Optimisation de requêtes dynamiques pour l'analyse de la biodiversité

Ndiouma Bame[1 2] — Hubert Naacke[2] — Idrissa Sarr [1] — Samba Ndiaye[1]

[1] Département de mathématique-informatique
Université Cheikh Anta Diop, Dakar, SENEGAL
prénom.nom@ucad.edu.sn

[2] Sorbonne Universités, UPMC Univ Paris 06
LIP6 Laboratory Paris, France
prenom.nom@lip6.fr

RÉSUMÉ. La quantité des données produites par de nombreux domaines augmente constamment et rend leur traitement de plus en plus difficile à gérer. Parmi ces domaines nous nous intéressons à la biodiversité pour laquelle le GBIF (*Global Biodiversity Information Facility*) vise à fédérer et partager les données de biodiversité produites par de nombreux fournisseurs à l'échelle mondiale. Aujourd'hui, avec un nombre croissant d'utilisateurs caractérisés par un comportement versatile et une fréquence d'accès aux données très aléatoire, les solutions actuelles n'ont pas été conçues pour s'adapter dynamiquement à ce type de situation. Par ailleurs, avec un nombre croissant de fournisseurs de données et d'utilisateurs qui interrogent sa base, le GBIF est confronté à un problème d'efficacité difficile à résoudre. Nous visons, dans cet article, à résoudre les problèmes de performances du GBIF. Dans cette perspective, nous proposons une approche d'optimisation de requête d'analyse de données de biodiversité qui s'adapte dynamiquement au contexte des environnements répartis à large échelle pour garantir la disponibilité des données. L'implémentation de notre solution et les résultats des expériences sont satisfaisants pour la garantie de performance et du passage à l'échelle.

ABSTRACT. The amount of data produced by many areas is constantly increasing and makes treatment more difficult to manage. One of those areas is the biodiversity field, which the GBIF (Global Biodiversity Information Facility) aims to federate and share data produced by many worldwide suppliers. Actually, with a growing number of users characterized by a versatile behavior and a high frequency and randomly data access, existing solutions are not well suited to face such challenge. Moreover, the GBIF system is not able to deal efficiently with the growing number of data providers and users. We aim in this paper to solve the performances drawback of the GBIF. In this respect, we propose a context-aware solution that dynamically adapts the query processing based on the data distribution and the access pattern. The implementation of our solution and the results of the experiments show the effectiveness of our solution in terms of scalability and availability.
MOTS-CLÉS : masses de données, réplication et distribution dynamiques, large échelle, optimisation de requêtes, biodiversité.
KEYWORDS : big data, dynamic replication and distribution, large escale, query optimization, biodiversité.

1. Introduction

1.1. Contexte et problématique

Le GBIF (Global Biodiversity Information Facility) est une initiative visant à fédérer et partager les données de biodiversité produites par de nombreux fournisseurs à l'échelle mondiale. Les données collectées par le GBIF décrivent principalement des occurrences de specimens vivants. Les données sont structurées : une occurrence est un nuplet dont les attributs décrivent la classification taxinomique du specimen, son lieu et sa date d'observation, l'observateur, le fournisseur, etc. En termes de taille, la base de données contient plus de 500 millions d'occurrences issues de plus de 14000 collections et 650 fournisseurs [19]. Ces données servent de matière première à une large communauté des chercheurs qui étudient plusieurs aspects de la biodiversité comme, par exemple, l'impact du réchauffement climatique sur certaines espèces. Le GBIF propose actuellement des services pour interroger ces données. Un utilisateur peut rechercher des occurrences en fonction de certains critères simples (nom scientifique d'un specimen, date et lieu d'observation). Il peut télécharger le résultat d'une requête afin de poursuivre son analyse avec ses propres outils. Un écologue qui étudie les abeilles invoquera le GBIF pour télécharger l'ensemble des occurrences d'abeilles. Puis, il devra préparer les données téléchargées pour les rendre compatibles avec ses outils qui calculeront la densité d'abeilles dans chaque zone étudiée. En plus des services d'interrogation et de téléchargement de données, le GBIF permet à l'utilisateur de visualiser sur une carte la position géographique des occurrences relatives à un pays ou à un fournisseur. Cette visualisation est figée et pré-calculée et un utilisateur ne peut ni filtrer les occurrences visualisées, ni choisir l'échelle de la visualisation En résumé, les services offerts par le GBIF sont une première réponse au besoin de partager des données dans de nombreux travaux de recherche en biodiversité ([12]).

Avec un nombre croissant de fournisseurs [19, 13] qui ajoutent de nouvelles données et d'utilisateurs qui expriment de nouveaux besoins d'interrogation, l'accès aux données du GBIF commence à poser des problèmes d'*expressivité* et d'*efficacité* difficiles à résoudre. Le manque d'expressivité empêche l'utilisateur d'exprimer des opérations pour filtrer, combiner, agréger les données avant de les télécharger. Par exemple, le scientifique ne peut pas demander à télécharger les fleurs localisées dans une région de France ayant également une forte densité d'abeilles. Il devra télécharger toutes les fleurs et toutes les abeilles de France, afin de les combiner pour ne garder finalement qu'une faible portion des données. Dans ce cas, la majeure partie des données a été transférée inutilement. Le manque d'efficacité est aussi mis en évidence quand des données sont modifiées. Lorsque le GBIF complète sa base avec une nouvelle collection de fleurs, chaque utilisateur concerné ré-exécute sa chaîne de traitement en commençant par télécharger les nouvelles données. Ceci génère une diffusion massive des nouvelles données qui pourrait être évitée si les utilisateurs partageaient un outil commun pour évaluer leurs requêtes. Le problème d'efficacité est d'autant plus crucial que le service de requêtes centralisé du GBIF pose plusieurs limites pour le *passage à l'échelle* face à la croissance des données et des demandes des usagers. Ceci ne garantit pas la disponibilité des données pour des requêtes interactives. Pour résoudre les problèmes d'expressivité et d'efficacité, deux exigences concernant l'interrogation des données et la disponibilité du service doivent principalement être considérées L'interrogation de ces masses de données nécessite d'une part, un langage de requête de haut niveau qui permet à l'utilisateur d'exprimer sim-

plement ses demandes (e.g., des requêtes SQL pour interroger des données structurées), et d'autre part des méthodes d'accès performantes pour évaluer des requêtes dans un délai imparti. A notre connaissance, il n'existe pas de système répondant à ces deux exigences d'expressivité et de performance. L'utilisateur doit faire un compromis entre l'expressivité et la performance. Deux options se présentent : (i) réduire la masse des données pour pouvoir les interroger de manière expressive et rapide, ou (ii) interroger la masse totale des données mais avec des requêtes très simples et des temps de réponse plus longs. Notre travail vise à éviter un tel compromis. Un aspect particulièrement difficile à prendre en compte est le comportement très versatile des utilisateurs. Cela génère des demandes très fluctuantes : la charge est variable en nombre de demandes, certaines données sont plus populaires (fréquemment demandées) que d'autres, la popularité est elle-même fluctuante (une donnée n'est populaire que pendant une durée limitée). Or les solutions actuelles n'ont pas été conçues pour s'adapter dynamiquement à ce type de situation.

1.2. Motivations

Le phénomène du « Big Data » est de plus en plus perçu comme l'un des grands défis informatique de la décennie en cours. De nombreux domaines font face à un 'déluge' de données sans précédent. La quantité des données produites augmente constamment et rend leur traitement de plus en plus difficile à gérer avec les outils actuels. Par exemple, Facebook enregistre plus de 800 millions d'utilisateurs actifs en moyenne par jour [17] et ajoute quotidiennement plus de 500 TB de données [22]. La base de données astrométrique européenne est complétée avec plus de 100 Go de données de nouvelles observations par jour [11]. En outre, selon une étude de Industrial Development Corporation (IDC) et EMC Corporation [24], la quantité de données générées en 2020 serait 44 fois supérieures par rapport à 2009 avec un taux de 35 ZB par an.

L'accès, l'interrogation et l'analyse de ces nouvelles masses de données sont essentiels pour élargir les connaissances du domaine y afférent et font parties des défis majeurs du Big Data. Ceci est particulièrement crucial dans les domaines tels que la génomique, la climatologie, l'astronomie, l'écologie, la biologie et la biodiversité.

La plupart des solutions actuelles pour la gestion de ces masses de données sont orientées pour des applications de types 'batch' où l'utilisateur soumet des lots de traitements non interactifs. Or, Intel [8] prédisait que si les applications 'batch' étaient de 50 % en 2012, alors en 2015 les deux-tiers des traitements sur ces masses de données seraient des applications interactives. Contrairement à une application 'batch', une application interactive réalise un dialogue avec l'utilisateur. L'utilisateur soumet une demande puis il attend la réponse avant de soumettre la demande suivante. Pour être fluide, cette interaction exige des temps de réponse courts. Nous constatons qu'en présence de masses de données, les besoins applicatifs et la nature des données sont très divers pour qu'une seule solution générale de gestion de données convienne à tous les contextes. Nous proposons d'exploiter les spécificités du contexte de la biodiversité pour concevoir une solution expressive et performante adaptée aux applications d'analyses pour les utilisateurs.

1.3. Contributions

Dans nos précédents travaux [5, 6, 7], nous avons identifié un certain nombre de problèmes liés au fonctionnement du GBIF notamment à son architecture centralisée et aux limites en termes

d'expressivité des requêtes supportées [5]. Nous avons pris en compte une contrainte imposée par le contexte applicatif : la nécessité d'utiliser autant que possible les machines localisées dans les laboratoires partenaires du GBIF. Autrement dit, l'infrastructure cible est un réseau mondial de machines ayant des capacités de calcul et de stockages et d'interconnexion très diverses. Nous avons proposé une solution décentralisée [6]. Cette solution rend possible l'évaluation répartie des requêtes sur des données de biodiversité et favorise le partage des ressources de calcul et de stockage dans l'infrastructure cible. Nous avons également proposé une stratégie de répartition dynamique des données de biodiversité [7]. La répartition s'appuie sur le modèle multidimensionnel hiérarchique (i.e. les dimensions taxonomiques et géographiques) des données pour identifier les données fréquemment accédées afin de les répliquer à la demande.

Ce présent travail vise à compléter nos précédents en s'intéressant à l'optimisation des requêtes complexes dans un environnement à large échelle. Cet environnement est caractérisé par des demandes versatiles (fluctuantes et disparates) venant des utilisateurs : les requêtes complexes sont de plus en plus nombreuses, la fréquence d'accès à chaque fragment fluctue. Nous partons du constat que les requêtes ne sont pas traitées assez rapidement lorsque de nombreux utilisateurs interrogent des données stockées sur un même site. Plus précisément, plusieurs requêtes peuvent solliciter simultanément un site et provoquer des attentes longues. Cela s'explique par le fait que le mécanisme de réplication dynamique proposé dans [7] est découplé de l'algorithme qui détermine le plan d'exécution répartie d'une requête. Le choix des sites à accéder dépend seulement des répliques existantes sans tenir compte des nouvelles répliques qui doivent être ajoutées pour supporter les requêtes. Afin d'éviter cette situation problématique, nous proposons un modèle d'exécution réparti des requêtes qui contrôle le placement dynamique des données à travers les sites. Ayant comme objectif de traiter la plupart des requêtes dans une durée impartie (temps de réponse borné), notre approche d'optimisation de requêtes considère plusieurs paramètres susceptibles d'impacter le coût d'une requête : la fluctuation et la disparité des charges, la disparité des capacités de calcul et des liens de communication, la localisation des données impliquées, le schéma de fragmentation des données et les prédicats des requêtes. Les contributions sont :

- Adaptation dynamique du schéma de placement des données. Notre approche évite la surcharge d'un site, en détectant les fragments de la requête qui se trouvent sur des sites surchargés et en les répliquant vers d'autres sites moins chargés. Ceci augmente les possibilités de choix pour les requêtes ultérieures qui invoqueront ces fragments. Le placement d'une nouvelle réplique permet de regrouper progressivement les données d'une même requête. Cela permet de traiter les requêtes impliquant les mêmes données sur un même site et ainsi réduire les surcoûts dus au traitement réparti de requête. Cela réduit aussi les situations de blocage inhérentes au traitement réparti (la probabilité de panne augmentant avec le nombre de sites).

- Garantie de temps de réponse borné pour les données populaires. L'approche utilise toutes les ressources (calcul et données) possibles et au besoin, la requête est décomposée pour accéder en parallèle aux fragments sollicités. De même, les sites non surchargés sont choisis toujours en priorité et les données populaires sont répliquées en fonction de la charge en cours. Notons que les principes sur lesquels s'appuient notre approche permettent de réduire le temps de réponse d'un plus grand nombre de requêtes accédant à des fragments populaires (voir la section 5.2 pour l'étude comparative). Dans nos expériences le temps de réponse est plafonné à environ 6 secondes.

- Modèle de coût dynamique. Notre approche d'optimisation s'appuie sur un modèle de coût dynamique qui s'adapte à un contexte où plusieurs requêtes sont en cours ou en attente d'exécution

dans le système et permet d'estimer le temps de réponse d'une nouvelle requête. Les paramètres intégrés dans ce modèle caractérisent un modèle de coût efficace pour l'optimisation de requête dans un environnement distribué hétérogène à large échelle.

2. Etat de l'art

L'optimisation de requêtes dans les environnements distribués à large échelle est assez complexe au regard des caractéristiques et des paramètres qui peuvent avoir un impact sur le coût. Plusieurs études [10, 23, 4, 1, 2, 25, 21, 18, 26] ont été menées dans ce sens. En 2003, Google a développé GFS [14] et BigTable [9] pour la gestion de ses applications. Ensuite, il a proposé MapReduce [10] qui est un paradigme de programmation pour faciliter le traitement parallèle sur des masses de données. Par la suite, beaucoup de travaux se sont basés sur le paradigme MapReduce pour proposer diverses extensions avec plus de fonctionnalités et plus d'expressivité. Par ailleurs des solutions non basées sur le paradigme MapReduce ont aussi été proposées.

2.1. MapReduce et ses extensions

MapReduce est un modèle de partitionnement de traitement (ou tâche) en vue d'une exécution parallèle sur plusieurs machines/sites. Le modèle MapReduce est principalement destiné à des traitements de type 'batch', de très grande taille, portant sur de très grands volumes de données. Toutefois, MapReduce se limite à deux fonctions (map et reduce), ce qui nécessitent un effort de programmation important : un traitement complexe doit être traduit en un enchaînement de nombreuses tâches. De ce fait, un programme MapReduce est généralement trop spécifique et peu réutilisable.

Hadoop implémente le modèle MapReduce. C'est un système conçu pour traiter en batch de très grandes quantités de données tout en tolérant les pannes. Il s'appuie sur le système de fichiers distribué HDFS [20] pour stocker les données. Il exploite la localité des données en déplaçant les traitements vers les données. Cependant, Hadoop présente des limites de flexibilité et de performance avec une latence minimale de l'ordre de dix secondes et des traitements pouvant atteindre plusieurs heures [25]. De ce fait, il n'est pas adapté à des situations où la quantité de données interrogées est petite et aux applications interactives. Pour améliorer la flexibilité et les performances, des extensions de Hadoop ont été proposées.

Hive [23] est une solution ciblant les application analytiques, basée sur Hadoop et HDFS. Hive offre un accès de plus haut niveau que MapReduce en proposant le langage de requête HiveQL proche de SQL. Ce qui permet à l'utilisateur d'interroger plus facilement les données. Hive a été conçu par Facebook qui l'utilise dans les traitements de ses gros volumes de données [23]. Cependant, il importe de noter que Hive tout comme Hadoop ne présente pas des avantages (en termes de performance) lorsque les données manipulées dans une requête sont de petites tailles (par exemple, de l'ordre de quelques MO). Ces solutions basées sur MapReduce, s'avèrent être lentes car une requête est traduite en un enchaînement de tâches map et reduce pour être exécuté, le résultat intermédiaire de chaque tâche est matérialisé dans le système de fichiers (HDFS ou GFS). Ceci engendre un nombre d'entrées/sorties important. Selon [25], les systèmes implémentant MapReduce présentent

une latence importante allant de quelques dizaines de secondes à plusieurs heures. Pour bénéficier des avantages des SGBD relationnels en termes d'optimisation locale des traitements, les auteurs de [4, 1] ont proposé HadoopDB. Dans ce système chaque machine héberge un SGBD relationnel (PostgreSql) pour stocker une partie des données et disposer d'un accès efficace aux données. La répartition des données est fixée lors du chargement des données dans les SGBD. HadoopDB utilise Hive pour calculer le plan d'exécution d'une requête et Hadoop pour bénéficier du parallélisme entre les machines et des facilités pour échanger des données entre les machines. Certes HadoopDB présente des avantages en utilisant les fonctionnalités des SGBD, mais cela n'empêche pas qu'il se comporte comme Hive en décomposant une requête en un enchaînement de tâches map et reduce. De fait, les résultats intermédiaires sont matérialisés dans les SGBD, ce qui a pour inconvénient de ne pas réduire efficacement la latence des requêtes.

Spark est un système de traitement parallèle de données proposant de stocker les données dans la mémoire principale des machines pour obtenir de meilleures performances. Il peut utiliser le système de fichiers HDFS pour le stockage persistent des données. L'abstraction que Spark utilise pour représenter les données est une collection générique appelée RDD (pour Resilient Distributed Dataset [26]). Cette abstraction permet de manipuler des données distribuées sur plusieurs machines aussi simplement que des données centralisées. Spark s'appuie sur le paradigme MapReduce pour proposer des opérations algébriques de manipulation de données (sélection, projection, regroupement, etc.) qui transforment des RDD en d'autres RDD. Une RDD peut ainsi être définie comme un enchaînement d'opérations algébriques et peut être recalculée si nécessaire. Cela permet à Spark de garder les données en mémoire en fonction des requêtes et de garantir leur recouvrement en cas de panne. Contrairement à Hadoop et Hive, Spark garde les résultats intermédiaires en mémoire et les déplace sur disque seulement lorsque c'est nécessaire. Le stockage en mémoire permet à Spark d'éviter les congestions dues aux E/S disques notamment pour les résultats intermédiaires d'une tâche map. La notion de RDD permet également à Spark de définir le partitionnement des données au moment de leur création.

Spark SQL [16] est une solution récente basée sur les ressources de développement de Shark [25]. Shark s'appuie sur Spark pour bénéficier des avantages des RDDs et utilise Hive pour disposer d'une interface de haut niveau pour interroger des données. En outre, il se sert du compilateur de requête de Hive pour analyser les requêtes afin de produire un plan logique. Ce plan est complété par des règles additionnelles fixées par Shark pour générer un plan algébrique composé d'opérations sur des RDDs. Pour optimiser une requête, Shark recueille des statistiques sur les résultats intermédiaires (taille, distribution des valeurs). Cela lui permet de choisir un algorithme efficace pour traiter les opérations algébriques telles de la jointure. Les performances de Shark sont 40 à 100 fois meilleures que celles obtenues avec Hadoop([25]). Basé sur Spark, Spark SQL permet de contrôler le partitionnement initial des données. L'optimisation ne considère pas les paramètres dynamiques et hétérogènes des environnement à large échelle notamment la différence de charge entre les sites et les fluctuations de la charge sur un même site au cours du temps.

2.2. Autres solutions d'optimisation

En plus des travaux implémentant ou étendant le modèle MapReduce, plusieurs recherches [18, 2, 21] ont été menées dans le domaine de l'optimisation de requête à large échelle. Nous présentons dans la suite quelques travaux qui ont été menés dans ce sens.

Travaux de Schaffner et al (2013) [18]. Les auteurs proposent un mécanisme d'optimisation de requêtes qui garantit des temps de réponse acceptables. Ils supposent que la charge est répartie de façon équitable à travers les applications, et proposent une solution de placement incrémentiel des données. Cette solution est guidée par la charge produites par les applications sur chaque fragment. Connaissant la charge de chaque application sur une donnée et l'ensemble des applications qui exploitent une donnée, ils proposent un algorithme qui détermine le schéma de placement optimal pour garantir un temps de réponse minimal avec le plus petit nombre de sites. Lorsqu'un site est surchargé, alors le schéma de placement est réadapté pour maintenir une charge acceptable. L'approche est mise en œuvre avec une estimation des comportements futurs des applications. Elle peut être efficace, lorsque ce comportement est statique ou régulier et que chaque application utilise les mêmes données en continuité. Cependant, elle peut engendrer un coût de replacement dynamique inutile lorsque le nombre et le comportement des applications sont dynamiques et irrégulier. De plus, elle se base sur le comportement antérieur des applications pour décider du placement qui serait adapté dans l'avenir. Ceci ne serait pas efficace lorsque de nouvelles applications intègrent le système après le replacement.

Travaux de Soliman et al (2014) [21]. Pour faire face à l'accroissement des données et aux limites des optimiseurs existants (avec des résultats non-optimaux), les auteurs d'Orca, proposent une solution d'optimisation de requêtes destinée aux applications analytiques à la demande. Orca utilise les statistiques sur les propriétés des données. Dans un premier temps, il explore un ensemble de plans en fonction des opérations de la requête. Ensuite, en fonction des informations sur les statistiques des données impliquées, il aménage (par des transformations) les plans d'exécution qui ont été calculés lors de la phase d'exploration. Il effectue une estimation du coût de chaque plan candidat en fonction des statistiques afin de déterminer le plan optimal qui est le plan de moindre coût. Les auteurs de Orca, considèrent que l'optimisation de la requête est coûteuse en calcul et proposent de paralléliser cette phase.

Travaux de Agarwal et al (2012) [2]. Les auteurs considèrent que la génération du plan d'exécution de la requête avant son exécution n'est pas efficace dans le contexte du large échelle, puisque les propriétés à partir desquelles le plan est calculé sont difficiles à estimer. En effet, pour eux, les propriétés des requêtes et des données varient largement au cours du temps. De ce fait, les utiliser de façon fixe pour des estimations a priori mènerait à des performances inefficaces. Ainsi ils proposent, RoPE, une approche de ré-optimisation de la requête au cours de son exécution. RoPE collecte les informations sur les requêtes et les propriétés des données pendant l'exécution de la requête pour alimenter l'optimiseur de requête. Ce dernier utilise les statistiques sur les propriétés des données et des opérations pour adapter les plans d'exécution des requêtes en cours. Les statistiques sont également utilisées pour adapter les plans d'exécution des futures invocations des mêmes tâches avec un optimiseur de requête basé sur le coût. Le nouveau plan nécessite un changement global du plan d'exécution de la requête : par exemple réordonner les opérations, choix de l'implémentation de l'opération appropriée et le regroupement des opérations ayant des tâches petites.

UBIQUEST (2012) [3]. Les auteurs proposent une abstraction de programmation de haut niveau pour des applications réparties dans un environnement dynamique. Pour ce qui concerne l'optimisation de requêtes, ils se basent sur l'historique des statistiques d'accès pour le calcul de plans d'exécution. Lorsqu'il n'existe pas d'historique pour la requête et les données concernées, l'optimiseur génère les plans possibles et utilise des heuristiques pour déduire le plan de moindre coût. Ce plan est utilisé pour l'exécution de la requête. Cette approche permet d'adapter le fonctionne-

ment du système en fonction des requêtes et des paramètres d'optimisation considérés. UBIQUEST considère les propriétés des requêtes et des données. Cependant, il ne prend pas en compte la disparité de charges et de performances entre les sites, encore moins de l'évolution dynamique de ces charges. Ceci pose un réel problème pour le passage à l'échelle de l'environnement et des applications. Une autre différence avec notre approche est que dans UBIQUEST le placement est des données n'est pas contrôlé par l'optimisation de requêtes.

2.3. Synthèse et contributions

L'état de l'art nous permet de constater que plusieurs paramètres et mécanismes pouvant améliorer les performances des requêtes sont pris en compte dans le processus d'optimisation. En effet, le placement dynamique en fonction de statistiques [21] sur les données ainsi que la factorisation de tâches redondantes ou encore la matérialisation de certains résultats peuvent adapter les performances du système en fonction des accès. De la même façon, la ré-optimisation [2] de requête en cours d'exécution peut adapter le plan d'exécution en fonction de paramètres réels du système. En outre, un mécanisme efficace [18] de placement des données en mémoire réduit les risques de congestion dus aux accès disque et accélère les traitements.

Cependant, des paramètres tels que la disparité et la fluctuation des charges à travers les sites ne sont pas prises en compte dans ces solutions. En outre, les solutions qui ré-optimisent la requête pendant son traitement, sont bénéfiques pour des traitements longs (de l'ordre de plusieurs minutes à plusieurs heures), mais peuvent être pénalisantes pour des requêtes de courte durée (moins de dix secondes par exemple) puisque la mise en œuvre de la ré-optimisation engendre un surcoût élevé par rapport au temps pris par la requête avec un modèle de coût efficace. Nous résumons sur le tableau suivant les paramètres d'optimisation qui sont pris en compte par chacune des solutions présentées en haut et notre approche.

Travaux	Placement dynamique	Propriétés des données	Opérations des requêtes	Charges courantes	Temps de réponse borné	Ré- optimisation
Hadoop-Hive	Non	Oui	Oui	Non	Non	Non
Spark SQL [25]	Oui	Oui	Oui	Non	Non	Oui
Travaux de Schaffner et al (2013) [18]	Oui	Oui	Oui	Oui	Non	Non
Travaux de Soliman et al (2014) [21]	Non	Oui	Oui	Non	Non	Oui
Travaux de Agarwal et al. (2012) [2]	Non	Oui	Oui	Non	Non	Non
UBIQUEST (2012) [3]	Non	Oui	Oui	Non	Non	Non
Notre approche	Oui	Oui	Oui	Oui	Oui	Non

Notre approche d'optimisation considère plusieurs paramètres susceptibles d'affecter les performances du système. En effet, notre approche d'optimisation guide la distribution dynamique des données dans l'optique de trouver un schéma de placement qui s'adapte davantage aux comportements des applications et qui se fait en fonction de la structure des données et des requêtes. Avec une contrainte de temps de réponse borné, elle prend en compte les charges fluctuantes des diffé-

rents sites. En outre, notre approche utilise les statistiques sur les temps de traitement des fragments sur les sites dans l'estimation du coût de la requête.

3. Traitement décentralisée de requête d'analyse

Cette section décrit l'exécution des requêtes dans un environnement décentralisé. Nous détaillons d'abord le modèle de requête décrivant des traitements indépendants qui peuvent être traités en parallèle. Puis, nous détaillons l'exécution décentralisée de plusieurs requêtes.

3.1. Modèle de requête

Une requête, notée $R(O, F)$, est caractérisée par un ensemble d'opérations O qui doivent être appliquées sur un ensemble de fragments F. Nous notons par O_k, un élément (une opération) de O et F_j un élément (un fragment) de F. Nous distinguons deux types d'opérations :

1) **Opération locale et traitement local**. Les opérations communes à tous les F_j et qui peuvent être exécutées en parallèle sur chaque F_j indépendamment des autres fragments. Nous les notons OL (pour Opérations Locales aux fragments), OL = $\{O_k \in O / O_k$ applicable à tous les $F_j\}$. Un **traitement local** (dénoté traitement OL) est l'ensemble des opérations de la requête qui sont locales à chaque fragment.

2) **Opération globale et traitement global**. Les opérations qui ne peuvent pas être exécutées en parallèle sur chaque F_j mais de façon globale sur les résultats des traitements OL. Nous les notons OG (pour Opérations Globales aux fragments). OG = $\{O_k \in O / O_k$ non-applicable à tous les $F_j\}$. Un **traitement global** (dénoté traitement OG) est l'ensemble des opérations de la requête qui sont exécutées sur les résultats des traitements OL de la requête.

Nous illustrons ce modèle de requête à travers la figure 1 où D_1 et D_2 correspondent respectivement aux densités des plantes (family = 'Plantae') et des abeilles (family = 'Apidae') dans la zone Z_x délimitée par min_decimallatitude = -10, max_decimallatitude = 20, min_decimallongitude = 30 et max_decimallongitude = 60.

Nous notons une requête par $R(\{OL, OG\}, F)$ et considérons que le coût d'un traitement (OL ou OG) est proportionnel à la quantité de données impliquées. Nous supposons que pour la plupart des requêtes de biodiversité, le coût des traitements OL est plus important que celui des traitements OG. En effet, les traitements OL sont des sélections suivies d'agrégations et de jointures et impliquent des traitements complexes, alors que les traitements OG sont des unions ou des jointures sur les résultats des OL (petites quantités de données). Par exemple, le calcul des cooccurrences de deux ou plusieurs espèces se décompose comme suit : (i) Traitement OL : créer le maillage de la zone d'étude et déterminer dans quelle maille est localisée chaque occurrence de l'espèce étudiée. Les occurrences de la même espèce sont dénombrées pour chaque maille afin d'obtenir les valeurs de densité. (ii) Traitement OG : pour chaque maille, les valeurs de densité des différentes espèces sont combinées pour obtenir les valeurs de cooccurrence.

Dans cet exemple, les traitements OL s'avèrent plus coûteux que les traitements OG. C'est un cas d'usage représentatif des requêtes généralement posées pour analyser les données de biodiver-

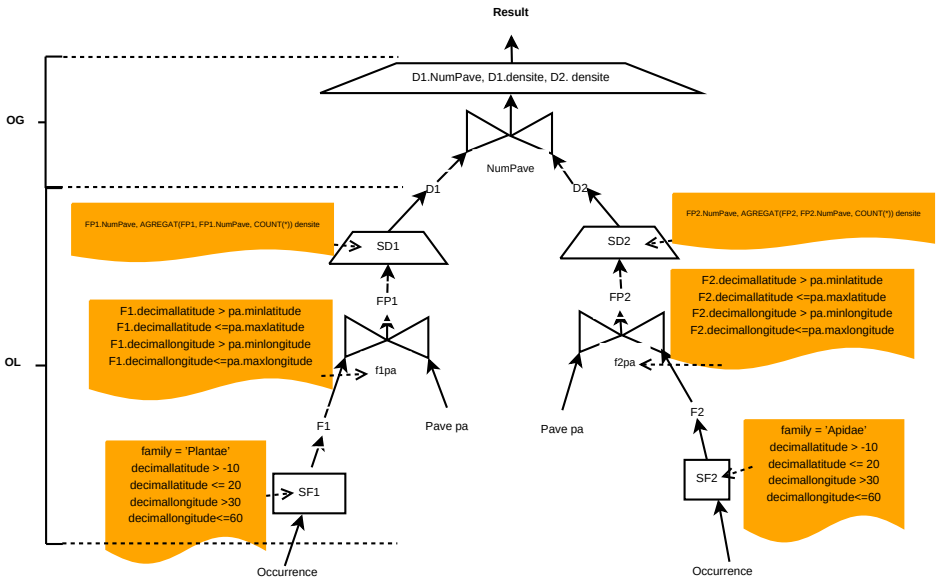


Figure 1. *Modèle de requête d'analyse de biodiversité*

sité. Les études de migration, de distribution, la modélisation de niche écologique, les relations de proie/prédation, nécessitent des calculs de cooccurrence et de densité d'espèces. Nous visons à apporter des solutions efficaces pour ces cas d'usage, en exploitant la possibilité d'évaluer en parallèle les traitements OL.

3.2. Exécution concurrente de plusieurs requêtes

Dans un contexte où de nombreux utilisateurs posent des requêtes, chaque site peut recevoir des requêtes et les traiter. Lorsqu'une requête arrive, le site qui coordonne les traitements OL et OG est appelé le coordinateur de la requête. Ce dernier soumet les OL aux sites concernés, en parallèle. Puis il attend les réponses de tous les OL avant d'évaluer le traitement OG. Chaque site a donc deux rôles, il peut coordonner une requête, et il peut traiter des OL pour une requête coordonnée par un autre site.

Le modèle d'exécution est conçu pour utiliser, autant que nécessaire, les ressources de calcul existantes. Sachant que chaque site dispose d'un SGBD relationnel capable d'exploiter toutes les ressources de calcul du site pour évaluer une requête, nous faisons le choix de traiter les OL en série (file) selon leur ordre d'arrivée sur un site. Le traitement en parallèle de plusieurs OL concerne des OL évaluées par des sites différents. Lorsqu'un site a terminé le traitement d'une OL, il vérifie si une OG est prête à être traitée ; le cas échéant il traite l'OG avant de poursuivre le traitement des OL en attente, et ainsi de suite.

Un coordinateur peut demander à un site S d'évaluer un traitement OL concernant un fragment qui n'existe pas encore sur le site S. Dans ce cas, le fragment est répliqué sur S, juste avant le traitement OL. Le but de la réplication étant de décharger rapidement un site trop sollicité, l'accès à un site pour lire un fragment à répliquer est traité immédiatement, sans tenir compte des éventuelles OL en attente. Cela prévient les situations d'inter-blocage à cause d'attentes mutuelles. De plus, la création d'une nouvelle réplique ne ralentit pas significativement le site qui transmet les données. L'exemple suivant illustre l'exécution d'une requête. Nous notons L_k (respectivement G_k) l'ensemble des OL (resp. des OG) affectés au site S_k.

Exemple
Considérons les trois requêtes R_1, R_2 et R_3 portant respectivement sur les ensemble de fragments $\{F_1, F_2\}$, $\{F_2, F_3\}$ et $\{F_1, F_3\}$ avec le schéma de placement à travers les sites S_1 et S_2 ci-après :
$S_1 : \{F_1, F_2\}$; $S_2 : \{F_3\}$
On soumet au système les deux requêtes R_1, R_2 et R_3 dans l'ordre. Notons : $OL_{i,j}$ le traitement OL de la requête R_i concernant le fragment F_j, et OG_i le traitement OG de la requête R_i.

La figure 2 illustre l'exécution des requêtes R_1, R_2 et R_3 à travers les sites S_1 et S_2. Nous avons $G_1 = \{OG_1\}$ en bleu sur S_1, $G_2 = \{OG_2, OG_3\}$ en bleu sur S_2 , $L_1 = \{OL_{11}, OL_{12}, OL_{22}\}$ en vert sur S_1 et $L_2 = \{OL_{23}, OL_{33}, OL_{31}\}$ en vert sur S_2. Tous les OL de R_1 sont traités au même site, S_1 qui traitera le OG de R_1. De la même façon, tous les OL de R_3 sont traités au même site, S_2 qui traitera le OG de R_3. Cependant les OL de R_2 sont réparti à travers S_1 et S_2 (étape 1). A la fin de tous les OL de R_1, le traitement du OG de la requête est démarré (étape 2). On remarque que OL_{33} portant sur le fragment F_3 est assigné au site S_2 qui ne dispose pas du fragment. De ce fait, il est nécessaire de répliquer le fragment F_3 vers S_2 pour le traitement de OL_{33} (étape 3).

4. Optimisation de requête basée sur un modèle de coût dynamique

Le processus d'optimisation d'une requête détermine le site à accéder, pour chaque fragment de donnée que la requête doit lire. L'objectif étant de garantir un temps de réponse borné pour la plupart des requêtes. L'utilisation combinée de plusieurs sites pour évaluer une requête tend à réduire le temps de réponse car les OL sont traitées en parallèle. L'originalité de notre approche repose sur la prise en compte, pendant le processus d'optimisation de requêtes, de la possibilité de choisir un site qui ne contient pas encore le fragment. Autrement dit, la stratégie de réplication des données est étroitement guidée par les requêtes qui viennent d'être posées et qui ne sont pas encore traitées.

Cependant, la réplication doit être déclenchée à bon escient en tenant compte du bénéfice et du surcoût qui en découlent. D'une part, adapter le placement des données en fonction des requêtes peut améliorer le temps de réponse du fait d'un meilleur usage des ressources disponibles. D'autre part, répliquer des données engendre des coûts de transfert de données et d'insertion. De plus, il faut tenir compte de la capacité de stockage limité d'une machine. Un mauvais choix de destination des nouvelles répliques peut dégrader fortement les performances du système.

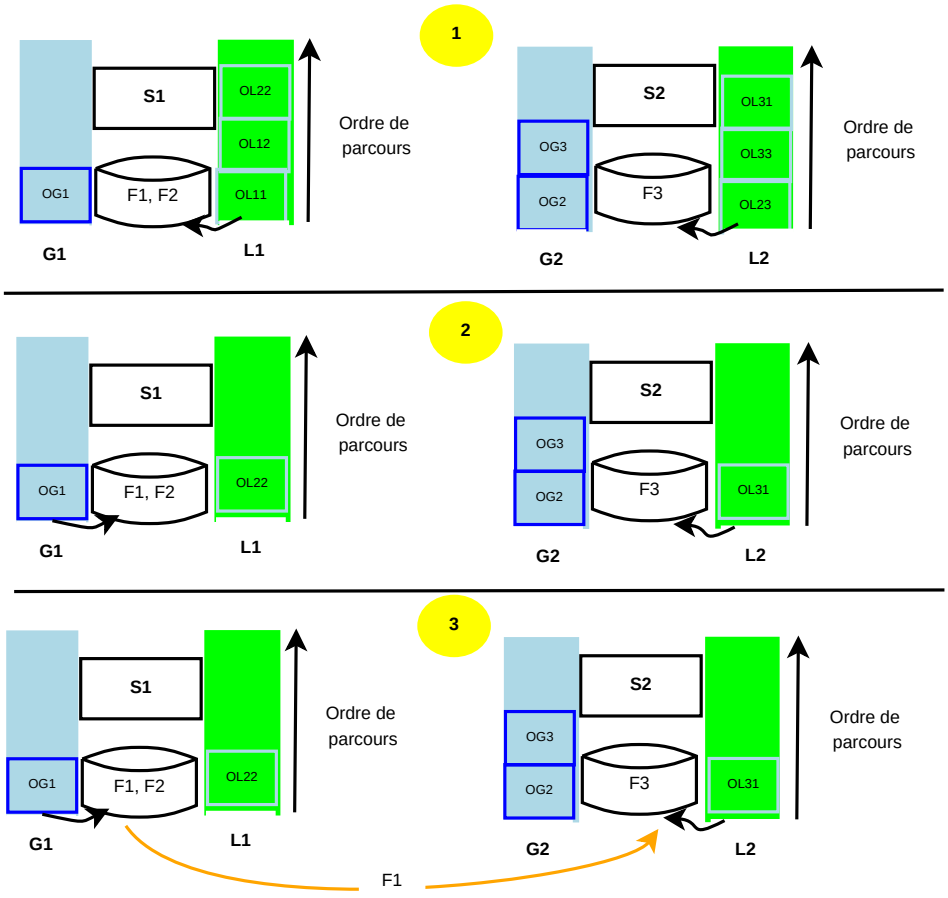


Figure 2. *Exemple d'exécution d'une requête*

Plusieurs stratégies d'optimisation peuvent être envisagées. Nous présentons notre approche d'optimisation basée sur une estimation du coût des requêtes, puis nous détaillons nos contributions par rapport à d'autres approches d'optimisation possibles.

4.1. Principes

Étant donné une requête, il s'agit de déterminer son plan d'exécution. Un plan précise les sites participants qui effectueront les traitements OL et le coordonnateur qui effectuera le traitement OG. Afin de réduire le risque d'un blocage lié à la surcharge d'un site, nous minimisons le nombre de participants tant que le temps de réponse estimé reste inférieur à un plafond fixé. Nous déterminons d'abord les sites qui peuvent traiter tous les OL de la requête sans réplication supplémentaire de

données. S'il n'existe pas de plan garantissant la contrainte de temps de réponse borné sans la moindre réplication supplémentaire de fragment, alors nous cherchons un plan qui garantit cette contrainte en répliquant les fragments stockés sur des sites surchargés. S'il n'existe aucun plan garantissant la contrainte de temps de réponse borné (par exemple, cela se produit lorsque les temps de réplication sont élevés), alors nous répliquons les fragments stockés sur des sites surchargés vers le site qui a le plus de données de la requête. Cela vise à regrouper davantage les fragments invoqués ensemble tout en augmentant les possibilités de choix pour les prochaines requêtes concernant les même données. Afin de réduire le volume du transfert des résultats des traitements OL, nous désignons comme coordinateur de la requête, le site devant évaluer le plus de traitement OL de la requête.

4.2. Définition des paramètres de coûts

Durée d'un traitement OL, TOL$_i(F_j)$
C'est la durée du traitement OL portant sur le fragment F_j sur le site S_i lorsque toutes les ressources (de calcul de mémoire) sont à disposition au site S_i. Ce paramètre dépend des performances du site et de la taille du fragment à traiter.

Durée d'un traitement OG, TOG$_i$
C'est la durée d'un traitement OG portant sur l'ensemble des résultats des OL d'une requête, sur le site S_i. Puisque les tailles des résultats des OL sont relativement petites, la durée du traitement OG dépend principalement des performances du site.

Durée d'une réplication, TREP$_{ik}(F_j)$
C'est la durée de réplication du fragment F_j depuis le site source S_k vers le site S_i. Ce paramètre totalise le temps de lire le fragment source, transférer les données et insérer le fragments sur S_i. Lorsque le fragment existe au site S_i, alors aucune réplication n'est effectuée. De ce fait, cette durée est nulle lorsque $i = k$.

Temps d'attente, TA$_k$
C'est la latence qu'un nouveau traitement aurait sur le site S_k. Elle est définie par la charge courante sur le site S_k. Cette charge est la somme des durées des OL et des OG qui sont en attente sur le site au moment de son évaluation. Puisqu'un OL peut nécessiter une réplication qui est coordonnée par le site lui-même, la durée de la réplication est une charge de plus induite. Ainsi, nous évaluons le temps ou charge d'un site S_k à l'aide de la formule ci-dessous :

$TA_k = \sum_{OLn} \in L_k(TOL_n + TREP_{jk}(F_n)) + \sum_{OGi} \in G_k(TOG_i)$

Temps de réponse maximal, TR_MAX
Le temps de réponse maximal, noté TR_MAX est la borne du temps de réponse à respecter pour la requête.

Durée de transfert d'un résultat partiel, TP$_{ki}$
La durée de transfert, notée TP$_{ki}$, d'un résultat partiel issu d'un OL traité au site S_k est le temps nécessaire pour acheminer ce résultat au coordonnateur S_i de la requête concernée.

Surcharge
La surcharge d'un site est son incapacité à effectuer au moins un traitement OL et un traitement OG

de plus pour une requête tout en respectant le temps de réponse maximal. On obtient l'information concernant la surcharge d'un site S_k grâce à la variable booléenne SS_k telle que :

$SS_k \longleftarrow (TA_k + TOL_k \geq TR\_MAX - TOG_k)$

où TOL_k correspond à TOL moyen au site S_k.

4.3. Plan d'exécution sans création de nouvelle réplique

4.3.1. Principes

La première phase du calcul du plan d'exécution d'une requête consiste à déterminer un plan sans la moindre réplication. Pour cela, on identifie les sites candidats qui satisfont à eux seuls la contrainte de temps de réponse borné sans la moindre réplication de donnée. Lorsque de tels sites existent, alors le site le moins chargé est utilisé pour traiter tous les OL de la requête. Par contre, lorsqu'il n'existe pas de site pouvant traiter seul toutes les OL de la requête, alors on détermine le site qui pourrait traiter le maximum d'OL sans réplication. Pour cela, on détermine les sites qui stockent des fragments de la requête et qui ne sont pas surchargés. Parmi ces derniers, on commence par le site qui pourrait traiter le plus grand nombre de OL tout en satisfaisant la contrainte de temps de réponse. On lui affecte en priorité les OL concernant les fragments les moins répliqués dont il dispose. Cet ordre d'attribution des OL permet de traiter les autres fragments (plus répliqués) dans les autres sites. Après chaque attribution d'OL à un site, on incrémente la charge du site concerné de la durée du traitement OL (TOL du fragment sur le même site). Les OL restants pour la requête sont affectés aux autres sites selon le même principe. Ce fonctionnement d'attribution des OL aux sites adaptés a pour avantage de favoriser la localité pour le traitement d'une requête. En d'autres termes, il vise à minimiser le nombre de participants au traitement d'une requête.

4.3.2. Algorithme

Cet algorithme nommé *calculerPlanSansReplication*, est constitué des étapes successives ci-dessous où F représente tous les fragments de la requête et S l'ensemble des sites.
Nous notons F^i l'ensemble des fragments de la requête stockés sur le site S_i
1. déterminer les sites non-surchargés (notés SNC) hébergeant au moins un fragment de la requête ;
2. ordonner de façon décroissante les sites de SNC par nombre de fragments hébergés de la requête ;
3. parcourir les sites S_i de SNC (en commençant par le site qui contient plus de fragments de la requête) ;
3.1. ordonner chaque ensemble F^i par degré de réplication croissant ;
3.2. enlever de F^i les fragments de la requête qui sont déjà assignés ;
3.3. vérifier que le site S_i peut traiter tous les fragments (${F_j}^i$) de la requête qu'il héberge et qui ne sont pas encore assignés (F^i)avec le respect de la contrainte de temps de réponse borné ;
3.3.1. lorsque la condition est vérifiée, on assigne les traitements OL des fragments concernés au site. Une fois un fragment assigné, il n'est plus considéré pour les sites suivants ;
3.3.2. dans le cas contraire, on vérifie que le site S_i peut traiter tous les fragments (F^i) de la requête qu'il héberge sauf le fragment plus répliqué (${F_{jmax}}^i$) en procédant de la même façon que précédemment ;
4. lorsque tous les fragments sont assignés pour leurs traitements OL (FA = F), alors un plan d'exécution sans réplication est retourné ;

5. lorsque les fragments ne sont pas tous assignés pour leurs traitements OL, alors l'algorithme retourne un résultat vide.

```
Fonction calculerPlanSansReplication(F : Fragments de la requête) : Plan
    plan = Nouveau Plan
    SNC ⟵ { S_k / SS_k ET (F^k ≠ ∅)}
    TrierSitesDesc(SNC, F)
    FA ⟵ ∅
    Pour chaque S_i de SNC faire
        TrierDegreFrag(F^i)
        F^i ⟵ F^i - FA
        FA^i ⟵ ∅
        OK ⟵ FAUX
        Tant que ((OK) OU (F^i = ∅))faire
            jmax ⟵ j / DegreRep(F_j^i) = max(DegreRep(F^i))
            Si (TA_i + Σ_{F_j ∈ F^i} TOL_i(F_j) ≤ TR_MAX − TOG_k)Alors
                FA^i ⟵ F^i
                OK ⟵ VRAI
            Sinon
                F^i ⟵ F^i - F_jmax^i
            Fin Si
        Fait
        Si (FA^i ≠ ∅)Alors
            Ajouter(plan, S_i, FA^i)
            FA ⟵ FA ⋃ FA^i
        Fin Si
    Fin Pour
    Si (FA = F) Alors
        Retourner plan
    Sinon
        Retourner NULL
    Fin Si
Fin
```

4.4. Plan d'exécution avec création de nouvelle réplique

Cette section détaille le cas où il n'existe aucun plan pour traiter la requête avec le respect du temps de réponse borné en utilisant les fragments existants. Il est nécessaire de répliquer davantage les données. Dans ce cas, nous proposons de répliquer des fragments stockés sur des sites surchargés vers des sites disponibles afin de borner les temps de réponse. La réplication a pour effet d'élargir les possibilités de plan d'exécution pour les requêtes ultérieures. Nous intégrons la réplication dans le processus d'optimisation de la requête. Nous prenons en compte le coût de la réplication pour déterminer le site sur lequel nous ajoutons une réplique.

4.4.1. Principes

Nous n'énumérons pas exhaustivement toutes les possibilité de répliquer chaque fragment sur chaque site car le nombre de possibilité est trop élevé en général (i.e. égal à S^F pour S sites et F fragments). Nous optons pour une approche de type *greedy* reposant sur les principes suivants :

Nous utilisons autant que possible les fragments existants. Nous accédons en priorité aux sites stockant le plus grand nombre de fragments de la requête. Pour un site, nous accédons en priorité aux fragments qui sont déjà les plus répliqués. Cette première étape permet d'attribuer un site à une partie des fragments de la requête. Nous décidons de répliquer les fragments restants pour lesquels aucun site n'a été attribué. Par conséquent, les fragments les moins répliqués ont plus de chance d'être répliqués. Cela tend à répartir l'effet de la réplication sur davantage de fragments ce qui convient bien à notre contexte fluctuant où tout fragment peut devenir très sollicité pendant un certain temps.

Quand la décision est prise de répliquer un fragment, nous choisissons le site source et le site destination comme suit : (i) choix du site source : s'il existe déjà plus d'une réplique, alors nous choisissons le site source le moins chargé. (ii) choix du site de destination : nous choisissons celui peut garantir le temps de réponse borné lorsque qu'il existe. Sinon, nous choisissons un site qui contient des données de la requêtes. Dans les deux cas, nous choisissons le site qui contient la plus grande portion de données de la requête. Ce choix tend à améliorer la localité des requêtes.

4.4.2. Algorithme

Les étapes successives de l'algorithme de calcul de plan avec la possibilité de réplication de données sont les suivantes :
Nous notons F^i l'ensemble des fragments de la requête stockés sur le site S_i
1. déterminer les sites non-surchargés (notés SNC) hébergeant au moins un fragment de la requête ;
2. ordonner de façon décroissante les sites par nombre de fragments hébergés de la requête ;
3. parcourir les sites S_i de SNC (en commençant par le site qui contient plus de fragments de la requête) 3.1. ordonner de façon croissante chaque ensemble de fragments F^i du site S_i par degré de réplication ;
3.2. enlever de F^i les fragments de la requête qui sont déjà assignés ;
3.3. vérifier que le site S_i pourrait traiter tous les fragments (${F_j}^i$) de la requête qu'il héberge et qui ne sont pas encore assignés (F^i)avec le respect de la contrainte de temps de réponse borné ; 3.3.1. lorsque la condition est vérifiée, on assigne les traitements OL des fragments concernés au site. Une fois un fragment assigné, il n'est plus considéré pour les sites suivants ;
3.3.2. dans le cas contraire, on vérifie que le site ${}_i$ pourrait traiter tous les fragments (F^i) de la requête qu'il héberge sauf le fragment moins répliqué (${F_{jmin}}^i$) en procédant de la même façon que précédemment ;
4. après avoir parcouru les sites ordonnés et que tous les fragments de la requête ne sont pas assignés, alors :
4.1. incrémente la charge de chaque site qui était non-surchargé des coûts des OL qu'on lui a déjà assignés pour la requête ;
4.2. calculer les sites non-surchargés SNC ;
4.3. ordonner de façon décroissante les sites SNC par nombre de fragments déjà assignés de la requête ;
4.4. parcourir les sites ordonné SNC en commençant par celui à qui on a déjà assigné plus de traitement OL pour la requête ;
4.4.1. déterminer FNA les fragments de la requête qui ne sont pas encore assignés. On a FNA = F - FA ; 4.4.2. ordonner de façon croissante les fragments FNA par degré de réplication ;
4.4.3. vérifier que le site courant S_i pourrait traiter tous les fragments restants (FNA) de la requête

lorsqu'ils sont répliqués dans la base locale avec le respect de la contrainte de temps de réponse borné ;
4.4.3.1. lorsque la condition est vérifiée, on assigne les traitements OL des fragments concernés au site. Une fois un fragment assigné, il n'est plus considéré pour les sites suivants ;
4.4.3.2. dans le cas contraire, on vérifie que le site S_i pourrait traiter tous les fragments restants (FNA) de la requête lorsqu'ils sont répliqués dans la base locale sauf le fragment moins répliqué FNA_{jmin} en procédant de la même façon que précédemment (4.4.3) ;
5. après avoir parcouru les sites ordonnés et que tous les fragments de la requête ne sont pas tous assignés, alors :
5.1. determiner le site S_{kmax} dont on a assigné le plus de OL de la requête ;
5.2. assigner les OL (fragments) restants FNA de la requête au site S_{kmax}.

L'algorithme de calcul de plan d'exécution avec la réplication de fragment peut être décomposé en trois phases successives :
-attribution de OL sans réplication de fragment en respectant la contrainte de temps de réponse borné : de 1. à 3.
-attribution de OL avec réplication de fragment en respectant la contrainte de temps de réponse borné : 4.
-attribution de OL avec réplication de fragment sans respecter la contrainte de temps de réponse borné : 5.

```
Fonction calculerPlanAvecReplication(F : Fragments de la requete) : Plan
    plan : Nouveau Plan
    FA ⟵ ∅
    assignerTraitementSansReplicationAvecSeuil(F, FA, plan)
    assignerTraitementAvecReplicationAvecSeuil(F, FA, plan)
    assignerTraitementAvecReplicationSansSeuil(F, FA, plan)
    Retourner plan
Fin
```

Les algorithmes ci-dessous correspondent aux trois phases du calcul de plan d'exécution des OLs d'une requête.

```
Procédure assignerTraitementAvecReplicationSansSeuil(F : Fragments de la requete, FA : Fragments assignés, plan : Plan)
    TrierSitesAssigneDesc(SNC, FA)
    S_kmax ⟵ SNC(0)
    FNA ⟵ F- FA
    FA^kmax ⟵ FA^kmax ⋃ FNA
    Ajouter(plan, S_kmax, FA^kmax)
Fin
```

Procédure assignerTraitementAvecReplicationAvecSeuil(F : **Fragments de la requete**, FA : **Fragments assignés**, plan : **Plan**)

- **Pour** chaque S_i de SNC **faire**
  - TA$_i$ ⟵ TA$_i$ + $\sum_{F_j \in FA^i} TOL_i(F_j)$
- **Fin Pour**
- SNC ⟵ { S$_k$ ∈ $S/\overline{SS_k}$}
- TrierSitesAssigneDesc(SNC, FA)
- **Pour** (chaque S_i de SNC **faire**
  - FNA ⟵ F - FA
  - **Si** (FNA ≠ ∅)**Alors**
    - OK ⟵ FAUX
    - **Tant que** ($(\overline{OK})$**ET** $(FNA \neq \emptyset)$)**faire**
      - cout ⟵ TA$_i$ + $\sum_{FNA_j \in FNA}(TOL_i(FNA_j) + TREP_{ik}$(FNA$_j$))
      - **Si** (cout ≤ $TR\_MAX - TOG_i$)**Alors**
        - FA^i ⟵ FA^i $\bigcup FNA$
        - FA ⟵ FA $\bigcup FA^i$ OK ⟵ VRAI
      - **Sinon**
        - jmin ⟵ j / DegreRep(FNA$_j$) = min(DegreRep(FNA))
        - FNA ⟵ FNA - FNA$_{jmin}$
      - **Fin Si**
    - **Fait**
    - **Si** (FA^i ≠ ∅)**Alors**
      - Ajouter(plan, S$_i$, FA^i)
      - FA ⟵ FA $\bigcup FA^i$
    - **Fin Si**
  - **Fin Si**
- **Fin Pour**

Fin

Procédure assignerTraitementSansReplicationAvecSeuil(F : **Fragments de la requete**, FA : **Fragments assignés**, plan : **Plan**)

- SNC ⟵{ S$_k$ / $\overline{SS_k}$**ET** $F^k \neq \emptyset$}
- TrierSitesDesc(SNC, F)
- FA = ∅
- **Pour** chaque S_i de SNC **faire**
  - TrierDegreFrag(F^i)
  - F^i ⟵ F^i - FA
  - FA^i ⟵ ∅
  - OK ⟵ FAUX
  - **Tant que** ($(\overline{OK})$**OU** $(F^i = \emptyset)$)**faire**
    - **Si** (TA$_i$ + $\sum_{F_j \in F^i} TOL_i(F_j) \leq TR\_MAX - TOG_i$)**Alors**
      - FA^i ⟵ F^i
      - OK ⟵ VRAI
    - **Sinon**
      - jmin ⟵ j / DegreRep(F$_j{}^i$) = min(DegreRep(F^i))
      - F^i ⟵ F^i - F$_{jmin}{}^i$
    - **Fin Si**
  - **Fait**
  - **Si** (FA^i ≠ ∅)**Alors**
    - Ajouter(plan, S$_i$, FA^i)
    - FA ⟵ FA $\bigcup FA^i$
  - **Fin Si**
- **Fin Pour**

Fin

4.4.3. Modèle de coût dynamique

Lors du traitement d'une requête, divers scénarios peuvent se présenter en fonction de la localisation des répliques à traiter : traitement local (OL et OG) de la requête sans création de nouvelle réplique, traitement local avec création de nouvelle réplique, traitement distribué avec création de nouvelle réplique et traitement distribué sans création de nouvelle réplique.

Nous proposons un modèle de coût global qui s'adapte à tous les scénarios et qui intègre plusieurs les paramètres susceptibles d'impacter le temps de réponse d'une requête. Le temps de réponse correspondant au délai qu'un utilisateur attend pour recevoir le résultat de sa requête depuis qu'il l'a soumise au système. Il dépend des charges des différents sites qui sont impliqués, des charges créées par la requête sur chaque site et des éventuels coûts de transferts de résultats partiels vers le coordinateur. Nous exprimerons le temps de réponse noté T_R, en nous basant sur le modèle d'exécution des requêtes et en utilisant les paramètres de coûts de notre approche d'optimisation.

D'après le modèle d'exécution défini en section , les traitements OL ainsi que leurs attentes à travers les sites se font en parallèle. Le traitement OG d'une requête est exécuté lorsque tous les OL de la requête sont finis.

Le coût d'une requête dépend :
-des charges courantes des différents sites impliqués au traitement de la requête TA_k
-des performances de tous les participants : TOL_k, TOG_i
-des capacités des liens de communication entre les sites : $TREP_{jk}(F_{OL_k})$, TP_{ki}
-des propriétés des fragments invoqués dans le traitement de la requête : TOL_k, $TREP_{jk}(F_{OL_k})$
-des propriétés de la requête : TOL_k, TOG_i

Le temps de réponse T_R de la requête R est :

$$T_R = \max_k \left[TA_k + \sum_{OL_k \in L_k^R} (TOL_k + TREP_{jk}(F_{OL_k})) + TP_{ki}\right] + TOG_i$$

5. Validation expérimentale

L'objectif de la validation expérimentale consiste à évaluer les performances et le passage à l'échelle de notre approche d'optimisation de requête. Dans la suite de cette section, nous présentons les outils et méthodes expérimentales que nous avons mis en œuvre et les résultats issus des expérimentations.

5.1. Méthodes expérimentales

Les méthodes expérimentales de la validation concernent l'environnement, les outils utilisées, les expériences effectuées et les approches comparatives. Nous présentons dans les sous-sections suivantes les détails des méthodes expérimentales.

5.1.1. Environnement et outils

Nous avons implémenté notre solution sur une infrastructure composée de onze machines. Chaque machine dispose des caractéristiques suivantes : 22 CPU de 2.394 GHz de fréquence, une

mémoire de 62 Go avec comme système d'exploitation Debian 3.2.54-2 x 64. Une première machine est utilisée pour représenter le portail GBIF et dispose de toutes les données dans une base locale. Une deuxième machine est utilisée pour générer des requêtes et simuler des utilisateurs qui envoient des requêtes d'analyse au système décentralisé. Une troisième machine contrôle les accès à la base de données du GBIF. Les huit machines restantes sont utilisées pour instancier chacune un site du système décentralisé. Chaque site dispose d'une base de données locale en plus des fonctions pour évaluer les OL et les OG. Nous avons également implémenté les catalogues locaux et global pour la localisation des données à travers le système distribué. Pour ce qui concerne les charges courantes des sites, elles sont demandées directement au sites concernés. Comme SGBD, nous avons utilisé MonetDB [15]. MonetDB est un SGBD qui compresse et stocke les données en mémoire [15]. Il a pour vocation d'exploiter les grandes mémoires physique des systèmes de calcul modernes de façon efficace lors du traitement des requêtes. De ce fait, il offre de bonnes performances en évitant les accès disques souvent coûteux. En plus, il supporte plusieurs modèles de données pour la gestion des données. Par exemples il supporte le modèle de données relationnelles et SQL, le format XML et XQuery et le format RDF et SPARQL [15].

5.1.2. Données

Nous avons utilisé dans nos expériences un miroir (dump) des données géo-référencées du GBIF que nous avons téléchargées et insérées dans le SGBD de la première machine afin qu'elle serve de portail GBIF. A partir des données d'occurrences de la base, nous avons construit le catalogue taxonomique décrivant la structure hiérarchique des données de biodiversité sur la dimension taxinomique. Pour ce qui concerne l'organisation hiérarchique selon la dimension géo-spatiale, nous avons divisé le globe en neuf cellules représentant chacune un continent. Chaque continent est découpé en neuf cellules qui correspondent aux pays. Enfin, chaque pays est divisé en 9 régions (cellules). Chaque fragment est défini par une espèce et une région.

5.1.3. Requêtes

Pour toutes les expériences, nous utilisons comme modèle de requête, le calcul de la cooccurrence de deux espèces dans une région. Une requête accède à deux fragments de la même région. Nous avons implémenté un générateur de requêtes SQL. Pour chaque requête, deux fragments sont choisis aléatoirement parmi les fragments contenant au moins 10000 occurrences dans la même région. Ceci rend les requêtes plus réalistes et évite les requêtes qui ne renvoient un résultat de trop petite taille. Le générateur produit un nombre fixe de requêtes au début de chaque expérience. Puis les requêtes sont soumises au système. Pour chaque requête nous mesurons le temps de réponse, le temps de chaque OL, le temps du traitement OG et le temps de chaque réplication éventuelle. Nous notons également tous les sites impliqués et leur tâche dans le traitement de la requête.

5.1.4. Expériences

Pour valider notre approche, nous commençons par étudier sa faisabilité. Dans un deuxième temps, nous évaluons les performances de notre approche d'optimisation comparativement à trois autres approches détaillées ci-dessous. Nous étudions la garantie du passage à l'échelle de notre approche en analysant le comportement des différents sites pour le traitement des requêtes et l'adaptation de notre approche d'optimisation par rapport au contexte dynamique.

5.1.5. Approches comparatives

Les approches utilisées pour évaluer les performances de notre solution sont MiniTrans, Equiload et DisNoRep. Nous rappelons brièvement le principe d'optimisation de chacune d'elles.

MiniTrans. Cette approche minimise le coût des transferts de données pendant l'évaluation d'une requête. Le coordinateur désigné pour évaluer la requête est le site qui stocke la plus grande partie des données (concernant la requête). Si les données concernant la requête ne sont sur aucun site, alors le coordinateur est le site ayant reçu la requête. Le coordinateur est complété avant d'évaluer la requête : les fragments manquants sont répliqués sur le coordinateur.

EquiLoad. Cette approche équilibre la charge des sites afin de réduire le temps d'attente pour les requêtes. Le coordinateur désigné pour évaluer la requête est le site le moins chargé. Les données manquantes sont répliquées sur le coordinateur.

DisNoRep. Dans cette approche il n'y a pas de réplication. Les fragments sont répartis entre les sites : un fragment est stocké sur un seul site. Un fragment n'existant sur aucun site est importé du GBIF pour être stocké sur le coordinateur. La requête est évaluée de manière répartie. Le coordinateur désigné pour évaluer la requête est le site qui reçoit la requête. Un avantage de cette approche est de réduire les transferts de données entre les sites.

5.2. Evaluation des performances

L'objectif de cette section consiste à évaluer les performances de notre solution. Cette évaluation concerne principalement l'approche d'optimisation qui détermine le schéma de placement dynamique des données à travers les sites et calcule le plan d'exécution de chaque requête selon divers paramètres. Nous évaluons les performances en fonction de l'évolution du débit transactionnel (nombre de requêtes traitées par intervalle de temps) du système et de l'approche d'optimisation mise en œuvre. Par la suite, nous comparons les performances de notre solution avec les approches Minitrans, Equiload et Disnorep.
Nous avons divisé la durée de chaque expérience en intervalles de temps de dix minutes et récapitulé le nombre de requêtes traitées (débit transactionnel) dans chaque intervalle pour chaque approche. Les différentes expériences sont effectuées dans les mêmes conditions : même nombre de machines, même nombre d'utilisateurs et même nombre de requêtes. Nous représentons uniquement les périodes complètes communes à toutes les expériences. La figure 3 représente les évolutions des débits transactionnels des différentes approches en fonction du temps.
Nous constatons que l'approche DisNoRep possède les plus faibles débits transactionnels dans toutes les périodes. Ceci s'explique par le fait qu'elle est la seule approche qui ne réplique aucun fragment. De ce fait, même si elle favorise le traitement parallèle, les accès concurrents aux mêmes fragments surchargent les sites qui les contiennent et par conséquent dégradent les performances. Puisque le schéma de placement de chaque fragment est statique, les performances ne sont pas améliorées au cours du temps.

Contrairement à DisNoRep, toutes les autres approches voient leur débit transactionnel augmenter au cours du temps. Ces approches reposent sur un mécanisme de réplication dynamique des fragments invoqués dans les requêtes. Ce qui donne davantage de possibilités à l'optimiseur de requête pour le calcul du plan d'exécution. Cependant, les approches MiniTrans et EquiLoad n'ex-

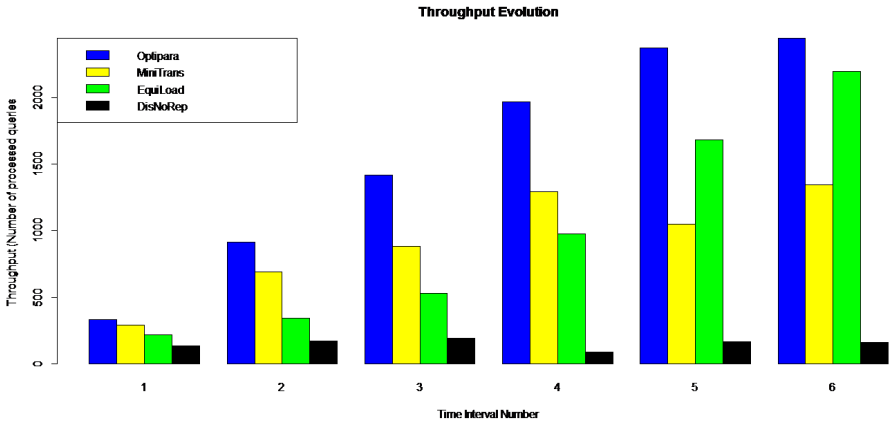


Figure 3. *Evolution des performances des différentes approches*

ploitent pas le parallélisme intra-requête qu'aurait offert le traitement réparti pour éviter certains transferts inutiles. L'augmentation de leur débit transactionnel s'explique par la prise en compte de paramètres importants qui ont un impact considérable sur le coût de la requête. En effet, MiniTrans réduit les transferts en répliquant les données manquantes sur le site qui dispose de la plus grande portion de données impliquées.

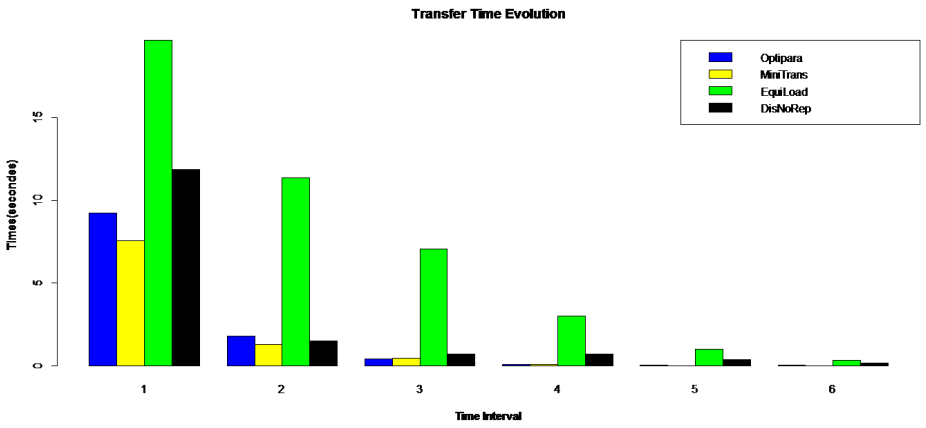
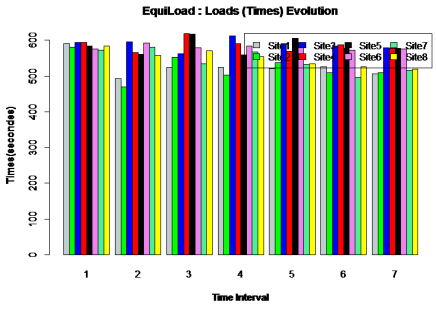


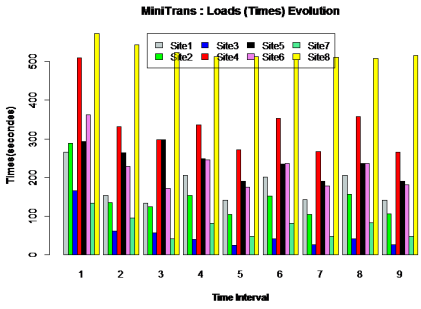
Figure 4. *Evolution des des temps de réplication*

Ainsi, les données d'une requête sont regroupées sur un seul site. Les prochaines requêtes qui invoquent les mêmes données ne nécessiteront pas de transfert. Ainsi les transferts de données deviennent de plus en plus rare pendant la durée de l'expérience. C'est ce qui explique l'augmentation progressive du débit transactionnel. Cependant cet accroissement est limité par le fait que MiniTrans utilise un petit nombre de sites pour regrouper la quasi-totalité des données invoquées par les requêtes. Seuls ces sites sont largement utilisés pour les traitements des requêtes. De ce fait, MiniTrans ne parvient pas à équilibrer la charge entre les sites.

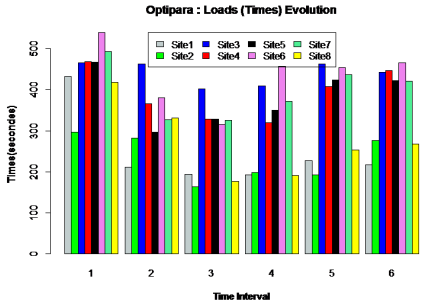
En ce qui concerne l'approche EquiLoad, elle règle ce problème d'équilibrage de la charge en traitant la requête sur le site le moins chargé. Elle permet ainsi de réduire le temps d'attente pour une requête. La figure 5(a) montre qu'EquiLoad répartit de façon quasi-équitable la charge globale à travers les sites. L'accroissement progressif du débit transactionnel s'explique par la réplication des données manquantes vers le site choisi pour traiter la requête. En effet, cette réplication dynamique crée davantage des répliques pour les fragments très sollicités et augmente ainsi le nombre de plans possibles pour exécuter les prochaines requêtes. Cependant, elle ne peut aboutir à une solution de réplication totale qui élimine complètement les transferts de données puisque la capacité de stockage de chaque base locale est limitée. Ce qui fait que les transferts de données restent toujours présents.



(a) EquiLoad



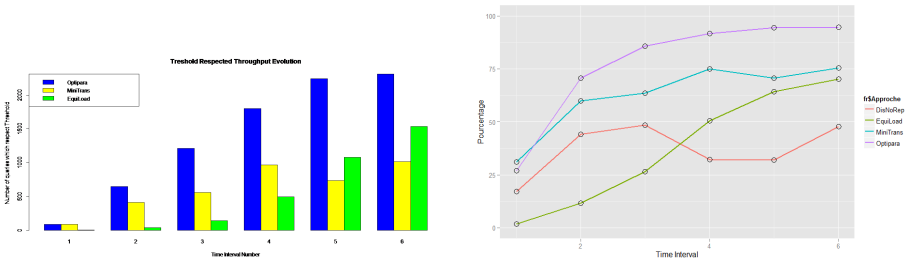
(b) MiniTrans



(c) OptiPara

Figure 5. *Evolution de la charge à travers les sites*

Notre approche, notée OptiPara, combine les avantages des trois approches en considérant la charge des sites, le coût de transfert et en exploitant les possibilités de traitement réparti pour une requête. En effet, notre approche s'appuie sur une contrainte de temps de réponse borné que le plan d'exécution de la requête doit respecter si cela est possible. Lorsque cette contrainte ne peut être respectée, notre approche en profite pour adapter dynamiquement le schéma de placement en répliquant les données qui se trouvent sur des sites surchargés (qui ne peuvent pas garantir un temps de réponse borné pour une nouvelle requête) vers des sites moins chargés. De cette façon, elle évite de surcharger les sites. La répartition de la charge entre les sites dépend donc de la capacité de chaque site. Cette stratégie ne garantit pas une charge équitable entre les sites, mais elle évite qu'un site soit surchargé en attribuant les nouveaux traitements aux sites libres qui pourraient garantir un temps de réponse borné. La réplication dynamique des données augmente le nombre de plans possibles pour exécuter une requête. L'évolution des temps de réplication nous permet de constater qu'après 30 minutes, les transferts de réplication de données tendent à être nuls. Ceci nous permet de conclure que dans la suite de l'expérience, le placement des données s'est adapté aux requêtes. En outre, OptiPara évite le transfert non nécessaire des fragments stockés sur des sites non-surchargés (pouvant respecter la contrainte de temps de réponse borné). La figure 6 montre l'efficacité d'OptiPara. En particulier, le pourcentage de requêtes qui respectent la contrainte de temps de réponse borné augmente progressivement durant l'expérience.



(a) Nombre (b) Pourcentage

Figure 6. *Evolution des requêtes qui respectent le seuil*

5.3. Passage à l'échelle

L'objectif de cette validation du passage à l'échelle est d'étudier les performances de notre solution en fonction de l'évolution des ressources et de la charge. Pour garantir le passage à l'échelle, les performances doivent évoluer dans le même sens que la variation des ressources et de la charge avec des coefficients proches. Par exemple, en doublant le nombre de sites, le débit observé devrait doubler.

Nous évaluons la garantie du passage à l'échelle de notre solution en évaluant les performances dans des scénarii différents. Chaque scénario est caractérisé par le nombre de sites utilisés et la charge soumise au système. Le nombre de sites et la charge d'un scénario à un autre varient avec le même facteur de proportionnalité. Pour chaque scénario nous notons ses débits transactionnels

stable et global. Le débit stable correspond à la moyenne des débits par intervalle de temps à partir desquels les variations sont inférieures à 10%. Nous récapitulons dans la figure 7 les évolutions des débits transactionnels par minute en fonction du nombre de sites qui composent le système décentralisé. Nous constatons que le débit transactionnel augmente proportionnellement avec le nombre

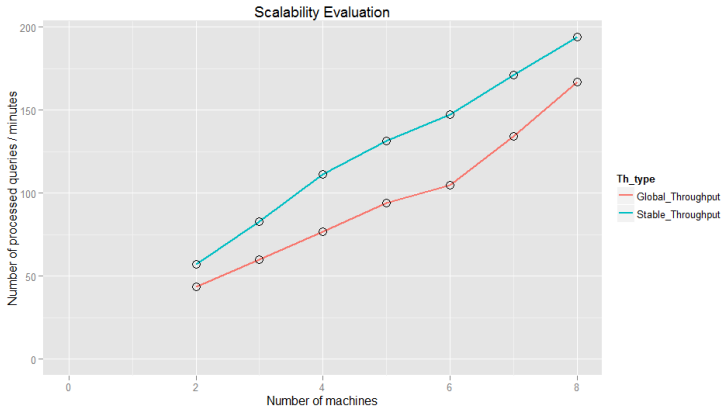


Figure 7. *Evolution des performances en fonction du nombre de sites*

de sites. En effet, le débit transactionnel évolue dans le même sens que le nombre de sites qui composent le système et avec presque le même facteur. En outre, on voit que le débit transactionnel stable est meilleur que le débit global. Ceci s'explique par le fait que le système nécessite une phase d'adaptation qui peut engendrer des réplications de données. L'adaptation dynamique du schéma de placement des données permet de réduire progressivement les transferts (ou réplication) pour les prochaines requêtes. De ce fait, avant la stabilisation du système, les sites, en plus des traitements, coordonnent aussi les réplications de données. Ce qui influe sur les performances globales du système. Lorsque le système est stable, alors les transferts dus aux réplications deviennent de plus en plus rares et les sites se concentrent entièrement aux traitements des requêtes. C'est ce qui explique que les débits stables sont supérieurs aux débits globaux. Avec cette adaptation dynamique du schéma de placement des données et le mécanisme d'optimisation basée sur le coût, notre solution garantit le passage à l'échelle.

6. Conclusions et perspectives

Cet article présente une approche innovante d'optimisation de requête dans un environnement réparti à large échelle caractérisé par la disparité des performances et des charges à travers les sites, l'irrégularité de la charge d'un site et du comportement des utilisateurs, la versatilité de la popularité d'une donnée et des ressources limitées pour chaque site individuel. Basée sur une contrainte de temps de réponse borné pour le traitement réparti de requête d'analyse de données de biodi-

versité, notre approche évite de surcharger un site en adaptant dynamiquement le placement des données pendant l'optimisation de la requête. Elle favorise la garantie de temps de réponse borné pour les requêtes qui sollicitent des fragments populaires en multipliant les possibilités de choix. En outre, elle adapte dynamiquement le placement de chaque donnée en fonction de sa popularité, des charges des sites et des classes d'accès des requêtes. L'exécution concurrente des requêtes aboutit à un modèle de coût dynamique qui prend en compte les paramètres dynamiques et disparates des caractéristiques d'un environnement réparti à large échelle. Nous avons implémenté notre approche et évalué les performances et le passage à l'échelle avec des données réelles du GBIF sur un cluster de 200 cœurs. Les résultats obtenus sont satisfaisants et converge vers un temps de réponse borné pour la plupart des requêtes. Nous envisageons d'étudier la participation du système décentralisé aux traitements intensifs d'intégration de données de biodiversité, pour supporter le passage à l'échelle des données qui est accru par l'automatisation progressive des techniques de collecte de données de biodiversité. Nous envisageons également d'étendre notre approche pour gérer plusieurs variétés de données massives telles des données climatiques, environnementales (caractéristiques physico-chimiques), des données portant sur les média sociaux, etc.

Références

[1] A. Abouzied, K. Bajda-Pawlikowski, J. Huang, D. J. Abadi, and A. Silberschatz. HadoopDB in action : building real world applications. *ACM Intl Conf. on Management of data (SIGMOD)*, pages 1111–1114, 2010.

[2] S. Agarwal, S. Kandula, N. Bruno, M. Wu, I. Stoica, and J. Zhoun. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 21–21, 2012.

[3] Ahmad Ahmad-Kassem, Christophe Bobineau, Christine Collet, Etienne Dublé, Stéphane Grumbach, Fuda Ma, Lourdes Martínez, and Stéphane Ubéda. Ubiquest, for rapid prototyping of networking applications. In *16th International Database Engineering & Applications Symposium, IDEAS '12, Prague, Czech Republic, August 8-10, 2012*, pages 187–192, 2012.

[4] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *ACM Intl Conf. on Management of Data (SIGMOD)*, pages 1165–1176, 2011.

[5] N. Bame, H. Naacke, I. Sarr, and S Ndiaye. Architecture répartie à large échelle pour le traitement parallèle de requête de biodiversité. In *African Conf. on Research in Computer Science and Applied Mathematics (CARI)*, pages 143–150, 2012.

[6] N. Bame, H. Naacke, I. Sarr, and S Ndiaye. Algorithmes de traitement de requêtes de biodiversité dans un environnement distribué. *Revue africaine de la recherche en informatique et mathématiques appliquées (ARIMA)*, 18 :1–18, 2014.

[7] N. Bame, H. Naacke, I. Sarr, and S Ndiaye. Bigbio : Utiliser les techniques de gestion du big data pour les données de la biodiversité. In *African Conf. on Research in Computer Science and Applied Mathematics (CARI)*, pages 273–284, 2014.

[8] Intel IT Center. Big data analytics, 2012.

[9] F. Chang et al. Bigtable : A Distributed Storage System for Structured Data (Best Paper Award). In *USENIX Symp. on Operating System Design and Implementation (OSDI)*, pages 205–218, 2006.

[10] J. Dean and S. Ghemawat. Mapreduce : simplified data processing on large clusters. *Commun. ACM*, 51 :107–113, 2008.

[11] GAIA and CNES Centre National dEtudes Spatiales. Project gaia at http ://smsc.cnes.fr/gaia/fr/index.htm, 2015.

[12] GBIF. GBIF data Use Case. `http://www.gbif.org/newsroom/uses`, 2015. Online.

[13] GBIF Secretariat. GBIF Annual Report 2013. Technical report, GBIF, 2014.

[14] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *ACM Symp. on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[15] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. Monetdb : Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull*, 2012.

[16] Databricks Inc. Shark, spark sql, hive on spark, and the future of sql on spark, 2015.

[17] Facebook newsroom. Statistics at http ://newsroom.fb.com/company-info/, 2014.

[18] J. Schaffner, T. Januschowski, M. Kercher, T. Kraska, H. Plattner, M. J. Franklin, and D. Jacobs. Rtp : Robust tenant placement for elastic in-memory database clusters. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 773–784, 2013.

[19] GBIF Secretary. Gbif data portal www.data.gbif.org, gbif web site www.gbif.org, 2013.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[21] M. A. Soliman, L. Antova L., V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahmanr, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca : A modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 337–348, 2014.

[22] TechCrunch. How big is facebooks data ? 2.5 billion pieces of content and 500+ terabytes ingested every day at http ://techcrunch.com/2012/08/22/how-big-is-facebooks-data-2-5-billion-pieces-of-content-and-500-terabytes-ingested-every-day/s, 2014.

[23] A. Thusoo et al. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *Intl Conf. on Data Engineering (ICDE)*, pages 996–1005, 2010.

[24] R. L. Villaaars, C. W. Olofson, and M. Eastwood. Big data : What it is and why you should care at http ://sites.amd.com/us/documents/idc_amd_big_data_whitepaper.pdf, 2011.

[25] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark : SQL and rich analytics at scale. In *ACM Intl Conf. on Management of Data (SIGMOD)*, pages 13–24, 2013.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.