcal{L}_k$

Maintenant, nous présentons les résultats des mêmes expériences mais en considérant $\mathcal{L}_k$ comme approximation du *top-k* courant. Pour chaque figure, nous présentons d'abord les courbes entières de tous les algorithmes, ensuite, nous présentons en bas de la figure un agrandissement de la deuxième moitié de la courbe pour mieux comparer les comportements des algorithmes.

S-Sources + R-sources + SR-sources :

La figure 3.7 présente les courbes coût-dsitance des algorithmes génériques, dans un contexte où les trois types de sources sont considérés, avec une distribution uniforme des scores. Dans ce contexte, les courbes de tous les algorithmes sont très similaires, stables et avec des bons potentiels d'approximation. Les algorithmes *BR-Cost** et *CA-gen* ont des courbes légèrement meilleures que celle de *NC* dans la première partie de la courbe, mais la différence est plus importante dans la deuxième partie. *BR-Cost** présente ici la meilleure θ-approximation, suivi de *NC*.

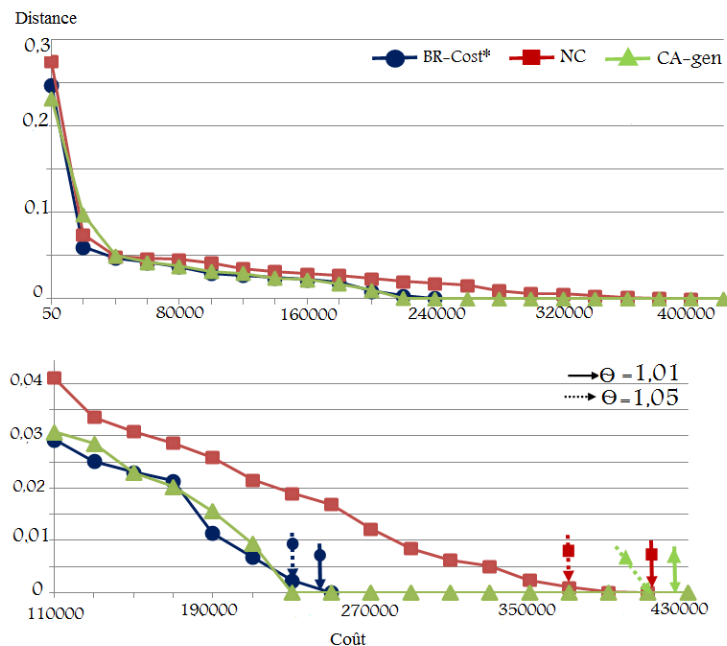


FIGURE 3.8 – Approximation avec $\mathcal{L}_k$, 6 S-sources + 6 R-sources + 6 SR-sources : distribution mixte

Les résultats de la même expérience, mais pour une distribution mixte des scores dans les sources, sont présentés dans la figure 3.8. Ces expériences montrent que l'algorithme *NC* s'adapte moins bien à une distribution mixte, et que *BR-Cost** et *CA-gen* présentent les meilleurs potentiels d'approximation dans ce contexte. La θ-approximation respecte le comportement signalé précédemment : les algorithmes avec les meilleurs coûts d'exécution, produisent une meilleure θ-approximation. Dans ce cas, *BR-Cost** produit la meilleure θ-approximation, alors que *CA-gen* et *NC* ont des θ-approximations similaires.

Nous remarquons que les courbes coût-distance avec $\mathcal{L}_k$ sont meilleures que celles avec $\mathcal{U}_k$. Par conséquent la différence entre le coût produit pour une θ-approximation et le coût d'approximation potentiel pour la même qualité est plus élevée avec $\mathcal{L}_k$.

S-Sources + SR-sources :

Les résultats des expériences sur un ensemble de sources composé de *S-sources* et de *SR-sources* sont représentés dans les figures 3.9 et 3.10, respectivement pour une distribution des scores uniforme et mixte. Dans la première figure (distribution uniforme), tous les algorithmes ont des bons potentiels d'approxi-

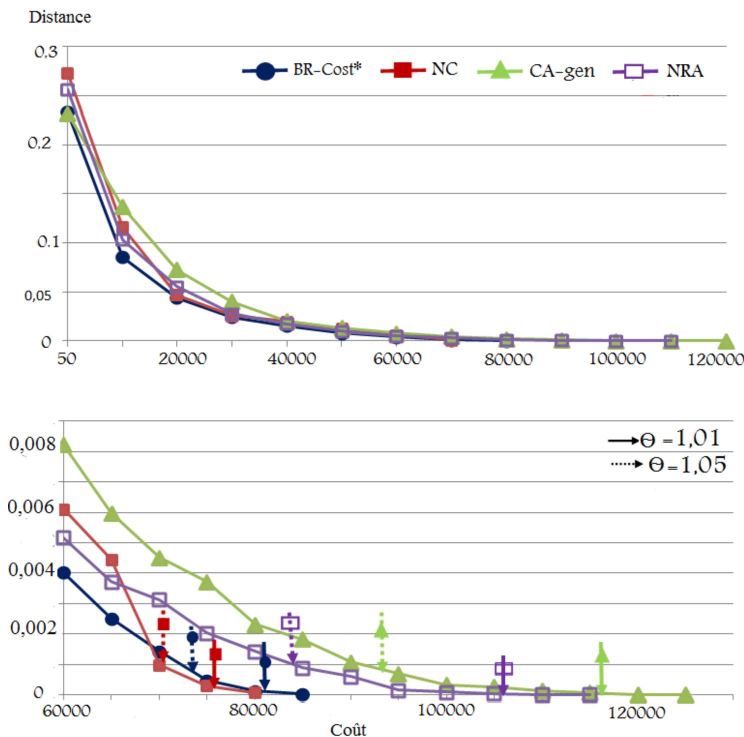


FIGURE 3.9 – Approximation avec $\mathcal{L}_k$, 6 S-sources + 6 SR-sources : distribution uniforme

mation et présentent des courbes avec des baisses monotones. Les algorithmes *NC* et *BR-Cost** proposent les meilleures θ-approximations devant *NRA* et *CA-gen*, qui présente les plus mauvaises courbes et θ-approximation.

Dans la figure 3.10, pour une distribution des scores mixte, l'algorithme *BR-Cost** est clairement supérieur aux autres algorithmes. Aidé par un coût d'exécution très bas, *BR-Cost** présente la meilleure courbe, ainsi que la meilleure θ-approximation. Les algorithmes *CA-gen* et *NRA* présentent des courbes similaires, avec un léger avantage à *NRA* dans la première partie de la courbe, et de bons potentiels d'approximation. De la même manière que les expériences précédentes, la performance de l'algorithme *NC* se dégrade pour une distribution mixte des scores.

R-Sources + SR-sources :

Dans ces expériences, nous testons les performances des algorithmes *Upper* et *TAz*, en plus des autres algorithmes génériques. La figure 3.11 présente les résultats de ces expériences pour une distribution de score uniforme. Les algorithmes génériques ont globalement de meilleures courbes que *Upper* et *TAz*. *BR-Cost**, *CA-gen* et *NC* ont des courbes similaires, mais *BR-Cost** présente une meilleure θ-approximation. Le coût élevé de l'exécution pour l'algorithme *TAz* est la cause de sa mois bonne courbe d'approximation sur la partie finale, alors que *Upper* présente le potentiel d'approximation le moins bon.

La figure 3.12 présente les mêmes expériences pour une distribution mixte des scores. Dans ce cadre,

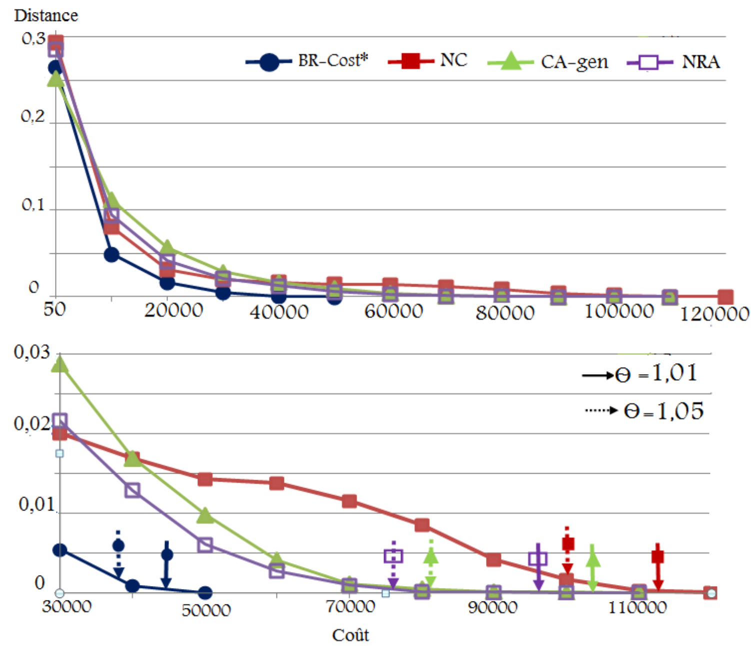


FIGURE 3.10 – Approximation avec $\mathcal{L}_k$, 6 S-sources + 6 SR-sources : distribution mixte

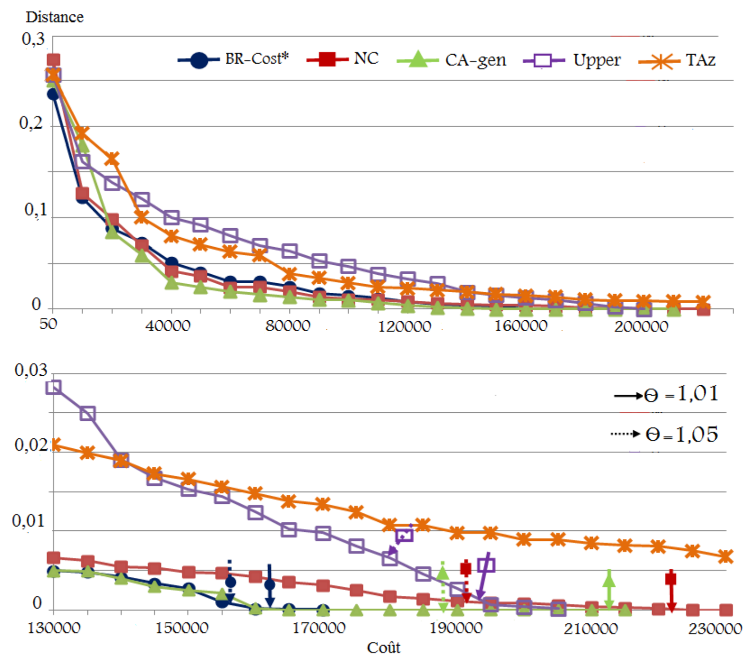


FIGURE 3.11 – Approximation avec $\mathcal{L}_k$, 6 R-sources + 6 SR-sources : distribution uniforme

les algorithmes *CA-gen* et *BR-Cost** ont les meilleurs potentiels d'approximation suivis par *NC*, tandis que *Upper* est globalement mieux que *TAz*. Comme dans les expériences précédentes, *BR-Cost** présente

la meilleure θ-approximation.

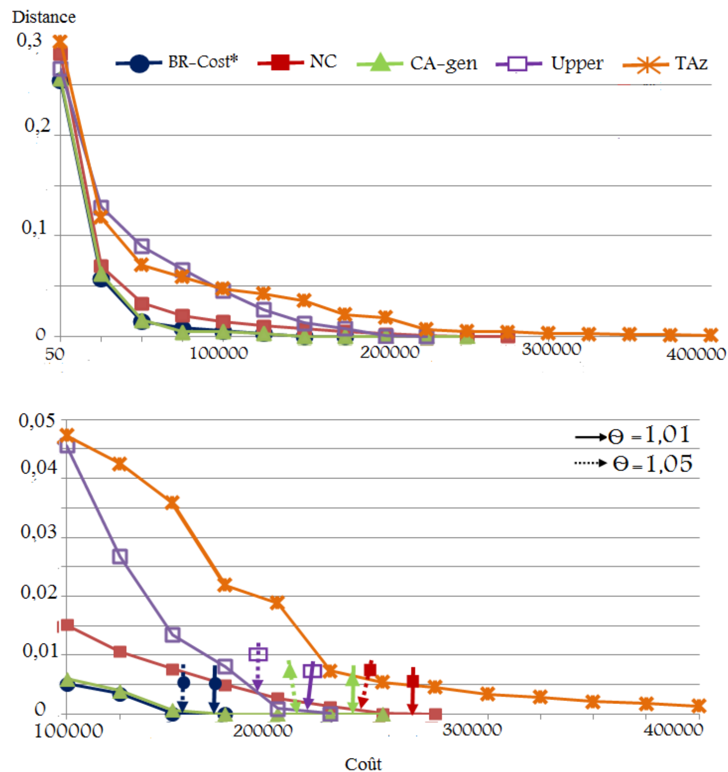


FIGURE 3.12 – Approximation avec $\mathcal{L}_k$, 6 R-sources + 6 SR-sources : distribution mixte

En conclusion, nous remarquons que l'approximation avec l'ensemble $\mathcal{L}_k$ permet une meilleure qualité d'approximation, et que les coûts d'approximation de tous les algorithmes sont moins élevés que ceux obtenus avec $\mathcal{U}_k$. En la comparant avec celle obtenue avec l'ensemble $\mathcal{U}_k$, l'approximation avec $\mathcal{L}_k$ est plus stable pour tous les algorithmes et présente des coûts plus bas pour la même qualité du résultat. Et même si *BR-Cost** présente globalement le meilleur potentiel d'approximation, la différence entre les performances des algorithmes génériques n'est plus très importante dans ce cas. Cependant, et dans tous les cas, *BR-Cost** présente le meilleur coût pour les θ-approximation. Nous constatons que, même si l'approximation avec $\mathcal{L}_k$ est meilleure qu'avec $\mathcal{U}_k$, les θ-approximations ne sont pas améliorées pour tous les algorithmes. Ceci, combiné avec la meilleure forme des courbes avec $\mathcal{L}_k$, mène à une différence encore plus grande entre le coût des θ-approximations et le coût potentiel pour la même qualité du résultat.

3.5 Conclusion

Dans ce chapitre, nous avons présenté une étude du potentiel d'approximation des algorithmes *top-k*, en se basant sur la technique d'approximation par arrêt prématuré. Nous avons proposé une généralisation de la θ-approximation dans le cadre *GF*, ensuite, nous avons présenté une comparaison expérimentale des algorithmes *top-k* en se basant sur deux ensembles d'approximation : *(i)* les k meilleurs candidats dans

l'ordre décroissant des scores maximum ($\mathcal{U}_k$), et *(ii)* les k meilleurs candidats dans l'ordre décroissant des scores minimums ($\mathcal{L}_k$). En comparant les courbes coût-distance, nous avons conclu que la stratégie *BR* présente globalement le meilleur potentiel d'approximation, avec un avantage conséquent par rapport aux autres algorithmes dans le cadre d'une approximation avec $\mathcal{U}_k$, l'ensemble sur lequel se basent tous les algorithmes *top-k* pour décider le type d'accès à effectuer et le candidat à évaluer par accès direct. Cependant, l'approximation avec $\mathcal{L}_k$ produit de meilleurs résultats pour tous les algorithmes *top-k* testés, et réduit considérablement la différence entre eux. Nous avons remarqué aussi que la θ-approximation est faiblement corrélée avec le potentiel d'approximation, mais dépend beaucoup du coût d'exécution. Cela réduit la différence entre l'approximation avec $\mathcal{U}_k$ et avec $\mathcal{L}_k$, et favorise encore la stratégie *BR* qui produit les meilleurs coûts d'exécution.

Dans le chapitre suivant, nous continuons l'étude sur la recherche approximative des k meilleurs réponses pour des requêtes de classement multicritères, dans le cas particulier de la recherche par le contenu dans les bases de données multimédia.
Dans ce contexte, nous proposons une méthode de recherche approximative des k-plus proches voisins qui s'appuie sur les algorithmes *top-k* approximatifs présentés ici. Cette méthode est ensuite comparée avec d'autres méthodes de recherche k-ppv approximative, notamment celles basées sur le hachage.

Placez votre main sur un poêle une minute et ça vous semble durer une heure. Asseyez vous auprès d'une jolie fille une heure et ça vous semble durer une minute. C'est ça la relativité.

– Albert Einstein

CHAPITRE 4

Application de la recherche *top-k* multicritères au contenu multimédia

Notre objectif est d'adapter les techniques *top-k* multicritères au cas particulier de la recherche par similarité de contenu dans les bases de données multimédia.

Nous proposons une nouvelle méthode de recherche des k-plus proches voisins, utilisant un index facile à construire, à mettre à jour et à exploiter dans un contexte distribué. Cette méthode, appelée *MSA* (Multicriteria Search Algorithm), est inspirée des algorithmes top-k présentés dans les chapitres précédents, et exprime la recherche k-ppv comme un problème de recherche *top-k* multicritères particulier. Bien que conçue pour une recherche approximative, *MSA* a l'avantage de pouvoir, si nécessaire, fournir également des résultats exacts.

Nous comparons de façon expérimentale les performances de *MSA* avec la méthode brute (*Brute-Force*) et avec *Multi-Probe LSH*, l'un des algorithmes de référence dans la recherche des k-plus proches voisins approximative, dans un contexte de très grande base de données stockée sur disque.

La recherche par similarité du contenu à grande échelle vise à trouver dans une grande collection d'images, le sous-ensemble des images les plus similaires visuellement à une image requête donnée. Les techniques courantes reposent sur l'extraction des descripteurs locaux, typiquement des descripteurs *SIFT* (Scale-Invariant Feature Transform) [Mikolajczyk et al., 2005]. Ensuite, les descripteurs de la requête sont comparés avec ceux des images de la base, les images avec le plus grand nombre de correspondances sont considérées les plus similaires [Mikolajczyk et al., 2005]. Comme alternative à cette méthode, les descripteurs sont combinés en un seul vecteur considéré comme signature de l'image, et la signature de la requête est comparée à celles des images de la base [Sivic and Zisserman, 2003] [Negrel et al., 2012]. Dans les deux cas, l'objectif est d'effectuer le calcul de similarité entre les images le plus rapidement possible. Pour effectuer cette recherche des plus proches voisins, des structures d'index et des algorithmes de recherche ont été proposés, tel que *LSH* [Indyk and Motwani, 1998], *inverted files* [Sivic and Zisserman, 2003], *NV-tree* [Lejsek et al., 2011], etc. Les principaux défis pour toutes ces techniques sont les suivants :

- Le temps de recherche doit être le plus court possible.
- Le coût de stockage de la structure d'index doit être aussi faible que possible.
- La procédure de mise à jour de la structure d'index (si de nouvelles images sont ajoutées) doit être aussi légère que possible.

Nous proposons *MSA*, un nouvel algorithme utilisant une nouvelle structure d'index et s'inspirant des algorithmes *top-k* multcritères pour effectuer une recherche par similarité de contenu à grande échelle. La méthode *MSA* exploite la nature multidimensionnelle des signatures des images pour reformuler le problème de la recherche des k-plus proches voisins à grande échelle en tant que problème de recherche *top-k* multicritères. Cette méthode offre un bon compromis entre la rapidité de la recherche, les besoins de stockage et le coût de la mise à jour. Ces propriétés proviennent d'une structure d'index très simple basée sur des listes triées, faciles à mettre à jour et à distribuer. L'algorithme *MSA* propose une nouvelle méthode approximative de recherche des k-ppv basée sur les techniques *top-k* multicritères, avec des garanties sur les faux négatifs, et l'émergence rapide de bonnes approximations, améliorées de façon monotone et conduisant, si nécessaire, à un résultat exact.

Dans la suite du chapitre, nous allons commencer, dans un premier temps par définir le résultat approximatif de la recherche des k-ppv produit par la méthode *MSA*, ensuite, nous présenterons la structure d'index utilisée et l'algorithme *MSA*.

4.1 Recherche approximative ϵ-exclusive des k-plus proches voisins

Nous considérons une base de données D comme étant un ensemble de points dans l'espace normé l_p^m, représentant l'espace Euclidien R^m avec la norme l_p et la distance $d(\cdot,\cdot)$ induite par cette norme. Pour un objet requête $q \in R^m$ et un paramètre d'approximation $\epsilon > 0$, nous appelons $P_{k,\epsilon}(q) \subset D$ le résultat approximatif de la recherche des k plus proches voisins retourné par l'algorithme de recherche.

***Définition* 5.** *(recherche des k-ppv ϵ-exclusive) Étant donné un point requête q et un seuil $\epsilon > 0$, la recherche approximative des k plus proches voisins de q dans une base de données D est appelée ϵ-exclusive, si, étant donné $P_{k,\epsilon}(q)$ le résultat approximatif de la recherche des k-ppv, et $P_k(q)$ le résultat exact, alors pour tout $x \in P_k(q) - P_{k,\epsilon}(q)$ nous avons $d(x,q) \geq \epsilon$.*

Intuitivement, la recherche k-ppv ϵ-exclusive garantit que même les bons candidats ratés dans le résultat approximatif de la recherche (faux négatifs), ont une *faible qualité*, car ils se trouvent à une distance d'au moins ϵ de la requête q. A noter que la condition ϵ-exclusive implique que pour tout point $x \in D - P_{k,\epsilon}(q)$: $d(x,q) \geq \epsilon$, puisque pour les points en dehors de $P_k(q)$, $d(x,q)$ est encore plus grande que pour ceux de $P_k(q)$.

La figure 4.1 illustre la recherche des k plus proches voisins de type ϵ-exclusive. Ici nous considérons $k = 5$ et le résultat approximatif $P_{k,\epsilon}(q) = \{p_1, ..., p_5\}$. Tous les points qui n'ont pas été retournés dans le résultat et même ceux qui sont plus proches que les points retournés p_4 et p_5, sont à une distance minimale de ϵ par rapport à la requête. Les points retournés dans le résultat $P_{k,\epsilon}(q)$ et qui sont à une distance maximale de ϵ de la requête (dans la figure : p_1, p_2, p_3) seront aussi dans le résultat exact $P_k(q)$. Si nous considérons $\zeta = max_{x \in P_{k,\epsilon}(q)}(d(x,q))$ la plus grande distance de q pour les points appartenant

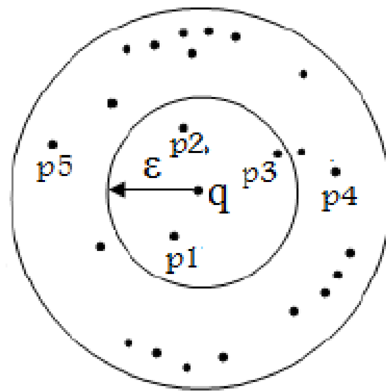


FIGURE 4.1 – Exemple de recherche des k plus proches voisins de type ϵ-exclusive

à la solution approximative, alors la qualité du résultat approximatif s'améliore quand la valeur de ϵ augmente et celle de ζ diminue. Nous montrons dans la suite que l'algorithme *MSA* produit une recherche des k plus proches voisins ϵ-exclusive, avec une augmentation monotone de ϵ et une baisse monotone de ζ.

4.2 La structure d'index *MSA*

L'algorithme *MSA* s'inspire des techniques de traitement de requêtes *top-k* multicritères pour effectuer efficacement une recherche approximative des k-ppv dans une grande base d'images. Il se base sur une structure d'index composée de m listes triées, une liste pour chaque dimension des signatures des images. Pour une image requête donnée, l'algorithme *MSA* exploite cet index pour générer m processus de recherche indépendants, fusionnés à travers une approche *top-k* multicritères.

L'idée générale de l'algorithme *MSA* est que pour chaque dimension, un processus permet de découvrir successivement des candidats pour intégrer le résultat final. Un paramètre d'approximation ϵ contrôle le moment où l'algorithme doit s'arrêter. Le résultat retourné est l'ensemble des k meilleurs candidats à cet instant. *MSA* produit une approximation monotone, l'augmentation de la valeur de ϵ conduit à un arrêt plus tard, permettant ainsi l'ajout de nouveaux candidats et l'amélioration de la qualité du résultat approximatif retourné.

La figure 4.2 présente la structure d'index proposée par la méthode *MSA*. Comme nous l'avons déjà mentionné, nous considérons une base de données D qui contient des signatures d'images (vecteurs de m dimensions). L'index utilisé dans *MSA* est composé de m listes triées : $L_1, ..., L_m$, une liste pour chaque dimension. Chaque élément d'une liste L_i est un couple (x, x_i) avec x l'identifiant de la signature de l'image dans D, et x_i le $i^{\text{ème}}$ composant du vecteur signature. Chaque liste L_i indexe tous les éléments de la base D, ces éléments sont triés dans l'ordre décroissant des valeurs de x_i.

Pour une requête $q = (q_1, ..., q_m)$, l'index permet une recherche binaire rapide de la position de q_i dans la liste triée L_i pour toute valeur de i. Le rôle de chaque liste L_i est de fournir successivement les couples (x, x_i) dans l'ordre croissant de la distance entre x_i et q_i, cela est réalisé par des accès séquentiels

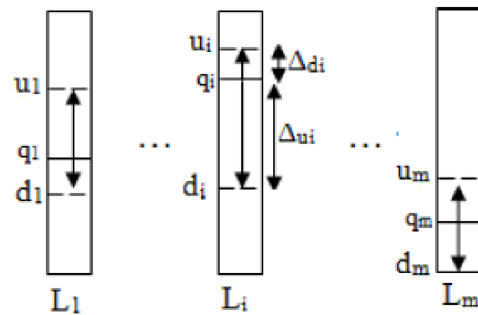


FIGURE 4.2 – Structure d'index utilisée dans MSA

à partir de q_i et dans les deux sens. A chaque étape, deux candidats sont sélectionnés, le premier est le plus proche de q_i dans le sens du haut (u_i), et le deuxième dans le sens inverse (d_i). Les deux candidats sont à des distances égales respectivement à $\Delta_{ui} = u_i - q_i$ et $\Delta_{di} = d_i - q_i$. Le candidat le plus proche à la requête q_i sera retourné, la distance par rapport à la requête est alors égale à $\Delta_i = \min(\Delta_{ui}, \Delta_{di})$. Bien sûr, si les candidats dans l'une des directions sont terminés, seul le plus proche candidat suivant dans l'autre direction est considéré.

Deux méthodes d'accès sont fournies par l'index *MSA* :

- $init(q)$ qui permet de trouver la position de chaque q_i dans la liste L_i.
- $getNext(L_i, q_i)$ qui retourne le prochain couple (x, x_i) dans la liste L_i dans l'ordre croissant de la distance par rapport à q_i.

L'index utilisé dans *MSA* pourrait être stocké en mémoire, mais, nous considérons ici le cas de très grandes bases de données d'images, où l'index et les signatures des images doivent être stockés sur disque. Nous avons implémenté les listes de l'index ainsi que les signatures des images en tant que fichiers avec accès directs et séquentiels. Les signatures des images sont stockées séquentiellement dans le fichier de la base, et le numéro d'ordre de chaque signature (vecteur), est utilisé comme identifiant du vecteur dans les listes L_i de l'index. A noter que les listes de l'index pourraient être implémentées comme des arbres $B+$ afin d'optimiser les mises à jour.

Les avantages de cette structure d'index peuvent être résumé par les points suivants :

- Pour la recherche dans l'index : $init$ réalise en temps logarithmique une recherche dichotomique, et $getNext$ est réalisé en temps constant.
- Pour la création et la mise à jour : très simples, car réalisées sur des listes triées ou des arbres $B+$.
- Pour la distribution de l'index : les listes de classement indépendantes par dimension peuvent êtres stockées sur des sites différents.
- Pour le stockage : la taille des données à stocker est de $m * n * (|ref| + |comp|)$, où n est le nombre de signatures, m le nombre de dimensions de ces signatures, $|ref|$ la taille d'une référence, et

$|comp|$ la taille d'une composante de signature. Comme la taille de la base D est $m * n * |comp|$, l'index est du même ordre de grandeur que la base.

Dans la suite, nous présentons l'algorithme *MSA* et les résultats expérimentaux montrant l'efficacité de cet algorithme dans la recherche des k plus proches voisins dans une vraie base d'images.

4.3 L'algorithme de recherche multicritères : MSA

L'algorithme *MSA* utilise une approche *top-k* multicritères pour fusionner toutes les informations sur les candidats les plus prometteurs et qui proviennent des listes de classement dans l'index.

Algorithm 5 L'algorithme *MSA*

Require: $q \in R^m$ and $\epsilon \in R_+^*$ and $k \in N^*$
  $cand \leftarrow \emptyset$
  $\Delta_i \leftarrow 0, i = \overline{1,m}$
  $\epsilon_{crt} \leftarrow \mathcal{F}(\Delta_1, ..., \Delta_m)$
  init(q)
  repeat
    $i \leftarrow$ *ChooseDimension*()
    $(x, x_i) \leftarrow$ *getNext*(L_i, q_i)
    exit loop when $x = nil$
    if x not in $cand$ **then**
      $d \leftarrow$ *ComputeDistance*(x, q)
      AddCandidate($cand, x, d$)
    end if
    $\Delta_i \leftarrow |x_i - q_i|$
    $\epsilon_{crt} \leftarrow \mathcal{F}(\Delta_1, ..., \Delta_m)$
  until $|cand| \geq k$ **and** ($\epsilon_{crt} \geq \epsilon$ **or** $\epsilon_{crt} \geq$ *KthDistance*($cand$))
  return *TopK*($cand$)

L'algorithme 5 présente l'idée générale de *MSA*. Trois entrées sont nécessaires : la signature de l'image requête q, le paramètre d'approximation ϵ, et le nombre d'images dans le résultat final k. *MSA* maintient un seuil ϵ_{crt}, et une liste de candidats $cand$ triée dans l'ordre croissant de la distance par rapport à q. $\mathcal{F}$ est une fonction d'agrégation monotone, qui permet de calculer la distance jusqu'à la requête q en combinant les distances Δ_i calculées sur chaque dimension. Cette fonction peut être instanciée, par exemple en une distance de type l_p, $d(x, y) = \sqrt[p]{\sum_{i=1}^{m} |x_i - y_i|^p}$. Le seuil ϵ_{crt} représente la distance minimale que peut avoir un nouveau candidat par rapport à la requête ; à chaque étape, il est calculé en fusionnant les distances courantes sur chaque dimension (Δ_i) en utilisant la fonction $\mathcal{F}$.
La position de recherche initiale dans chaque liste de l'index est fixée par un appel à la fonction $init$.
A chaque itération, *MSA* commence par sélectionner la dimension i à interroger. Plusieurs stratégies peuvent êtres utilisées et implémentées pour la fonction $ChooseDimension$. Par exemple dans *TA* [Fagin et al., 2001], la stratégie utilisée consiste simplement à interroger successivement toutes les dimensions, tandis que dans *Quick Combine* [Güntzer et al., 2000] une stratégie basée sur le calcul d'un bénéfice est

préférée, avec le choix de la dimension ayant le meilleur bénéfice. En sélectionnant une dimension i, la liste L_i est interrogée avec la fonction $getNext$ pour retourner l'actuel meilleur objet de la liste. Si tous les objets de la liste L_i sont déjà retournés ($x = nil$) l'algorithme s'arrête car tous les objets de la base ont été retrouvés. Dans le cas où l'objet x retourné par la fonction $getNext$ est découvert pour la première fois (nouvel objet), il sera intégré à la liste des candidats $cand$ après évaluation de sa distance globale par rapport à la requête. Cette distance est calculée avec la fonction $ComputeDistance$ qui accède à la base de données pour récupérer la signature x. A la fin de chaque itération, la valeur de ϵ_{crt} est mise à jour en prenant en considération la nouvelle valeur de Δ_i. *MSA* s'arrête quand la liste $cand$ contient au moins k candidats, et la valeur de ϵ_{crt} devient supérieure ou égale soit à la valeur du paramètre d'approximation ϵ, soit à $\zeta_{crt} = KthDistance(cand)$, la $k^{\text{ème}}$ meilleure distance. Le premier cas correspond à une solution approximative ϵ-exclusive, comme nous le démontrons ci-dessous. Le deuxième cas correspond à un résultat exact de la recherche des k plus proches voisins, puisque tout nouveau candidat sera au moins à une distance de ϵ_{crt} de la requête q. Par conséquent, les k premiers candidats resteront les meilleurs et constituent le résultat exact. Dans les deux cas *MSA* retourne les k meilleurs candidats de la liste $cand$, *TopK*($cand$).

L'algorithme *MSA* est comparable aux algorithmes *top-k* présentés dans les chapitres précédents, il est proche du cadre général *GF* avec approximation, et utilise une approche *SR-source* plus efficace qu'une approche *S-source* : à cause du grand nombre de dimensions, un algorithme de type *NRA* converge trop lentement. L'avantage de l'approche utilisée dans *MSA* est qu'elle permet d'éviter le maintien d'une liste de candidats avec des intervalles de scores, méthode utilisée dans *NRA* où un accès séquentiel à une source nécessite la mise à jour des intervalles de scores de tous les candidats qui ne sont pas encore trouvés dans la source. Dans notre cas, la mise à jour de la liste des candidats consiste simplement à insérer le nouvel objet. Dans *MSA*, l'objectif est de minimiser la distance, tandis que dans les algorithmes *top-k* l'objectif est plutôt de maximiser le score. La fonction d'agrégation utilisée ici, pour calculer la distance entre les images de la base et la requête, dépend du contexte et de l'application en général. La condition d'arrêt utilisée dans *MSA* est adaptée à son contexte, mais reste proche de celles vues dans les algorithmes *top-k* (ϵ_{crt} correspond à U_{unseen}, et *KthDistance* à L_k).

Le caractère monotone de la fonction d'agrégation $\mathcal{F}$ et la propriété de l'index de MSA qui permet de dégager les candidats avec des valeurs croissantes de Δ_i garantissent des valeurs toujours croissantes du seuil ϵ_{crt}. Aussi, toute signature qui n'est pas encore retournée par l'index est à une distance d'au moins ϵ_{crt} de q. Avec ces remarques, les deux propriétés suivantes sont garanties pour l'algorithme *MSA* :

***Théorème* 4.** *L'algorithme MSA permet une recherche approximative ϵ-exclusive des k plus proches voisins*

Démonstration. En effet, tout bon candidat x non retourné dans le résultat (*faux négatif*), est un objet qui n'a jamais été découvert dans les listes de l'index, alors la distance par rapport à la requête q : $d(x, q) \geq \epsilon_{crt}$, et comme $\epsilon_{crt} \geq \epsilon$ au moment de l'arrêt de l'algorithme, nous avons : $d(x, q) \geq \epsilon$ □

***Théorème* 5.** *MSA est monotone en ϵ et en temps d'exécution*

Démonstration. *(i)* L'exécution de *MSA(q,ϵ_1,k)* correspond à un arrêt prématuré de l'exécution de *MSA(q,ϵ_2,k)* si $\epsilon_1 < \epsilon_2$. *(ii)* Les durées d'exécution croissantes correspondent à des valeurs croissantes de ϵ et des valeurs décroissantes de ζ.

Le point *(i)* est la conséquence de l'augmentation monotone de ϵ_{crt} dans le temps, et du fait que $TopK(cand)$ à un moment donné correspond au résultat de *MSA(q,*ϵ_{crt}*,k)*. Le point *(ii)* est vérifié car les valeurs de ϵ_{crt} sont croissantes dans le temps, alors que pour ζ_{crt} elles sont décroissantes car des nouveaux candidats sont rajoutés à $cand$ améliorant l'ensemble courant des k meilleurs candidats.

A noter que la monotonie de *MSA* permet une approche pratique alternative, en fixant une durée d'exécution maximale si la valeur de ϵ est difficile à décider. □

La méthode *MSA* propose un mélange de traitement de requêtes k-ppv et ϵ (points dans un rayon), dans lequel les faux négatifs ne sont pas les objets les plus importants, puisqu'ils se trouvent au moins à une distance ϵ de q. La monotonie permet ici d'améliorer continuellement la solution. Avec cette méthode, nous visons des applications proches de la recherche de quasi-copies, où peu de points sont intéressants, donc les erreurs dans le résultat approximatif concernant les candidats qui sont à une distance au delà de ϵ sont moins importants. A tout moment de l'exécution, le résultat approximatif de *MSA* propose les meilleurs points à moins de ϵ et des candidats prometteurs au delà de ϵ pour compléter le *top-k*.

Dans la suite, nous proposons une évaluation expérimentale de la méthode *MSA*.

4.4 Évaluations expérimentales

Pour évaluer les performances de notre méthode et la comparer avec les techniques de l'état de l'art, nous avons utilisé la base de données d'images de vacances *Holidays* [Jegou et al., 2012]. Cette base d'images contient des photos personnelles de vacances et a été créée pour tester les méthodes qui permettent la recherche de quasi-copies. Elle contient 1491 images réparties en 500 sous-groupes, où chaque groupe représente une scène ou un objet différent. La figure 4.3 présente un exemple d'images appartenant à la base utilisée dans nos expériences. Toutes les images de cette base sont en couleur et haute résolution. *Holidays* contient aussi des descripteurs locaux SIFT (Scale-invariant feature transform). Pour avoir suffisamment de données pour nos expériences, nous rajoutons à la base de données *Holidays* un ensemble de 1 million d'images sans rapport avec celles de la base, à partir de la base de données *Flickr-10M*. Pour chaque image nous extrayons une signature compacte VLAT de 64 dimensions, voir [Negrel et al., 2012]. Les vecteurs des signatures sont tous normalisés.

Toutes les expériences sont réalisées sur une machine avec $4GB$ de mémoire DDR, un processeur *dual core (2.13 GHz)* et un système d'exploitation Linux. L'index utilisé dans *MSA* est implémenté sur disque comme nous l'avons expliqué précédemment, avec un fichier par liste triée de l'index, et un fichier pour la base des signatures. Les algorithmes sont implémentés en Java, et pour optimiser les entrées/sorties sur disque nous avons utilisé le package *java.nio*

4.4.1 Algorithmes testés

Nous avons tout d'abord comparé deux variantes de l'algorithme *MSA* que nous avons appelées MSA_{TA} et MSA^*, avec la méthode brute et la technique *Multi-Probe LSH*, présentée dans le chapitre 1 :

- La variante MSA_{TA} utilise la stratégie considérée dans l'algorithme *TA*. Ici, chaque liste triée de l'index est considérée comme une source indépendante. A chaque itération, une liste est interrogée

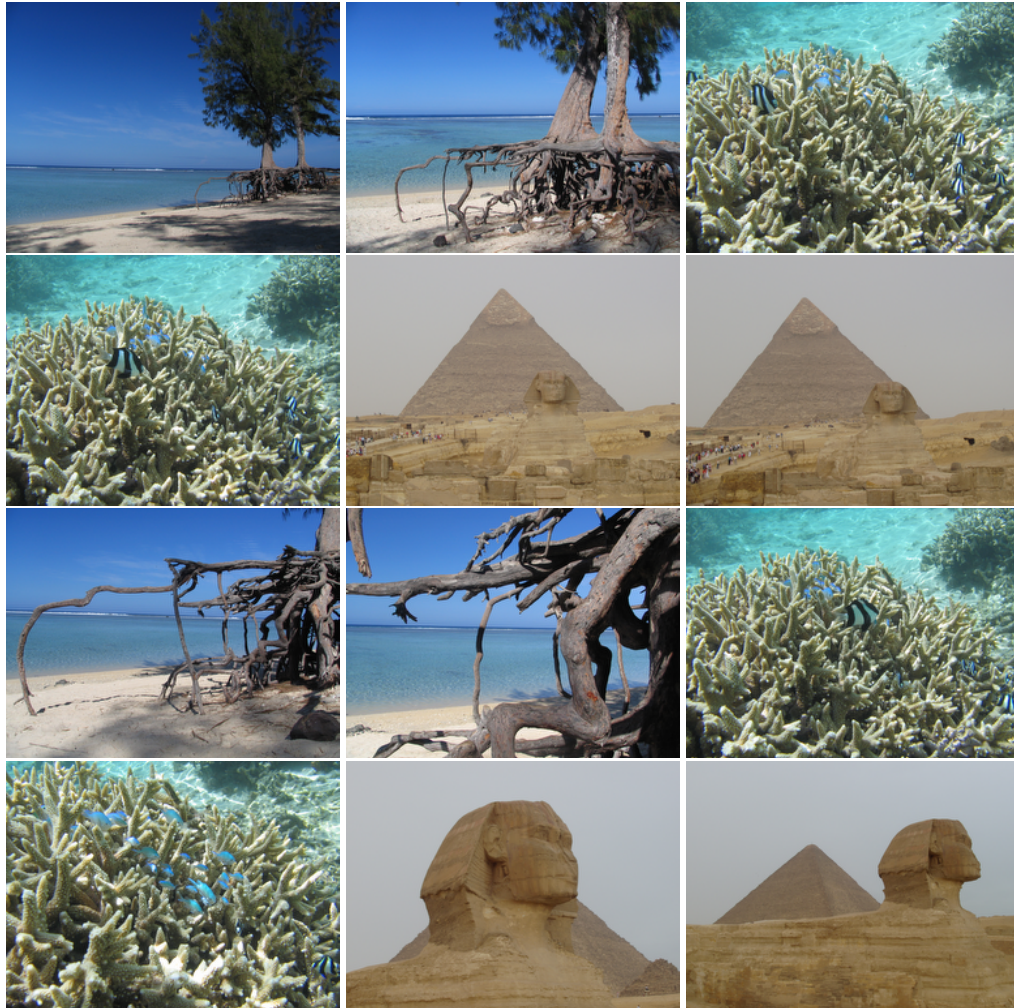


FIGURE 4.3 – Exemple d'images de la base de données Holidays

pour retourner un objet (le plus proche de la requête). Si l'objet est déjà connu, la liste suivante est interrogée, sinon la base est consultée pour récupérer la signature du candidat et calculer la distance à la requête.

- La variante MSA^* a comme objectif de minimiser le nombre d'entrées/sorties disque dans l'index. Ici, une seule liste triée est consultée dans l'index. Dans ce cas, la fonction $chooseDimension$ retourne toujours la même valeur pour choisir la même dimension (liste). Dans ce cas, nous choisissons celle avec la plus grande amplitude de la valeur indexée, qui est potentiellement la plus discriminante pour les candidats.

- La méthode brute parcourt toute la base de signatures stockée sur disque et calcule à chaque étape la distance entre une signature et la requête. Les k meilleures signatures sont retournées dans le résultat final en tant que k plus proches voisins.

- La méthode *MLSH* est une méthode de référence dans la recherche approximative des k-ppv, et

nous l'avons adaptée à notre contexte pour qu'elle soit comparable à *MSA*. Comme dans les méthodes *MSA*, l'index est stocké sur disque. Chaque cellule (bucket) se trouve donc sur disque et contient les identifiants des signatures contenus dans la cellule. Comme pour *MSA*, les signatures sont stockées dans un fichier à part (la base de données).
Pour chaque table de hachage, l'algorithme *MLSH* commence par retourner les identifiants des signatures dans la cellule où se trouve la requête, et continue de la même manière avec les cellules voisines jusqu'à effectuer T accès dans chaque table de hachage. La liste des candidats (identifiants des signatures) trouvés est maintenue en mémoire. Ensuite, la base de données est accédée pour récupérer les signatures des éléments de la liste des candidats, afin d'évaluer la distance avec la requête pour chaque signature. Enfin, la liste des signatures est triée par ordre croissant de distance, et les k premières sont retournées.

4.4.2 Paramètres

Dans chaque expérience, nous considérons 500 requêtes ; chaque requête est une signature d'image appartenant à un sous groupe de la base *Holidays*. Les résultats présentés sont à chaque fois la moyenne des mesures sur les 500 requêtes. Dans toutes les expériences, nous considérons une valeur de $k = 10$, et pour trouver les 10 plus proches voisins, nous comparons les signatures des images en utilisant la distance euclidienne l_2. Pour deux signatures x et y : $d(x, y) = \sqrt[2]{\sum_{i=1}^{m} |x_i - y_i|^2}$, avec m le nombre de dimensions de chaque signature.

4.4.3 Résultats expérimentaux

Dans les expériences réalisées, nous avons tout d'abord comparé les performances des algorithmes *MSA* avec celles de la méthode brute, en nous concentrant sur deux points essentiels dans la recherche approximative des k plus proches voisins :

- Le temps d'exécution : notre objectif est de montrer que notre technique permet d'accélérer le processus de recherche des k-ppv.
- La qualité du résultat : en mesurant le mAP (*Mean Average Precision*) par rapport au temps de recherche, nous montrons que les techniques utilisées dans *MSA* permettent d'avoir un résultat de très bonne qualité très rapidement.

La première expérience analyse la variation du temps d'exécution par rapport à la valeur de ϵ pour les algorithmes *MSA*. La figure 4.4 présente les résultats obtenus suite à cette expérience, et montre que le temps d'exécution augmente avec la valeur de ϵ de manière différente pour les deux variantes MSA_{TA} et MSA^*. Le temps d'exécution de la méthode brute est indépendant de la valeur de ϵ, et la recherche exacte des k plus proches voisins par cette méthode prend environ 1100 ms. Concernant les variantes de *MSA*, pour des petites valeurs de ϵ (inférieures à 0.25), MSA_{TA} et MSA^* sont plus rapides que la méthode brute, et ont des comportements très similaires. La différence entre les deux variantes apparait clairement pour des valeurs plus grandes de ϵ, la performance de MSA_{TA} se dégrade pour atteindre le triple de temps d'exécution de la méthode brute pour un résultat exact des k-ppv. Ce phénomène peut être expliqué par le grand nombre d'entrées/sorties disque, effectuées sur les 64 fichiers de l'index. L'avantage de l'algorithme MSA^* est l'utilisation d'un seul fichier index, ce qui permet un comportement différent

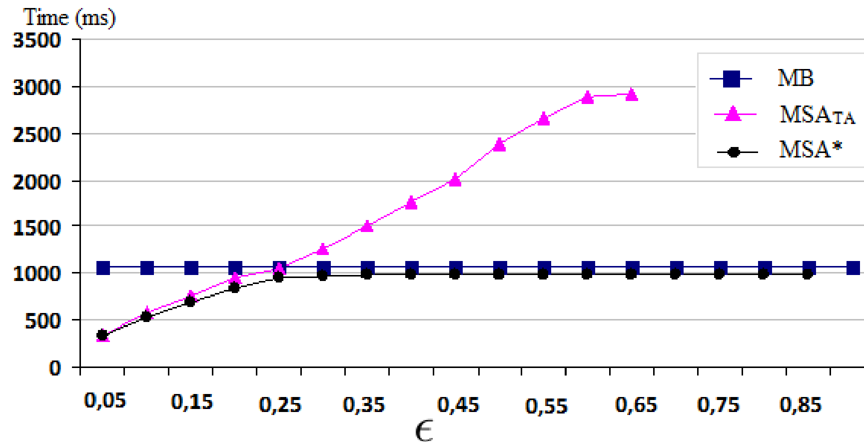


FIGURE 4.4 – Comparaison des algorithmes *MSA* avec la méthode brute (MB) : temps d'exécution en fonction de la valeur de ϵ

et un temps de recherche trois fois plus rapide. Pour des valeurs très grandes de ϵ, MSA^* permet d'obtenir un résultat exact sans être très couteûx et en restant toujours plus rapide que la méthode brute.

En conclusion de cette expérience, nous avons montré que *MSA* est plus rapide que la méthode brute en termes de temps d'exécution pour des petites valeurs de ϵ, mais il est moins approprié pour une recherche exacte des k plus proches voisins. Cependant, *MSA* a l'avantage d'améliorer de façon monotone la qualité de son résultat en fonction du temps d'exécution, et peut aboutir si nécessaire à un résultat exact tout en restant plus rapide que la méthode brute (cas de MSA^*).

L'objectif de la deuxième expérience réalisée est d'analyser la qualité du résultat approximatif proposé par les algorithmes *MSA* quand le temps de recherche est inférieur à celui de la méthode brute, et de la comparer avec celle de la technique *MLSH*. Dans ces expériences, nous considérons les paramètres par défaut suivants pour la méthode *MLSH* :

- L, le nombre de tables de hachage est 5.
- T, le nombre d'accès (probe) pour chaque table est par défaut 16.
- M, le nombre de fonctions de hachage pour chaque table est 12.
- La valeur de W est ici 0.8.

Les valeurs choisies pour ces paramètres sont le choix typique pour la méthode *MLSH*, tout de même, nous avons réalisé des expériences dans lesquelles nous avons varié les valeurs de ces paramètres (L, W et M), et nous avons gardé celles qui permettent une meilleure qualité du résultat avec un minimum de temps d'exécution.

Les résultats obtenus sont présentés dans la figure 4.5 qui présente l'évolution de la qualité du résultat (*mAP*) en fonction du temps d'exécution. Le *mAP* est la métrique la plus utilisée pour mesurer la qualité de recherche. Pour mesurer la qualité du résultat, la base d'images utilisée doit être munie d'une *vérité terrain*. Cette vérité terrain définit la catégorie à laquelle appartient chaque image de la base. Nous rappelons que dans la base *Holidays*, nous avons 500 catégories, et nous considérons une requête pour chaque

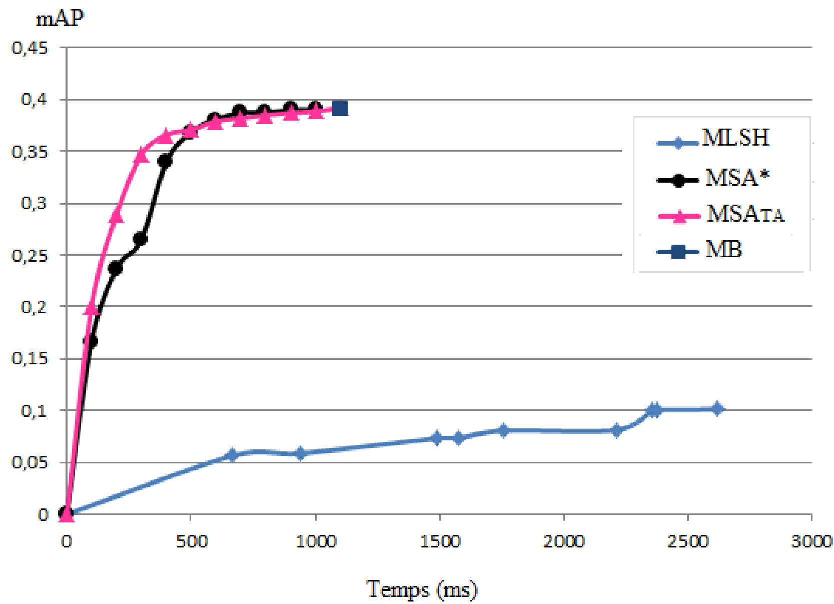


FIGURE 4.5 – Comparaison des algorithmes *MSA* avec la méthode brute (MB) et la technique *MLSH* : *mAP* en fonction du temps d'exécution

catégorie. Dans cette évaluation, nous utilisons la mesure du *mAP* des k premiers éléments. Le *mAP* représente la moyenne de précision des 500 requêtes. La précision du résultat d'une requête est mesurée par *AP*.
Pour calculer le *AP*d'une requête, nous considérons tout d'abord les trois ensembles suivants :

- K, le *top-k* exact de la requête (vérité terrain).
- R^k, le *top-k* retourné par la méthode utilisée pour une requête donnée.
- $rank$, permettant de garder les classements (rangs) des candidats de K dans R^k.

Deux cas sont possibles pour calculer le *AP* :

1. $R^k \cap K = \emptyset$. Dans ce cas l'ensemble R^k est composé que de faux positifs, et par conséquent $AP_{R^k} = 0$.
2. $R^k \cap K \neq \emptyset$. Dans ce cas, le *AP* est mesuré de la manière suivante :

$$AP_{R^k} = \frac{1}{k} \sum_{j=1}^{|rank|} \frac{j}{rank[j]} \tag{4.1}$$

Ici l'ordre des images pertinentes n'est pas important. Par exemple, si pour une catégorie donnée, une requête possède 3 quasi-copies (images pertinentes), trouvées en tant que *top-3* dans la liste des candidats, le *AP* de la requête sera 1 (la valeur maximale). L'ordre entre les 3 images n'influence pas la valeur de *AP*. Le *mAP* est ensuite calculé en moyennant les valeurs de précision (*AP*) pour toutes les requêtes.

La méthode brute est ici (figure 4.5) présentée par un seul point indiquant le temps total de la recherche des k-ppv et la valeur maximale possible du *mAP*. Dans notre cas, la valeur maximale du *mAP* est 0.39, cela est dû à l'imprécision des descripteurs numériques. La méthode brute trouve le meilleur *mAP* possible avec des descripteurs donnés. La qualité du résultat s'améliore de façon monotone pour les deux variantes de *MSA*, avec une meilleure approximation du résultat pour l'algorithme MSA_{TA} pour des petites valeurs de ϵ. Cet avantage est dû à l'utilisation de toutes les listes de classement de l'index, ce qui permet d'avoir les signatures les plus proches dans chaque dimension. A l'inverse de MSA_{TA}, l'algorithme MSA^* est moins performant puisqu'il utilise une seule liste de classement dans son index et compare les images dans l'ordre proposée par cette liste. L'algorithme MSA_{TA} atteint de très bonnes valeurs de *mAP* très rapidement et à partir de 300 ms de temps d'exécution, nous obtenons une valeur de *mAP* (0.35) comparable à celle de la méthode brute (0.39). Une valeur similaire est obtenue avec l'algorithme MSA^* à partir de 450 ms de temps d'exécution. Concernant la méthode MLSH, la figure montre que la qualité du résultat s'améliore aussi de façon monotone, mais reste toujours moins performant que les méthode *MSA*. Pour un temps d'exécution d'environ 2600 ms, le *mAP* mesuré pour *MLSH* est de $0,11$, cette mesure montre clairement que les méthodes *MSA* sont plus efficaces et permettent une meilleure qualité du résultat avec un temps d'exécution relativement court.

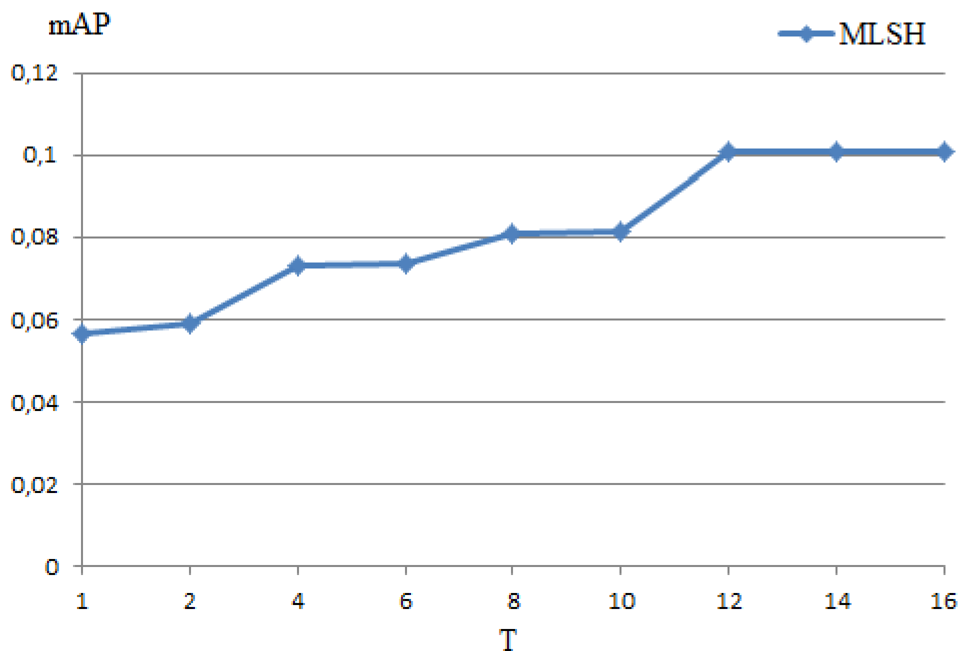


FIGURE 4.6 – Évaluation de la qualité du résultat approximatif (*mAP*) en fonction du nombre d'accès pour *MLSH*

Dans la figure 4.6, nous proposons une évaluation du résultat approximatif retourné par la méthode MLSH en fonction de la valeur de T (nombre d'accès à une table). La figure montre que la qualité du résultat retourné s'améliore en augmentant la valeur de T, et le *mAP* passe de 0.05 pour $T = 1$ à 0.1 pour $T = 16$. Même en doublant la valeur du *mAP*, la qualité du résultat retourné reste loin derrière celle

obtenue par la méthode brute et les algorithmes *MSA*. A noter ici que la valeur du *mAP* pour les trois derniers points de la figure (entre $T = 12$ et $T = 16$) reste pratiquement la même, ce qui signifie qu'une augmentation supplémentaire de la valeur de T ne va pas améliorer la qualité du résultat. En effectuant les mesures pour cette expérience, nous avons testé dans le cas où $T = 30$, et nous avons obtenu une valeur très similaire du *mAP* (environ 0.11).

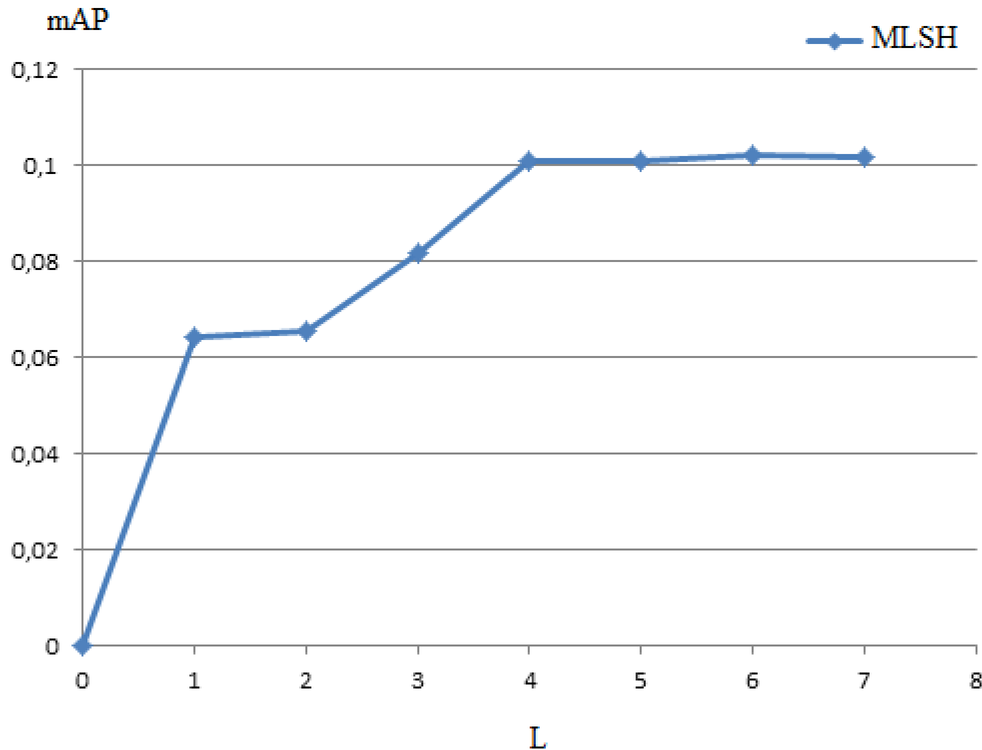


FIGURE 4.7 – Évaluation de la qualité du résultat approximatif (*mAP*) en fonction du nombre de table pour *MLSH*

La dernière expérience concerne aussi la méthode *MLSH* et analyse la variation de la qualité du résultat (*mAP*) par rapport au nombre de tables de hachage. La figure 4.7 présente les résultats de cette expérience, et montre une amélioration de la qualité du résultat approximatif retourné en augmentant le nombre de tables de hachage. La valeur du *mAP* passe de 0.064 si nous considérons une seule table de hachage, à 0.1 dans le cas où $L = 4$. Nous remarquons aussi que la qualité du résultat reste pratiquement la même en augmentant encore le nombre de tables, par exemple, le *mAP* pour $L = 7$ est très similaire à celui obtenu pour quatre ou cinq tables, mais avec un temps d'exécution beaucoup plus élevé, environ 5000 ms contre 2600 ms pour $L = 5$.

En conclusion, nous avons commencé dans ces expériences par analyser le comportement des algorithmes *MSA* en mesurant *(i)* le temps d'exécution nécessaire pour une bonne approximation des k plus proches voisins, et *(ii)* la qualité du résultat proposé pour différentes valeurs de ϵ. Nous avons comparé les performances des variantes MSA_{TA} et MSA^* avec celles de la *méthode brute* dans les cas de grands volumes de données multidimensionnelles. Cette comparaison a permis de montrer que *MSA* permet

d'avoir un résultat approximatif de très bonne qualité beaucoup plus rapidement que la *méthode brute*, trois à quatre fois plus vite dans notre cadre expérimental. Dans un deuxième temps, nous avons comparé la méthode *MSA* à *MLSH*, une des méthodes de référence dans la recherche approximative des k-ppv. Nous avons évalué la qualité du résultat par rapport au temps d'exécution, et nous avons montré que pour le même temps d'exécution, les algorithmes *MSA* sont plus efficaces que *MLSH* et proposent un résultat approximatif de meilleure qualité. Même si *MLSH* s'améliore dans le temps, il reste toujours moins bon en termes de qualité de résultat et de temps d'exécution que les algorithmes *MSA* et que la méthode brute dans ce contexte. Nous avons fini par analyser la qualité du résultat obtenu par la méthode *MLSH* en variant d'abord le nombre d'accès (probe) dans les tables, ensuite, le nombre de tables de hachage. Les deux expériences ont montré une amélioration du résultat au début, mais la qualité du résultat reste stable dans la dernière partie de chaque courbe.

Nous continuons dans la suite notre étude sur la recherche approximative des k-plus proches voisins, et nous proposons une optimisation de la méthode *MLSH* dans le contexte de *MSA*.

4.5 Dynamic Multi-Probe LSH : recherche efficace des k-ppv sur disque

Dans cette section, nous proposons une amélioration d'une des méthodes de référence pour la recherche approximative des k-ppv, Multi-Probe LSH, dans le contexte de *MSA* : les très grandes bases de données stockées sur disque. Ce travail a été effectué en collaboration avec S.Yin durant son post-doctorat, et en parallèle aux travaux sur la méthode *MSA*. Il constitue un complément aux travaux sur la recherche approximative des k-ppv dans les grandes bases de d'images.

La méthode *Multi-Probe LSH* (*MLSH*), présentée dans le chapitre 1, permet de réduire le nombre de tables de hachage nécessaires à la méthode *LSH*, en effectuant plusieurs accès dans chaque table de hachage, dans les cellules voisines à celle de la requête. Cette technique a montré son efficacité dans le cadre d'une recherche des plus proches voisins où les données sont stockées en mémoire principale. Nous considérons dans ce travail un contexte différent, avec un grand volume de données, où le stockage dans une mémoire secondaire est nécessaire. Dans ce contexte particulier, les données de cellules voisines peuvent êtres stockées dans différents endroits sur disque, ce qui augmente le nombre d'entrées-sorties dans cette méthode. Nous proposons *Dynamic Mulit-Probe LSH* (DMLSH), une nouvelle méthode permettant de réduire le nombre d'entrée sorties dans le cadre d'une recherche des k-plus proches voisins dans un espace multidimensionnel.

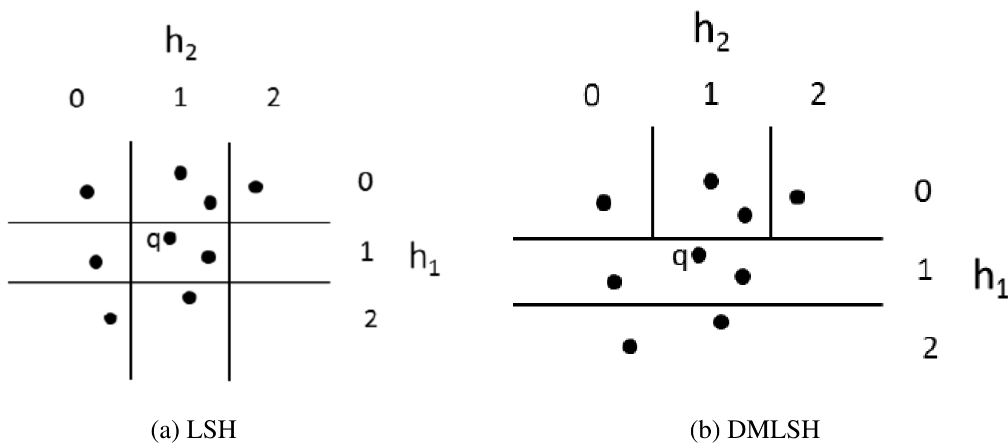
4.5.1 DMLSH : idée générale

L'idée générale de *DMLSH* est de faire varier dynamiquement la granularité des cellules (buckets) pour adapter le nombre d'objets à la taille d'une page disque. Pour cela, nous utilisons les mêmes fonctions de hachage (sensibles à la proximité) et le même ordre de consultation des cellules que celui utilisé dans *MLSH*. Plus précisément, l'idée de *DMLSH* peut être résumée en deux points principaux :

1. Au lieu de construire directement une table de hachage en utilisant les M fonctions de hachage, nous commençons tout d'abord par construire une table de hachage en utilisant seulement une fonction de hachage. Si l est le nombre d'objets que peut contenir une page disque et si une cellule

contient plus de l objets, une deuxième fonction de hachage est utilisée pour diviser cette cellule en plusieurs petites cellules. Et si des petites cellules continuent d'avoir plus de l objets, d'autres fonctions de hachage sont utilisées jusqu'à ce que toutes les cellules contiennent au plus l objets ou que le nombre de fonctions de hachage utilisé atteigne M. Nous stockons les signatures de toutes les cellules dans un arbre $B+$. Les signatures ont des tailles différentes et par conséquent, les clés dans l'arbre $B+$ ont des tailles différentes.

2. Nous utilisons l'algorithme proposé dans *MLSH* pour générer les signatures des cellules à consulter. Si la signature générée existe dans l'arbre $B+$, nous retournons tous les objets de la cellule correspondante, sinon, nous recherchons la cellule dont la signature est un préfixe de la signature générée.



(a) LSH (b) DMLSH

FIGURE 4.8 – Utilisation des fonctions de hachage pour la création des cellules

Nous considérons dans la figure 4.8, un exemple permettant d'expliquer ces deux principes. La figure 4.8(a), montre la construction d'une table de hachage en utilisant deux fonctions de hachage h_1 et h_2. Pour une requête q, en utilisant l'algorithme MLSH pour générer 6 signatures, les cellules qui seront choisies sont celles avec les signatures 11, 01, 10, 00, 21 et 20. Ceci nécessite l'accès à 6 pages sur disque pour retourner 7 candidats. Dans la figure 4.8(b), au lieu d'utiliser directement les deux fonctions de hachage, nous commençons par utiliser la fonction h_1, ensuite, si le nombre d'objets dans une cellule dépasse le seuil l ($l = 2$ dans cet exemple), nous la divisons en appliquant la deuxième fonction de hachage h_2. Pour la même requête q, nous utilisons le même algorithme proposé dans *MultiProbe LSH* pour générer les signatures des cellules à interroger, qui sont donc 11, 01, 10, 00, 21 et 20. Dans ce cas, pour une signature qui ne correspond pas à une cellule (par exemple 11), nous considérons la cellule dont la signature est un préfixe de la première, dans ce cas, la cellule 1 pour la signature 11. Par conséquent, l'ensemble des signatures des cellules à interroger est : 1, 01, 00, et 2. Seulement 4 pages disque sont accédées dans ce cas.

4.5.2 Construction de l'index

De la même manière que dans *MLSH*, nous utilisons L tables de hachage, et pour chaque table nous générons M fonctions de hachage. A la différence de *MLSH*, les M fonctions ne sont pas automatiquement utilisées, et les signatures obtenues sont stockées dans un arbre $B+$. Cependant, la recherche que nous réalisons dans l'arbre $B+$ n'est pas exacte, car nous pouvons retourner un préfixe de la clé recherchée. Par exemple, en cherchant la clé 001010, nous pouvons retourner 001 comme résultat. Pour cela, nous avons légèrement modifié l'algorithme classique de recherche dans l'arbre $B+$.

L'arbre *DMLSH*

Nous appelons arbre *DMLSH*, l'arbre $B+$ [Comer, 1979] que nous utilisons en mémoire pour stocker les signatures (clés) des cellules produites par la méthode *DMLSH* avec des informations supplémentaires sur les cellules. Dans l'arbre DMLSH, seules les signatures des cellules non-vides sont stockées. A noter que les clés (signatures) stockées dans l'arbre $B+$ sont des séquences de "*digits*" produits par les fonctions de hachage individuelles.

La seule modification apportée à l'arbre $B+$ est la surcharge des opérateurs de comparaison des signatures. Les nouvelles définitions des opérateurs $==$, $<$ et $>$ sont les suivantes :

- Considérant deux clés k_1 et k_2 : $k_1 == k_2$ si une des clés k_1 ou k_2 est un préfixe de l'autre, ce qui inclut le cas où $k_1 = k_2$.
- Si $k_1 == k_2$ retourne faux, il est simple de montrer que k_1 et k_2 ont la même forme : $k_1 = k + d_1 + s_1$ et $k_2 = k + d_2 + s_2$, avec k, s_1, s_2 des séquences de *digits* qui peuvent êtres vides, et d_1, d_2 sont des *digits*, avec $d_1 ! = d_2$. Nous disons que $k_1 > k_2$ (respectivement $k_1 < k_2$) si $d_1 > d_2$ (respectivement $d_1 < d_2$).

Par exemple : $001 == 00101$, $0101 < 011$ et $10110 > 100010$, etc. Pour les signatures des cellules dans *DMLSH*, la propriété suivante est vérifiée :

***Propriété* 1.** *Pour deux signatures distinctes k_1 et k_2, nous avons soit $k_1 < k_2$ ou $k_1 > k_2$*

Démonstration. Considérons deux signatures différentes produites par DMLSH, avec $k_1 == k_2$ où k_1 est le préfixe de k_2. Dans ce cas, la cellule qui a la signature k_2 est incluse dans celle de k_1. Ceci est en contradiction avec le principe de partitionnement de l'espace produit par *DMLSH*. Alors $k_1 == k_2$ n'est pas possible dans notre cas, et à partir des définitions des opérateurs $==$, $>$ et $<$ vues précédemment, nous concluons que $k_1 < k_2$ ou $k_1 > k_2$. Par conséquent, les signatures produites par *DMLSH*, respectent un ordre total induit par l'opérateur $<$. □

BucketNumber	*NbHashesUsed*	*NbVectors*	*@*	*HV*

FIGURE 4.9 – Structure d'une entrée dans la feuille de l'arbre *DMLSH*

Chaque table de hachage est maintenue comme un arbre *DMLSH* de la manière suivante : chaque signature d'une cellule non-vide est considérée comme une clé, et toutes les clés sont insérées dans l'arbre. La figure 4.9 présente la structure d'une entrée dans une feuille de l'arbre *DMLSH*. Dans cette entrée, la clé est suivie par :

Algorithm 6 Construction de la structure d'index

for $i = 1$ to L **do**
  for $j = 1$ to M **do**
    Générer aléatoirement une fonction LSH $h_{i,j}$
  end for
  Créer un arbre vide DMLSH tree $Tree_i$
end for
for all $v \in S$ **do**
  $InsertVector(v)$ *//(Algorithm 7)*
end for

- Un compteur pour calculer le nombre de fonctions de hachage utilisées pour avoir cette signature : $NbHashUsed$.
- Un compteur indiquant le nombre de vecteurs (objets) contenus dans la cellule correspondante : $NbVectors$.
- L'adresse de la page contenant ces vecteurs : @.
- Un ensemble *HV* contenant les signatures complètes (avec M fonctions de hachage) des vecteurs de la cellule, en d'autres termes, les signatures des cellules *MLSH* non vides sont contenues dans la cellule *DMLSH*.

Construction de l'index *DMLSH*

L'algorithme 6 présente le processus de construction d'index *DMLSH*. Pour chacune des L tables de hachage, nous générons aléatoirement M fonctions *LSH* et un arbre *DMLSH* initialement vide. Ensuite, pour chaque vecteur v dans la base de données S, nous insérons v dans chacun des arbres *DMLSH* en utilisant l'algorithme 7 (*InsertVector*). Comme signalé précédemment, l'insertion d'un vecteur respecte la stratégie *DMLSH* qui génère des cellules en utilisant le moins possible de fonctions de hachage. L'utilisation d'une nouvelle fonction de hachage est nécessaire uniquement quand la taille de la cellule dépasse le seuil. Si les M fonctions sont déjà utilisées, nous stockons les nouveaux objets insérés dans des pages de débordement.

L'insertion d'un objet

L'algorithme 7 montre le processus d'insertion des objets. L'insertion d'un vecteur, se fait dans chacune des L tables de hachage. Pour chaque vecteur v, nous calculons sa valeur de hachage $g(v)$ en utilisant les M fonctions de hachage, ensuite, nous cherchons sa signature dans l'arbre $B+$ correspondant. Si un préfixe de $g(v)$ existe dans l'arbre, la fonction $Tree_{i.}Find(g(v))$ retourne l'entrée dans l'arbre qui correspond au préfixe de la signature. Il est possible que le préfixe n'existe pas dans l'arbre, puisque les cellules vides ne sont pas stockées dans l'arbre $B+$. Dans ce cas, l'entrée retournée est $NULL$, alors, nous ajoutons une entrée dans l'arbre avec le plus petit préfixe de $g(v)$ qui n'est pas un préfixe d'une autre signature. Cette étape est réalisée par la fonction $Tree_{i.}Insert(g'(v), item)$. La taille de ce préfixe

Algorithm 7 Insertion d'un vecteur : $InsertVector(v)$

```
for i = 1 to L do
  g(v) = (h_{i,1}(v), h_{i,2}(v), ..., h_{i,M}(v))
  item = Tree_i.Find(g(v))
  if item == NULL then
    item = NewLeafItem()
    NbHashesUsed = NbHash(v, i) //(algorithme 8)
    g'(v) = (h_{i,1}(v), h_{i,2}(v), ..., h_{i,NbHashesUsed}(v))
    Tree_i.Insert(g'(v), item)
  end if
  AddVectorToItem(v, item, i) //(algorithme 9)
end for
```

est calculée par la fonction *NbHash* (algorithme 8). Enfin, nous insérons le vecteur dans la cellule correspondante au préfixe trouvé (ou ajouté), et nous mettons à jour l'entrée d'index correspondante avec la fonction *AddVectorToItem* (algorithme 9).

Algorithm 8 Calcul du nombre des fonctions de hachage utilisées pour le vecteur v dans $Tree_i$: $NbHash(v,i)$

```
g(v) = (h_{i,1}(v), h_{i,2}(v), ..., h_{i,M}(v))
pred = Tree_i.FindPred(g(v))
succ = Tree_i.FindSucc(g(v))
lccPred = length(longestCommonPrefix(pred, g(v)))
lccSucc = length(longestCommonPrefix(succ, g(v)))
lcc = max(lccPred, lccSucc)
return lcc + 1
```

L'algorithme 8 montre la méthode utilisée pour calculer le nombre des fonctions de hachage à utiliser pour un nouveau vecteur v dont la signature globale (avec M fonctions) ne correspond à aucun préfixe de l'arbre. Dans un premier temps, nous calculons sa signature $g(v)$ en utilisant les M fonctions de hachage, ensuite, nous cherchons dans l'arbre la plus longue signature inférieure à $g(v)$: *pred*, et la plus petite signature supérieure à $g(v)$: *succ*. Dans un deuxième temps, *lcc*, la taille maximale du plus long préfixe commun entre $g(v)$ et *pred*/*succ* est calculée. Si *pred* ou *succ* n'existe pas, la taille du préfixe commun est 0. Enfin, nous retournons $lcc+1$, le nombre de fonctions de hachage à utiliser pour l'insertion du vecteur v ; ce sera la longueur de la plus courte signature (préfixe) qui se distingue des préfixes existants.

L'algorithme 9 montre la méthode utilisée pour l'insertion d'un vecteur v dans l'entrée d'index qui lui correspond. L'insertion d'un vecteur est possible uniquement dans deux cas : *(i)* quand le nombre de vecteurs de l'entrée est inférieur à l (avec $l = B/sizeof(v)$, et où B est la taille d'une page disque) ou *(ii)* si toutes les M fonctions sont utilisées. Cette insertion implique l'ajout de v dans la page @ et la signature globale $g(v)$ dans *HV*. Si l'insertion n'est pas possible, la cellule est trop pleine et sera divisée de la manière suivante : l'entrée est supprimée de l'arbre $B+$, et tous ses vecteurs sont réinsérés dans d'autres cellules, en utilisant une fonction de hachage supplémentaire (algorithme 10).

Exemple d'un index DMLSH :

Algorithm 9 L'ajout d'un vecteur v à l'entrée *item* de l'arbre i : $AddVectorToItem(v, item, i)$

```
if item.NbVectors < l or item.NbHashesUsed == M then
    item.NbVectors + +
    g(v) = (h_{i,1}(v), h_{i,2}(v), ..., h_{i,M}(v))
    Insérer g(v) dans item.HV
    Insérer v dans la page d'adresse item.@ ou dans une page de débordement
else
    Tree_i.Remove(item)
    for all v_j ∈ item do
        ReinsertVector(v_j, item.NbHashesUsed + 1, i) //(algorithme 10)
    end for
end if
```

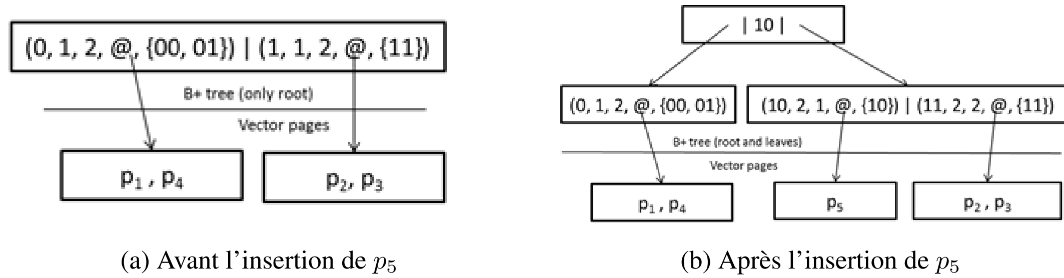
Algorithm 10 Réinsérer un vecteur v dans l'arbre i avec k fonctions de hachage : $ReinsertVector(v, k, i)$

```
g(v) = (h_{i,1}(v), h_{i,2}(v), ..., h_{i,k}(v))
item = Tree_i.Find(g(v))
if item == NULL then
    item = NewLeafItem()
    item.NbHashesUsed = k
    Tree_i.Insert(g(v), item)
end if
AddVectorToItem(v, item, i) //(algorithme 9)
```

Dans la figure 4.10, nous présentons un exemple avec une seule table de hachage. Pour simplifier, nous considérons un seuil $l = 2$ et une nombre de fonctions maximal $M = 2$. Initialement, nous avons quatre objets : p_1, p_2, p_3 et p_4, avec $h_1(p_1) = 0$, $h_1(p_2) = 1$, $h_1(p_3) = 1$ et $h_1(p_4) = 0$. La signature complète de chaque objet avec la fonction $g = (h_1, h_2)$ est : $g(p_1) = 00$, $g(p_4) = 01$, $g(p_2) = g(p_3) = 11$. La figure 4.10(a) montre l'index correspondant. En voulant insérer un objet p_5 avec $h_1(p_5) = 1$ et $g(p_5) = 10$, le compteur de la cellule 1 devient 3, ce qui est interdit (plus grand que l). L'action nécessaire, ici, est de diviser la cellule (en utilisant la deuxième fonction de hachage h_2), et par conséquent, nous réinsérons tous les objets de la cellule 1. La figure 4.10(b) présente le résultat de cette insertion : deux nouvelles cellules sont créées avec les clés 10 et 11 et insérées dans l'arbre $B+$. La cellule de signature 0 n'est pas divisée.

4.5.3 Recherche des k-plus proches voisins

Nous présentons dans cette section la méthode de la recherche des k-plus proches voisins en utilisant la structure d'index présentée précédemment.



(a) Avant l'insertion de p_5

(b) Après l'insertion de p_5

FIGURE 4.10 – Exemple d'index *DMLSH*

Algorithme

L'algorithme 11 montre la technique utilisée pour la recherche des k-ppv. Pour une requête q, nous répétons le processus suivant pour chaque table de hachage :

Nous construisons en mémoire un arbre $B+$ (*ProbedTree*), initialement vide, utilisé pour mémoriser les signatures des cellules accédées. Cet arbre a les mêmes propriétés qu'un arbre *DMLSH* (section 4.5.2), mais contient seulement les signatures des cellules. L'arbre $B+$ est ici utilisé pour accélérer la recherche si l'index devient très grand. Nous calculons la signature de q en utilisant les M fonctions de hachage : $g(q) = (h_1(q), ..., h_M(q))$ et nous générons la séquence d'accès aux cellules pour $g(v)$ en utilisant l'algorithme proposé par *MLSH* [Lv et al., 2007]. T est le nombre de cellules à interroger dans chaque table. Pour chaque cellule à interroger, nous cherchons la signature dans l'arbre ($ProbedTree_i$). Si son préfixe est trouvé, cela veut dire que la cellule à déjà été interrogée, alors nous ignorons cette étape. Dans le cas contraire, nous cherchons la clé dans l'arbre *DMLSH*. Si un préfixe pk de la signature est trouvé et $g(q)$ existe dans l'ensemble *HV*, alors la cellule qui correspond à la signature $g(q)$ n'est pas vide et elle est incluse dans la cellule *physique* de signature pk. Enfin, nous ajoutons les vecteurs qui se trouvent dans la page de signature pk à l'ensemble des candidats trouvés, et nous insérons pk dans l'arbre $B+$ $ProbedTree_i$ avec les cellules déjà interrogées. Après avoir trouvé tous les candidats, nous les trions dans l'ordre croissant de leurs distances à q, et nous retournons le *top-k*.

Exemple

Nous considérons ici, l'exemple de la figure 4.10(b). Nous supposons $g(q) = (h_1(q), h_2(q)) = 10$, $T = 3$, et la séquence suivante est générée : 10, 00, 01. L'arbre *DMLSH* est interrogé pour la signature 10, la page(@) de l'entrée pour 10 est chargée, p_5 est ajouté à la liste des candidats et la signature 10 est ajoutée à l'arbre des signatures interrogées ($ProbedTree$). Pour la deuxième signature 00, nous trouvons le préfixe 0 dans l'index. Puisque la signature 00 existe dans l'ensemble *HV* ($00 \in \{00, 01\}$), les objets p_1 et p_4 sont ajoutés à la liste des candidats. Pour la dernière signature 01, nous trouvons déjà son préfixe dans l'arbre $ProbedTree$, ce qui veut dire que la cellule est déjà accédée et donc nous n'avons pas besoin d'une entrée/sortie supplémentaire. Pour cet exemple, nous réalisons uniquement deux accès au lieu de trois pour *MLSH*.

Algorithm 11 Recherche des k-plus proches voisins

```
CS ← ∅ //(Initialisation de l'ensemble des candidats)
for i = 1 to L do
  Create a B+ tree ProbedTree_i in memory
  g(q) = (h_{i,1}(q), h_{i,2}(q), ..., h_{i,M}(q))
  Generate the probing sequence Probes[T]
  for j = 1 to T do
    found = ProbedTree_i.exists(Probes[j])
    if !found then
      item = Tree_i.Find(Probes[j])
      if item! = NULL and g(q) ∈ item.HV then
        CS = CS ∪ vectors(item.@)
        pk = item.BucketNumber
        ProbedTree_i.Insert(pk)
      end if
    end if
  end for
end for
trier CS par rapport à la distance de q
return top-k (k candidats les plus proches)
```

Propriétés

La méthode *DMLSH* que nous proposons possède deux propriétés importantes par rapport à la méthode originale *MLSH* :

- P_1 : Pour les mêmes paramètres, le nombre d'entrées/sorties (disque) réalisées pour *DMLSH* pour une requête q, est plus petit ou égal à celui réalisé par *MLSH*.
- P_2 : Toujours pour les mêmes paramètres, la rappel de la recherche des k-ppv obtenue par la méthode *DMLSH* pour une requête q est plus grand que celui obtenu par *MLSH*.

Il est simple de prouver ces deux propriétés. D'abord, P_1 est la conséquence du fait que les cellules *MLSH* occupent en principe des pages différentes, tandis que les cellules *DMLSH* (surtout voisines) peuvent partager le même préfixe et par conséquent, elles seront stockées dans la même page disque. Ensuite, P_2 vient du fait que l'ensemble des candidats produit par MLSH est un sous-ensemble de celui produit par la méthode *DMLSH*, puisque les cellules *MLSH* sont les mêmes ou incluses dans celles de *DMLSH*.

4.5.4 Évaluation expérimentale

Dynamic MultiProbe LSH est une variante de *MLSH* qui permet de réduire le nombre d'entrées/sorties disque. Nous présentons une évaluation expérimentale de la méthode *DMLSH*, comparée avec *MLSH*, où nous varions les paramètres suivants :

- Le nombre de fonctions de hachage M,
- Le nombre de tables de hachage L,
- Le nombre d'accès T.

La méthode *DMLSH* peut être aussi combinée avec la méthode basique *LSH* et ses variantes, en organisant chaque table de hachage comme un arbre DMLSH. Mais l'impact serait moins important que pour la méthode *MLSH*, puisqu'un seul accès est effectué pour chaque table de hachage, et qu'un grand nombre de tables est utilisé. Pour cette raison nous limitons notre étude à la méthode *MLSH*.

Données utilisées

Pour notre évaluation expérimentale, nous avons choisi deux ensembles de données très utilisées dans les expériences réalisées pour les méthodes présentées en état de l'art. Ces ensembles sont les suivants :

- **Couleur** : cet ensemble de données contient 68040 vecteurs de 32 dimensions représentant des histogrammes de couleur extraites des images de la collection *Corel* [1]. Les valeurs des dimensions sont des nombres réels dans l'intervalle $[0, 1]$ et avec un maximum de 6 décimales.
- **Audio** : cette base contient 54387 vecteurs de 192 dimensions. Ces données sont extraites de la collection LDC SWITCHBOARD-1 [2]. Les valeurs sont des nombres réels entre -1 et 1.

Nous augmentons la taille des deux ensembles de données en ajoutant des vecteurs *bruit* pour atteindre 1 million d'objets, et nous utilisons à chaque fois 100 vecteurs choisis aléatoirement, dans les bases de départ, comme exemples de requêtes.

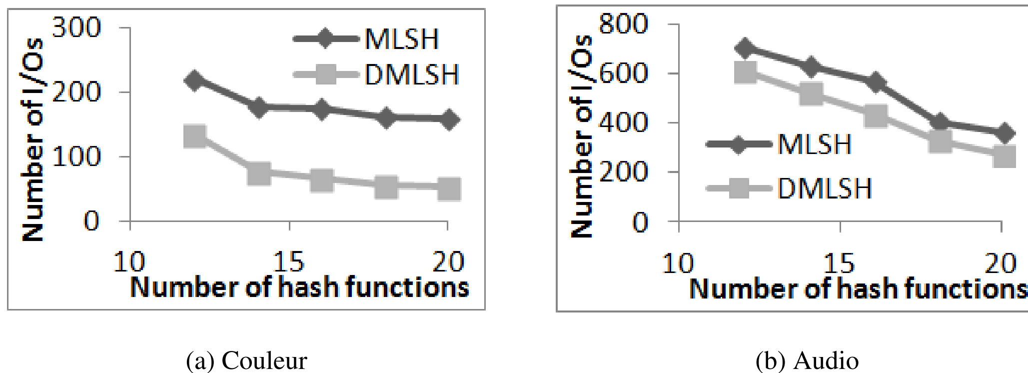
Métriques

Efficacité de la recherche
Comme tous les vecteurs sont stockés en mémoire secondaire (disque), l'efficacité de la recherche est mesurée par le coût des entrées/sorties. Dans ces expériences, nous considérons une taille de page disque égale à la taille de 100 vecteurs. La méthode *DMLSH* nécessite une charge *CPU* supplémentaire pour calculer les distances entre les vecteurs et la requête, car le nombre de candidats retournés est plus important que celui de *MLSH*. Les mesures montrent seulement une petite différence (environ 5%), négligeable par rapport au gain en termes d'entrées/sorties disque.

Qualité de la recherche
La qualité est mesurée par la valeur du *rappel* (*Recall*) qui est le rapport entre le nombre de réponses correctes trouvées et le nombre total de réponses correctes. Le résultat est la moyenne des 100 requêtes, avec $k = 20$. Pour une requête q, nous considérons $E(q)$ l'ensemble des vrais k-plus proches voisins et $F(q)$ l'ensemble des candidats trouvés dans le résultat k-ppv retourné. La valeur du *rappel* est définie de la façon suivante :

$$rappel = \frac{|E(q) \cap F(q)|}{|E(q)|} \tag{4.2}$$

[1] http ://kdd.ics.uci.edu/databases/CorelFeatures/
[2] http ://www.cs.princeton.edu/cass/audio.tar.gz



(a) Couleur

(b) Audio

FIGURE 4.11 – Effet de la variation de M sur le coût d'entrée/sorties

Dans notre cas particulier, en considérant que les k-ppv réels sont discernables (pas d'égalité de distances à la limite du *top-k*), nous avons $|E(q)| = k$ et $|E(q) \cap F(q)|$ est le nombre de vecteurs trouvés parmi les k.

Résultats expérimentaux

Variation du nombre des fonctions de hachage M

Nous mesurons la valeur du *rappel*, et le coût d'entrée/sortie pour les deux méthodes en variant le nombre maximal de fonctions pour chaque table de hachage. Pour les deux ensembles des données, nous considérons $L = 3$ et $T = 100$.

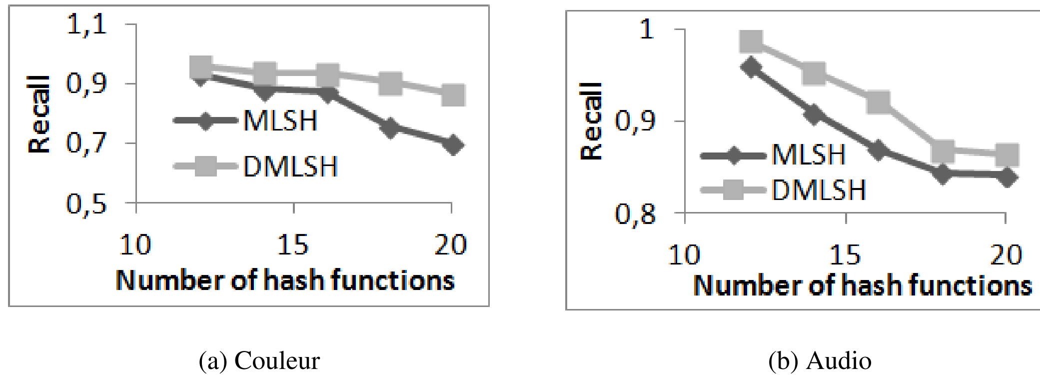
La figure 4.11 présente les résultats montrant l'impact de la valeur de M sur le coût d'entrées/sorties, alors que la figure 4.12 présente l'impact sur le *rappel*. Pour l'ensemble des données couleur, nous considérons $W = 0.6$, valeur utilisée par plusieurs travaux utilisant ces données [Lv et al., 2007] [Gan et al., 2012]. La méthode *DMLSH* réduit de 39% à 67% le coût d'entrées/sorties (figure 4.11(a)). La figure 4.12(a) montre que DMLSH permet d'augmenter de 3% à 23% la valeur du *rappel* par rapport à la méthode *MLSH*.

Concernant l'ensemble des données audio, nous considérons $W = 3.5$, valeur utilisée par plusieurs travaux utilisant ces données. La figure 4.11(b) montre que la méthode *DMLSH* permet de réduire le coût d'entrées/sorties de 13% à 25%, et la figure 4.12(b) montre une augmentation du *rappel* du résultat qui varie entre 3% et 6%

En termes de variation, nous remarquons qu'en augmentant le nombre de fonctions de hachage, le coût d'entrée/sortie et le *rappel* diminuent. Cela s'explique par le fait que pour chaque ajout de fonction de hachage, la taille des cellules baisse. Cela augmente les chances d'avoir des cellules regroupées dans une même page disque, mais diminue le nombre de candidats produits pour les valeurs de L et T inchangées.

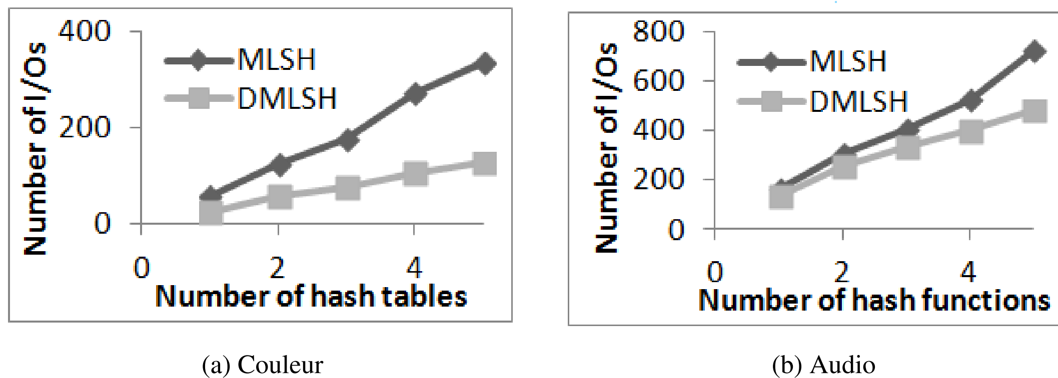
Variation du nombre de tables de hachage L

Les figures 4.13 et 4.14 montrent l'effet du nombre de tables de hachage sur le coût d'entrée/sortie et le *rappel* du résultat. T, le nombre d'accès par table est fixé ici à 100. Pour l'ensemble des données



(a) Couleur (b) Audio

FIGURE 4.12 – Effet de la variation de M sur le rappel



(a) Couleur (b) Audio

FIGURE 4.13 – Effet de la variation de L sur le coût d'entrée/sorties

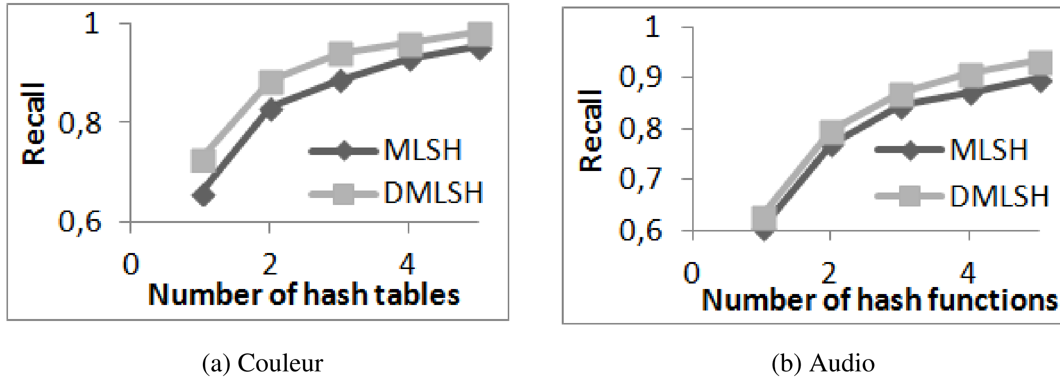
couleur, nous considérons $M = 14$ et $W = 0.6$. La méthode DMLSH permet de réduire le coût de 53% à 62% (figure 4.13(a)), et d'augmenter le *rappel* d'environ 3% à 10% (figure 4.14(a)).

Concernant l'ensemble des données audio, nous considérons $M = 18$ et $W = 3.5$. Les expériences montrent que DMLSH permet une réduction du coût d'entrées/sorties entre 16% et 33% (figure 4.13(b)), et une augmentation du *rappel* entre 1% et 9% (figure 4.14(b)).

Dans ces expériences, nous remarquons qu'en augmentant le nombre de tables de hachage, le coût d'entrées/sorties, ainsi que le *rappel* augmentent. Ceci est naturel, puisqu'en utilisant $L + 1$ tables de hachage, nous faisons plus d'accès disque et l'ensemble de candidats retrouvés est toujours un *sur-ensemble* de celui produit en utilisant L tables. Pour choisir le meilleur nombre de tables de hachage, il faut trouver le compromis entre l'efficacité, la qualité et la consommation d'espace mémoire. Les expériences montrent que pour le même *rappel*, notre méthode a besoin de moins de tables de hachage que la méthode *MLSH*, et par conséquent elle consomme moins de mémoire.

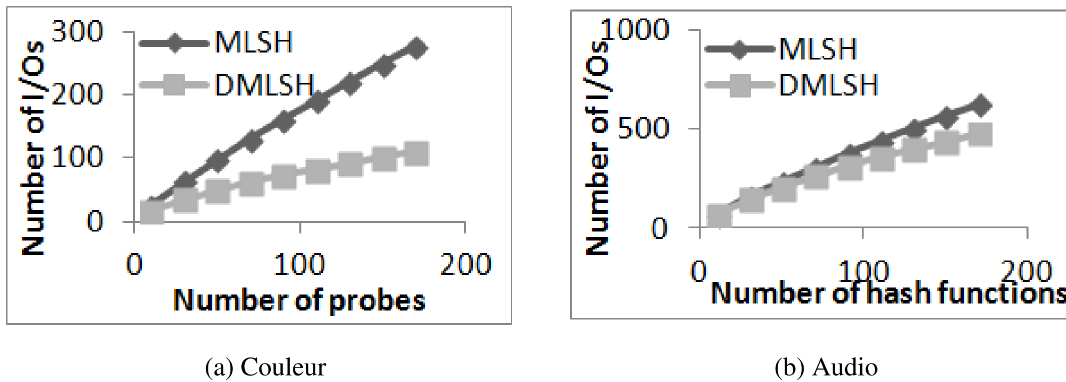
Variation du nombre d'accès T

Dans cette expérience, nous mesurons l'effet du nombre d'accès par table de hachage sur le coût total d'entrée/sortie et sur le *rappel*. Les figures 4.15 et 4.16 montrent les résultats pour les deux ensembles



(a) Couleur (b) Audio

FIGURE 4.14 – Effet de la variation de L sur le rappel



(a) Couleur (b) Audio

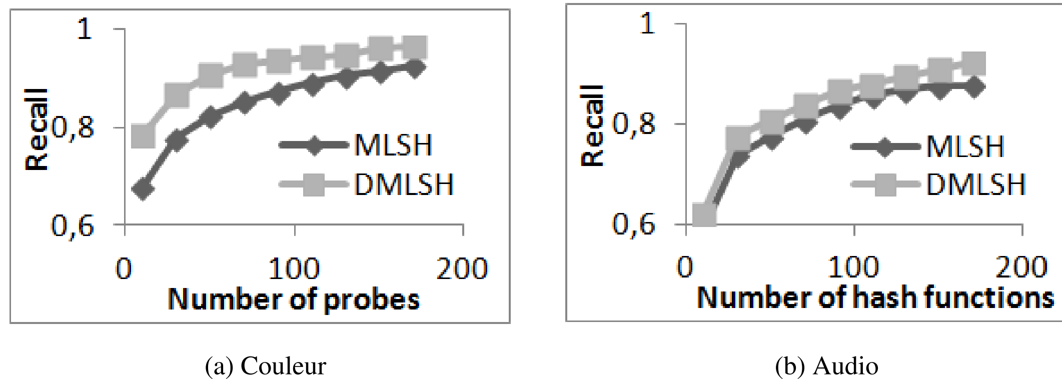
FIGURE 4.15 – Effet de la variation de T sur le coût d'entrée/sorties

de donnéess couleur et audio. Nous considérons des valeurs de T entre 10 et 170, avec $L = 3$. Pour l'ensemble des données couleur, nous considérons $M = 14$ et $W = 0.6$. La réduction du coût d'entrée/sortie que permet la méthode DMLSH varie de 33% à 60%. Plus T augmente, plus le gain de coût d'entrées/sorties augmente, figure 4.15(a). Notre méthode permet aussi une augmentation du *rappel* de 4% à 16% (figure 4.16(a)).

Concernant l'ensemble des données audio, nous fixons $M = 18$ et $W = 3.5$. *DMLSH* permet de réduire le coût d'entrée/sortie de 2% à 24% (figure 4.15(b)). Le *rappel* augmente de 3% à 5% pour notre méthode (figure 4.16(b)), et pour la même qualité nous avons besoins de moins d'accès que la méthode *MLSH*.

4.6 Conclusion

Dans ce chapitre, nous avons, dans un premier temps, proposé *MSA*, une nouvelle technique de recherche approximative des k plus proches voisins dans le cadre de la recherche d'images par similarité du contenu (CBIR) dans des bases de grande taille. Dans cette méthode, nous nous sommes inspirés des techniques de traitement des requêtes *top-k* multicritères. *MSA* utilise une structure d'index très simple qui offre un bon compromis entre la rapidité de la recherche, les besoins de stockage et la rapidité des mises à jour. L'al-



(a) Couleur

(b) Audio

FIGURE 4.16 – Effet de la variation de T sur le rappel

gorithme *MSA* effectue une recherche approximative ϵ-exclusive des k plus proches voisins, permettant d'avoir des garanties concernant les faux négatifs, avec une émergence rapide de bonnes approximations, améliorées de façon monotone et pouvant conduire à la solution exacte si nécessaire.

Les expériences réalisées sur une base de 1 million d'images, avec implémentation de l'index sur disque, ont permis de confirmer expérimentalement l'émergence rapide de bonnes approximations avec les algorithmes *MSA*, obtenues trois à quatre fois plus rapidement qu'avec la méthode brute. Nous avons aussi comparé les performances des algorithmes *MSA* avec celles de la méthode *MLSH* adaptée à notre contexte. Dans cette comparaison, nous avons montré que pour le même temps d'exécution, la qualité du résultat approximatif des algorithmes *MSA*, mesuré par le *mAP*, est supérieure à celle de *MLSH*. Nous avons analysé aussi l'impact du nombre de tables hachage L et du nombre d'accès à chaque table T sur la valeur du *mAP*. Ces expériences ont montré que la qualité du résultat approximatif s'améliore en augmentant la valeur de T dans la première expérience, et en augmentant la valeur de L dans la deuxième. Cependant, le *mAP* reste stable à partir de valeurs incluses dans nos mesures ($L = 4$, et $T = 12$) et ne s'améliore pas pour des valeurs de L et T qui donnent des temps d'exécution raisonnables.

Dans la deuxième partie de ce chapitre, nous avons présenté la méthode *DMLSH*, qui peut être considérée comme une variante de *MLSH* plus efficace en terme d'entrées/sorties disque. Elle utilise un nombre de fonctions de hachage différent pour chaque cellule, afin d'adapter chacune d'entre elles à la taille d'une page disque. Pour la construction de l'index, la méthode *DMLSH* commence par l'utilisation d'une seule fonction de hachage et quand la taille de la cellule dépasse la taille d'une page disque, une nouvelle fonction est rajoutée. Dans une table de hachage, les cellules ne sont pas construites par le même nombre de fonctions de hachage, ceci, permet d'avoir des signatures de tailles différentes. Les signatures des cellules sont indexées dans un arbre $B+$ légèrement modifié pour accélérer la recherche. Pour une requête donnée, nous commençons par générer la séquence des cellules à accéder, et ensuite les interroger. Comme plusieurs cellules se partagent le même préfixe de signature et donc, sont stockées dans la même page disque, nous avons besoin seulement d'un accès disque pour retourner le contenu de ces cellules. Par conséquent, le nombre total d'entrées/sorties est réduit par rapport à *MLSH*. Au même temps, l'ensemble des candidats retournés est toujours un sur-ensemble de celui produit par *MLSH*, ce qui explique une meilleure qualité du résultat approximatif retourné.

Un expert, c'est quelqu'un qui a fait toutes ses erreurs dans un champ réduit d'applications.

– Niels Bohr

Conclusion

Dans cette thèse nous nous sommes intéressés au traitement des requêtes de classement *top-k*, qui est un axe très actif dans le domaine de la recherche d'information. Plusieurs applications nécessitent le traitement de requêtes *top-k* multicritères, telles que la recherche dans le web, la recherche par similarité du contenu dans les bases de données multimédia, etc. Nous nous sommes focalisés dans un premier temps sur la recherche *top-k* dans un contexte général, où l'accès aux scores est très coûteux. Nous avons proposé un cadre général *GF* permettant d'exprimer n'importe quel algorithme *top-k* dans notre contexte, et un nouvel algorithme *BR*. Nous avons également exploré la recherche *top-k* approximative en se basant sur la stratégie d'arrêt prématuré de l'exécution. Ensuite nous nous sommes placés dans le contexte de l'application de la recherche multicritères à la recherche des k-plus proches voisins dans les bases de données multimédia, et particulièrement les bases d'images. En s'inspirant des algorithmes *top-k*, nous avons proposé *MSA*, une nouvelle technique permettant une recherche approximative des k-ppv. Toujours dans ce contexte, nous avons contribué à une amélioration d'une des méthodes de référence dans la recherche approximative des k-ppv, *Multi-Probe LSH* dans un contexte de stockage sur disque.
Dans ce chapitre, nous résumons nos principales contributions réalisé pendant la thèse, et nous discutons quelques perspectives.

Bilan

Nous avons commencé dans un premier temps par passer en revue les techniques de traitement de requêtes *top-k* multicritères dans un cadre général. D'abord, nous avons présenté les différents paramètres qui distinguent les diverses catégories de la recherche *top-k* tel que le type de la requête, la fonction d'agrégation, le niveau d'exécution, le type d'accès aux données, etc. Ensuite, nous avons défini le cadre que nous avons considéré : requêtes *top-k* de sélection, cas générique pour les types d'accès aux scores, traitement de la requête en dehors du moteur de la base de données, données certaines avec résultat exact, et une fonction de classement monotone. Nous avons aussi défini le modèle de coût considéré, avec des critères permettant différents classement des mêmes données, le coût d'accès aux scores étant l'opération la plus coûteuse. Ensuite, nous avons présenté les techniques utilisées pour adapter les algorithmes *top-k* à un contexte de résultat approximatif. L'objectif dans ce cadre est de trouver un compromis entre le coût d'accès et la qualité du résultat retourné.

Dans la deuxième partie du premier chapitre, nous avons considéré un contexte particulier de la recherche multicritères, la recherche basée sur la similarité du contenu dans les bases de données multimédia, en particulier les bases d'images, l'objectif étant de trouver les k images les plus similaires à une image requête. Les images sont ici décrites par des vecteurs numériques représentant des signatures. Trouver le *top-k* dans ce contexte consiste à retourner les k-plus proches voisins dans un espace multidimensionnel, en se basant sur une notion de distance entre les signatures des images de la base et la signature de la requête. Dans ce cadre, nous avons focalisé notre étude sur la recherche approximative des k-ppv, et nous avons présenté principalement la technique *LSH* et ses variantes tel que *Multi-Probe LSH*, techniques de référence dans ce contexte.
Dans le reste du manuscrit, nous avons présenté notre contribution en trois chapitres.

Dans le deuxième chapitre, nous avons commencé par proposer un nouveau cadre générique *GF* permettant d'exprimer n'importe quel algorithme de traitement de requêtes *top-k* de sélection. *GF* considère un algorithme *top-k* comme une suite d'accès aux sources permettant à chaque fois de découvrir un score local d'un seul objet. Nous nous sommes basés sur *GF* pour exprimer les algorithmes de l'état de l'art, et nous avons donné plusieurs exemples d'algorithmes *top-k* exprimé dans *GF*. Ensuite, nous avons présenté une nouvelle technique, *Breadth-Refine*, pour le traitement de requêtes *top-k* se basant sur une nouvelle stratégie de type *breadth-first* (en largeur d'abord). Cette stratégie considère le *top-k* comme un tout, et essaye à chaque étape de maintenir le meilleur *top-k* possible en évaluant le candidat ayant l'intervalle de score le moins précis plutôt que le meilleur candidat. Nous avons présenté, dans un premier temps, trois variantes de *BR*. *BR-Basic* correspond à la version basique de *BR*, *BR-First* utilise la méthode classique pour choisir le candidat à évaluer (choisir toujours le meilleur candidat). La comparaison de ces deux algorithmes nous a permis de comparer concrètement notre stratégie à celle utilisée habituellement. La troisième variante proposée est *BR-Cost*, elle inclut une approche qui ressemble à celle utilisée dans l'algorithme *CA* pour réduire le nombre d'accès directs. Ensuite nous avons visé la comparaison de notre stratégie avec d'autres stratégies génériques.

En l'absence d'algorithmes génériques comparables, nous avons proposé une variante de *SR/G*, l'algorithme de référence de *NC*, dans le contexte *BR* (objets avec des scores incomplets dans le *top-k* final). Nous avons proposé aussi une variante générique de l'algorithme *CA* que nous avons appelé *CA-gen*. Enfin, nous avons proposé *BR-Cost**, une alternative de l'algorithme de *BR-Cost*, dans laquelle nous avons adopté une nouvelle stratégie pour calculer le ratio r en le considérant comme un rapport des bénéfices et non plus un rapport de coûts d'accès.
A la fin du chapitre, nous avons présenté une évaluation expérimentale très riche, dans laquelle nous avons montré la capacité des algorithmes *BR* à s'adapter à différentes configurations de type et de coût d'accès aux sources. Nous avons montré que *BR-Cost* présente à chaque fois la meilleure performance dans des cas favorables aux algorithmes de l'état de l'art. Cette comparaison a montré aussi la supériorité de la stratégie *breadth-first* en comparant les algorithmes *BR-First* et *BR-Basic*, et l'impact de la prise en considération de la différence de coût d'exécution en montrant que *BR-Cost* est plus performant que *BR-Basic* quand l'accès direct est plus coûteux que l'accès séquentiel. Ensuite, nous avons considéré un cadre générique, dans lequel, nous avons proposé une première comparaison des algorithmes *top-k* génériques, dans laquelle l'algorithme *BR-Cost** a montré sa supériorité par rapport aux variantes de *NC* et *CA*.

Le troisième chapitre présente une étude du potentiel d'approximation des algorithmes *top-k*, en se basant sur la technique d'approximation par arrêt prématuré de l'exécution. Nous avons proposé une

généralisation de la θ-approximation dans le cadre *GF*, ensuite, nous avons présenté une comparaison expérimentale des algorithmes *top-k* en se basant sur deux ensembles d'approximation : *(i)* les k meilleurs candidats dans l'ordre décroissant des scores maximum ($\mathcal{U}_k$), et *(ii)* les k meilleurs candidats dans l'ordre décroissant des scores minimum ($\mathcal{L}_k$). En comparant les courbes coût-distance, nous avons montré que la stratégie *BR* présente globalement le meilleur potentiel d'approximation, avec un avantage conséquent par rapport aux autres algorithmes dans le cadre d'une approximation avec $\mathcal{U}_k$, l'ensemble sur lequel se basent tous les algorithmes *top-k* pour décider le type d'accès à effectuer et le candidat à évaluer. Cependant, l'approximation avec $\mathcal{L}_k$ produit de meilleurs résultats pour tous les algorithmes *top-k*, et réduit considérablement la différence entre eux.

Dans le dernier chapitre, nous avons dans un premier temps proposé *MSA*, une nouvelle technique de recherche approximative des k-plus proches voisins dans le cadre de la recherche d'images par similarité du contenu (CBIR) dans des bases de grande taille (sur disque). La méthode *MSA* s'inspire des algorithmes *top-k*. *MSA* utilise une structure d'index très simple qui offre un bon compromis entre la rapidité de la recherche, les besoins de stockage et la rapidité des mises à jour. L'algorithme *MSA* effectue une recherche approximative ϵ-exclusive des k plus proches voisins, permettant d'avoir des garanties concernant les faux négatifs, avec une émergence rapide de bonnes approximations, améliorées de façon monotone et pouvant conduire à la solution exacte si nécessaire.
Les expériences réalisées sur une vraie base d'images ont permis de confirmer expérimentalement l'émergence rapide de bonnes approximations avec les algorithmes *MSA*, obtenues trois à quatre fois plus rapidement que avec la méthode brute. Nous avons aussi comparé les performances des algorithmes *MSA* avec celles de la méthode *Multi-Probe LSH* adaptée à notre contexte. Dans cette comparaison, nous avons montré que pour le même temps d'exécution, la qualité du résultat approximatif des algorithmes *MSA*, mesuré par le *mAP* (mean Average Precision), est supérieure à celle de *Mulit-Probe LSH*.

Dans la deuxième partie de ce chapitre, nous avons présenté la méthode *Dynamic Multi-Probe LSH* (*DMLSH*), qui peut être considérée comme une amélioration de *MLSH* dans un contexte de grande base de données sur disque. *DMLSH* utilise un nombre de fonctions de hachage différent pour chaque cellule, afin d'adapter chacune d'entre elles à la taille d'une page disque. Pour la construction de l'index, la méthode *DMLSH* commence par l'utilisation d'une seule fonction de hachage et quand la taille de la cellule dépasse la taille d'une page disque, une nouvelle fonction est rajoutée. Dans une table de hachage, les cellules ne sont pas construites par le même nombre de fonctions de hachage, ceci, permet d'avoir des signatures de tailles différentes. Les signatures des cellules sont indexées dans un arbre $B+$ légèrement modifié pour accélérer la recherche. Les résultats des évaluations expérimentales sur deux bases de données (couleur et audio) ont montré d'abord que la méthode *DMLSH* permet un meilleur rappel pour différentes configurations des paramètres de nombre de tables de hachage, nombre d'accès à une table, etc. Ces expériences ont montré aussi, une importante réduction du nombre des entrées/sorties disque pour la méthode *DMLSH* par rapport à la méthode *MLSH*.

Résumé des contributions

Les contributions apportées pendant cette thèse peuvent être résumées en plusieurs points :

- Un cadre générique permettant d'exprimer n'importe quel algorithme *top-k*, qui peut servir de base d'analyse et de comparaison des différentes stratégies utilisées dans les algorithmes.

- Une nouvelle stratégie de recherche du *top-k Breadth-Refine* (*BR*), considérant l'ensemble des candidats du *top-k* comme un tout, en essayant d'avoir à chaque étape le meilleur *top-k* possible. *BR* est un algorithme générique qui s'adapte à n'importe quelle configuration de types d'accès aux scores.

- Une comparaison expérimentale de *BR* avec des algorithmes *top-k* spécifiques, montrant la supériorité de *BR* et sa capacité d'adaptation à des configurations de type d'accès spécifiques.

- Une première comparaison expérimentale entre des techniques génériques, montrant l'efficacité de notre stratégie par rapport aux méthodes de l'état de l'art adaptées au contexte *BR*.

- Une adaptation des algorithmes *BR* pour une recherche approximative du *top-k*, et une généralisation de cette approche au cadre *GF*. Une première étude expérimentale permettant d'évaluer et de comparer le potentiel d'approximation des algorithmes *top-k* par arrêt prématuré.

- Une application des algorithmes top-k dans le cadre de la recherche par similarité du contenu dans les bases de données multimédia, et proposition d'un algorithme pour la recherche approximative des k-plus proches voisins dans une grande base sur disque (*MSA*). Des évaluations expérimentales montrant la supériorité de *MSA* par rapport à la méthode brute et une adaptation de la méthode *Multi-Probe LSH*.

- Une amélioration de la méthode *MultiProbe LSH* dans un contexte où l'utilisation d'une mémoire secondaire est nécessaire (grands volumes de données), avec optimisation du nombre d'entrées/sorties disque.

Perspectives

Plusieurs perspectives s'ouvrent suite à ces travaux. Nous pouvons les classer par rapport aux contributions de la thèse.

- La méthode *Breadth-Refine* permettant le traitement de requêtes *top-k* multicritères de sélection dans un cadre général.
  Dans ce contexte, plusieurs travaux futurs peuvent être menés :

  - Une expérimentation plus approfondie des algorithmes *BR*, sur des données réelles, permettra de mieux évaluer la stratégie *breadth-first* dans un cas plus concret.
  - Exploration de nouvelles variante de *BR*, notamment un calcul dynamique (variable dans le temps) du rapport r entre le bénéfice des accès séquentiels et directs (variante *BR-Cost**).
  - Exploration des variantes parallèles de *BR*, basées sur un accès en parallèle aux sources, et adaptation à une approche *Map-Reduce* dans le *cloud*.
  - Exploration des variantes *BR* avec des contraintes différentes, par exemple permettant d'accéder plusieurs éléments à la fois pour un accès comme dans *TPUT* [Cao and Wang, 2004].
  - Intégration des algorithmes *BR* dans un moteur de base de données, en définissant l'interaction avec les autres opérateurs et les techniques d'optimisation de requêtes.
  - Adaptation de *BR* à d'autres types de requêtes *top-k* (requêtes de jointure ou d'agrégation)[Natsev et al., 2001] [Ilyas et al., 2004] [Ré et al., 2007], ou encore à des données imprécises [Zhan et al., 2012].

- La méthode *MSA* pour la recherche par similarité du contenu dans les bases de données images, et la méthode *DMLSH* permettant une amélioration de la technique de recherche des k-ppv, MLSH, dans un espace multidimensionnel.
  Dans ce contexte, plusieurs travaux futurs peuvent être menés :
  - Une nouvelle mesure sur des bases de données de très grande taille.
  - Une comparaison avec d'autres méthodes k-ppv approximatives de l'état de l'art (*VA-File*, *Spaces-Filling Curves*, etc).
  - Une comparaison avec la méthode *DMLSH*, mieux adaptée que *MLSH* au disque.
  - Un comparaison plus générale, en incluant le temps de création et de mise à jour de l'index.
- Pour l'adaptation des algorithmes *top-k* aux données multimédia, d'autres travaux futurs peuvent être réalisés :
  - Adaptation des algorithmes *top-k* à une recherche approximative basée sur des index multimédia retournant des informations imprécises (une zone dans l'espace, et non une localisation précise).
  - Adaptation des algorithmes *top-k* à des applications nécessitant une recherche multimédia hétérogène, en explorant des structures d'index spécifiques (en tant que sources triées dans les algorithmes *top-k*) pour des critères liés au texte, données géographiques, etc.

Publications personnelles

Badr, M. and Vodislav, D. (2011b). A general top-k algorithm for web data sources. In *Proceedings of the 22nd international conference on Database and expert systems applications - Volume Part I*, DEXA'11, pages 379–393

Badr, M. and Vodislav, D. (2011a). Breadth-first strategies for top-k algorithms over web data sources. In 27èmes *journées Bases de Données Avancées*

Badr, M. and Vodislav, D. (2013). Generic top-k query processing with breadth-firststrategies. In *Proceedings of the 24nd international conference on Database and EXpert systems Applications*, DEXA'13

Yin, S., Badr, M., and Vodislav, D. (2013). Dynamic multi-probe lsh: an i/o efficient index structure for approximate nearest neighbor search. In *Proceedings of the 24nd international conference on Database and expert systems applications*, DEXA'13

Badr, M., Vodislav, D., Picard, D., Yin, S., and Gosselin, P.-H. (2013). Multi-criteria search algorithm: an efficient approximate k-nn algorithm for image retrieval. In *IEEE International Conference on Image Processing*, ICIP'13

Bibliographie

Abiteboul, S., Kanellakis, P., and Grahne, G. (1987). On the representation and querying of sets of possible worlds. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, pages 34–48, New York, NY, USA. ACM. 26

Akbarinia, R., Pacitti, E., and Valduriez, P. (2007). Best position algorithms for top-k queries. *VLDB*, pages 495–506. 28

Amato, G., Rabitti, F., Savino, P., and Zezula, P. (2003a). Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inf. Syst.*, 21(2) :192–227. 26

Amato, G., Rabitti, F., Savino, P., and Zezula, P. (2003b). Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inf. Syst.*, 21(2) :192–227. 36

Badr, M. and Vodislav, D. (2011a). Breadth-first strategies for top-k algorithms over web data sources. In 27èmes *journées Bases de Données Avancées*.

Badr, M. and Vodislav, D. (2011b). A general top-k algorithm for web data sources. In *Proceedings of the 22nd international conference on Database and expert systems applications - Volume Part I*, DEXA'11, pages 379–393.

Badr, M. and Vodislav, D. (2013). Generic top-k query processing with breadth-firststrategies. In *Proceedings of the 24nd international conference on Database and EXpert systems Applications*, DEXA'13.

Badr, M., Vodislav, D., Picard, D., Yin, S., and Gosselin, P.-H. (2013). Multi-criteria search algorithm : an efficient approximate k-nn algorithm for image retrieval. In *IEEE International Conference on Image Processing*, ICIP'13.

Barbará, D., Garcia-Molina, H., and Porter, D. (1992). The management of probabilistic data. *IEEE Trans. on Knowl. and Data Eng.*, 4(5) :487–502. 26

Bawa, M., Condie, T., and Ganesan, P. (2005). Lsh forest : self-tuning indexes for similarity search. In *WWW*. 39

Beis, J. S. and Lowe, D. G. (1997). Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, pages 1000–, Washington, DC, USA. IEEE Computer Society. 38

Bellman, R. and Kalaba, R. (1959). On adaptive control processes. In *Automatic Control, IRE Transactions on In Automatic Control, IRE Transactions on*. 38

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9) :509–517. 38

Berchtold, S., Keim, D. A., and Kriegel, H.-P. (1996). The x-tree : An index structure for high-dimensional data. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 28–39, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 38

Beyer, K. S., Goldstein, J., Ramakrishnan, R., and Shaft, U. (1999). When is "nearest neighbor" meaningful ? In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 217–235, London, UK, UK. Springer-Verlag. 36

Böhm, C., Berchtold, S., and Keim, D. A. (2001). Searching in high-dimensional spaces : Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3) :322–373. 14, 38, 39

Bruno, N., Gravano, L., and Marian, A. (2002). Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–380. 22, 25, 31, 32

Börzsönyi, S., Kossmann, D., and Stocker, K. (2001). The skyline operator. In *ICDE*, pages 421–430. 27

Buhler, J. (2001). Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5) :419–428. 42

Burghouts, G. J. and Geusebroek, J.-M. (2009). Performance evaluation of local colour invariants. *Comput. Vis. Image Underst.*, 113(1) :48–62. 37

C, L., E. Y, C., Garcia-Molina, H., and Wiederhold, G. (2002). Clindex : Approximate similarity queries in high-dimensional spaces,. In *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, pages 792–808, Washington, USA. IEEE Computer Society. 38

Cao, P. and Wang, Z. (2004). Efficient top-k query calculation in distributed networks. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 206–215, New York, NY, USA. ACM. 28, 120

Carrilero, A. C. (1999). Les espaces de représentation de la couleur. *Technical Report 99D006*. 37

Chang, K. C.-C. and Hwang, S.-w. (2002). Minimal probing : supporting expensive predicates for top-k queries. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 346–357, New York, NY, USA. ACM. 25, 32, 34

Cheng, R., Kalashnikov, D. V., and Prabhakar, S. (2004). Querying imprecise data in moving object environments. *IEEE Trans. on Knowl. and Data Eng.*, 16(9) :1112–1127. 26

Ciaccia, P. and Patella, M. (2000). Pac nearest neighbor queries : Approximate and controlled search in high-dimensional and metric spaces. In *ICDE*, pages 244–255. 36

Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree : An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 38

Comer, D. (1979). Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2) :121–137. 106

Considine, J., Li, F., Kollios, G., and Byers, J. (2004). Approximate aggregation techniques for sensor databases. In *Proceedings of the 20th International Conference on Data Engineering*, pages 449–, Washington, DC, USA. IEEE Computer Society. 26

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, New York, NY, USA. ACM. 41, 42

de Almeida, V. T. and Güting, R. H. (2005). Supporting uncertainty in moving objects in network databases. In *Proceedings of the 13th annual ACM international workshop on Geographic information systems*, GIS '05, pages 31–40, New York, NY, USA. ACM. 26

de Sande, K. E. A. V., Gevers, T., and Snoek, C. G. M. (2008). Evaluation of color descriptors for object and scene recognition. In *CVPR*. 37

de Sande, K. E. A. V., Gevers, T., and Snoek, C. G. M. (2010). Evaluation of color descriptors for object and scene recognition. 37

Donjerkovic, D. and Ramakrishnan, R. (1999). Probabilistic optimization of top n queries. In *VLDB*, pages 411–422, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 36

Fagin, R., Kumar, R., and Sivakumar, D. (2003). Efficient similarity search and classification via rank aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 301–312, New York, NY, USA. ACM. 26, 38, 39

Fagin, R., Lotem, A., and Naor, M. (2001). Optimal aggregation algorithms for middleware. In *PODS*. 22, 24, 30, 32, 35, 56, 95

Fagin, R., Lotem, A., and Naor, M. (2002). Optimal aggregation algorithms for middleware. *CoRR*, cs.DB/0204046. 22, 25, 28, 35, 76

Fournier, J., Cord, M., and Philipp-Foliguet, S. (2001). Retin : A content-based image indexing and retrieval system. *Pattern Analysis and Applications Journal, Special issue on image indexation*, 4 :153–173. 37

Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. (1999). Learning probabilistic relational models. In *In IJCAI*, pages 1300–1309. Springer-Verlag. 26

Fuhr, N. (1990). A probabilistic framework for vague queries and imprecise information in databases. In *VLDB*, pages 696–707. Morgan. 26

Gan, J., Feng, J., Fang, Q., and Ng, W. (2012). Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 541–552, New York, NY, USA. ACM. 43, 113

Gionis, A., Indyk, P., and Motwani, R. (1999). Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 38, 39, 42, 45

Güntzer, U., Balke, W.-T., and Kießling, W. (2001). Towards efficient multi-feature queries in heterogeneous environments. In *In Proc. of the IEEE International Conference on Information Technology : Coding and Computing (ITCC 2001)*, pages 622–628, LAS VEGAS, USA. 22, 25, 29, 55, 56

Güntzer, U., Balke, W.-T., and Kießling, W. (2000). Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428. 22, 25, 31, 95

Gorisse, D. (2010). *Passage à l'échelle des méthodes de recherche sémantique dans les grandes bases d'images*. Thèse de doctorat, Université de Cergy-Pontoise. 11, 41

Griebel, M. and Zumbusch, G. (2002). Hash based adaptive parallel multilevel methods with space-filling curves. *NIC Series*, 9 :479–492. 38

Guttman, A. (1984). R-trees : a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2) :47–57. 38

Heinrichs, A., Koubaroulis, D., Levienaise-Obadia, B., Rovida, P., and Jolion, J.-M. (1959). Image indexation and content based search using pre-attentive similarities. In *RIAO*, pages 1–9. 37

Hwang, S. and Chang, K. C.-C. (2007). Optimizing top-k queries for middleware access : A unified cost-based approach. *ACM Trans. Database Syst.*, 32(1) :5. 25, 33, 51, 60

Ilyas, I. F., Aref, W. G., and Elmagarmid, A. K. (2004). Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3) :207–221. 22, 25, 27, 120

Ilyas, I. F., Beskales, G., and Soliman, M. A. (2008). A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4). 21

Indyk, P. and Motwani, R. (1998). Approximate nearest neighbors : towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, New York, NY, USA. ACM. 39, 40, 91

Jegou, H., Perronnin, F., Douze, M., Sannchez, J., Perez, P., and Schmid, C. (2012). Aggregating local image descriptors into compact codes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34(9) :1704–1716. 97

Katayama, N. and Satoh, S. (1997). The sr-tree : an index structure for high-dimensional nearest neighbor queries. *SIGMOD Rec.*, 26(2) :369–380. 38

Lakshmanan, L. V. S., Leone, N., Ross, R., and Subrahmanian, V. S. (1997). Probview : A flexible probabilistic database system. *ACM TRANSACTIONS ON DATABASE SYSTEMS*, 22(3) :419–469. 26

Lejsek, H., Þór, B., and Amsaleg, J. L. (2011). Nv-tree : Nearest neighbors at the billion scale. In *In proceedings of ICMR*, pages 1–54. 91

Li, C., Chang, K. C.-C., and Ilyas, I. F. (2006). Supporting ad-hoc ranking aggregates. In *SIGMOD Conference*, pages 61–72. 23

Li, C., Chang, K. C.-C., Ilyas, I. F., and Song, S. (2005). Ranksql : Query algebra and optimization for relational top-k queries. In *SIGMOD Conference*, pages 131–142. 13, 25, 26

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2) :91–110. 37

Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K. (2007). Multi-probe lsh : efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 950–961. VLDB Endowment. 38, 39, 40, 42, 45, 110, 113

Ma, W. Y. and Manjunath, B. S. (1997). Netra : a toolbox for navigating large image databases. In *ICIP*, pages 568–, Washington, DC, USA. IEEE Computer Society. 37

Mamoulis, N., Yiu, M. L., Cheng, K. H., and Cheung, D. W. (2007). Efficient top-k aggregation of ranked inputs. *ACM Trans. Database Syst.*, 32(3). 28

Marian, A., Bruno, N., and Gravano, L. (2004). Evaluating top-*k* queries over web-accessible databases. *ACM Trans. on Database Syst.*, 29(2) :319–362. 25, 32

Michel, S., Triantafillou, P., and Weikum, G. (2005). Klee : A framework for distributed top-k query algorithms. *VLDB*, pages 637–648. 28

Mikolajczyk, K., Tuytelaars, T., Schmid, C., Zisserman, A., Matas, J., Schaffalitzky, F., Kadir, T., and Gool, L. V. (2005). A comparison of affine region detectors. *Int. J. Comput. Vision*, 65(1-2) :43–72. 91

Natsev, A., Chang, Y.-C., Smith, J. R., Li, C.-S., and Vitter, J. S. (2001). Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 281–290, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 22, 27, 36, 120

Negrel, R., Picard, D., and Gosselin, P. (2012). Compact tensor based image representation for similarity search. In *ICIP*. 91, 97

Oliva, A. and Torralba, A. (2006). Building the gist of a scene : the role of global image features in recognition. In *Progress in Brain Research*. 37

Philipp-Foliguet, S., Logerot, G., Constant, P., Gosselin, P. H., and Lahanier, C. (2006). Multimedia indexing and fast retrieval based on a vote system. In *Proceedings of the 2006 IEEE International Conference on Multimedia and Expo, ICME 2006, July 9-12 2006, Toronto, Ontario, Canada*, pages 1781–1784. IEEE. 26

Ré, C., Dalvi, N., and Suciu, D. (2007). Efficient top-k query evaluation on probabilistic data. In *in ICDE*, pages 886–895. 23, 120

Shustek, L., Friedman, J., and Baskett, F. (2006). An algorithm for finding nearest neighbors. In *Computing, IEEE Transactions*, pages 1000–1006, Washington, USA. IEEE Computer Society. 38, 39

Sivic, J. and Zisserman, A. (2003). Video google : A text retrieval approach to object matching in videos. In *ICCV*, pages 1470–1477. 37, 91

Smeulders, A. W. M., Worring, M., Santini, S., Gupta, A., and Jain, R. (2000). Content-based image retrieval at the end of the early years. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 22(12) :1349–1380. 37, 39

Tao, Y., Yi, K., Sheng, C., and Kalnis, P. (2009). Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 563–576, New York, NY, USA. ACM. 43

Theobald, M., Schenkel, R., and Weikum, G. (2005). An efficient and versatile query engine for topx search. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 625–636. VLDB Endowment. 26

Theobald, M., Weikum, G., and Schenkel, R. (2004). Top-k query evaluation with probabilistic guarantees. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 648–659. VLDB Endowment. 36

Valle, E. (2008). *Local-descriptor matching for image identification systems*. Thèse de doctorat, Université de Cergy-Pontoise. 38

Weber, R., Schek, H.-J., and Blott, S. (1998). A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 194–205, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 38

Yin, S., Badr, M., and Vodislav, D. (2013). Dynamic multi-probe lsh : an i/o efficient index structure for approximate nearest neighbor search. In *Proceedings of the 24nd international conference on Database and expert systems applications*, DEXA'13.

Yuan, Y., Lin, X., Liu, Q., Wang, W., Yu, J. X., and Zhang, Q. (2005). Efficient computation of the skyline cube. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 241–252. VLDB Endowment. 27

Zhan, L., Zhang, Y., Zhang, W., and Lin, X. (2012). Finding top k most influential spatial facilities over uncertain objects. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 922–931, New York, NY, USA. ACM. 26, 120

Zhang, Z., Hwang, S.-w., Chang, K. C.-C., Wang, M., Lang, C. A., and Chang, Y.-c. (2006). Boolean + ranking : querying a database by k-constrained optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 359–370, New York, NY, USA. ACM. 23