Allocation de ressources élastique pour l'optimisation de requêtes

Mohamed Mehdi Kandi

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *29/11/2019* **par :**

MOHAMED MEHDI KANDI

Allocation de ressources élastique pour l'optimisation de requêtes

---

JURY

LADJEL BELLATRECHE	Professeur des universités, ENSMA, Poitiers	Rapporteur
MARINETTE SAVONNET	Maître de Conférences HDR, Université de Bourgogne	Rapportrice
DJAMAL BENSLIMANE	Professeur des Universités, Université Lyon 1	Examinateur
ABDELMALEK BENZEKRI	Professeur des universités, Université Toulouse 3	Examinateur
ABDELKADER HAMEURLAIN	Professeur des universités, Université Toulouse 3	Directeur de Thèse
SHAOYI YIN	Maître de Conférences, Université Toulouse 3	Co-Directrice de Thèse

---

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (UMR 5505)

Directeur de Thèse :
Professeur Abdelkader HAMEURLAIN

Rapporteurs :
Professeur Ladjel BELLATRECHE et Madame Marinette SAVONNET

Remerciements

Je souhaite avant tout exprimer mes sincères remerciements au Professeur Abdelkader HAMEURLAIN (directeur de thèse) et à Madame Shaoyi YIN (co-directrice de thèse), pour avoir dirigé mes travaux de thèse, mais aussi pour leur confiance, leur soutien permanent, leurs conseils avisés et leurs encouragements tout au long de ces trois années de thèse.

Je remercie très sincèrement les rapporteurs de ma thèse, le Professeur Ladjel BELLATRECHE et Madame Marinette SAVONNET, qui m'ont fait l'honneur d'apporter leur précieuse expertise dans l'évaluation de mes travaux.

Je souhaite aussi exprimer mes remerciements aux Professeurs Abdelmalek BENZEKRI et Djamal BENSLIMANE, qui ont gracieusement accepté de siéger dans mon jury de thèse en tant qu'examinateurs. Au-delà de son rôle d'examinateur, je remercie Monsieur BENZEKRI d'avoir été mon tuteur à l'IUT Paul Sabatier, et aussi pour son soutien et encouragements.

Je remercie aussi les autres membres de l'équipe PYRAMIDE, Professeur Frank MORVAN et Monsieur Riad MOKADAM pour leur disponibilité, leur soutien et leurs nombreux conseils.

Mes remerciements vont également à tous les collègues du laboratoire IRIT, pour les moments agréables partagés durant cette période.

Un grand merci à tous mes amis de Toulouse : Yacine, Amine, Salim, Moh, Wahab, Soufian... et aussi à Hamid et Sawsan, avec qui j'ai passé des moments inoubliables.

Je remercie de tout mon cœur mes chers parents, mes sœurs (Nadra et Lilia) et mon frère (Mohamed Ali) pour leur enthousiasme, leur disponibilité, leur patience et leur amour, sans lesquels cette thèse n'aurait pas été possible.

Résumé

Le Cloud Computing est devenu un moyen largement utilisé pour l'interrogation de bases de données. Les fournisseurs de cloud actuels proposent une variété de services implémentés sur des architectures parallèles. Les objectifs de performances et les éventuelles pénalités dans le cas de violation sont établis au préalable dans un contrat appelé Service-Level Agreement (SLA). Le but du fournisseur est de maximiser son bénéfice tout en respectant les besoins des locataires.

Avant la naissance des systèmes cloud, plusieurs travaux ont considéré le problème d'allocation de ressources pour l'interrogation de bases de données sur des architectures parallèles. Le plan d'exécution de chaque requête est un graphe de tâches dépendantes. L'expression "allocation de ressources" dans ces travaux sous-entend souvent le placement des tâches sur les ressources disponibles et aussi leur ordonnancement qui tient compte des liens de dépendance. Le but consistait principalement à minimiser le temps d'exécution de requêtes et maximiser l'utilisation de ressources. Par contre, dans le cloud ce but ne garantit pas nécessairement le meilleur bénéfice économique pour le fournisseur. Afin de maximiser le bénéfice et satisfaire les besoins des locataires, il est essentiel d'inclure le modèle économique et les SLAs dans le processus d'allocation de ressources. En effet, les besoins des locataires en terme de performances sont différents, il serait donc intéressant d'allouer les ressources d'une façon qui favorise les locataires les plus exigeants en assurant quand même une certaine qualité de service pour les locataires les moins exigeants. De plus, dans le cloud le nombre de ressources attribuées peut augmenter/diminuer selon la demande (élasticité) et le coût monétaire dépend du nombre de ressources attribuées, il devient donc intéressant de mettre en place un mécanisme pour choisir automatiquement le moment adéquat pour ajouter ou supprimer des ressources en fonction de la charge (dimensionnement automatique).

Nous nous intéressons dans le cadre de cette thèse à la conception de méthodes d'allocation de ressources élastique pour les services d'interrogation de bases de données dans le cloud : (1) une méthode d'allocation de ressources statique en deux phases pour as-

surer un bon compromis entre le bénéfice du fournisseur et la satisfaction des locataires, tout en garantissant un coût d'allocation raisonnable, (2) une méthode de réallocation de ressources dirigée par les SLAs pour limiter l'impact des erreurs d'estimation sur le bénéfice et (3) une méthode de dimensionnement automatique basée sur l'apprentissage par renforcement qui répond aux spécificités de l'interrogation de bases de données.

Afin d'évaluer nos contributions, nous avons implémenté nos méthodes dans un environnement cloud simulé et nous les avons comparées à des méthodes de l'état de l'art en terme de coût monétaire de l'exécution de requêtes ainsi que le coût d'allocation.

Mots clés : Cloud computing, Bases de données, Allocation de ressources, Dimensionnement automatique, Service-Level Agreement, Modèle économique.

Abstract

Cloud computing has become a widely used way to query databases. Today's cloud providers offer a variety of services implemented on parallel architectures. Performance targets and possible penalties in case of violation are established in advance in a contract called Service-Level Agreement (SLA). The provider's goal is to maximize its benefit while respecting the needs of tenants.

Before the birth of cloud systems, several studies considered the problem of resource allocation for database querying in parallel architectures. The execution plan for each query is a graph of dependent tasks. The expression "Resource allocation" in this context often implies the placement of tasks within available resources and also their scheduling that takes into account dependencies between tasks. The main goal was to minimize query execution time and maximize the use of resources. However, this goal does not necessarily guarantee the best economic benefit for the provider in the cloud. In order to maximize the provider's benefit and meet the needs of tenants, it is important to include the economic model and SLAs in the resource allocation process. Indeed, the needs of tenants in terms of performance are different, so it would be interesting to allocate resources in a way that favors the most demanding tenants and ensure an acceptable quality of service for the least demanding tenants. In addition, in the cloud the number of assigned resources can increase/decrease according to demand (elasticity) and the monetary cost depends on the number of assigned resources, so it would be interesting to set up a mechanism to automatically choose the right moment to add or remove resources according to the load (auto-scaling).

In this thesis, we are interested in designing elastic resource allocation methods for database queries in the cloud. This solution includes : (1) a static two-phase resource allocation method to ensure a good compromise between provider benefit and tenant satisfaction, while ensuring a reasonable allocation cost, (2) an SLA-driven resource reallocation to limit the impact of estimation errors on the benefit and (3) an auto-scaling method based on reinforcement learning that meet the specificities of database queries.

In order to evaluate our contributions, we have implemented our methods in a simulated cloud environment and compared them with state-of-the-art methods in terms of monetary cost of the execution of queries as well as the allocation cost.

Keywords : Cloud computing, Databases, Resource allocation, Auto-scaling, Service-Level Agreement, Economic model.

Sommaire

Table des figures

Liste des tableaux

Chapitre 1

Introduction générale

Contenu

1.1 Contexte

Le cloud computing est devenu aujourd'hui un moyen largement utilisé pour répondre à des besoins en informatique. Depuis son lancement dans les années 2000, le marché du cloud computing continue d'évoluer. Le chiffre d'affaires d'exploitation des activités cloud chez Amazon par exemple s'élève à 7,7 milliards de dollars (avec un bénéfice de 2,2 milliards de dollars) au premier trimestre 2019.

Le bénéfice du fournisseur est défini comme étant la différence entre le revenu et les dépenses. Le revenu est la somme monétaire facturée aux différents locataires suite à l'utilisation des services. Les dépenses regroupent le coût monétaire lié à l'exécution des services (CPU, mémoire, disque, réseau, licences des logiciels...) ainsi que les éventuelles pénalités dans le cas de violation des objectifs de performances et/ou fiabilité.

Les services sollicités dans le cloud sont souvent critiques. Les objectifs de performances et/ou fiabilité sont formulés dans un contrat appelé Service-Level Agreements (SLAs). Le contenu de ce contrat est négocié au préalable entre le fournisseur et le locataire. Il précise d'une façon formelle les différents objectifs (temps d'exécution, taux de disponibilité...),

mais aussi les actions à prendre dans le cas où ils ne sont pas respectés. En général, le fournisseur paye des pénalités aux locataires en cas de violation des objectifs.

Les fournisseurs de cloud actuels proposent une variété de services allant de simples capacités de calcul à des solutions complexes visant à répondre aux différents besoins des locataires. Les services cloud peuvent être classifiés selon leur nature, à savoir : Infrastructure as a Service (IaaS), Plateform as a Service (PaaS) et Software as a Service (SaaS). L'IaaS offre des services de niveau infrastructure. Le fournisseur est responsable des machines physiques, le réseau et la virtualisation. Le PaaS offre des services du niveau plateforme. Le fournisseur est responsable des systèmes de gestion de bases de données et/ou des outils de développement logiciel. Le SaaS offre des services du niveau applicatif. Le fournisseur est responsable des applications de natures variées : analyse, business, bureautique, web, robotique... Les trois classes de services peuvent être proposés par le même fournisseur. Il peut y avoir aussi des fournisseurs qui proposent des services appartenant à une seule classe. Dans ce cas, un fournisseur PaaS peut solliciter, par exemple, un fournisseur IaaS pour la gestion de l'infrastructure.

Nous nous intéressons dans le cadre de la thèse aux services d'interrogation de bases de données (fournisseur PaaS). Les systèmes de gestion de bases de données sont des éléments indispensables pour toute organisation de nos jours. Étant donné les avantages, plusieurs organisations ont choisi de déployer leurs bases de données dans le cloud (Chauhan et al., 2014; Verbitski et al., 2017), notamment via : Amazon Aurora [1], Amazon RDS [2], Amazon EMR [3], Azure SQL Database [4], Azure HDInsight [5], Oracle Database Cloud Service [6]...

Une base de données est un ensemble structuré de données modélisant les objets d'une partie du monde réel et servant de support à une application informatique (Gardarin, 2003). Nous considérons une structuration relationnelle pour les données. Les requêtes d'interrogation sont souvent exprimées avec le langage déclaratif SQL (Date and Darwen, 1987). Reconnu comme norme internationale dans les années 80, SQL est toujours utilisé dans les systèmes de gestion de bases de données relationnels, mais aussi comme extension (langage quasi-SQL) dans les outils big data récents comme Hadoop/Hive (Thusoo et al., 2010; White, 2012; Saha et al., 2015) et Spark SQL (Zaharia et al., 2010b; Armbrust et al.,

1. https ://aws.amazon.com/fr/rds/aurora/
2. https ://aws.amazon.com/fr/rds/
3. https ://aws.amazon.com/fr/emr/
4. https ://azure.microsoft.com/fr-fr/services/sql-database/
5. https ://azure.microsoft.com/fr-fr/services/hdinsight/
6. https ://cloud.oracle.com/fr_FR/database

2015).

L'optimiseur de requêtes est un composant essentiel pour l'interrogation des bases de données. Son rôle consiste à trouver le meilleur plan d'exécution pour les requêtes soumises. Chaque requête est d'abord analysée pour vérifier si elle est lexicalement et syntaxiquement correcte. Ensuite, elle est transformée en un arbre d'opérateurs relationnels (sélections, projections, jointures et agrégations) appelé un plan d'exécution logique. Enfin, un plan d'exécution physique séquentiel est généré. Ce dernier indique l'ordre des opérateurs et leur algorithme d'exécution.

Dans un environnement d'exécution parallèle, le plan d'exécution séquentiel est transformé en un plan d'exécution parallèle. Chaque opérateur est exécuté par un ensemble de tâches parallèles. Une tâche donnée effectue un traitement sur une partie des données. Les tâches dépendantes communiquent en mode producteur-consommateur. L'exécution parallèle d'une requête nécessite une infrastructure matérielle constituée de plusieurs unités de calcul (CPU) et un réseau de communication.

Nous considérons qu'une ressource est constituée d'une CPU et une certaine quantité mémoire. À un instant donnée, une ressource peut exécuter une seule tâche à la fois. Nous nous intéressons dans cette thèse à l'allocation de ressources et le dimensionnement automatique. Ces derniers ont un impact sur le bénéfice du fournisseur et la satisfaction des locataires.

1.1.1 Allocation de ressources

L'allocation de ressources consiste à déterminer "quand" et "où" exécuter chaque tâche du plan. Le choix de la ressource d'exécution (où ?) est appelé placement. La détermination de l'instant de déclenchement (quand ?) par rapport aux autres tâches placées sur la même ressource et en tenant compte des dépendances entre les tâches de la même requête est appelé ordonnancement.

L'allocation de ressources se base sur un modèle de coûts. Ce dernier utilise des statistiques contenues dans un catalogue (notamment le nombre de tuples et leur taille en octets, nombre de valeurs uniques d'un attribue donné, histogrammes) pour estimer le nombre et la taille des tuples des relations intermédiaires de la requête. À partir de ces estimations, des algorithmes sont utilisés pour trouver une allocation de ressources qui minimise (ou maximise, selon la formulation du problème) une fonction objectif tout en

respectant un ensemble de contraintes.

Avant la naissance des systèmes cloud, plusieurs travaux ont considéré le problème d'allocation de ressources pour l'interrogation de bases de données dans des architectures parallèles. Certains ont résolu le problème d'une manière statique avant l'exécution de la requête (Nag and DeWitt, 1998; Yu and Cornell, 1993) alors que d'autres d'une manière dynamique pendant l'exécution de la requête (Bouganim et al., 1998). Il y a aussi des travaux qui supposent l'existence d'une allocation statique puis cette dernière est modifiée dynamiquement pendant l'exécution de la requête si les estimations initiales du modèle de coûts sont significativement différentes de la réalité (Kabra and DeWitt, 1998). Ceci est appelé la réallocation de ressources. Suite à l'arrivée du cloud, certaines méthodes ont été étendues pour répondre aux spécificités de cet environnement, notamment les SLAs, le modèle économique et le multi-locataire (Kllapi et al., 2011; Pietri et al., 2019). Dans cette thèse, nous étudions le problème d'allocation de ressource statique en environnement cloud, et aussi le réallocation.

1.1.2 Dimensionnement automatique

L'utilisation du cloud a de nombreux avantages par rapport à une solution en local au sein des locaux de l'entreprise. Une solution en local nécessite un investissement initial pour la mise en place de l'infrastructure et l'installation des logiciels. De plus, elle peut donner lieu à une infrastructure sous-chargée ou surchargée. En effet, le nombre de ressources est défini lors de la mise en place de l'infrastructure alors que la charge peut varier dans le temps d'une façon imprévisible. La surcharge implique une diminution des performances de service alors que la sous-charge est considérée comme du gaspillage de ressources. Par contre, dans le cloud, il n'y a pas d'investissement initial pour le locataire car les ressources sont au niveau de l'infrastructure du fournisseur. De plus, grâce à l'élasticité (Kouki and Ledoux, 2012), le nombre de ressources attribuées peut changer à tout moment ce qui permet de garder des performances acceptables et au même temps éviter le gaspillage.

Afin d'assurer l'élasticité du cloud, il est important de décider le moment d'augmenter ou diminuer le nombre de ressources attribuées au service (combien ?). Le dimensionnement automatique est le mécanisme permettant de choisir le moment d'augmenter et diminuer le nombre de ressources. La plupart des fournisseurs actuels utilisent une ap-

proche intuitive basée sur des seuils (Khatua et al., 2010; Chieu et al., 2011; Simmons et al., 2011; Ghanbari et al., 2011; Han et al., 2012). L'idée principale de cette approche est d'ajouter (ou de supprimer) des ressources si une certaine mesure est supérieure (ou inférieure) à un seuil prédéfini. Le principal inconvénient de l'approche basée sur les seuils est le fait qu'elle exige une compréhension approfondie des tendances de la charge pour choisir les bons seuils, ce qui n'est pas facile à réaliser. Afin d'éviter les inconvénients de l'approche de seuils, de nombreux travaux scientifiques ont été consacrés à la conception de méthodes de dimensionnement automatique indépendantes de l'humain, basées sur l'apprentissage par renforcement (Dutreilh et al., 2011; Rao et al., 2011). L'apprenant est un agent qui fait des actions successives dans un environnement et reçoit une récompense pour chaque action effectuée. Après une suite d'essais, il devrait apprendre à prendre de bonnes décisions ce qui permet d'effectuer le dimensionnement automatique sans intervention humaine. Dans cette thèse, nous avons adopté la deuxième approche.

1.2 Problématique

La thèse répond à trois questions notée respectivement Q1, Q2 et Q3. Dans cette section, nous présentons une question à la fin de chaque paragraphe.

Le but des travaux dédiés à l'allocation statique pour les architectures parallèles (Yu and Cornell, 1993; Nag and DeWitt, 1998; Bouganim et al., 1998) est de minimiser le temps d'exécution de requêtes et maximiser l'utilisation de ressources, dont le nombre est supposé fixe. Ils sont destinés plutôt à une installation en local au sein des locaux de l'entreprise. En effet, le fait de minimiser le temps d'exécution de requêtes et maximiser l'utilisation de ressources ne garantit pas nécessairement le meilleur bénéfice économique pour le fournisseur. Dans le cloud, les besoins en terme de performances et les pénalités en cas de violation sont différents d'un locataire à l'autre, il serait donc intéressant d'allouer les ressources d'une façon qui favorise les locataires les plus exigeants en assurant quand même une certaine qualité de service pour les locataires les moins exigeants. Les méthodes existantes orientées cloud (Kllapi et al., 2011; Pietri et al., 2019) sont généralement basées sur des méthodes heuristiques qui présentent l'avantage d'un temps d'allocation rapide et la simplicité de leur conception. Cependant l'optimalité n'est pas garantie et dans certains cas la solution trouvée par ces méthodes est loin de l'optimal. Ce qui laisse la

voie à la communauté scientifique pour proposer des nouvelles méthodes qui répondent à la question : *(Q1) Comment permettre à l'allocation statique d'assurer un bon compromis entre le bénéfice du fournisseur et la satisfaction des locataires, tout en garantissant un coût d'allocation raisonnable ?*

L'allocation statique se base sur des estimations calculées grâce au modèle de coûts. Néanmoins, les estimations peuvent contenir des erreurs. La taille et le nombre de tuples réels des relations intermédiaires peuvent être significativement différents de ceux estimés. Les erreurs d'estimation sont dues au fait que les estimations sont calculées avec des formules analytiques approximatives. De plus, les statistiques du catalogue ne sont pas maintenues à jour. Dans le cadre d'une requête avec plusieurs opérateurs successifs, l'erreur se propage d'une façon exponentielle (Ioannidis and Christodoulakis, 1991). Si la taille des données réelle est très différente de la taille estimée, la qualité du plan d'allocation initialement établi (plan statique) se dégrade d'une façon significative. Certains travaux ont considéré le problème d'erreurs d'estimation (Kabra and DeWitt, 1998; Agarwal et al., 2012; Bruno et al., 2013). L'idée est de placer des opérateurs qui collectent des statistiques pendant l'exécution de la requête. Si les statistiques collectées sont significativement différentes des statistiques estimées pendant l'allocation statique alors l'allocateur est sollicité pour modifier l'allocation du reste de la requête. Ces méthodes considèrent que les requêtes ont le même niveau de priorité. Par contre, le cloud est un environnement multi-locataire dans lequel les exigences de performances et les éventuelles pénalités sont différentes d'un locataire à un autre. Par exemple un locataire de profil *premium* exige des seuils stricts sur le temps d'exécution de ses requêtes et des pénalités élevées dans le cas du dépassement de seuils alors q'un locataire de profil *basic* est tolérant en matière de seuils et en même temps les pénalités sont moindres dans le cas de violation. Étant donné que le but du fournisseur est de satisfaire les besoins des différents locataires tout en maximisant son bénéfice, il est important de prendre en compte les SLAs durant le processus de collecte de statistiques et la réallocation. Une question à poser est : *(Q2) Comment limiter l'impact des erreurs d'estimation sur le bénéfice du fournisseur, sachant qu'il s'agit d'un environnement multi-locataire ?*

Enfin, par rapport au dimensionnement automatique basé sur l'apprentissage par renfoncement, il n'y a pas une méthode efficace pour toutes les applications cloud. Chaque solution doit tenir compte des spécificités de l'application. Nous constatons que la plupart

des méthodes existantes se concentrent sur les applications Web. Une requête dans ces applications peut être considérée comme une tâche atomique assignée à une seule ressource pour une courte période de temps (Dutreilh et al., 2011; Rao et al., 2011). Ces méthodes ne conviennent pas à l'interrogation de bases de données. En effet, la structure d'une requête dans ce cas est complexe comparée à une requête web. Le plan d'exécution physique contient de nombreuses tâches. Certaines sont parallèles, d'autres sont liées par une relation producteur-consommateur. Le temps d'exécution d'une requête sur des données massives peut aller de quelques minutes à plusieurs heures. En outre, dans une requête de base de données, si le consommateur n'est pas lancé au moment où les données sont générées par le producteur, alors les données intermédiaires peuvent être stockées sur disque. Le stockage disque dans le cloud peut avoir un coût monétaire. Étant donné que la plupart des méthodes existantes ne conviennent pas à l'interrogation de bases de données. La question que nous pouvons poser est : *(Q3) Comment améliorer les méthodes de dimensionnement automatique basées sur l'apprentissage par renforcement afin qu'elles correspondent aux spécificités de l'interrogation de bases de données ?*

1.3 Contributions

Nous avons proposé des méthodes pour répondre aux questions Q1, Q2 et Q3 de la section précédente. Les contributions de thèse peuvent être résumées comme suit :

— Une méthode d'allocation de ressources statique en deux phases pour assurer un bon compromis entre le bénéfice du fournisseur et la satisfaction des locataires, tout en garantissant un coût d'allocation raisonnable. La méthode est constituée de deux phases successives : *1)* placer les tâches sur les ressources disponibles (placement) et *2)* ordonner les tâches allouées pour chaque ressource en tenant compte des dépendances entre les tâches d'une même requête (ordonnancement). Chaque phase est modélisée par un modèle de Programmation Linéaire en Nombres Entiers (PLNE), de sorte que la solution optimale pour chaque phase peut être faite avec n'importe quel algorithme d'optimisation PLNE. Décomposer le problème en deux phases successives permet de réduite la complexité et donc le temps d'exécution de l'allocation. Nous avons comparé la méthode en deux phases avec une méthode PLNE monophase (que nous avons conçu dans le but de comparer et montrer l'avantage de la

méthode en deux phases) et quelques méthodes gloutons existantes : G-BRT, G-MPT et G-MPM (Kllapi et al., 2011). Nous avons montré que notre méthode offre un bon compromis entre le coût monétaire engendré par l'exécution des requêtes et la satisfaction des locataires, tout en garantissant un coût d'allocation raisonnable.

— Une méthode de réallocation de ressources dirigée par les SLAs pour limiter l'impact des erreurs d'estimation sur le bénéfice. La méthode choisit les collecteurs de statistiques les plus pertinents en se basant d'une part sur la structure de la requête et le contenu du catalogue, et d'une autre part sur la nature des différents SLAs. La méthode favorise de mettre plus de collecteurs dans les requêtes des locataires les plus exigeants. Nous avons comparé notre méthode avec une solution existante qui considère que les requêtes ont le même niveau de priorité (Kabra and DeWitt, 1998; Agarwal et al., 2012; Bruno et al., 2013). Nous avons montré que notre méthode dirigée par SLAs assure un bénéfice meilleur.

— Une méthode de dimensionnement automatique basée sur l'apprentissage par renforcement pour répondre aux spécificités de l'interrogation de bases de données. La principale originalité de la méthode réside dans le fait que nous estimons l'utilisation de ressources et les pénalités à payer dans une fenêtre temporelle future afin de fournir une description plus précise du modèle d'apprentissage. Le calcul des estimations est basé sur le modèle de coûts ainsi que l'allocation de ressources statique. Ces estimations sont injectées dans l'algorithme d'apprentissage par renforcement pour améliorer la capacité d'apprentissage des agents. Une comparaison de notre méthode avec une méthode existante (Dutreilh et al., 2011; Barrett et al., 2013) a montré l'avantage d'introduire le modèle de coûts et la sortie de l'allocation statique dans le modèle d'apprentissage.

Le dimensionnement automatique, le placement et l'ordonnancement constituent le processus d'allocation de ressources élastique.

1.4 Liste des publications

Le travail de thèse a donné lieu à deux articles scientifiques acceptés :

— Mohamed Mehdi Kandi, Shaoyi Yin, and Abdelkader Hameurlain. Avril 2018. An integer linear-programming based resource allocation method for SQL-like queries

in the cloud. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC). Pau, France. ACM, 161–166.

— Mohamed Mehdi Kandi, Shaoyi Yin, and Abdelkader Hameurlain. Août 2019. Resource Auto-scaling for SQL-like Queries in the Cloud based on Parallel Reinforcement Learning. International Journal of Grid and Utility Computing (IJGUC). Inderscience Publishers. vol. 10, no 6, p. 654-671.

Ainsi qu'un article soumis :

— Mohamed Mehdi Kandi, Shaoyi Yin, and Abdelkader Hameurlain. (Under review). SLA-Driven Ressource Re-Allocation for SQL-like Queries in the Cloud. Knowledge and Information Systems (KAIS). Springer.

1.5 Organisation du manuscrit

Ce document est constitué de six chapitres. Après la description du contexte, la problématique et les contributions dans le ***chapitre 1***, le reste de document est organisé comme suit :

Le ***chapitre 2*** est dédié à l'état de l'art. Nous positionnons notre travail par rapport au monde du cloud computing et de l'interrogation parallèle de bases de données. Nous présentons aussi des méthodes d'allocation de ressources et de dimensionnement automatique existantes.

Le ***chapitre 3*** est consacré à la description du service cloud considéré dans la thèse, à notre méthode d'allocation statique ainsi que la méthode de réallocation. Pour la méthode d'allocation statique nous expliquons le principe de la PLNE, puis nous présentons la formulation PLNE du placement et la formulation PLNE d'ordonnancement. Pour la méthode de réallocation, nous détaillons l'algorithme du choix des collecteurs de statistiques et l'algorithme de réallocation.

Le ***chapitre 4*** décrit notre méthode de dimensionnement automatique. Nous commençons par décrire le processus d'allocation de ressources élastique. Ensuite, nous expliquons le principe de l'algorithme Q-learning. Enfin, nous présentons la méthode de dimensionnement automatique basée sur cet algorithme.

Le ***chapitre 5*** est dédié à l'évaluation des performances. Après la description du simulateur utilisé et la génération des plans d'exécution parallèles, nous présentons l'éva-

luation des méthodes proposées (allocation statique, réallocation et dimensionnement automatique). Pour chaque méthode nous décrivons les méthodes avec lesquelles on s'est comparé, les métriques de comparaison, les paramètres et les scénarios de simulation, ainsi que l'analyse des résultats obtenus.

Le ***chapitre 6*** conclue le manuscrit de thèse avec une synthèse de nos contributions ainsi qu'une discussion des voies possibles pour la continuation de notre travail.

Chapitre 2

État de l'art

Contenu

2.1 Introduction

Le contexte général de cette thèse est l'interrogation de bases de données sur des architectures parallèles dans un environnement de cloud computing. Nous nous intéres-

sons principalement aux problèmes de l'allocation de ressources et le dimensionnement automatique.

L'idée derrière le cloud computing a été introduite pour la première fois en 1961, quand John McCarthy [1] présente lors d'une conférence un concept qui recommande à offrir les ressources informatiques comme n'importe quel service public (Parkhill, 1966; Garfinkel, 1999). Cependant, le terme n'est devenu populaire qu'à partir de 2006, quand les premières solutions commerciales ont été officiellement lancées (notamment Amazon Web Services). Depuis, le terme cloud a été largement adopté dans une variété de contextes. Ceci a donné lieu à une confusion par rapport d'un côté à la définition précise du cloud et d'un autre côté aux caractéristiques qui le différencient des grilles de calcul conventionnelles (Bégin et al., 2008; Jha et al., 2009). Cependant, des organismes de standardisation ont proposé des définitions formelles largement adoptées par la communauté (Mell et al., 2011). Des efforts ont aussi été fournis pour différencier les services et modèles de déploiement de cloud.

Le déploiement de bases de données dans le cloud est une pratique très adoptée par les entreprises de nos jours [2] [3] [4] [5]. Les entreprises favorisent l'utilisation du cloud car il permet d'éviter un grand investissement matériel, humain et technique pour la gestion de l'infrastructure en local. Par contre, le déploiement dans le cloud a donné lieu à de nouveaux défis en ce qui concerne la gestion de ressources.

Dans ce chapitre, nous commençons par donner une définition du cloud, ses caractéristiques, les classifications existantes et les concepts clés (section 2.2). Ensuite, nous présentons un aperçu de l'interrogation des bases données sur des architectures parallèles dans un cadre général et dans le cloud (section 2.3 et 2.4). Enfin, nous présentons quelques travaux qui portent sur l'allocation de ressources (section 2.5) et le dimensionnement automatique (section 2.6).

1. l'un des fondateurs de la discipline de l'intelligence artificielle et la technique du temps partagé
2. https ://aws.amazon.com/fr/products/databases/
3. https ://aws.amazon.com/emr/ ?nc1=h$_l s$
4. https ://azure.microsoft.com/en-gb/services/sql-database/
5. https ://azure.microsoft.com/en-gb/services/hdinsight/

2.2 Cloud Computing

2.2.1 Définition et caractéristiques

De nombreux travaux proposent des définitions pour le cloud (Bragg, 2008; de Haaff, 2008; Geelan et al., 2009; Zhang et al., 2010). La principale raison de l'existence de plusieurs définitions est le fait qu'il ne s'agit pas d'une nouvelle technologie, mais plutôt d'un nouveau modèle qui exploite un ensemble de technologies existantes pour répondre à des exigences économiques. La plupart des technologies utilisées par le cloud, notamment la virtualisation, ne sont pas nouvelles (Zhang et al., 2010).

Pour éliminer la confusion par rapport à la définition, des organismes de standardisation ont proposé des définitions formelles pour le Cloud. L'Institut National des Standards et de la Technologie (NIST, Etats-Unis) adopte la définition suivante (Mell et al., 2011) :

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications...) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

Cette dernière peut être traduite en français comme suit :

"Le cloud computing est un modèle permettant un accès réseau à la demande, pratique et omniprésent, à un pool partagé de ressources informatiques configurables (Réseaux, serveurs, stockage, applications...) qui peuvent être rapidement réservées et libérées avec un effort de gestion ou une interaction minimale avec les fournisseurs de services".

Le même organisme définit les caractéristiques principales du cloud comme suit (Mell et al., 2011) :

— Libre-service à la demande : un locataire peut utiliser unilatéralement des capacités informatiques en fonction de ses besoins, sans avoir nécessairement besoin d'une interaction humaine avec le fournisseur du service.

— Large accès au réseau : les services sont accessibles via le réseau par le biais de mécanismes standard qui encouragent l'utilisation de plateformes hétérogènes d'équipement légers ou lourds (téléphones mobiles, ordinateurs portables...).

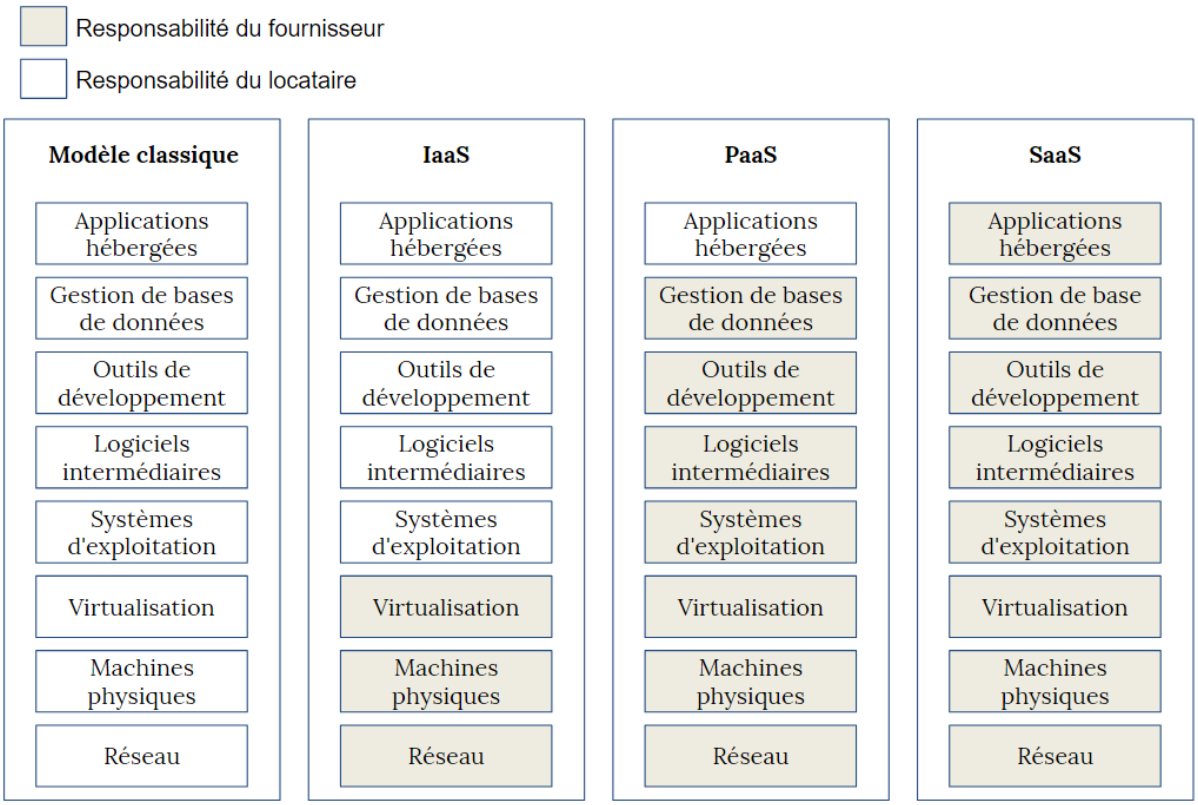


FIGURE 1 – Modèles de service cloud (Kouki, 2013)

— Mise en commun des ressources : les ressources informatiques du fournisseur sont mises en commun pour servir de multiples locataires. Le locataire n'a généralement aucun contrôle ou connaissance sur l'emplacement exact des ressources fournies, mais peut être en mesure de spécifier l'emplacement à un niveau d'abstraction plus élevé, par exemple le pays ou l'état.

— Élasticité rapide : les capacités peuvent être provisionnées et libérées automatiquement afin de s'adapter rapidement à la demande. Pour le locataire, les capacités disponibles semblent souvent illimitées.

— Service mesuré : les systèmes cloud contrôlent et optimisent automatiquement l'utilisation des ressources qui peut être surveillée, contrôlés et signalés, ce qui assure la transparence tant pour le fournisseur que pour le locataire.

2.2.2 Classifications

Les environnements cloud peuvent être classifiés sur la base du modèle de services et aussi le modèle de déploiement (Mell and Grance, 2011; Santana, 2016).

2.2.2.1 Modèles de service

La classification selon le modèle de service se base sur la nature du service fourni. Il y a trois classes de services : Infrastructure as a Service (IaaS), Plateform as a Service (Paas) et Software as a Service (SaaS). La Figure 1 montre la responsabilité du fournisseur et le locataire dans chaque classe. Nous proposons dans ce qui suit une définition basée sur (Mell and Grance, 2011) vu qu'elle a été largement adoptée. Dans la pratique la distinction entre les différents modèles de service n'est pas nette et peut-être légèrement différente d'un fournisseur à un autre.

L'IaaS (Malawski et al., 2015; Karthikeyan, 2018; Talebian et al., 2019) offre des services de bas niveau notamment les ressources de calcul et de stockage. Le fournisseur est responsable du réseau, des machines physiques et la virtualisation. Le locataire peut installer le système d'exploitation et les applications qu'il souhaite. Ce service est destiné principalement aux administrateurs systèmes qui préfèrent louer des ressources de calcul au lieu d'acheter et gérer des ressources en interne.

Le PaaS (Pahl, 2015; Pokahr and Braubach, 2016; Zhong and Yuan, 2019) fournit au locataire la capacité de déployer des applications grâce à des outils de développement pris en charge par le fournisseur. Ce dernier est responsable du réseau, des machines physiques, de la virtualisation, des systèmes d'exploitation, des logiciels intermédiaires, des outils de développement et de la gestion de bases de données. Un fournisseur PaaS peut être vu comme un locataire qui fait appel un fournisseur IaaS pour la gestion du réseau, des machines physiques et la virtualisation. Le PaaS est destiné principalement aux développeurs d'applications et administrateurs de bases de données qui souhaitent avoir un environnement complet pour leur travail sans se soucier des aspects bas niveau.

Le SaaS (Stavrinides and Karatza, 2016; Iranpour and Sharifian, 2018; Stavrinides and Karatza, 2019) est destiné aux locataires qui souhaitent accéder à des applications complètes. Le fournisseur est responsable de tous les niveaux : matériel, virtualisation, système d'exploitation... Il doit assurer une fiabilité et des performances acceptables et similaires à celles obtenues avec un déploiement en local.

2.2.2.2 Modèles de déploiement

La classification selon le modèle de déploiement s'intéresse à la position du cloud par rapport au locataire. Il existe trois modèles de déploiement : public, privé et hybride.

Dans les chapitres suivants, le mot cloud désigne plutôt un fournisseur public si aucune précision n'est rajoutée.

Le cloud public (Firestone et al., 2018; Persico et al., 2018; Xue et al., 2019) est une infrastructure destinée à une utilisation ouverte pour un grand public. C'est la catégorie la plus connue par rapport à celles qui seront présentées par la suite. Malgré ses avantages, il existe des risques pour ce type de déploiement : perte de données, des informations sensibles peuvent être lues par des attaquants, les identifiants et les mots de passe peuvent être volés, des informations confidentielles peuvent être placées par erreur dans un espace de stockage public, la facilité d'allocation peut engendrer une utilisation exagérée et donc des coûts inutiles. De plus pour des raisons de sécurité nationale, certaines informations ne peuvent pas être stockées dans une localisation (pays) non autorisée. Néanmoins les fournisseurs de cloud actuels sont conscients de ces risques, ils proposent donc des mécanismes pour les éviter.

Le cloud privé (Hale and Egerstedty, 2015; Chang, 2015; Griebler et al., 2018) est une infrastructure destinée à l'utilisation par une seule organisation. Par rapport au cloud public, il permet une meilleure sécurité, un meilleur contrôle et gestion des coûts. Le cloud privé peut se situer physiquement au sein même de l'organisation et être géré par ses employés. Néanmoins, le cloud privé peut-être géré par une société externe si l'organisation manque de moyens ou de compétences nécessaires.

Le cloud hybride (Farokhi et al., 2016; Yangui et al., 2016; Azumah et al., 2018) est une composition de plusieurs infrastructures cloud (public, privé). Le but est de profiter des avantages des différentes catégories. Le locataire peut par exemple allouer des ressources dans le cloud public dans les périodes où le cloud privé est surchargé. Le locataire peut aussi utiliser un cloud privé quand une garantie en terme de sécurité est exigée, sinon il utilise le cloud public pour profiter du passage à l'échelle. La difficulté de déploiement d'un cloud hybride se trouve dans le fait que les infrastructures qui le constituent utilisent des technologies et des standards qui peuvent être différents.

2.2.3 Concepts clés

2.2.3.1 Service-Level Agreements (SLAs)

Un SLA est un contrat établi entre le fournisseur et le locataire. Il précise d'une façon formelle les différents objectifs et les actions à prendre dans le cas où ils ne sont pas

respectés. Ce concept était introduit pour la première fois dans les années 80 pour gérer la qualité de service dans la télécommunication. Puis il s'est développé avec l'apparition d'Internet, les architectures orientées services (Ludwig et al., 2003; Andrieux et al., 2007) et le cloud (Nassar, 2018; Zhou et al., 2018; Ibrahim et al., 2018).

Les principaux composants d'un SLA sont les suivants (Jin et al., 2002) :

— L'objet : objectifs à atteindre en utilisant le SLA.

— Les restrictions : les mesures ou actions nécessaires qui doivent être prises pour s'assurer que le niveau demandé services est fourni.

— Période de validité : période de temps d'application du SLA.

— Portée : les services qui seront fournis au locataire et les services qui ne seront pas couverts dans le cadre du SLA.

— Parties : les organisations ou les personnes concernées et leurs rôles (principalement le fournisseur et le locataire).

— Service-level objectives (SLOs) : niveaux de services sur lesquels les parties s'entendent. Parmi les indicateurs, on trouve : la disponibilité et la performance.

— Pénalités : si le service fourni n'atteint pas les SLO des pénalités seront imposées.

— Services facultatifs : services qui ne sont pas obligatoires, mais qui pourraient être requis.

— Administration : processus utilisés pour garantir la réalisation des SLOs et les responsabilités organisationnelles connexes en matière de contrôle de ces processus.

Le cycle de vie des SLA peut être représenté par six étapes (Maarouf et al., 2015) :

— Étape 1 - découvrir les fournisseurs de services : les fournisseurs de services sont situés en fonction des exigences du locataire.

— Étape 2 - définir du SLA : cette étape inclut la définition des services, des parties, les pénalités et les SLOs.

— Étape 3 - établissement d'un accord : le SLA est établi et les parties s'engagent à le respecter.

— Étape 4 - surveillance de la violation du SLA : la performance du service est mesurée par rapport au contrat.

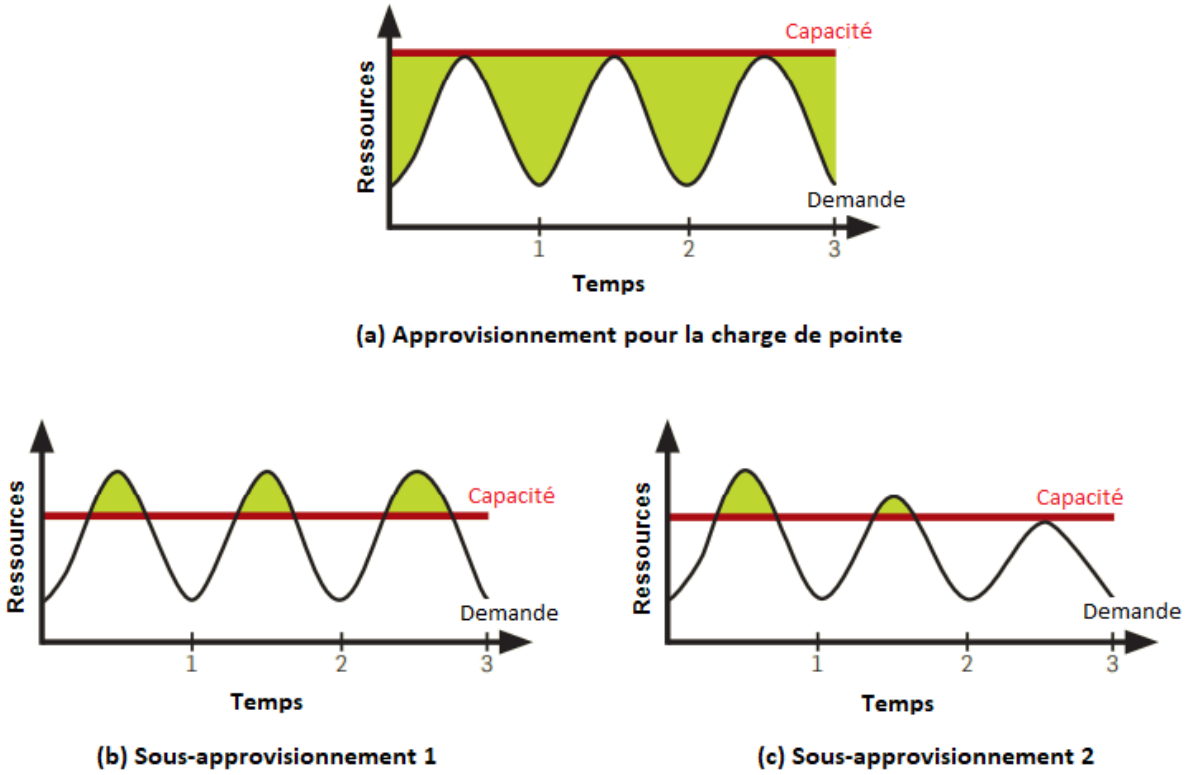


FIGURE 2 – Exemple de sur-approvisionnement et sous-approvisionnement (JoSEP et al., 2010)

— Étape 5 - Terminaison du SLA : le SLA se termine à cause d'un timeout ou d'une violation de la part d'une partie.

— Étape 6 - l'application des pénalités en cas de violation du SLA : si l'une des parties ne respecte pas les termes du contrat, les clauses correspondantes sont invoquées et exécutées.

2.2.3.2 Élasticité

Comme mentionné dans la section 2.2.1, l'élasticité rapide fait partie des caractéristiques principales du cloud. L'élasticité est définie comme étant la capacité d'un système à ajouter et à supprimer des ressources, ou modifier leur configuration, pour s'adapter à la variation de charge en temps réel (Al-Dhuraibi et al., 2017).

Le terme élasticité est associé à l'évolutivité. Cette dernière est définie comme étant la capacité d'un système à supporter des charges de travail croissantes en utilisant des ressources supplémentaires. L'évolutivité est indépendante du temps, il s'agit donc d'une propriété statique. L'élasticité peut être considérée comme une automatisation du concept

d'évolutivité, mais elle vise aussi à optimiser au mieux et le plus rapidement possible les ressources à un moment donné (Al-Dhuraibi et al., 2017).

Nous pouvons citer trois cas qui montrent l'intérêt d'un système élastique par rapport à un système non élastique. Le premier cas est celui où la demande d'un service varie dans le temps. Dans ce cas, la capacité réservée dans un système non élastique doit satisfaite la charge de pointe ce qui entraîne une sous-utilisation à d'autres moments. Au lieu de cela, un système élastique permet de rajouter ou supprimer des ressources en fonction de la charge ce qui limite le gaspillage. Un deuxième cas est celui où la demande est inconnue à l'avance. Par exemple, une organisation qui démarre devra prendre en charge une hausse de la demande lorsqu'elle deviendra populaire, suivie éventuellement d'une réduction lorsque certains visiteurs se détourneront. Enfin, les organisations qui réalisent occasionnellement des traitements parallèles massifs peuvent profiter de l'élasticité pour réserver un nombre important de ressources uniquement quand il y a le besoin (JoSEP et al., 2010).

Le sur-approvisionnement et le sous-approvisionnement sont deux facteurs qui caractérisent un système élastique. Le système est dans un état de sur-approvisionnement quand les ressources attribuées sont supérieures aux ressources nécessaires. Comme c'est le cas de l'approvisionnement qui correspond à la charge de pointe de la Figure 2-(a). La qualité de service est satisfaite par contre cet état entraîne des coûts supplémentaires et inutiles. Le système est dans un état de sous-approvisionnement quand les ressources attribuées sont inférieures aux ressources requises. Comme c'est le cas de l'approvisionnement qui de la Figure 2-(b). Ceci entraîne une dégradation des performances et éventuellement le départ des locataires non satisfaits (Al-Dhuraibi et al., 2017). La Figure 2-(c) montre une baisse des demandes suite au départ des locataires.

Al-Dhuraibi et al. (2017) proposent d'autres classifications des travaux qui concernent l'élasticité. Les solutions sont classées en fonction de la configuration (rigide, configurable), de la portée (infrastructure, plateforme), de l'objectif (performances, coûts, énergie, disponibilité), du mode (réactif, prédictif), de la méthode (horizontale, verticale, hybride), de l'architecture (centralisée, décentralisée) et du fournisseur (unique, multiple).

Le terme élasticité est aussi associé à l'efficacité. Cette dernière dépend de la quantité de ressources consommées pour traiter une quantité de travail donné, plus cette quantité est faible, plus l'efficacité d'un système est élevée. L'allocation et le dimensionnement au-

tomatique visent à répondre aux compromis de la maximisation du bénéfice du fournisseur et la satisfaction des locataires. La définition de l'allocation de ressources et dimensionnement automatique et l'état de l'art des travaux existants sont présentés respectivement dans les sections 2.5 et 2.6.

2.2.3.3 Modèle économique

L'objectif du fournisseur cloud est de garantir le plus grand bénéfice, tandis que l'objectif de chaque locataire est d'obtenir un service maximal à faible coût. Le bénéfice du fournisseur est la différence entre le revenu et les dépenses. Le revenu est la somme d'argent facturée aux locataires qui utilisent les services. Les dépenses sont la somme du coût monétaire lié à l'exécution des services et les éventuelles pénalités dans le cas de violation des SLAs.

Le prix des services est l'une des mesures les plus importantes que le fournisseur doit contrôler pour encourager l'utilisation de ses services et la maximisation de son bénéfice. Il existe différents schémas de tarification, principalement : la tarification fixe et dynamique (Mazrekaj et al., 2016). La tarification fixe comprend le paiement à l'utilisation (les locataires paient en fonction du temps et de la quantité qu'il consomme sur un service spécifique) et l'abonnement (les locataires s'abonnent à une combinaison pré sélectionnée d'unités de service et s'engagent sur une période longue, généralement mensuelle ou annuelle (Chun and Choi, 2014)). Dans la tarification dynamique, le prix est déterminé en fonction de l'offre et la demande. Ce modèle permet au fournisseur de mieux exploiter le potentiel de paiement des locataires et donc de réaliser plus de bénéfice. Le prix peut aussi être choisi suite à une négociation entre le fournisseur et le locataire (Yin et al., 2018). Dans ce cas, un protocole de négociation est utilisé afin de satisfaire les deux parties.

2.3 Bases de données dans le cloud

Les services de bases de données dans le cloud sont considérés comme des produits de niveau PaaS. Ces services sont utilisés soit directement par des utilisateurs finaux (généralement des administrateurs de bases de données) ou bien par des services du niveau SaaS. Les services de bases de données utilisent eux-mêmes des services du niveau IaaS qui fournissent des capacités de calcul et de stockage (Figure 1) La Figure 3 montre les

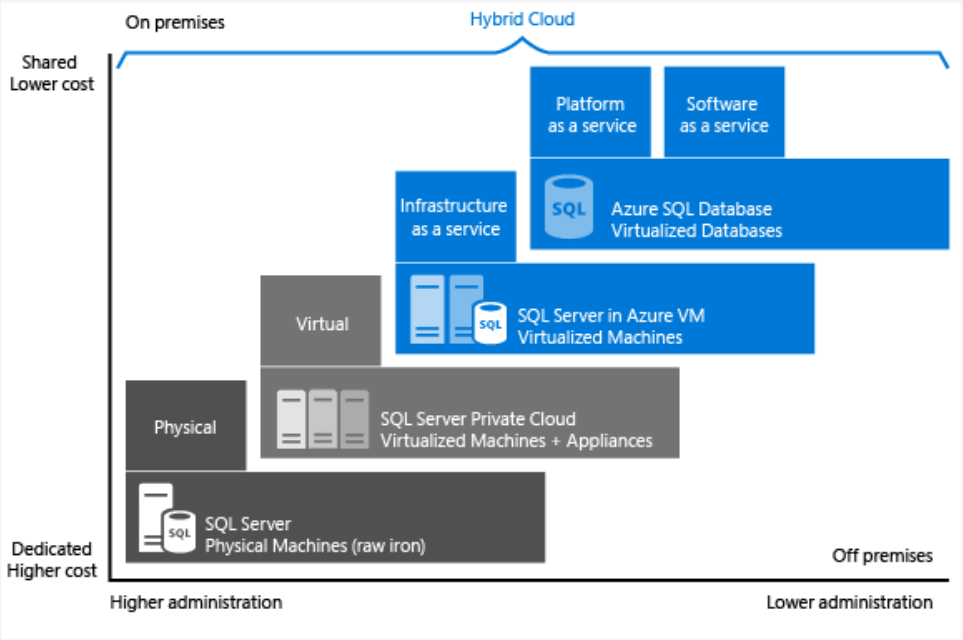


FIGURE 3 – Produits bases de données Cloud de Microsoft Azure

différentes possibilités fournies par Microsoft. Sur la Figure nous avons de gauche vers la droite : (1) la solution privée physique non cloud, (2) la solution cloud privé virtuel, (3) la solution cloud publique virtuel IaaS et (4) la solution cloud publique virtuel PaaS.

Contrairement aux systèmes de gestion des bases de données classiques qui sont installés et maintenus au sein de l'organisation, les services cloud sont proposés généralement par un fournisseur externe. Le fournisseur propose un tarif qui tient compte de la durée de service, des prix proposés par les concurrents et d'un certain nombre de paramètres de performances et de disponibilité.

Le Tableau 2.3.1 présente une comparaison des services de bases de données des principaux fournisseurs du marché. Les critères de comparaison que nous avons considérés sont les suivants :

— **C1 : le type de bases de données considéré.** relationnel (REL), NoSQL, environnement Hadoop (HAD).

— **C2 : infrastructure de calcul.** le service cloud IaaS proposé par le fournisseur pour le calcul.

— **C3 : infrastructure de stockage.** le service cloud IaaS proposé par le fournisseur pour le stockage.

— **C4 : paramètres pris en compte pour la tarification.** nombre ou performances de processeurs (CPU), capacité mémoire (MEM), capacité de stockage (CST), per-

formances réseau (PRS), débit d'entrée/sortie (DEB), nombre de noeuds (NBN), nombre de bases de données (NBD), moteur de traitement (MOT), région géographique (REG), nombre de zones de réplication (NBZ), mode de tarification (MOD), services additionnels inclus (SER).

— **C5 : élasticité.** horizontale : possibilité de changer le nombre de MVs (H), verticale : possibilité de changer le nombre de ressources <cpu, mémoire...> d'une même MV (V).

— **C6 : Service-level agreement.** garanties sur la disponibilité de service (DSP), garanties sur les performances (PRF).

Bien que la plupart des services considèrent surtout la disponibilité pour les SLAs, nous constatons que les fournisseurs actuels commencent à s'intéresser aussi aux performances.

2.4 Interrogation parallèle des bases de données

2.4.1 Analyse et optimisation de requêtes

Les requêtes d'interrogation de bases de données sont souvent exprimées avec un langage déclaratif. Avec un tel langage, l'utilisateur décrit les données qu'il souhaite obtenir sans fournir les algorithmes d'accès aux données. Chaque requête est d'abord analysée lexicalement et syntaxiquement pour vérifier qu'elle est exprimée correctement, puis traduite en un graphe d'opérateurs. Ce graphe subit une optimisation (Gruenheid et al., 2011). Le but de l'optimisation est de trouver un plan d'exécution optimal ou proche de l'optimal. Depuis les années 80, l'optimisation de requêtes a reçu une attention considérable (Jarke and Koch, 1984; Mitschang, 1995). Un optimiseur transforme le graphe d'opérateurs de l'analyse en un plan d'exécution composé d'opérations de bas niveau réalisant l'accès aux données. L'optimisation de requêtes est constituée de deux étapes successives (Figure 4) :

1. ***Optimisation logique :*** permet de réécrire la requête sous une forme logiquement optimisée, sans tenir compte des coûts d'accès aux données. Le but est de réduire la taille de données manipulées en appliquant des règles de transformation des arbres algébriques.
2. ***Optimisation physique :*** choisit le meilleur plan d'exécution (déterminer les algorithmes d'exécution des opérateurs et leur ordre) en tenant compte de la taille

Tableau 2.3.1 – Comparaison des services de bases de données cloud existants

Fournisseur	Service	C1 : modèle de bases de données.	C2 : infra. de calcul.	C3 : infra. de stockage.	C4 : paramètres pris en compte pour la tarification.	C5 : élasticité.	C6 : Service-level agreement.
Amazon	Amazon RDS	REL	EC2	S3	CPU,MEM,PRES, MOT,REG,NBZ, MOD	V	DSP
	DynamoDB	NoSQL			REG,DEB, CST,MOD	V	PRF
	Elastic MapReduce	HAD			REG,CPU,MEM, CST,NBN,MOD	HV	/
Microsoft	Azure SQL DataBase	REL	Azure VM	Azure Disks	REG,ELA,CPU, MEM,DEB,CST, NBD	V	DSP, PRF
	DocumentDB	NoSQL			REG,DEB,CST	V	DSP
	HDInsight	HAD			REG,SER,NBN, CPU,MEM,CST	HV	DSP
Google	Cloud SQL	REL	Google Compute Engine	Google Cloud Storage	REG,CPU,MEM, CST,DEB	V	DSP
	Cloud Bigtable	NoSQL			REG,NBN, CST,DEB	V	DSP
	Cloud Dataproc	HAD			REG,NBN, CPU,MEM	HV	DSP
Oracle	Oracle Database Cloud Service	REL	Oracle Compute	Oracle Storage	SER,MEM	V	DSP
	Big Data Cloud Service	HAD			NBN,CPU	HV	DSP

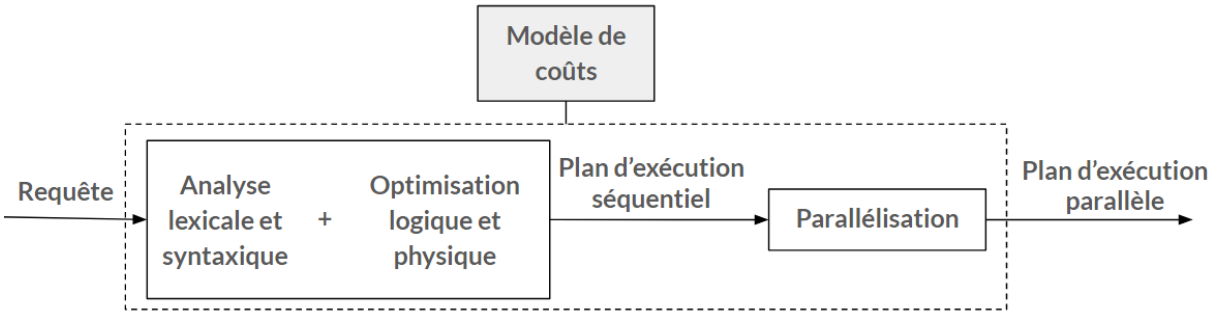


FIGURE 4 – Optimisation de requêtes

des données et des chemins d'accès aux données disponibles.

Pour chaque plan d'exécution, un coût approché est calculé. Pour cela, l'optimisation physique nécessite un modèle de coûts (Hameurlain, 1996). Ce dernier permet de calculer un ensemble de métriques exploitées par l'optimiseur pour trouver le plan d'exécution séquentiel optimal (ou proche de l'optimal). Le plan d'exécution séquentiel optimal (ou proche de l'optimal) généré à l'issue de l'optimisation physique est transformé en plan d'exécution parallèle suite à l'étape de parallèlisation. L'exécution parallèle de requêtes est abordée dans la section suivante.

2.4.2 Parallélisation de requêtes : modèle d'exécution sans-pipeline VS. modèle d'exécution avec-pipeline

Dans le cadre de l'exécution parallèle des requêtes, nous distinguons deux niveaux de parallélisme : le parallélisme inter-requête et le parallélisme intra-requête. Le parallélisme inter-requête consiste à exploiter plusieurs processeurs de l'architecture pour exécuter plusieurs requêtes en même temps. Le parallélisme intra-requête consiste à exploiter plusieurs processeurs de l'architecture pour exécuter une requête donnée. Il y a deux formes de parallélisme intra-requête : le parallélisme inter-opération et le parallélisme intra-opération. Le parallélisme inter-opération consiste à exécuter plusieurs opérations de la même requête en même temps. Ces opérations sont soit indépendantes ou bien consécutives (liées par des flux de données). Le parallélisme intra-opérateur consiste à décomposer une opération donnée en un ensemble de tâches, chacune d'elles est placée sur un seul processeur et exécute l'algorithme de l'opération sur une partie des données (Bonneau, 1999). Certains opérateurs, comme la sélection et la projection, peuvent être facilement décomposés en tâches parallèles. Par contre, pour d'autres opérateurs, comme la jointure, la décom-

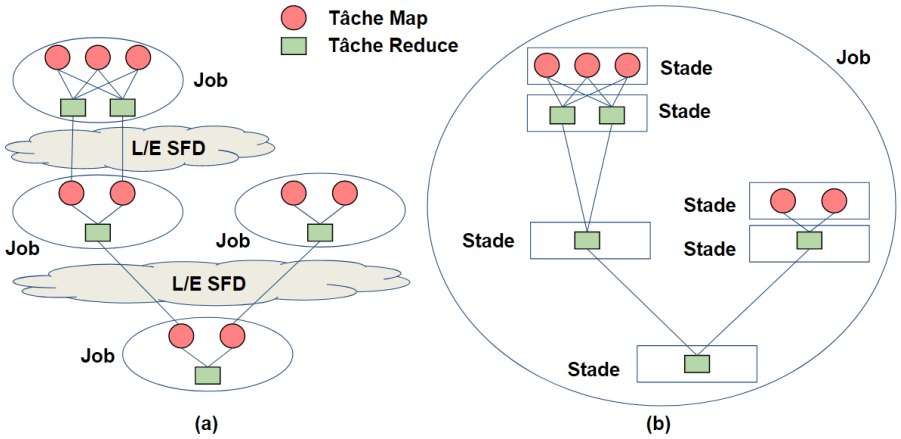


FIGURE 5 – Comparaison de la structure du plan d'exécution parallèle dans (a) le modèle d'exécution sans-pipeline et (b) le modèle d'exécution avec-pipeline

position est plus complexe.

MapReduce est un modèle de programmation pour le traitement de grands ensembles de données. Les utilisateurs spécifient le calcul sous forme de fonctions Map et Reduce (Dean and Ghemawat, 2008). Afin de profiter le parallélisme implicite et la tolérance aux fautes des plateformes qui implémentent MapReduce, des outils ont été développés pour représenter le plan d'exécution parallèle par un ensemble de jobs MapReduce dépendants (White, 2012) (Figure 5-a. Un job MapReduce contient une phase Map et une phase Reduce. Chaque phase est instanciée par un ensemble de tâches parallèles de type Map ou Reduce. L'inconvénient de ce modèle, c'est qu'il y a une opération d'écriture puis une opération de lecture sur un système de fichiers distribué (SFD) entre deux jobs successifs, ce qui ralentit le temps d'exécution des requêtes.

Récemment, un autre modèle a été proposé et déjà intégré dans des outils existants, principalement Hive/Tez (Saha et al., 2015) et SparkSQL (Armbrust et al., 2015). Les opérateurs relationnels sont regroupés en stades (Floratou et al., 2014). Chaque stade est instancié par un ensemble de tâches parallèles (Figure 5-b). Une tâche donnée exécute l'ensemble des opérateurs du stade sur une partie des données en entrée. La communication entre les tâches de deux stades qui se suivent se fait soit par diffusion (i.e. un tuple généré par une tâche du stade producteur est envoyé à toutes les tâches du stade consommateur) ou bien par répartition (i.e. un tuple généré par une tâche du stade producteur est envoyé à une seule tâche du stade consommateur). La lecture à partir du système de fichiers distribué est faite uniquement au début de l'exécution de la requête. L'écriture est faite uniquement à la fin de l'exécution de la requête.

Dans ce document, nous nous référons à l'ancien modèle MapReduce comme le modèle d'exécution sans-pipeline et à la nouvelle représentation comme le modèle d'exécution avec-pipeline.

2.4.3 Architectures parallèles

La communauté de bases de données classe les architectures parallèles selon l'organisation de leur mémoire et disque. Quatre architectures ont été proposées (Figure 6) : architecture parallèle à mémoire partagée (Ranganathan et al., 1998), architecture parallèle à disques partagés, architecture parallèle à mémoire distribuée (Chen et al., 2015; Wan et al., 2016) et architecture hybride.

L'architecture à mémoire partagée présente l'avantage de la simplicité de programmation par contre le réseau d'interconnexion est utilisé pour tout accès mémoire, ce qui fait que le problème d'interférence devient critique. La bande passante du réseau doit correspondre aux besoins de la somme du besoin des processeurs. Donc le nombre de processeurs est limité. L'architecture à mémoire distribuée réduit les interférences car elle minimise les ressources partagées et la mémoire est exploitée indépendamment du réseau d'interconnexion. Cette architecture est plus facilement extensible. Les avantages de l'architecture à disques partagés par rapport aux architectures à mémoire commune, c'est une meilleure extensibilité (car chaque processeur dispose d'une mémoire privée) et la disponibilité (en cas de problème sur un noeud donné, la base de données reste disponible car un autre noeud détecte le problème et réalise une récupération automatique). L'architecture hybride tente de combiner les avantages de l'architecture à mémoire commune et l'architecture à mémoire distribuée (Hameurlain, 1996; Bonneau, 1999). Sur la Figure 6-(d), chaque noeud dispose d'une architecture à mémoire partagée.

2.4.4 Répartition des données

Dans une architecture parallèle, chaque relation est répartie sur plusieurs disques. La répartition doit éviter d'accéder souvent à des données distantes afin de réduire le coût de communication. Il existe deux approches de répartition de données : la répartition totale et la répartition partielle. La répartition totale répartit chaque relation sur tous les disques disponibles. Elle est adoptée dans les architectures à mémoire commune. En effet, les architectures à mémoire partagée contiennent beaucoup de noeud. Le surcoût

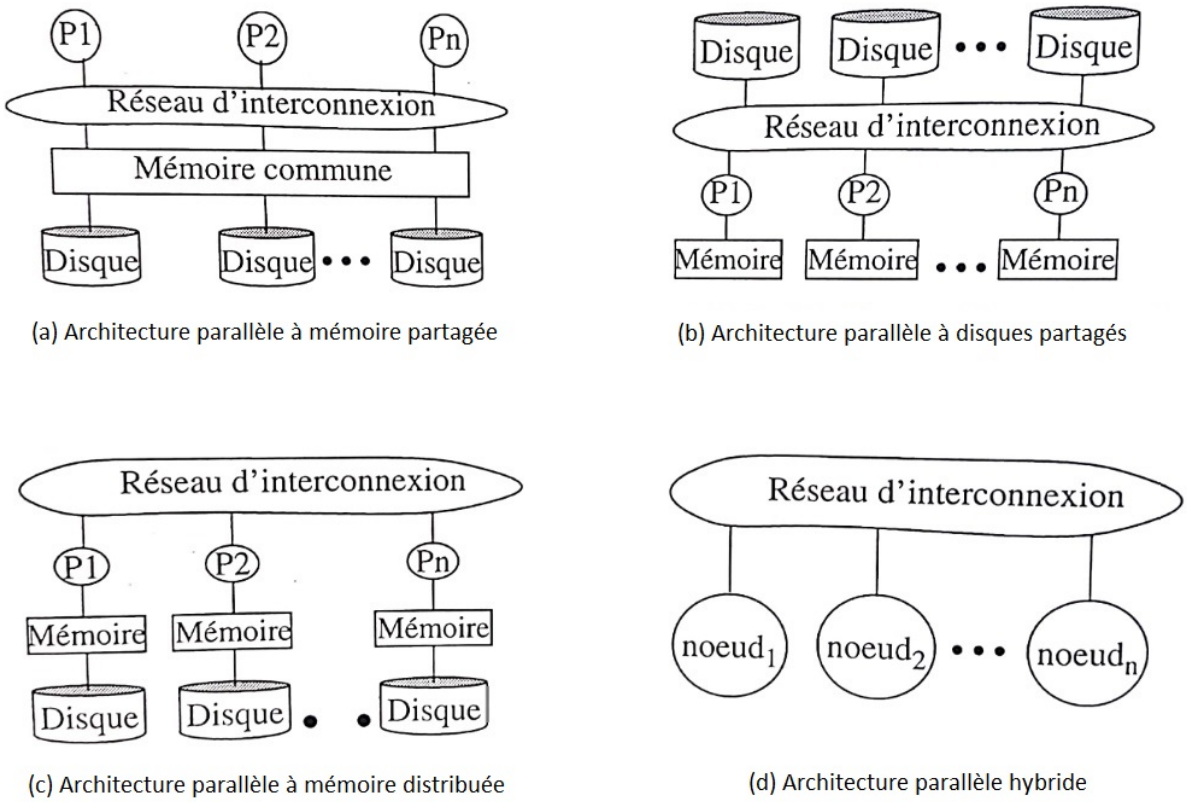


FIGURE 6 – Modèles d'architecture parallèle (Hameurlain, 1996)

de communication engendré par cette approche est très important donc le compromis entre les traitements et la communication ne peut pas être établie. La répartition partielle répartit chaque relation sur un sous-ensemble de disques. Cette approche est plus performante. La difficulté consiste à déterminer le nombre de disques adéquats pour la répartition (Hameurlain, 1996).

Il y a plusieurs façons d'effectuer la répartition de données : répartition circulaire, répartition par intervalle et répartition par hachage (Mehta and DeWitt, 1997). Avec la répartition circulaire, les relations sont réparties tuple par tuple dans l'ordre sur les disques. Avec répartition par intervalle, les relations sont réparties avec des opérations de restriction sur des intervalles par rapport à un attribut choisi par l'administrateur. Avec la répartition par hachage, une fonction de hachage est appliquée à chaque tuple pour déterminer le disque associé. La répartition par hachage peut générer une distribution non équilibrée des données. Il est donc important de bien choisir les attributs de répartition et la fonction de hachage.

2.5 Allocation de ressources

Étant donné les plans d'exécution parallèles d'un ensemble de requêtes à exécuter sur une architecture parallèle, l'allocation de ressources consiste à déterminer "où" et "quand" lancer chaque tâche des plans. En effet, dans une architecture parallèle plusieurs ressources sont susceptibles d'exécuter les tâches. Le choix de la ressource de l'exécution de chaque tâche est appelé placement (où ?), alors que la détermination de l'instant de déclenchement de l'exécution de la tâche est appelé ordonnancement (quand ?).

Formellement, un placement est une application *plac* qui associe une ressource r pour une tâche t :

$$plac : T \rightarrow R \; \forall t \in T, \exists r \in R, plac(t) = r \tag{2.1}$$

L'ordonnancement est une application *ordo* qui associe un instant de déclenchement i pour une tâche t :

$$ordo : T \rightarrow \mathbb{I} \; \forall t \in T, \exists i \in I, ordo(t) = i \tag{2.2}$$

Avec T est l'ensemble des tâches, P est l'ensemble des ressources de l'architecture parallèle et I est un ensemble d'instants possibles.

En fonction du problème, le but est de trouver un placement (resp. ordonnancement) qui maximise (ou minimise) une fonction objectif et respecte un ensemble de contraintes. En général, l'espace de recherche est très grand pour explorer toutes les possibilités en un temps raisonnable.

Nous présentons d'abord les travaux qui portent sur l'allocation de ressources pour l'interrogation parallèle de bases de données non cloud (section 2.5.1), puis les travaux qui portent sur l'allocation de ressources pour l'interrogation parallèle de bases de données dans le cloud (section 2.5.2).

2.5.1 Pour l'interrogation parallèle de bases de données non cloud

Il existe de nombreux travaux qui portent sur l'allocation de ressources pour l'interrogation parallèle des bases de données dans un contexte non cloud. Les anciens travaux

qui datent des années 90 sont destinés aux systèmes de gestion de bases de données relationnels parallèles. Dans les 10 derniers années, de nouveaux travaux sur l'allocation de ressources ont été proposés. Ils sont principalement dédiés aux technologies big data, notamment l'écosystème Hadoop. Le but principal de ces travaux (anciens et nouveaux) est de minimiser le temps de réponse et maximiser l'utilisation des ressources. Certains proposent des méthodes statiques (i.e. allocation de ressources avant l'exécution de la requête) alors que d'autres proposent des méthodes dynamiques (i.e. allocation de ressources pendant l'exécution de la requête). Il y a aussi des travaux qui supposent une allocation statique qui sera modifiée pendant l'exécution en cas d'imprévu. Ceci est appelé la réallocation de ressources. Dans la suite de cette section nous présentons quelques travaux existants.

Salagnon (1994) aborde le problème d'ordonnancement et placement des tâches obtenues après parallélisation des requêtes SQL. Le but de l'ordonnancement est de minimiser le temps de réponse. L'auteur propose un algorithme basé sur des heuristiques. Cet algorithme utilise les dates de début au plus tôt, ainsi que l'aptitude des tâches à se déplacer dans le temps sans augmenter le temps de réponse. Il ajoute la possibilité de modifier la granularité des tâches s'il vient à manquer des processeurs pour leur exécution. Il emploie également une heuristique pour le placement dont le rôle est l'équilibrage des charges. L'auteur a étudié la mise en ouvre dans le cadre des systèmes parallèles à mémoire partagée, puis de ceux a mémoire distribuée.

Garofalakis and Ioannidis (1996) présentent une méthode statique de placement-ordonnancement sur une architecture parallèle à mémoire distribuée. L'algorithme proposé repose sur un modèle multi-dimensionnel d'utilisation de ressources. À chaque opérateur est associé un vecteur possédant une dimension par ressource. L'algorithme proposé prend en compte les parallélismes intra-opération, pipeline et indépendant et la localité des données. Les auteurs supposent qu'il n'y a pas de limitation mémoire. L'algorithme découpe l'arbre d'opérateurs en phases contenant chacune des chaînes pipeline. Les phases sont ordonnées en respectant les contraintes de précédente. Les tâches contraintes par la répartition initiale des données sont placées sur les ressources qui contiennent les données qu'ils utilisent. Le travail est étendu dans (Garofalakis and Ioannidis, 1997) pour une architecture hybride.

Certains travaux ont abordé principalement l'allocation de la mémoire. Nag and De-

Witt (1998) examinent la question de savoir comment allouer de la mémoire aux différents opérateurs d'une requête de manière à minimiser le temps d'exécution total de la requête. Les auteurs considèrent l'allocation statique avant l'exécution de la requête. Ils ont proposé quatre stratégies d'allocations. Tous les algorithmes présentés dans ce travail peuvent fonctionner dans un environnement parallèle avec quelques modifications.

Yu and Cornell (1993) utilisent le concept de retour sur consommation comme base pour l'allocation. Le retour sur consommation mesure l'efficacité de la réduction du temps d'exécution grâce à une consommation mémoire supplémentaire. Une stratégie d'optimisation globale utilisant l'algorithme recuit simulé est développée. Le but est de minimiser le temps d'exécution pour toutes les requêtes, avec la contrainte que la consommation de mémoire totale doit être inférieure à la taille de la mémoire disponible. La méthode sélectionne l'algorithme de jointure et l'allocation pour tous les types de requêtes simultanément. Néanmoins, les auteurs se limitent à des requêtes avec un seul opérateur de jointure.

Parmi les travaux sur la réallocation de ressources pendant l'exécution de la requête, nous citons (Kabra and DeWitt, 1998; Agarwal et al., 2012; Bruno et al., 2013). Dans (Kabra and DeWitt, 1998), le plan produit par l'optimiseur est annoté avec les estimations utilisées par l'optimiseur. Des statistiques sont collectées au moment de l'exécution de la requête. Ces statistiques sont comparées aux statistiques estimées initialement et la différence sert à indiquer si le plan initial est sous-optimal. Les nouvelles statistiques peuvent ensuite être utilisées pour optimiser le reste de la requête. La réoptimisation dans ce travail concerne à la fois l'allocation mémoire mais aussi le plan d'exécution de la requête. Néanmoins, ce travail ne considère pas le parallélisme intra-opérations.

Agarwal et al. (2012); Bruno et al. (2013) conçoivent RoPE, un ré-optimiseur pour le traitement parallèle des données. RoPE collecte certaines propriétés de données lors de l'exécution de la requête. Il adapte les plans d'exécution en transmettant ses propriétés à un optimiseur de requêtes. Parmi les principales nouveautés de ce travail, c'est le fait que la collecte des statistiques se fait dans un environnement parallèle. Les données locales collectées par les différentes tâches sont envoyées à un gestionnaire global qui procède à une agrégation afin d'obtenir les statistiques globales.

Il existe aussi des travaux d'allocation de ressources conçus pour les technologies big data, principalement l'écosystème Hadoop (YARN (Vavilapalli et al., 2013)). L'algorithme

de base implémenté est premier arrivé premier servi. L'allocateur affecte la plus ancienne tâche en attente à la première ressource disponible. L'inconvénient c'est lorsque des tâches longues sont soumises, les tâches courtes qui viennent après doivent attendre longtemps.

FAIR (Zaharia et al., 2010a) est un algorithme d'allocation de ressources qui résout ce problème en considérant l'équité. Cet algorithme garantit que chaque requête reçoit une capacité de ressources minimale tant qu'il y a une demande suffisante. Lorsqu'un utilisateur n'a pas besoin de sa capacité minimale, d'autres utilisateurs sont autorisés à la prendre. Malgré ses avantages, FAIR n'offre pas de mécanismes pour gérer les dates limites de la fin d'exécution.

ARIA (Verma et al., 2011) propose une solution à ce problème. Pour ce faire, il établit un profil qui reflète les caractéristiques de performance pour les phases Map et reduce, puis définit un modèle de performance qui estime le nombre de tâches map et reduce. Enfin, ARIA détermine l'ordre d'exécution des tâches pour le respect des délais basé sur la politique le délai le plus court d'abord.

Cheng et al. (2017) proposent une solution pour améliorer l'utilisation des ressources et les performances des applications dans les clusters multi-locataires Spark-on-YARN. Les auteurs conçoivent un mécanisme de placement de tâches qui tient compte des réservations afin de sélectionner des ressources pour réduire le temps de réservation. Ils conçoivent aussi un mécanisme d'ajustement des ressources en qui tient compte des dépendances afin de contrôler d'une manière adaptative les ressources sous-utilisées par les tâches Reduce. Il développent également un mécanisme d'attribution de tâches afin de coordonner la prise en compte de la localité entre des tâches MapReduce et le placement des tâches Spark. Ce mécanisme vise à augmenter les opportunités d'accès aux données locales sur les noeuds qui hébergent à la fois Spark et des applications MapReduce.

HaSTE (Yao et al., 2019) est un algorithme d'ordonnancement Hadoop YARN basé sur la dépendance de tâches et la demande de ressources. Il vise à utiliser efficacement les ressources pour planifier les tâches MapReduce dans Hadoop YARN et améliorer le temps de réponse. L'algorithme choisit dynamiquement les tâches à exécuter lorsque les ressources deviennent disponibles en fonction de l'aptitude et l'urgence de chaque tâche. L'aptitude fait référence à l'écart entre la demande de ressources des tâches et la capacité résiduelle des nœuds en ressources. La priorité vise à quantifier l'importance d'une tâche. Elle permet de prioriser toutes les tâches et tenir compte des dépendances entre les tâches.

2.5.2 Pour l'interrogation parallèle de bases de données dans le cloud

Parmi les travaux existants d'allocation de ressources dédiés au cloud, (Kllapi et al., 2011) est le plus proche de notre contexte. Ce travail prend en compte les aspects économiques. Les auteurs explorent trois problèmes différents : (1) minimiser le temps d'exécution compte tenu d'un budget fixe, (2) minimiser le coût monétaire compte tenu des dates limites et (3) trouver le bon compromis entre le temps d'exécution et le coût monétaire. Ils proposent des méthodes gourmandes et un algorithme de recherche locale pour allouer les ressources aux tâches dépendantes. Ils montrent que la recherche locale n'améliore pas significativement les résultats par rapport aux méthodes gourmandes. Cependant, les approches gourmandes ne garantissent pas théoriquement la qualité de la solution. Cela peut avoir un impact négatif sur le bénéfice du fournisseur.

Zhao et al. (2016) se concentrent sur l'allocation de ressources pour les applications de type Analytics-as-a-Service (AaaS). Le but est de maximiser le bénéfice du fournisseur AaaS tout en respectant les SLAs. Pour cela, les auteurs modélisent le problème d'allocation de ressources afin de maximiser le bénéfice du fournisseur AaaS tout en minimisant les temps de réponse des requêtes avec des garanties SLA. Ensuite, ils proposent un algorithme d'allocation des ressources pour traiter des charges de travail de requêtes hautement dynamiques et permettre aux utilisateurs d'obtenir des solutions analytiques en temps opportun. Le problème a été formulé avec la programmation linéaire en nombre entier. La formulation des auteurs se situe dans un cadre moins spécifique que celle que nous présenterons par la suite dans cette thèse. En effet, les requêtes peuvent concentrent différentes applications. Elles sont de natures variées : interrogation d'une ou plusieurs tables, un algorithme de machine learning qui joue le rôle de classifieur... De plus, les auteurs considèrent qu'une machine virtuelle est une ressource atomique.

Tan and Babu (2016) positionnent leur travail dans le contexte des bases de données parallèles à plusieurs locataires. Leur solution prend en charge les SLA. Néanmoins, ils ne tiennent compte que des paramètres de performances mais pas les aspects économiques.

Pietri et al. (2019) proposent un algorithme d'allocation de ressources équitables qui permet l'exécution de plusieurs flux de données dans un cloud hétérogène. L'algorithme cherche un équilibre entre le temps d'exécution, le coût monétaire et l'équité, en explorant l'espace de la solution de manière efficace. La méthode proposée est statique. De plus, elle

suppose que l'ensemble des requêtes sont soumises par le même locataire et non pas des locataires différents.

2.6 Dimensionnement automatique de ressources

Le dimensionnement automatique est le mécanisme permettant de choisir le moment d'augmenter et diminuer le nombre de ressources. Les méthodes existantes qui traitent le dimensionnement automatique sont diverses en ce qui concerne l'approche de prise de décision. Chaque méthode a été conçue avec des objectifs particuliers et se concentre sur une architecture cible.

2.6.1 Pour un cadre applicatif général

Les travaux peuvent être classés de différentes manières. Dans cette section, nous donnons un bref aperçu des méthodes existantes et nous nous concentrons sur deux approches : le dimensionnement basé sur les seuils et le dimensionnement basé sur l'apprentissage par renforcement. D'une part, en raison de son aspect intuitif, l'approche basée sur les seuils est adoptée par les fournisseurs actuels de cloud. D'autre part, l'apprentissage par renforcement suscite actuellement l'intérêt de la communauté scientifique et son adoption pour différentes applications et architectures de cloud est une tendance prometteuse.

2.6.1.1 Dimensionnement basé sur les seuils

L'utilisation de seuils est une approche bien connue pour le dimensionnement automatique dans le cloud. L'idée est d'ajouter de nouvelles ressources lorsqu'une certaine mesure dépasse un seuil d'ajout et de supprimer des ressources lorsque la mesure est en dessous d'un seuil de suppression. Les travaux existants peuvent être classés en deux catégories. Certains travaux sont basés sur les valeurs observées de la métrique (Han et al., 2012; Hasan et al., 2012) tandis que d'autres appliquent des techniques de prévision (Khatua et al., 2010). Han et al. (2012) affirment que le dimensionnement ne nécessite pas toujours d'ajouter ou de supprimer des machines virtuelles en permanence. La modification de la capacité des machines virtuelles (CPU et mémoire) peut être effectuée pour obtenir un dimensionnement automatique avec moins de coûts et moins de temps. Ils ont introduit un algorithme léger pour permettre le dimensionnement au niveau du processeur et de

la mémoire. La solution utilise des seuils séparés pour le processeur et la mémoire. Les mesures considérées sont basées sur l'observation et non sur des prédictions. De plus, les coûts de communication ne sont pas pris en compte.

L'aspect communication a été pris en compte dans certains travaux. Hasan et al. (2012) utilise la charge, la gigue et le retard des liens comme métriques. Ils se concentrent sur la relation entre le calcul et le réseau. Ils soulignent le fait que ces domaines sont souvent considérés séparément pour le dimensionnement. Pour remédier à cette limitation, ils ont proposé un mécanisme de seuil qui combine les métriques des domaines de calcul et réseau. Ce travail ne prend également en compte que les valeurs observées pour calculer les mesures.

D'autres travaux proposent d'associer l'approche des seuils à un processus de vote. Chaque MV vote pour une action d'ajout ou suppression de ressources. Le vote est basé sur une ou plusieurs règles gérées par des métriques et des seuils. L'action de dimensionnement est déclenchée si la majorité des MV sont d'accord. Cette solution a été adoptée par Chieu et al. (2011) et Simmons et al. (2011). Il existe également des travaux qui combinent l'approche des seuils avec d'autres approches de dimensionnement. Ghanbari et al. (2011) proposent une politique d'élasticité utilisant à la fois la théorie du contrôle et des approches basées sur les seuils. Contrairement à l'utilisation de seuils qui est plus intuitive, la théorie du contrôle est basée sur la modélisation mathématique.

Tous les travaux cités ci-dessus utilisent les valeurs observées pour calculer les métriques. D'autres travaux se sont penchés sur la prévision des valeurs futures. Nous mentionnons, par exemple, Khatua et al. (2010) qui utilise la théorie des séries chronologiques (Mills and Mills, 1991) pour prévoir les valeurs futures. Si l'une des valeurs prévues dépasse le seuil prédéfini, un événement est déclenché.

La nature intuitive de l'approche des seuils a attiré les fournisseurs de cloud. Toutefois, le choix des paramètres à prendre en compte et les valeurs des seuils de manière efficace exigent une intervention humaine et une compréhension approfondie des tendances de la charge de travail, ce qui n'est pas facile à réaliser. Une autre approche indépendante de l'intervention humaine intéresse la communauté scientifique. Il est basé sur le renforcement de l'apprentissage.

2.6.1.2 Dimensionnement basé sur l'apprentissage par renforcement

L'apprentissage par renforcement a été adopté pour le dimensionnement automatique dans certains travaux cloud. Les méthodes existantes peuvent être différenciées par le mode de dimensionnement, l'algorithme d'apprentissage et la technique utilisée pour accélérer l'apprentissage.

Le mode de dimensionnement dans le cloud peut être horizontal ou vertical. Dans le dimensionnement horizontal, les actions possibles sont d'ajouter ou de supprimer des ressources (Dutreilh et al., 2011). Dans le dimensionnement vertical, au contraire, le nombre de ressources est fixe et les actions possibles consistent à ajuster leur configuration en termes de CPU et de mémoire (Rao et al., 2009, 2011).

L'algorithme typique d'apprentissage par renforcement est le Q-learning classique mais certains travaux utilisent une variante appelée Sarsa (Tesauro et al., 2006). Plus de détails sur la différence entre le Q-learning standard seront donnés dans la section 4.3 de ce document.

Malgré leurs avantages, les algorithmes d'apprentissage par renforcement présentent certains problèmes, notamment le temps d'apprentissage important et la taille de l'espace d'état. Dutreilh et al. (2011) introduisent une politique gourmande pour trouver une bonne initialisation des valeurs d'apprentissage, une technique d'accélération de la convergence, et la détection des changements de modèles de performance. Rao et al. (2009) proposent dans un premier temps d'adopter un modèle d'apprentissage par renforcement global pour le dimensionnement vertical. Dans ce modèle, l'état est décrit par la quantité de CPU/mémoire de toutes les machines virtuelles du cloud. Cette approche donne lieu à un très grand nombre d'états et donc beaucoup de temps pour explorer suffisamment le modèle. Ensuite, ils considèrent dans (Rao et al., 2011) que chaque MV a son propre modèle local et dans ce cas, l'état est défini par la quantité de CPU/mémoire de cette MV uniquement pour que la complexité soit réduite.

Barrett et al. (2013) utilisent une version parallèle de l'apprentissage mais le niveau de généralité est limité au niveau MV (pas au niveau des ressources logiques), le dimensionnement automatique est indépendant de la programmation du placement et les spécificités des requêtes d'interrogation de bases de données ne sont pas prises en compte dans la fonction description d'état et récompense.

Le niveau de granularité de la plupart des travaux existant sur le dimensionnement

automatique est limité au niveau machine virtuelle (MV). Les requêtes sont considérées comme des entités atomiques. Dans notre travail, nous considérons un niveau de granularité plus fin. Une MV contient un ensemble de ressources logiques et chaque requête est décomposée en un graphe de tâches avec dépendances. Chaque ressource logique utilise une quantité spécifique de ressources physiques (CPU, mémoire et disque) sur une machine spécifique. Un job (ou un stade) contient un ensemble de tâches parallèles. Le problème est donc beaucoup plus complexe si on le compare aux applications web.

Enfin, la gestion des données intermédiaires est généralement négligée dans les travaux existants. C'est une caractéristique importante pour les applications de bases de données. En effet, si le prochain consommateur de données intermédiaires n'est pas disponible immédiatement, ces données peuvent être stockées sur le disque. L'utilisation de la capacité de stockage dans le cloud a un coût monétaire qui peut ne pas être négligeable. Les disques sur Amazon S3, par exemple, sont facturés en fonction de la taille et de la durée de stockage.

2.6.2 Pour l'interrogation de bases de données

Peu de travaux de dimensionnement automatique dans le cloud sont dédiés à l'interrogation de bases de données. Certains travaux ont proposé des solutions de dimensionnement automatique pour les bases de données dans le cloud, mais ils se sont concentrés sur des technologies spécifiques. Par exemple, MongoDB (Huang et al., 2013) ou Hadoop (Gandhi et al., 2016). Ces travaux ont considéré des métriques de performances (et non monétaires) dans leurs propositions.

Certains travaux se concentrent sur les bases de données NoSQL. Par exemple, TIRAMOLA est un framework open-source permettant d'effectuer un dimensionnement automatique des clusters NoSQL selon des politiques définies par l'utilisateur (Konstantinou et al., 2012; Angelou et al., 2012; Tsoumakos et al., 2013). Il existe également des travaux sur le dimensionnement horizontale des bases de données NoSQL en tenant compte des coûts (Naskos et al., 2015, 2017, 2018). Ces propositions sont également fondées sur les processus de décision markoviens (PDM) et utilisent la vérification du modèle probabiliste comme principal mécanisme de décision.

2.7 Conclusion

Dans cette thèse, nous adoptons la définition de cloud présentée par Mell et al. (2011) (section 2.2.1). Nous nous intéressons principalement, dans le cadre de notre problématique de recherche, à la mise en commun des ressources et l'élasticité rapide. Nous considérons un fournisseur PaaS publique. Pour cela, le mot fournisseur désigne un fournisseur PaaS publique dans la suite du document si aucune précision n'est rajoutée. Par rapport aux SLOs, nous nous intéressons aux performances. Nous supposons qu'un seuil est fixé pour chaque requête soumise au cloud, et le fournisseur paye des pénalités au locataire si le temps d'exécution est supérieur au seuil fixé. Nous considérons une élasticité à un niveau abstrait : rajouter et supprimer des ressources logiques. Une ressource logique est définie par la suite dans la section 3.2.2. Par rapport au modèle économique, nous supposons que le prix est négocié au préalable entre le fournisseur et ses locataires. La détermination du prix ne fait pas partie des contributions de cette thèse.

Nous considérons le parallélisme inter-requête et intra-requête. Notre travail traite à la fois le modèle d'exécution sans-pipeline et avec-pipeline. L'architecture parallèle considérée est de type hybride.

Les contributions qui seront présentées dans les deux chapitres suivants portent sur les problèmes d'allocation de ressources et dimensionnement automatique.

Chapitre 3

Stratégie d'allocation de ressources

Contenu

3.1 Introduction

Dans le chapitre précédent, nous avons donné une synthèse des travaux qui portent d'un côté sur l'allocation de ressources et d'un autre côté sur le dimensionnement automatique. Dans ce chapitre, nous présentons notre stratégie d'allocation de ressources (réponse aux questions Q1 et Q2 posées dans la section 1.2). Notre méthode de dimensionnement automatique est abordée dans le chapitre suivant.

Nous rappelons que l'allocation de ressources est constituée du placement des tâches du plan d'exécution parallèle et de leur ordonnancement. Le placement détermine "où" exécuter chaque tâche alors que l'ordonnancement détermine "quand" commencer leur

exécution. Le problème d'allocation de ressources est abordé bien avant la naissance du cloud. Par contre, ce dernier a introduit de nouveaux défis. En effet, le contrat établi peut-être différent d'un locataire à un autre. Pour maximiser le bénéfice du fournisseur et garantir la satisfaction des locataires, il faut allouer les ressources d'une façon qui favorise les locataires les plus exigeants et assurant au même temps une certaine qualité pour les locataires les moins exigeants. Le coût de l'allocation doit aussi être raisonnable. Nous avons présenté les travaux existants qui portent sur l'allocation de ressources dans la section 2.5 du chapitre précédent. Certaines méthodes existantes sont statiques (i.e. le plan d'allocation est établi avant le lancement de l'exécution de requête) alors que d'autres sont dynamiques (i.e. l'allocation est effectuée pendant l'exécution de requête). Il y a aussi des travaux qui portent sur la réallocation de ressources (i.e. ils supposent l'existence d'une allocation statique déjà établie, puis cette dernière est modifiée dynamiquement si une sous optimalitée est détectée). L'allocation statique se base sur le modèle pour calculer des estimations pour le nombre de tuples et la taille des relations intermédiaires. Néanmoins, les estimations peuvent contenir des erreurs. Ce qui fait que le plan d'allocation statique choisi peut être sous optimal. Il devient donc important d'adopter un mécanisme pour limiter l'impact des erreurs d'estimation sur le bénéfice du fournisseur sachant qu'il s'agit d'un environnement multi-locataire.

Nous proposons dans ce chapitre notre stratégie d'allocation de ressources constituée d'une méthode d'allocation de ressources statique et une méthode de réallocation pour limiter l'impact des erreurs d'estimations. La méthode d'allocation de ressources statique est décomposée en deux phases. Chaque phase est modélisée par un modèle de Programmation Linéaire en Nombres Entiers (PLNE). La méthode de réallocation de ressources choisit les collecteurs de statistiques les plus pertinents en se basant sur la structure de la requête, le contenu du catalogue, mais aussi de la nature du contrat établi entre le fournisseur et le locataire.

Nous commençons d'abord par donner, dans la section 3.2, une description du service cloud considéré. Ensuite, nous présentons, dans la section 3.3, notre méthode d'allocation de ressources statique. Enfin, nous abordons, dans la section 3.4, la méthode de réallocation de ressources.

```
SELECT l_orderkey, sum(l_extendedprice * (1 - l_discount))
       as revenue, o_orderdate, o_shippriority
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING'
      and c_custkey = o_custkey
      and l_orderkey = o_orderkey
      and o_orderdate < '1995-03-22'
      and l_shipdate > '1995-03-22'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, o_orderdate
LIMIT 10;
```

FIGURE 7 – Exemple de requête quasi-SQL

3.2 Description du service cloud considéré

Nous considérons qu'un système de gestion de bases de données est mis en place chez un fournisseur de cloud et qu'un ensemble de requêtes $\mathcal{Q}$ appartenant à différents locataires sont soumises via des interfaces client. D'abord, l'optimiseur de requêtes transforme chaque requête en un plan d'exécution physique parallèle (sous-section 3.2.1). Ensuite, le gestionnaire d'allocation de ressources global effectue l'allocation de ressources statique pour l'ensemble des requêtes. Enfin, l'exécution des requêtes est lancée sur l'infrastructure matérielle à disposition du fournisseur (sous-section 3.2.2). Si une erreur d'estimation est détectée pendant l'exécution alors une réallocation de ressources est envisagée. L'allocation statique et la réallocation sont guidées par un modèle de coûts (sous-section 3.2.3) et un modèle économique (sous-section 3.2.4).

3.2.1 Paradigme d'exécution de requêtes

Chaque requête soumise par un locataire est d'abord analysée pour vérifier si elle est lexicalement et syntaxiquement correcte. Ensuite, elle est transformée en un arbre d'opérateurs relationnels (sélections, projections, jointures et agrégations) appelé plan d'exécution logique. Enfin, un plan d'exécution physique séquentiel est généré. Ce dernier indique l'algorithme d'exécution de chaque opérateur. Un opérateur du plan d'exécution physique séquentiel peut contenir plusieurs phases, comme c'est le cas de la jointure par hachage qui contient une phase build et une phase probe.

Prenant l'exemple de la requête quasi-SQL représentée sur la Figure 7. La requête concerne trois relations : *customer*, *orders* et *lineitem*. La requête est transformée en un

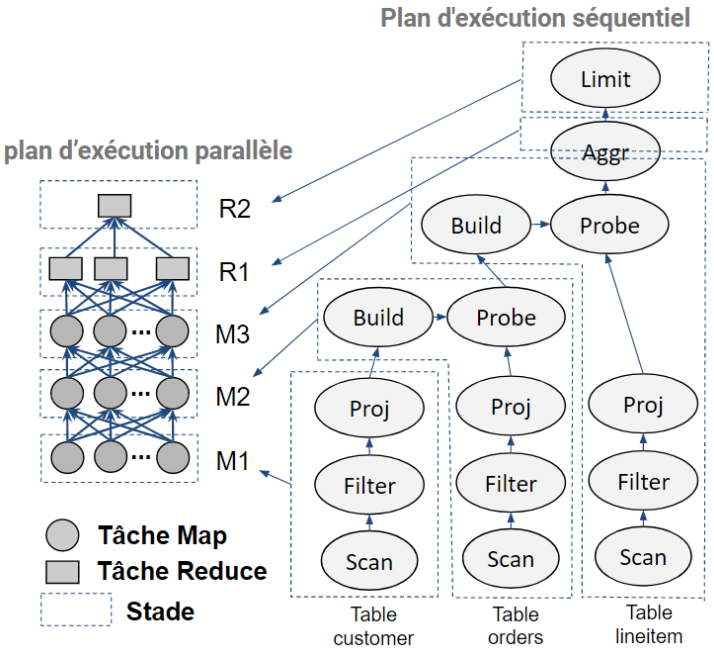


FIGURE 8 – Du plan d'exécution séquentiel au plan d'exécution parallèle

plan d'exécution logique puis un plan d'exécution physique séquentiel.

Comme indiquée sur la partie droite de la Figure 8, le plan d'exécution physique séquentiel de la requête de l'exemple contient trois parcours de tables ($Scans$) trois sélections ($Filter$), trois projections ($Proj$), deux equi-jointures par hachage ($Build + Probe$) et une agrégation ($Aggr$).

Le plan d'exécution séquentiel est ensuite transformé en un plan d'exécution parallèle. Le plan d'exécution parallèle avec-pipeline de l'exemple précédent (requête de la Figure 7) est représenté sur la partie gauche de la Figure 8. Il est constitué de trois stades Map (M1,M2,M3) et deux stades Reduce (R1, R2). Les communications M1→M2 et M2→M3 sont de type diffusion alors que les communications M3→R1 et R1→R2 sont de type répartition. Le stade M1 effectue un parcours de table, une sélection et une projection sur la relation *customer*. Le stade M2 effectue un parcours de table et une sélection sur la relation *orders*, ensuite, une equi-jointure entre le résultat de la sélection et le résultat de la phase M1. Le stade M3 effectue un parcours de table, une sélection et une projection sur la relation *lineitem*, puis une equi-jointure entre le résultat de la projection et le résultat de la phase M2, et enfin une agrégation locale. R4 effectue une agrégation globale alors que R5 limite le résultat à 10 tuples puis les écrit dans le système de fichiers distribué.

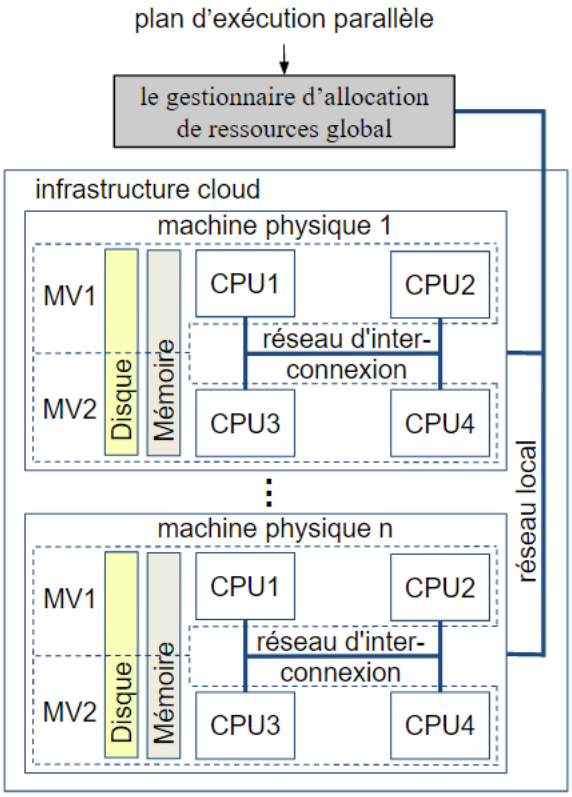


FIGURE 9 – Architecture de l'infrastructure cloud considérée

3.2.2 Architecture de l'infrastructure considérée

L'architecture que nous considérons est de type hybride. Cette dernière correspond à une architecture parallèle à mémoire globalement distribuée et localement partagée (Hua et al., 1991; Valduriez, 1993). L'infrastructure est constituée d'un ensemble de machines physiques. Chaque machine physique contient des ressources (CPU, mémoire et disque) distribuées entre des machines virtuelles. La mémoire d'une machine virtuelle donnée est partagée entre ses CPUs. Les différentes machines physiques communiquent via un réseau local (LAN). Les CPU appartenant à la même machine physique communiquent via un réseau interne d'interconnexion. Dans l'exemple de la Figure 9, nous avons deux machines virtuelles par machine physique et chaque machine virtuelle dispose de deux CPU.

Le gestionnaire d'allocation de ressources global reçoit le plan d'exécution parallèle des requêtes soumises afin d'effectuer l'allocation de ressources. Chaque tâche élémentaire est affectée à une ressource logique particulière pendant une certaine période donnée. Nous définissions une ressource logique comme étant une représentation abstraite d'une CPU et une certaine quantité de mémoire. Une ressource logique donnée est caractérisée par la vitesse du CPU, l'emplacement physique dans l'infrastructure et la taille mémoire

minimale garantit. À un instant donné, une ressource logique peut contenir zéro ou une seule tâche.

Pour que l'exécution d'un opérateur relationnel soit la plus rapide possible, il est nécessaire que la quantité mémoire attribuée soit suffisante pour contenir les données manipulées en faisant au plus un seul accès disque. Néanmoins, il n'est pas toujours possible de satisfaire complètement le besoin en mémoire étant donné le nombre important de requêtes. Chacune des tâches ne peut utiliser qu'une partie de la mémoire disponible. La répartition de la mémoire entre les tâches doit respecter la limite de disponibilité mémoire tout en garantissant le minimum entrés/sorties disque possible. Dans l'idéal, il est souhaitable de récupérer les données du disque uniquement une fois (one-pass algorithme), voir deux fois si ce n'est pas possible de le faire en une seule fois (two-pass algorithme).

La sélection et la projection parcourent les tuples de la relation un par un, effectuent un traitement sur chaque tuple indépendamment des autres, puis déplacent les tuples sélectionnés ou les tuples projetés vers la sortie. Ainsi, le même bloc de tuples est récupéré une seule fois seulement à partir du disque (one-pass algorithme). De plus, une seule page mémoire suffit quelle que soit la taille de la relation en entrée.

D'autres opérateurs comme la jointure et l'agrégation ont besoin de plus de mémoire. Par exemple pour exécuter un algorithme de jointure par hachage il est préférable dans l'idéal que la mémoire attribuée soit supérieure à la plus petite des deux relations (one-pass algorithme). Si ce n'est pas le cas, les données doivent d'abord être chargées en mémoire pour les répartir en seaux qui rentrent dans la mémoire disponible, puis charger une deuxième fois, un sceau à la fois, pour exécuter l'algorithme de jointure par hachage (two-pass algorithme). Dans ce cas, si la taille de la petite des deux relations est B blocs alors la taille mémoire minimale nécessaire est approximativement de l'ordre de $\sqrt{B}$ pages (Garcia-Molina, 2008).

3.2.3 Modèle de coûts

Nous présentons dans cette section le modèle de coûts de l'exécution avec-pipeline. Le modèle de coûts de l'exécution sans-pipeline est présenté dans l'annexe A.

Le temps d'exécution estimé (ET_{stat}) d'une requête $q \in \mathcal{Q}$ peut être calculé en se basant sur le plan d'exécution parallèle, l'allocation de ressources ainsi que les paramètres des Tableaux 3.2.1, 3.2.2 et 3.2.3. Il est représenté comme suit dans le cas idéal où les

Tableau 3.2.1 – Paramètres de base du modèle de coûts

Paramètre	**Signification**
R_s	la première (ou l'unique) relation en entrée du stade s
S_s	la deuxième relation en entrée du stade s
O_s	la relation en sortie du stade s
$\|R\|$	nombre de tuples dans R
σ	sélectivité des sélections
ψ	sélectivité des jointures
π	la fraction du tuplet conservée après la projection
$limit$	la taille de données retournées suite à l'application de l'opérateur $limit$
f	la fraction des données qui peut être gardée en mémoire dans le cas d'un algorithme two-pass
pd_s	nombre de tâches dans le stade s (degré de parallélisme)

Tableau 3.2.2 – Paramètres système du modèle de coûts

Paramètre	**Signification**
db_l	bande passante du disque local
db_{sfd}	bande passante du SFD
cpu	vitesse du traitement
nb	bande passante du réseau
nd	latence du réseau
nif	nombre d'instructions pour appliquer une sélection sur un tuple
nip	nombre d'instructions pour appliquer une projection sur un tuple
nih	nombre d'instructions pour appliquer une fonction de hachage sur un tuple
nis	nombre d'instructions pour trouver une correspondance dans la table de hachage
nij	nombre d'instructions pour joindre deux tuples
nia	nombre d'instructions pour appliquer une agrégation à un tuple

Tableau 3.2.3 – Paramètres déduits du modèle de coûts

Paramètre	**Signification**
$\|\|R\|\|$	taille de R en octets
t_{filter}	le temps pour appliquer un filtre sur un tuple (t_{filter}=nif/cpu)
$t_{projection}$	le temps pour appliquer une projection sur un tuple ($t_{projection}$=nip/cpu)
t_{hash}	le temps pour appliquer une fonction de hachage sur un tuple (t_{hash}=nih/cpu)
t_{search}	le temps pour trouver une correspondance dans la table de hachage (t_{search}=nis/cpu)
t_{join}	le temps pour joindre deux tuples (t_{join}=nij/cpu)
t_{agg}	le temps pour appliquer une agrégation à un tuple (t_{agg}=nia/cpu)

différents stades de la requête q sont exécutés l'un après l'autre sans chevauchement avec d'autres requêtes qui partagent les mêmes ressources :

$$ET_{stat}(q) = \sum_{s \in \mathcal{S}_q} et_{stat}(s) \tag{3.1}$$

Avec $\mathcal{S}_q$ est l'ensemble des stades de la requête q. $et_{stat}(s)$ est le temps d'exécution estimé d'une tâche du stade s. En cas de chevauchement avec d'autres requêtes qui partagent les mêmes ressources, le calcul de $ET_{stat}(q)$ doit tenir compte de l'ordonnancement imposé par l'allocation de ressources.

Le temps d'exécution estimé d'une tâche du stade s est calculé comme suit :

$$et_{stat}(s) = T_{op}(s) + T_{rep}(s) + T_{tr}(s) \tag{3.2}$$

$T_{op}(s)$ est le temps estimé des opérateurs exécutés par le stade s. Il dépend donc de la nature des opérateurs qui constituent s, mais aussi de l'algorithme utilisé (one-pass ou two-pass). Nous donnons comme exemple, dans l'annexe B, le détail de calcul de $T_{op}(s)$ pour les stades de la requête de la Figure 7.

$T_{rep}(s)$ est le temps estimé pour répartir les données en sortie du stade s. Ce temps est calculé seulement dans le cas où la communication est en répartition entre le stade s et le stade consommateur des sorties de s :

$$T_{rep}(s) = \begin{cases} t_{hash} * (|O_s|/pd_s) & \text{, si répartition.} \\ 0 & \text{, si diffusion ou stade final.} \end{cases} \tag{3.3}$$

$T_{tr}(s)$ est le temps estimé pour transférer les données entre le stade s et le stade suivant $(next(s))$:

$$T_{tr}(s) = \begin{cases} ((||O_s||/(pd_s * pd_{next(s)}))/nb + nd) * max(pd_s; pd_{next(s)}) & \text{, si répartition.} \\ ((||O_s||/pd_s)/nb + nd) * max(pd_s; pd_{next(s)}) & \text{, si diffusion.} \\ 0 & \text{, si stade final.} \end{cases} \tag{3.4}$$

3.2.4 Modèle économique

Chaque requête $q \in \mathcal{Q}$ a un prix PR_q, une date limite DL_q et un poids de pénalités PW_q. Les valeurs de ces paramètres sont négociés au préalable entre le fournisseur et le locataire (Yin et al., 2018). PR_q est le montant que le fournisseur reçoit de la part du locataire suite au succès de l'exécution de la requête. Le fournisseur paye des pénalités au locataire si le temps d'exécution de la requête dépasse la date limite DL_q. Le poids de pénalités PW_q est la somme d'argent que le fournisseur paye pour chaque unité de temps d'exécution après la date limite. Le temps d'exécution $ET(q)$ est la durée qui sépare l'instant de la soumission de la requête à l'infrastructure et la fin de son exécution. Le coût des pénalités par requête $PC(q)$ est calculé comme suit :

$$PC(q) = \begin{cases} (ET(q) - DL_q) * PW_q \text{ , si } ET(q) > DL_q \\ 0 \text{ , sinon} \end{cases} \tag{3.5}$$

La requête q utilise un ensemble de ressources logiques ($\mathcal{R}_q$), de l'espace disque et le réseau. Chaque ressource logique $r \in \mathcal{R}_q$ contient une CPU et une partie de la mémoire disponible. Le coût monétaire d'utilisation de l'infrastructure $IC(q)$ est la somme du coût monétaire de l'utilisation des ressources logiques $RC(q)$, le disque $DC(q)$ et le réseau $NC(q)$. Il est calculé comme suit :

$$IC(q) = RC(q) + DC(q) + NC(q) \tag{3.6}$$

Le bénéfice par requête q est :

$$B(q) = PR_q - (PC(q) + IC(q)) \tag{3.7}$$

Le bénéfice global du fournisseur est le suivant :

$$B = \sum_{q \in \mathcal{Q}} B(q) \tag{3.8}$$

3.3 Méthode d'allocation de ressources statique

Nous proposons une méthode basée sur la Programmation Linéaire en Nombres Entiers (PLNE) pour le problème de l'allocation des ressources statique. Étant donné les ressources logiques attribuées au service cloud, l'objectif de notre méthode d'allocation est de trouver un placement et un ordonnancement qui réduisent le coût monétaire pour le fournisseur du cloud. La solution proposée adopte une approche en deux phases. Tout d'abord, le placement implique le choix d'une ressource logique pour chaque tâche des requêtes soumises (phase 1). Ensuite, une fois le placement effectué, l'ordonnancement consiste à ordonner les tâches placées en tenant compte des dépendances entre les stades et la disponibilité des ressources logiques (phase 2). Nous avons proposé une formulation PLNE pour chaque phase. Décomposer le problème en deux phases successives permet de réduire la complexité et donc le coût de l'allocation. Nous compensons par expliquer le principe de la Programmation Linéaire en Nombres Entiers (sous-section 3.3.1). Ensuite, nous présentions notre formulation PLNE pour le modèle d'exécution avec-pipeline (sous-section 3.3.2), en commençant par la phase de placement (sous-section 3.3.2.1) puis la phase d'ordonnancement (sous-section 3.3.2.2). La formulation PLNE pour le modèle d'exécution sans-pipeline est présentée dans l'annexe C.

3.3.1 Principe de la Programmation Linéaire en Nombres Entiers (PLNE)

Un problème de Programmation Linéaire (PL) est un problème d'optimisation dans lequel nous cherchons à trouver les valeurs d'un ensemble de variables (x_1, x_2..., x_n) qui maximise (ou minimise) une fonction objectif linéaire z en satisfaisant un ensemble de contraintes linéaires. Un problème de Programmation Linéaire en Nombres Entiers (PLNE) est un problème de programmation linéaire dans lequel au moins l'une des variables est limitée à des valeurs entières (Chen et al., 2010). Un problème PLNE est formulé mathématiquement comme suit, dans le cas d'une fonction objectif à minimiser :

$$\begin{aligned}
\text{minimize} \quad & z = \sum_j c_j * x_j + \sum_k d_k * y_k \\
\text{subject to} \quad & \sum_j a_{ij} * x_j + \sum_k g_{ik} * y_k \leq b_i \quad (i = 1, 2, ...m) \\
& x_j \geq 0 \qquad\qquad (j = 1, 2, ...n) \\
& y_j = 0, 1, 2... \qquad\qquad (k = 1, 2, ...p)
\end{aligned}$$

Avec m est le nombre de contraintes, n est le nombre de variables continues, p est le nombre de variables entières, $(c_1, c_2, ...c_n)$ et $(d_1, d_2, ...d_p)$ sont les coefficients de la fonction objectif, $(a_{11}, a_{12}, ...a_{1n}, ...a_{m1}, a_{m2}, ...a_{mn})$, $(g_{11}, g_{12}, ...g_{1p}, ...g_{m1}, g_{m2}, ...g_{mp})$ et $(b_1, b_2, ...b_n)$ sont les coefficients des contraintes, $(x_1, x_2, ...x_n)$ sont les variables continues, $(y_1, y_2, ...y_p)$ sont les variables entières.

Plusieurs approches de résolution des problèmes de programmation linéaires, et leurs variantes, ont été proposées par le passé, à savoir : $branch - and - bound$ (Lawler and Wood, 1966), *cutting plane* (Balas et al., 1993) et *group theoretic* (Shapiro, 1968). La solution dominante aujourd'hui est $branch - and - cut$ (Padberg and Rinaldi, 1987; Audet et al., 2007; Yu et al., 2019). Il existe aujourd'hui plusieurs solveurs logiciels de PLNE, notamment : GLPK, CPLEX et Gurobi. L'utilisateur introduit la formulation mathématique de son problème dans un langage haut niveau puis c'est le solveur qui s'occupe de trouver la solution optimale du problème.

3.3.2 Formulation PLNE pour le modèle d'exécution avec-pipeline

La formulation PLNE (contraintes et fonction objective linéaires) du placement et d'ordonnancement avec-pipeline est présentée dans ce qui suit. Les Tableaux 3.3.1, 3.3.2 et 3.3.3 définissent respectivement les ensembles, les paramètres et les variables utilisés.

3.3.2.1 Formulation PLNE du placement (1^{ere} phase)

Nous introduisons les familles de variables x :

— $x_{i,m,r}$ indique si la tâche $m \in \mathcal{T}_i$ du stage $i \in \mathcal{S}$ est placée sur la ressource $r \in \mathcal{P}$ $(= 1)$ ou non $(= 0)$.

Il s'agit donc de variables binaires :

Tableau 3.3.1 – Notation utilisée pour les ensembles de la formulation PLNE avec-pipeline

Ensembles	
$\mathcal{S}$	Ensemble des stades de toutes les requêtes soumises
$\mathcal{T}_i$	Ensemble des tâches du stade $i \in \mathcal{S}$
$\mathcal{P}$	Ensemble de ressources logiques
$\mathcal{F}$	Ensemble des stades finals des requêtes soumises
$\mathcal{C}$	Ensemble de types de ressources, chaque type est caractérisé par sa capacité de mémoire et son coût monétaire

$$x_{i,m,r} \in \{0,1\},\ \forall i \in \mathcal{S}, m \in \mathcal{T}_i, r \in \mathcal{P} \tag{3.9}$$

La quantité mémoire minimale C^*_{mem} pour une tâche map ne doit pas dépasser la quantité mémoire minimale garantit C_{mem} par la ressource choisie :

$$C^*_m(i) * x_{i,m,r} \leq C_m(T_y(r)),\ \forall i \in \mathcal{S}, m \in \mathcal{T}_i, r \in \mathcal{P} \tag{3.10}$$

Chaque tâche est placée sur une et une seule ressource logique :

$$\sum_{r \in \mathcal{P}} x_{i,m,r} = 1,\ \forall i \in \mathcal{S}, m \in \mathcal{T}_i \tag{3.11}$$

Afin d'assurer le parallélisme intra-stage, deux tâches appartenant au même stage ne peuvent pas être placées dans la même ressource :

$$\sum_{m \in \mathcal{T}_i} x_{i,m,r} \leq 1,\ \forall i \in \mathcal{S}, r \in \mathcal{P} \tag{3.12}$$

Nous ajoutons la contrainte (3.14) pour améliorer la répartition équitable des tâches entre les ressources. Sans cette contrainte, nous avons constaté que les tâches ne sont pas réparties de manière équilibrée sur les ressources. Les tâches sont plus susceptibles d'être sur la même ressource. Cela réduit les coûts réseau, mais il y a un risque d'être dans des situations où nous avons des ressources trop occupées (risque de dépassement des délais) et d'autres ressources sous-exploitées. L'ajout de la contrainte (3.14) permet de prendre en compte la répartition équilibrée des tâches sur les ressources disponibles. Le compromis entre l'équilibrage de charge et les coûts réseau est exprimé par la suite dans la fonction

Tableau 3.3.2 – Notation utilisée pour les paramètres de la formulation PLNE avec-pipeline

paramètres	
$T_y(r)$	Type de ressource $r \in \mathcal{P}$, $Ty(r) \in \mathcal{C}$
$C^*_{mem}(i)$	Quantité mémoire minimale pour une une tâche du stade $i \in \mathcal{S}$
$C_{mem}(c)$	uantité mémoire minimale garantit par la ressource de type $c \in \mathcal{C}$
$F_{r,t}$	Indique si la ressource a est initialement disponible à l'instant t $(= 1)$ ou non $(= 0)$, $r \in \mathcal{P}$
$Q_{i,j}$	Quantité de données transférées entre une tâche du stage $i \in \mathcal{S}$ et une tâche du stage $j \in \mathcal{S}$
T_i	Temps de réponse local d'une tâche du stage $i \in \mathcal{S}$
T	Nombre de fenêtres temporelles futures prises en compte
$Dist(r_1, r_2)$	Distance entre la ressource $r_1 \in \mathcal{P}$ et la ressource $r_2 \in \mathcal{P}$
W_{proc}	Poids d'utilisation du processeur
W_{mem}	Poids d'utilisation de la mémoire
W_{stor}	Poids associé au stockage des résultats intermédiaires
W_{com}	Poids de la communication
W_{rep}	Poids de l'équilibrage de charge
$C(r)$	Coût de la ressource logique $r \in \mathcal{P}$ $(C(r) = W_{proc} + W_{mem} * C_{mem}(T_y(r)))$
$C_{com}(r_1, r_2)$	Coût de la communication entre les ressources logiques r_1 et $r_2 \in \mathcal{P}$ $(C_{com}(r_1, r_2) = W_{com} * Dist(r_1, r_2))$
D_i	Date limite de la requête à laquelle appartient le stage $i \in \mathcal{S}$
W_i	Poids de pénalité associé à chaque fenêtre de temps d'exécution après la date limite du stage $\in \mathcal{S}$
q_i	Quantité estimée de données générées des tâches du stage $i \in \mathcal{S}$
$A_{i,m}$	Indique la ressource logique dans laquelle la tâche $m \in \mathcal{T}_i$ du stade $i \in \mathcal{S}$ est placés suite à la phase de placement
$S_{i,j}$	Indique si le stage $i \in \mathcal{S}$ et $j \in \mathcal{S}$ sont liés par des liens non pipeline, $S_{i,j} \in \{0, 1\}$ (non pipeline signifie que la tâche j peut commencer à partir du moment où la tâche i se termine complètement)
$P_{i,j}$	Indique si le stade $i \in \mathcal{S}$ et $j \in \mathcal{S}$ sont liés par des liens pipeline, $P_{i,j} \in \{0, 1\}$ (pipeline signifie que la tâche j peut démarrer dès que la tâche i génère sa première sortie)

Tableau 3.3.3 – Notation utilisée pour les variables de la formulation PLNE avec-pipeline

Variables du placement	
$x_{i,m,r}$	Indique si la tâche $m \in \mathcal{T}_i$ du stage $i \in \mathcal{S}$ est placée sur la ressources $r \in \mathcal{P}$ $(= 1)$ ou non $(= 0)$
z_{r_1,r_2}	Quantité maximale de données transférées entre la tâche placée sur la ressource $r_1 \in \mathcal{P}$ et la tâche placée sur la ressource $r_2 \in \mathcal{P}$
α	variable fictive utilisée dans la contrainte (3.14)
Variables de l'ordonnancement	
$v_{i,m,t}$	Indique si la tâche $m \in \mathcal{T}_i$ du stade $i \in \mathcal{S}$ commence avant, pendant $(= 1)$ ou après $(= 0)$ l'instant $t \in 0, ..., T$
$u_{i,m,t}$	Indique si les résultats intermédiaires de la tâche $m \in \mathcal{T}_i$ du stade $i \in \mathcal{S}$ sont stockés sur disque à l'instant $t \in \{0, ..., T\}$ $(= 1)$ ou non $(= 0)$
β	Famille de variables fictives utilisées dans les contraintes (3.28) et (3.29)

objectif. Nous introduisons la variable α. La fonction objective que nous présentons plus loin inclut α comme variable à minimiser.

$$\alpha \in \{0, 1....., T\} \tag{3.13}$$

$$\sum_{i \in \mathcal{S}} \sum_{m \in \mathcal{T}_i} T_i * x_{i,m,r} + \sum_{t<T} (1 - F_{r,t}) \leq \alpha, \ \forall r \in \mathcal{P} \tag{3.14}$$

La variable z_{r_1,r_2} indique la quantité maximale de données transférées entre la tâche placée sur la ressource $r_1 \in \mathcal{P}$ et la tâche placée sur la ressource r_2. Cette définition satisfait :

$$z_{r_1,r_2} \in \{0, 1...., UpperBound(z)\}, \ \forall r_1, r_2 \in \mathcal{P} \tag{3.15}$$

$$x_{i,m,r_1} = 1 \text{ and } x_{j,r,r_2} = 1 \Rightarrow z_{r_1,r_2} \geq Q_{i,j}$$

$$\forall i, j \in \mathcal{S}, m \in \mathcal{T}_i, r \in \mathcal{T}_j, r_1, r_2 \in \mathcal{P}, Q_{i,j} > 0 \tag{3.16}$$

Cette contrainte peut être exprimée linéairement comme suit :

$$Q_{i,j} * x_{i,m,r_1} + Q_{i,j} * x_{j,r,r_2} - z_{r_1,r_2} \leq Q_{i,j},$$

$$\forall i, j \in \mathcal{S}, m \in \mathcal{T}_i, r \in \mathcal{T}_j, r_1, r_2 \in \mathcal{P}, Q_{i,j} > 0 \tag{3.17}$$

La fonction d'objectif f prend en compte les coûts des ressources et réseau. Elle tient

également compte de l'équilibrage de charge :

$$f = \sum_{i \in \mathcal{S}} \sum_{m \in \mathcal{T}_i} \sum_{r \in \mathcal{P}} C(r) * T_i * x_{i,m,r} + \sum_{r_1 \in \mathcal{P}} \sum_{r_2 \in \mathcal{P}} C_{com}(r_1, r_2) * z_{r_1,r_2} + W_{rep} * \alpha$$

La formulation PLNE du problème du placement est :

$$\begin{aligned} &\text{minimize} && f \\ &\text{subject to} && (3.9), (3.10), (3.11), (3.12), (3.13), (3.14), (3.15), (3.17) \end{aligned}$$

La solution du problème est trouvée avec le logiciel GLPK.

3.3.2.2 Formulation PLNE de l'ordonnancement (2^{eme} phase)

Le résultat du modèle de placement est considéré comme une entrée pour le modèle d'ordonnancement que nous présentons dans cette section.Nous recherchons maintenant les fenêtres de temps allouées à chaque tâche. Les variables suivantes sont introduites :

— $v_{i,m,t}$ indique si la tâche $m \in \mathcal{T}_i$ du stade $i \in \mathcal{S}$ commence avant, pendant ($= 1$) ou après ($= 0$) l'instant t.

Nous pouvons déduire de la définition de la famille de variables v que :

$$v_{i,m,t} \in \{0, 1\},\ \forall i \in \mathcal{S}, m \in \mathcal{T}_i, t < T \tag{3.18}$$

$$v_{i,m,t} \leq v_{i,m,t+1},\ \forall i \in \mathcal{S}, m \in \mathcal{T}_i, t < T \tag{3.19}$$

Une ressource ne peut pas exécuter plus d'une tâche à la fois (contrainte d'exclusivité). De la définition de la famille de variables v et sachant qu'une tâche ne peut être interrompue avant sa fin, on peut le déduire que $v_{i,m,t} - v_{i,m,t-T_i} = 1$ si la tâche m du stade i utilise la ressource r au moment t ; $= 0$ sinon. La formulation linéaire de la contrainte d'exclusivité est la suivante :

$$\sum_{i\in\mathcal{S}}\sum_{\substack{m\in\mathcal{T}_i\\A_{i,m}=r}}\sum_{t-T_i\geq 1}(v_{i,m,t}-v_{i,m,t-T_i})$$
$$+\sum_{i\in\mathcal{S}}\sum_{\substack{m\in\mathcal{T}_i\\A_{i,m}=r}}\sum_{t-T_i<1}v_{i,m,t}\leq F_{r,t},\ \forall r\in\mathcal{P},t<T \tag{3.20}$$

Nous proposons la formulation suivante pour la contrainte de dépendance entre les tâches. Pipeline signifie que la tâche j peut démarrer dès que la tâche i génère sa première sortie. Non pipeline signifie que la tâche j peut commencer à partir du moment où la tâche i se termine.

$$v_{j,r,t}-v_{i,m,t-T_i}\leq 1-S_{i,j},$$
$$\forall i,j\in\mathcal{S},m\in\mathcal{T}_i,r\in\mathcal{T}_j,t-T_i\geq 1 \tag{3.21}$$
$$v_{j,r,t}\leq 1-S_{i,j},\ \forall i,j\in\mathcal{S},r\in\mathcal{T}_j,t-T_i<1 \tag{3.22}$$
$$v_{j,r,t}-v_{i,m,t}\leq 1-P_{i,j},$$
$$\forall i,j\in\mathcal{S},m\in\mathcal{T}_i,r\in\mathcal{T}_j,t<T \tag{3.23}$$

Les résultats intermédiaires d'une tâche sont maintenus sur l'espace de stockage local jusqu'à ce que les tâches successives commencent. La variable $u_{i,m,t}$ indique si les résultats intermédiaires de la tâche $m\in\mathcal{T}_i$ du stade $i\in\mathcal{S}$ sont stockés sur disque à l'instant t $(=1)$ ou non $(=0)$:

$$u_{i,m,t}\in\{0,1\},\ \forall i\in\mathcal{S},m\in\mathcal{T}_i,t<T \tag{3.24}$$
$$v_{i,m_1,t}=1 \text{ and } v_{j,m_2,t}=0\Rightarrow u_{i,m_1,t}=1$$
$$\forall i,j\in\mathcal{S},m_1\in\mathcal{T}_i,m_2\in\mathcal{T}_j,t<T,S_{i,j}=1 \text{ or } P_{i,j}=1 \tag{3.25}$$

La contrainte (3.25) peut être exprimée linéairement comme suit :

$$S_{i,j} * v_{i,m_1,t} + S_{i,j} * (1 - v_{j,m_2,t}) - u_{i,m_1,t} \leq 1,$$
$$\forall i, j \in \mathcal{S}, m_1 \in \mathcal{T}_i, m_2 \in \mathcal{T}_j, t < T, S_{i,j} > 1 \tag{3.26}$$
$$P_{i,j} * v_{i,m_1,t} + P_{i,j} * (1 - v_{j,m_2,t}) - u_{i,m_1,t} \leq 1,$$
$$\forall i, j \in \mathcal{S}, m_1 \in \mathcal{T}_i, m_2 \in \mathcal{T}_j, t < T, P_{i,j} > 1 \tag{3.27}$$

Les coûts monétaires qui influencent l'ordonnancement des tâches sont les pénalités et le stockage des résultats intermédiaires. L'objectif est de trouver la combinaison de v et u qui minimise ce coût. Chaque requête a une date limite spécifiée dans les SLAs. Le cumul des pénalités commence lorsque l'exécution de la requête dépasse le délai. La fonction objective à minimiser est la suivante. La première (resp. deuxième) ligne représente le coût de pénalité (resp. coût disque) :

$$g = \sum_{i \in \mathcal{F}} \sum_{D_i - T_i < t < T} W_i * \max_{r \in \mathcal{T}_i} (1 - v_{i,r,t})$$
$$+ W_{stor} * \sum_{i \in \mathcal{S}} \sum_{m \in \mathcal{T}_i} \sum_{t < T} q_i * u_{i,m,t}$$

Cette fonction objectif est non linéaire. Pour avoir une forme linéaire, nous introduisons la famille des variables β telles que :

$$\beta_{i,t} \in \{0, 1\}, \ \forall i \in \mathcal{J}, t < T \tag{3.28}$$
$$1 - v_{i,r,t} \leq \beta_{i,t} \ \forall i \in \mathcal{S}, r \in \mathcal{T}_i, t < T \tag{3.29}$$

La fonction objectif peut être exprimée linéairement comme suit :

$$g' = \sum_{i \in \mathcal{F}} \sum_{D_i - T_i < t < T} W_i * \beta_{i,t} +$$
$$W_{stor} * \sum_{i \in S} \sum_{m \in T_i} \sum_{t < T} q_i * u_{i,m,t}$$

La formulation PLNE du problème d'ordonnancement est :

$$\begin{aligned} &\text{minimize} && g' \\ &\text{subject to} && (3.18), (3.19), (3.20), (3.21), (3.22), (3.23), \\ & && (3.24), (3.26), (3.27), (3.28), (3.29) \end{aligned}$$

La solution du problème est trouvée avec le logiciel GLPK.

3.4 Méthode de réallocation de ressources

La méthode d'allocation de ressources statique présentée dans la section 3.3 est guidée par le modèle de coûts. Ce dernier utilise des formules analytiques et des statistiques contenues dans le catalogue pour estimer le nombre et la taille des tuples des relations intermédiaires. Ces estimations permettent de mesurer le coût de plusieurs plans afin de choisir le meilleur. Nous notons le meilleur plan statique trouvé $Popt_{stat}$.

Néanmoins, le nombre de tuples pendant l'exécution peut être différents des estimations calculées au préalable à l'étape statique. Ceci est dû à la nature approximative des formules analytiques, mais aussi le fait que le contenu du catalogue n'est pas nécessairement à jour. Les erreurs d'estimation se propagent d'une façon exponentielle sur les opérateurs successifs d'une requête complexe q (Ioannidis and Christodoulakis, 1991). La conséquence, c'est que le temps d'exécution réel peut être significativement supérieur au temps d'exécution initialement estimé puis éventuellement $B_{real} < B_{stat}$ (B_{real} est le bénéfice réel suite à l'exécution des requêtes, B_{stat} est le bénéfice estimé suite à l'allocation initiale de ressources). Ce qui fait que le plan d'allocation $Popt_{stat}$ n'est pas nécessairement le meilleur au cours de l'exécution de la requête (sous optimalité).

Le problème consiste d'une part à trouver un moyen pour détecter les erreurs d'estimation pendant l'exécution de la requête avec peu de surcharge et en considérant le fait que les locataires ont différents niveaux d'exigence. D'autre part, générer une autre allocation de ressources dynamiquement $Popt_{dyna}$ si des erreurs sont détectées afin de maximiser le bénéfice défini par la formule (3.8).

Afin de remédier aux erreurs d'estimation, des collecteurs de statistiques sont injectés dans les différentes requêtes (Figure 10). Certaines statistiques peuvent être calculées facilement avec un minimum de charge (comme c'est le cas du nombre de tuple d'une re-

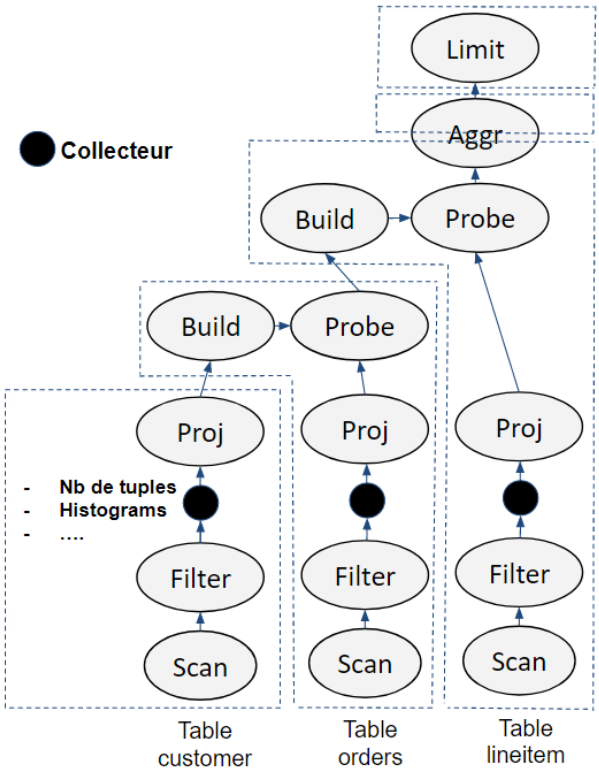


FIGURE 10 – Plan d'exécution séquentiel avec des collecteurs de statistiques

lation intermédiaire) alors que d'autres sont plus gourmandes en terme de charge (nombre de valeurs uniques d'un attribut et histogrammes). Les statistiques collectées permettent d'obtenir de meilleures estimations de la taille des données et nombre de tuples des relations intermédiaires.

Le fait d'ajouter tous les collecteurs potentiels possibles d'une requête q peut être significativement coûteux. Pour cette raison, la méthode place uniquement les collecteurs potentiels les plus pertinents (sous-section 3.4.1). Étant donné que nous considérons un environnemt parallèle, les statistiques collectées par une tâche donnée ne constituent qu'une partie locale des statistiques globales. Chaque tâche envoie les statistiques locales à un manager qui calcule les statistiques globales puis décide de déclencher la réallocation de ressources si une sous optimalité est détectée (sous-section 3.4.2). Contrairement aux travaux existants, nous proposons une méthode dirigée par les SLAs.

3.4.1 Algorithme du choix des collecteurs de statistiques pertinents

Les travaux précédents (Kabra and DeWitt, 1998; Agarwal et al., 2012; Bruno et al., 2013) placent les collecteurs pertinents en supposant que l'ensemble des requêtes ont le même niveau de priorité. Cependant, locataires dans le cloud ont différents niveaux d'exigences. Certains sont moins tolérants au retard que d'autres. Le but du fournisseur est de satisfaire les besoins des locataires tout en maximisant son bénéfice. Contrairement aux travaux de l'état de l'art, nous proposons une approche dirigée par les SLAs (Algorithme 1).

Dans l'algorithme 1, la liste des collecteurs potentiels est d'abord triée par ordre de pertinence via une fonction $sort()$. Les travaux existants proposent un ordre de pertinence basé sur deux critères (Kabra and DeWitt, 1998) : (1) la probabilité que l'estimation soit inexacte (potentiel d'inexactitude) ainsi que (2) le nombre d'opérateurs affectés par la statistique et n'ayant pas encore été exécutés. Ce deuxième critère ne prend pas en compte le parallélisme. Nous proposons donc de remplacer le critère (2) par le nombre de tâches des stades affectées et n'ayant pas encore été exécutées. En effet, certains stades sont constitués d'un nombre important de tâches alors que d'autres ne contiennent que peu, notre nouveau critère constitue donc un paramètre plus représentatif dans un environnement parallèle. Pour définir l'ordre de pertinence, les collecteurs sont d'abord comparés avec le critère (1), puis selon le critère (2) en cas d'égalité.

Le temps total de la collecte globale de statistiques (l'ensemble des requêtes) ne doit pas dépasser un certain seuil calculé à partir du temps total de chaque requête :

$$q.tempsTotal = \sum_{s \in \mathcal{S}_q} et_{stat}(s) * pd_s, \ \forall q \in \mathcal{Q} \tag{3.30}$$

$$tempsCollectGlobal = \alpha * \sum_{q \in \mathcal{Q}} q.tempsTotal; 0 < \alpha < 1 \tag{3.31}$$

Ce temps de collecte global est réparti sur l'ensemble des requêtes $q \in Q$ en se basant sur un poids par requête. Le poids est calculé à partir du contenu des SLAs. Chaque requête q est caractérisée par sa date limite DL_q ainsi que la somme monétaire que le fournisseur paye pour chaque unité de temps d'exécution après la date limite PW_q (pénalités).

Algorithm 1 Choix des collecteurs de statistiques pertinents dirigée par les SLAs

1: // Trier les collecteurs potentiels par ordre de pertinence
2: **for** chaque requête $q \in \mathcal{Q}$ **do**
3: $\mathcal{C}_q \leftarrow \emptyset$
4: **for** chaque stade $s \in \mathcal{S}_q$ **do**
5: $\mathcal{C}_q \leftarrow \mathcal{C}_q \cup$ { les collecteurs potentiels du stade s }
6: **end for**
7: $\mathcal{C}_{sort}(q) \leftarrow sort(\mathcal{C}_q)$
8: **end for**
9: // Calculer le temps total Global de la collecte de statistiques
10: $tempsTotalGlobal \leftarrow 0$
11: **for** chaque requête $q \in \mathcal{Q}$ **do**
12: $q.tempsTotal \leftarrow 0$
13: **for** chaque stade $s \in \mathcal{S}_q$ **do**
14: $q.tempsTotal \leftarrow q.tempsTotal + et_{stat}(s) * pd_s$
15: **end for**
16: $tempsTotalGlobal \leftarrow tempsTotalGlobal + q.tempsTotal$
17: **end for**
18: $tempsCollectGlobal \leftarrow \alpha * tempsTotalGlobal$
19: // Définir la répartition du temps de collecte total
20: $somme \leftarrow 0$
21: **for** chaque requête $q \in \mathcal{Q}$ **do**
22: $q.poids \leftarrow (\beta * (1/DL_q) + \gamma * PW_q) * q.tempsTotal$
23: $somme \leftarrow somme + q.poids$
24: **end for**
25: **for** chaque requête $q \in \mathcal{Q}$ **do**
26: $q.tempsTotalCollect \leftarrow (q.poids/somme)*$tempsCollectGlobal
27: **end for**
28: // Sélectionner les collecteurs à retenir
29: **for** chaque requête $q \in \mathcal{Q}$ **do**
30: $temps \leftarrow 0$
31: $\mathcal{C}_q \leftarrow \emptyset$
32: **for** chaque collecteur $c \in \mathcal{C}_{sort}(q)$ **do**
33: $c.tempsTotal \leftarrow (T_{local}(c) + T_{tr}(c)) * pd_{stade(c)} + T_{global}(c)$
34: **if** $temps + c.tempsTotal < q.tempsTotalCollect$ **then**
35: $\mathcal{C}_q \leftarrow \mathcal{C}_q \cup$ {c}
36: $temps \leftarrow temps + c.tempsTotal$
37: **else**
38: Break
39: **end if**
40: **end for**
41: **end for**

$$q.poids = (\beta * (1/DL_q) + \gamma * PW_q) * q.tempsTotal \tag{3.32}$$

$$q.tempsTotalCollect = (q.poids / \sum_{q' \in \mathcal{Q}} q'.poids) * tempsCollectGlobal,\ \forall q \in \mathcal{Q} \tag{3.33}$$

Enfin, pour chaque requête $q \in \mathcal{Q}$ nous commençons à rajouter les collecteurs les plus pertinents à l'ensemble des collecteurs à retenir ($\mathcal{C}_q$) un par un (dans l'ordre du tri) jusqu'à ce que le temps total de collecte cumulé de l'ensemble des collecteurs retenus soit en dessus du temps total de collecte de q.

Le temps total d'un collecteur $c \in \mathcal{C}_q$ situé dans un stade $s \in \mathcal{S}$ inclue le traitement local au niveau de chaque tâche (T_{local}), le transfert réseaux entre les tâches et le manager global (T_{tr}), ainsi que le traitement au niveau du Manager (T_{global}).

$$c.tempsTotal = (T_{local}(c) + T_{tr}(c)) * pd_s + T_{global}(c) \tag{3.34}$$

3.4.2 Algorithme de réallocation

Pendant l'exécution d'une requête q, les collecteurs insérés calculent les statistiques locales et les transmettent au manager. Ce dernier lit les statistiques locales reçues et les combinent pour trouver les statistiques globales. Dans le cas où le nouveau bénéfice estimé B_{new} est significativement inférieur au bénéfice estimé de l'allocation statique B_{stat} (i.e. $B_{stat} - B_{new} > seuil_{reAlloc}$) alors un processus de réallocation est déclenché. D'abord, l'algorithme effectue une réallocation mémoire (expliquée dans le paragraphe suivant). Si le nouveau bénéfice est toujours significativement inférieur du bénéficie estimé de l'allocation statique alors un re-placement de tâches est envisagé. Étant donné la surcharge réseau engendrée par le re-placement (transfert de données), ce dernier n'est déclenché que s'il garantit une amélioration significative du bénéfice (i.e. $B_{stat} - B_{new} \leq seuil_{reAlloc}$). La réallocation de ressources est présenté dans l'algorithme 2.

La réallocation mémoire commence d'abord par définir s_{init} le premier stade non encore exécuté et qui sera impacté par cette erreur d'estimation. Par exemple, si l'erreur concerne un histogramme sur un attribut alors le premier stade impacté pourrait être le

Algorithm 2 Collecte des statistiques du gestionnaire global et réallocation

for chaque MV $v \in \mathcal{V}$ **do**
  lire statistiques locales de v
end for
calculer les statistiques globales
calculer le nouveau bénéfice avec les nouvelles statistiques B_{new}
if $B_{stat} - B_{new} > seuil_{reAlloc}$ **then**
  effectuer une réallocation mémoire (Algortihme 3)
  calculer le nouveau bénéfice estimé après la réallocation mémoire B_{reMem}
  if $B_{stat} - B_{reMem} > seuil_{reAlloc}$ **then**
    for chaque tâche appartenant à un stade impacté par l'erreur d'estimation **do**
      choisir la ressource logique cible qui satisfait le nouveau besoin et qui garantit la moins de temps de transfert de données
    end for
    calculer le nouveau bénéfice estimé dans le cas du re-placement B_{rePlac}
    if $B_{rePlac} - B_{reMem} > seuil_{rePlac}$ **then**
      déclencher le re-placement
    end if
  end if
end if

Algorithm 3 réallocation mémoire

Définir $s_{init} \in \mathcal{S}_q$ le premier stade impacté par l'erreur d'estimation
$s \leftarrow s_{init}$
while $s \neq null$ **do**
  for chaque opérateur o du stade s **do**
    $o.memory \leftarrow 0$
    recalculer $o.Mmin$ et $o.Mmax$ en se basant sur les statistiques collectées
  end for
  $s \leftarrow next(s)$
end while
for chaque MV $v \in \mathcal{V}$ **do**
  $s \leftarrow s_{init}$
  while $s \neq null$ **do**
    allocationMemoire(v,s) // (Nag and DeWitt, 1998)
    $s \leftarrow next(s)$
  end while
end for

prochain qui contient une jointure dont l'attribut est celui de l'histogramme. La réallocation mémoire concerne le stade s_{init} ainsi que tous les stades qui viennent après ($next(.)$). Tout d'abord, pour chaque opérateur o, la mémoire allouée ($o.memory$) est initialisé à zéro et la mémoire minimale ($o.Mmin$) et maximale ($o.Mmax$) recalculées en se basant sur les nouvelles estimations. Ensuite, la réallocation est déclenchée pour chaque couple : (machine virtuelle, stade). Toute méthode existante de l'allocation mémoire peut être utilisée pour les stades concernés par la réallocation (Nag and DeWitt, 1998). La réallocation mémoire est effectuée via l'algorithme 3.

3.5 Conclusion

Dans ce chapitre, nous avons présenté notre méthode d'allocation de ressources statique ainsi que la méthode de réallocation. Le but de la méthode d'allocation de ressources statique est d'assurer le meilleur compromis entre le bénéfice du fournisseur et la satisfaction des locataires, tout en garantissant un coût d'allocation raisonnable. La réallocation a pour rôle de limiter la dégradation des performances causée par les erreurs d'estimation du modèle de coût. Notre solution est dirigée par les SLAs. Nos méthodes sont évaluées et comparées avec des travaux existants dans le chapitre 5. Le chapitre suivant aborde notre méthode de dimensionnement automatique qui permet de choisir le moment d'augmenter et diminuer le nombre de ressources en fonction de la charge.

Chapitre 4

Méthode de dimensionnement automatique

Contenu

4.1 Introduction

Dans le chapitre précédent, nous avons présenté notre stratégie d'allocation de ressources (réponse aux questions Q1 et Q2 de la problématique de thèse). Ce chapitre est consacré à notre nouvelle méthode de dimensionnement automatique (réponse à la question Q3 de la problématique de thèse). L'évaluation des performances de l'ensemble des méthodes proposées est abordée dans le chapitre suivant.

Le dimensionnement automatique est le mécanisme permettant de choisir le moment d'augmenter et diminuer le nombre de ressources en fonction de la charge afin d'assurer des performances acceptables et éviter le gaspillage. Actuellement les fournisseurs de cloud utilisent généralement une approche basée sur des seuils. Les inconvénients de cette approche ont été abordés dans la section 2.6.1.1. Afin d'éviter les inconvénients, de nombreux travaux scientifiques ont été consacrés à la conception de méthodes de dimensionnement automatique basées sur l'apprentissage par renforcement. Par rapport à cette approche, nous avons constaté que la plupart des travaux existants se concentrent sur des applications dans lesquelles les requêtes sont constituées d'une seule tâche atomique placée sur

une ressource pour une courte période de temps. Ceci ne convient pas à l'interrogation de bases de données. En effet, le plan d'exécution physique contient de nombreuses tâches. Le temps d'exécution d'une requête sur des données massives peut être très long. De plus, les données intermédiaires peuvent être stockées sur le disque ce qui engendre un coût monétaire supplémentaire. Ce qui fait que les méthodes de dimensionnement automatique basées sur l'apprentissage par renforcement devraient être améliorées afin qu'elles correspondent aux spécificités de l'interrogation de bases de données.

Nous proposons dans ce chapitre une méthode de dimensionnement automatique basée sur l'apprentissage par renforcement dédiée pour les requêtes d'interrogation de bases de données. La nouveauté de la méthode, c'est le fait que nous estimons la disponibilité des ressources et les pénalités futures afin de fournir une description plus précise du modèle d'apprentissage. Le calcul de ces estimations se base sur le modèle de coûts et l'allocation de ressources statique (présentés respectivement dans les sections 3.2.3 et 3.3 du chapitre précédent). Les estimations sont injectées dans l'algorithme d'apprentissage par renforcement pour améliorer l'efficacité des agents. Le dimensionnement automatique, le placement et l'ordonnancement constituent le processus d'allocation de ressources élastique.

Nous commençons dans ce chapitre par présenter le processus d'allocation de ressources élastique dans la section 4.2. Ensuite, nous donnons dans la section 4.3, des détails par rapport à l'algorithme d'apprentissage par renforcement utilisé. Enfin, nous abordons notre méthode de dimensionnement automatique dans la section 4.4.

4.2 Processus d'allocation de ressources élastique

La Figure 11 illustre le processus d'allocation élastique. Ce processus se déroule en quatre étapes : (1) dimensionnement automatique, (2) choix de la méthode de placement/ordonnancement, (3) placement et (4) ordonnancement. Nous appelons une exécution complète des quatre étapes une itération. Une nouvelle itération est effectuée chaque unité de temps fixe dt. Dans une itération lancée à un instant t, le système traite les plans d'exécution parallèles générés entre $t - dt$ et t.

Le dimensionnement automatique permet de décider si le nombre de ressources devrait être augmenté, diminué ou rester inchangé en fonction de la charge. Nous donnons plus

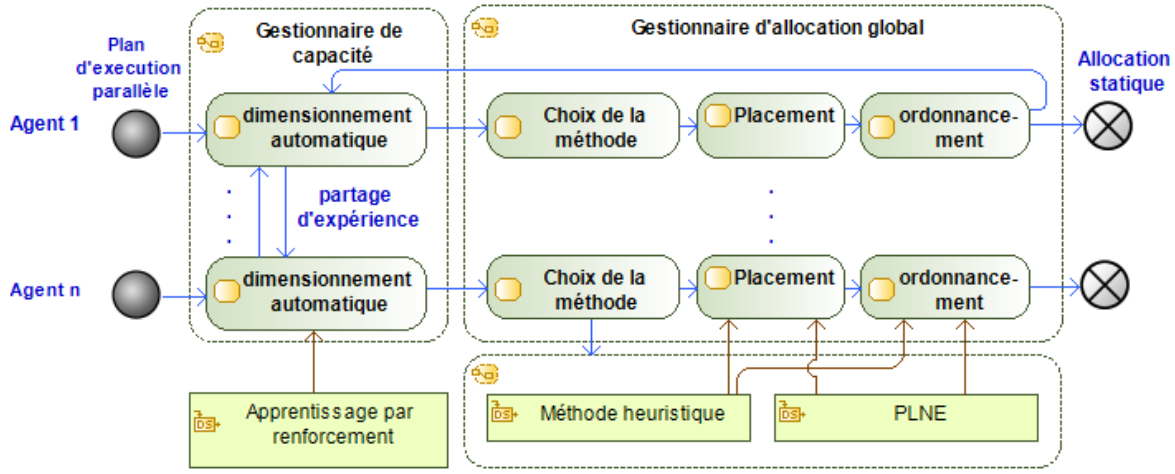


FIGURE 11 – Processus d'allocation de ressources élastique

de détails sur la méthode de dimensionnement automatique proposée dans la section 4.4. Suite au dimensionnement automatique, l'agent choisit la meilleure méthode de placement et d'ordonnancement des tâches en fonction du nombre de requêtes et le nombre de ressources affectées. À la fin de chaque itération de ce processus, le plan d'allocation statique, ainsi que les coûts estimés des pénalités et du stockage des données intermédiaires sont injectés dans le gestionnaire de capacité afin d'être utilisés pour le dimensionnement de la prochaine itération.

Nous supposons qu'il y a n des agents qui travaillent indépendamment. Chaque agent est responsable de plusieurs plans d'exécution parallèles. Il y a deux avantages à utiliser plusieurs agents travaillant en parallèle : (1) les expériences peuvent être partagées entre agents, ce qui accélère l'apprentissage, (2) le processus d'allocation élastique est plus rapide lorsqu'il y a plusieurs agents. Pour expliquer le deuxième point, nous considérons l'exemple suivant. Nous supposons qu'à l'instant t, il y a 50 ressources logiques allouées et 20 requêtes qui arrivent dans le système. Considérons le premier cas où nous n'avons qu'un seul agent, ce dernier doit faire le placement et l'ordonnancement des 20 requêtes sur 50 ressources logiques. Considérons un deuxième cas où nous avons 5 agents parallèles. Dans ce cas, chaque agent doit faire le placement et l'ordonnancement de 4 requêtes (20/5) sur environ 10 ressources logiques (50/5). Le problème est moins complexe dans le deuxième cas puisqu'il y a moins de requêtes et de ressources logiques à gérer par un agent, ce qui fait que le temps du placement et ordonnancement est plus rapide.

Notre solution de dimensionnement automatique dépend de l'allocation de ressources statique. Notre processus d'allocation de ressources élastique supporte notre méthode

d'allocation statique présentée dans le chapitre précédent ainsi que la méthode proposée dans (Kllapi et al., 2011). Le choix de l'approche se fait d'abord en fonction de la complexité des requêtes reçues et du nombre de ressources disponibles.

4.3 Principe de l'algorithme Q-learning

Tous les concepts, formules et algorithmes présentés dans cette section (4.3) existent dans la littérature (Alpaydin, 2014). Ils sont décrits ici vu qu'ils constituent la base de notre méthode de dimensionnement automatique présentée dans la section suivante. Nous rappelons que l'idée de base de l'apprentissage par renforcement est que nous avons un agent qui prend des décisions dans un environnement complexe. L'agent est caractérisé par son autonomie dans la prise de décision. À tout moment, l'environnement se trouve dans un certain état qui fait partie d'un ensemble d'états possibles. À chaque état, l'agent dispose d'un certain nombre d'actions possibles. Une fois qu'une action est choisie par l'agent, l'état change. Suite à l'exécution de l'action, l'environnement fournit une récompense ou une pénalité (selon le problème traité [1]) à l'agent - Figure 12. Après un ensemble d'itérations, l'agent doit apprendre la séquence des actions qui maximisent la récompense totale (ou minimisent la pénalité totale).

Le fonctionnement d'un tel système est modélisé par un processus de décision markovien (PDM). Le temps est discret $t = 0, 1, 2, ...$ À un instant t, l'agent est dans un état $s_t \in \mathcal{S}$, avec $\mathcal{S}$ est l'ensemble des états possibles. Puis il choisit une action $a_t \in \mathcal{A}(s_t)$, tel que $\mathcal{A}(s_t)$ est l'ensemble des actions possibles lorsque l'environnement est dans l'état s_t. Suite à l'action, l'agent reçoit une récompense $r_{t+1} \in \mathcal{R}$ puis passe à un état s_{t+1}. La valeur de la récompense et le nouveau état suivent des distributions de probabilité : $p(r_{t+1}/s_t, a_t)$, $p(s_{t+1}/s_t, a_t)$. Dans certaines applications, il y a un état terminal. Toutes les actions dans cet état terminal passent à lui-même et avec zéro récompense. La séquence d'actions du début jusqu'à l'état terminal est appelé un épisode.

Une politique est une fonction π qui associe une action à un état : $a_t = \pi(s_t)$. $V^{\pi}(s_t)$, est la récompense cumulative qui sera reçue suite à l'exécution de la politique π en démarrant de l'état s_t. Les récompenses du future éloigné sont pénalisées par rapport aux récompenses du future proche en utilisant un facteur γ (avec $E(.)$ est l'espérance mathé-

1. pour simplifier la lecture nous utilisons par la suite uniquement le terme récompense

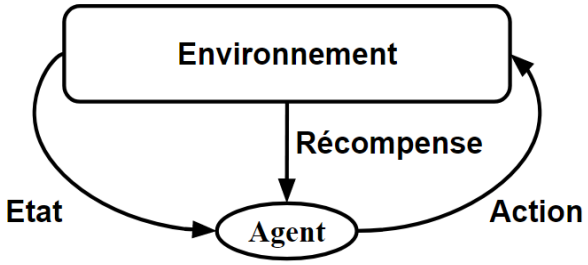


FIGURE 12 – Interaction entre l'agent et l'environnement

matique) :

$$V^{\pi}(s_t) = E(\sum_{t=1}^{\infty} \gamma^{t-1} r_{t+1}), 0 \leq \gamma < 1 \tag{4.1}$$

Il peut y avoir plusieurs politiques possibles. La politique optimale π^* est celle qui maximise la récompense cumulative.

Nous définissons $Q(s_t, a_t)$ comme étant la récompense cumulative en démarrant de l'état s_t et en effectuant l'action a_t. Elle indique à quel point il est bon d'effectuer l'action a_t lorsqu'on se trouve dans l'état s_t. $Q^*(s_t, a_t)$ est récompense cumulative en appliquant la politique optimale :

$$V^{\pi^*}(s_t) = \max_{a_t} Q^*(s_t, a_t) \tag{4.2}$$

$$Q^*(s_t, a_t) = E(r_{t+1}) + \gamma \sum_{s_{t+1}} p(s_{t+1}/s_t, a_t) * \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \tag{4.3}$$

Si nous avons les valeurs Q^* alors il faut que l'agent choisisse à chaque instant t l'action a_t avec $Q^*(s_t, a_t)$ le plus élevé (ou la plus faible si r est considéré comme une pénalité).

Lorsque la structure du modèle est bien connue, les valeurs de Q^* peuvent être calculées avec la programmation dynamique. Cependant, la structure du système et son comportement ne sont généralement pas connus à l'avance, et dans ce cas, les algorithmes d'apprentissage par renforcement, tels que le Q-learning classique (ou sa variante Sarsa) sont plus appropriés (Watkins, 1989; Watkins and Dayan, 1992).

L'algorithme Q-learning classique (Algorithme 4) utilise l'apprentissage pour trouver des valeurs estimées ($\hat{Q}$). $\hat{Q}$ sont initialisées avec une valeur arbitraire (ligne 1). La stratégie de choix de la prochaine action fait un compromis entre l'exploration de l'espace d'états et la maximisation de la récompense. Pour cela, l'agent choisit à chaque itération une action aléatoirement avec une probabilité ϵ (lignes 6-7) et choisit l'action avec le

$\hat{Q}(s,a)$ le plus élevé avec la probabilité $1-\epsilon$ (lignes 8-9). $\hat{Q}$ est amélioré (ligne 12) en se basant sur le nouvel état et la récompense observée (ligne 11). Les paramètres η, γ et la fonction ϵ sont définis à l'avance.

Algorithm 4 Q-Learning classique

1: initialiser tous les $\hat{Q}(s,a)$ arbitrairement
2: initialiser s
3: $iteration \leftarrow 1$
4: **repeat**
5: générer un nombre aléatoire *"rand"* entre 0 et 1
6: **if** $rand < \epsilon(iteration)$ **then**
7: choisir une action $a \in A(s)$ aléatoirement
8: **else**
9: choisir une action $a \in A(s)$ avec $\hat{Q}$ le plus élevé
10: **end if**
11: observer s' (le nouveau état) et $R(s,a)$ (la récompense)
12: mettre à jour $\hat{Q}(s,a)$:
$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \eta * (R(s,a) + \gamma * \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a))$
13: $s \leftarrow s'$
14: $iteration \leftarrow iteration + 1$
15: attendre la prochaine itération
16: **until** le système s'arrête

L'algorithme Q-learning classique utilise la meilleure action possible a' de l'état suivant s' pour mettre à jour $\hat{Q}$ (ligne 12 de l'algorithme 4) tandis que la variante Sarsa choisit d'abord une action a' en utilisant la politique courante (lignes 7-12 de l'algorithme 5) et revient ensuite pour mettre à jour $\hat{Q}(s,a)$ (ligne 13 de l'algorithme 5). Les deux algorithmes combinent l'exploration et l'optimisation dans la prise de décision mais le Q-learning classique considère que l'agent prend toujours la politique optimale lors de la mise à jour des valeurs de $\hat{Q}$ alors que Sarsa considère le fait que la politique réelle combine l'exploration et l'optimisation. Sarsa permet à l'agent de savoir que certaines de ses décisions sont aléatoires. L'expérience stockée à Sarsa est donc plus précise que le Q-learning classique.

4.4 Méthode de dimensionnement automatique

Nous appliquons l'apprentissage par renforcement pour nous assurer que le service cloud dispose du nombre correct de ressources. Nous définissons l'état comme un triplet : 1) le nombre de ressources logiques attribuées au service, 2) l'heure actuelle de la journée

Algorithm 5 Q-Learning Sarsa

initialiser tous les $\hat{Q}(s,a)$ arbitrairement
initialiser s
choisir une action $a \in A(s)$ aléatoirement
$iteration \leftarrow 1$
repeat
  observer s' (le nouvel état) et $R(s,a)$ (la récompense)
  générer un nombre aléatoire *"rand"* entre 0 et 1
  if $rand < \epsilon(iteration)$ **then**
    choisir une action $a' \in A(s')$ aléatoirement
  else
    choisir une action $a' \in A(s')$ avec $\hat{Q}$ le plus élevé
  end if
  mettre à jour $\hat{Q}(s,a)$:
  $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \eta * (R(s,a) + \gamma * \hat{Q}(s',a') - \hat{Q}(s,a))$
  $s \leftarrow s'$
  $a \leftarrow a'$
  $iteration \leftarrow iteration + 1$
  attendre la prochaine itération
until le système s'arrête

et 3) le niveau de disponibilité des ressources.

Formellement un état $s = (n, h, d)$. Tel que $n = (n_1, n_2, ...n_{card(\mathcal{C})})$, $\mathcal{C}$ est l'ensemble des types de ressources, chaque type est caractérisé par sa capacité mémoire et son coût monétaire, n_c est le nombre de ressources de type c, h est une représentation distincte de l'heure actuelle, $0 \leq d \leq 1$ définit le niveau de disponibilité des ressources.

Contrairement aux applications Web, la durée d'une requête de base de données peut être longue. Le niveau de disponibilité des ressources dans notre méthode n'est pas seulement basé sur l'instant présent (c'est le cas des travaux existants) mais aussi sur une fenêtre temporelle future calculée à partir du plan de placement des ressources. Le niveau de disponibilité d est calculé avec la formule (4.4). Avec $\mathcal{P}(iter)$ est l'ensemble des ressources logiques potentielles dans l'itération courante "iter", T est le nombre de fenêtres temporelles futures considérées, $F_{r,t}$ indique si la ressource logique $r \in \mathcal{P}(iter)$ est libre à l'instant t ($F_{r,t} = 1$) ou occupée ($F_{r,t} = 0$) selon l'allocation statique.

$$d = round(\frac{\sum_{r \in \mathcal{P}(iter)} \sum_{t<T} (1 - F_{r,t})}{T * card(\mathcal{P}(iter))}, nbDigits) \tag{4.4}$$

Afin d'expliquer l'intuition derrière la formule (4.4), nous considérons l'exemple de la Figure 13. Le numérateur de la formule représente la surface de la zone grise (les ressources

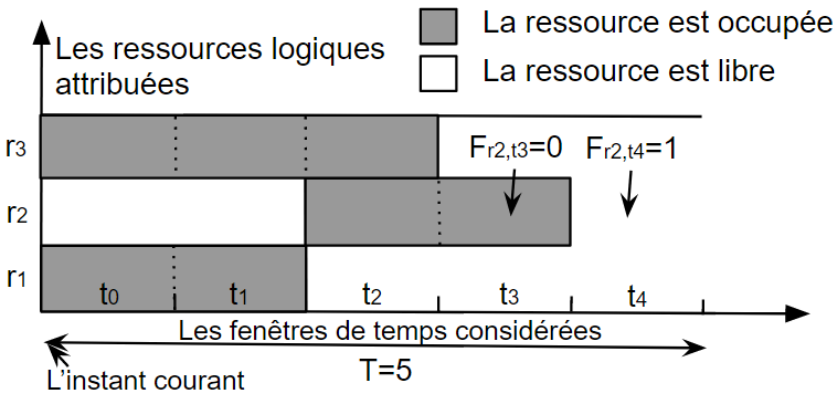


FIGURE 13 – Exemple de placement - ordonnancement

sont occupées). Le dénominateur représente la surface de la zone grise plus la surface de la zone blanche (les ressources sont occupées + libres). Donc d est un nombre entre 0 et 1, avec d est proche de 0 (resp. proche de 1) quand il y a beaucoup de ressources disponibles (resp. beaucoup de ressources occupées) dans la fenêtre temporelle future considérée T. $nbDigits$ est le nombre de chiffres considérés après la virgule quand on applique la fonction *round*. La fonction *round* est nécessaire pour obtenir un espace d'états avec un nombre fini d'états.

Les actions possibles sont de garder le même nombre (0), ajouter (+1) ou libérer (−1) une ressource. Une action $a = (a(1), a(2), ...a(card(\mathcal{C})))$ avec $a(c) \in \{-1; 0; 1\} \forall c \in \mathcal{C}$ et si $\exists c_1 \in \mathcal{C}, a(c_1) = 1 \vee a(c_1) = -1 \Rightarrow a(c_2) = 0 \ \forall c_2 \neq c_1$ (c.-à-d. nous nous rajoutons/supprimons pas plus d'une seule ressource logique pendant la même itération).

La récompense suite d'une action a comprend : 1) le coût d'utilisation des ressources physiques C_{ress}, 2) le coût d'affectation et de libération des ressources C_{ajust}, 3) le coût des pénalités payées par le fournisseur en cas de violation des SLAs C_{pen} et 4) le coût d'utilisation de l'espace de stockage C_{stor} :

$$R(s, a) = C_{ress}(s) + C_{ajust}(a) + C_{pen}(s) + C_{stor}(s) \tag{4.5}$$

Le coût d'utilisation des ressources comprend le coût du processeur C_{proc} et la mémoire C_{mem} :

$$C_{ress}(s) = C_{proc}(s) + C_{mem}(s) \tag{4.6}$$

Le coût du processeur (resp. de la mémoire) est calculé comme suit :

$$C_{proc}(s) = W_{proc} * \sum_{c \in \mathcal{C}} s.n_c \tag{4.7}$$

$$C_{mem}(s) = W_{mem} * \sum_{c \in \mathcal{C}} C_m(c) * s.n_c \tag{4.8}$$

W_{proc} est le poids du coût du processeur, W_{mem} est le poids du coût de la mémoire, $s.n_c$ est le nombre de ressources logiques de type c lorsque le système est en état s, $C_m(c)$ est la mémoire disponible dans les ressources de type c.

Le coût d'ajustement dépend du coût d'ajout des ressources C_{asg} et du coût de suppression des ressources C_{rel} ($1_{condition} = 1$ si la condition est vraie, $= 0$ sinon) :

$$C_{ajust}(a) = \sum_{c \in \mathcal{C}} (C_{asg}(c) * 1_{a(c)>0} + C_{rel}(c) * 1_{a(c)<0}) \tag{4.9}$$

Les coûts de pénalités (C_{pen}) et de stockage (C_{stor}) dépendent de l'ordonnancement des tâches. Basé sur la fonction objective de la formulation PLNE de l'ordonnancement et en supposant que la solution trouvée de l'ordonnancement est $\{v^*, u^*, \beta^*\}$, nous avons :

$$C_{pen}(s) = \sum_{i \in \mathcal{F}} \sum_{D_i - T_i < t < T} W_i * \beta^*_{i,t} \tag{4.10}$$

$$C_{stor}(s) = W_s * \sum_{i \in S} \sum_{m \in T_i} \sum_{t < T} q_i * u^*_{i,m,t} \tag{4.11}$$

L'objectif est de minimiser la pénalité cumulative. La meilleure action est celle qui a la valeur la plus basse de Q. Dans notre problème, le comportement du PDM n'est pas connu à l'avance donc, comme mentionné à la section 4.3, un algorithme tel que Q-learning peut être utilisé pour apprendre une valeur estimée $\hat{Q}$. La valeur de $\hat{Q}$ est améliorée à chaque itération en fonction de la récompense observée et de la description de l'état.

Dans l'apprentissage par renforcement, le nombre de fois qu'un état est visité détermine la qualité de la décision. En effet, plus de visites impliquent une meilleure expérience. Il est donc intéressant d'adopter des méthodes qui accélèrent l'évolution de l'expérience de l'agent d'apprentissage. Nous proposons d'adopter le renforcement parallèle de l'apprentissage (Kretchmar, 2002) pour accélérer l'apprentissage. Le système a un ensemble

Algorithm 6 Q-learning classique parallèle pour le dimensionnement automatique (Agent i)

initialiser tous les $\hat{Q}_l^i(s,a)$ d'une façon arbitraire
initialiser tous les $k^i(s,a)$ à 1
for chaque état $s \in S$ **do**
  $C_{proc}(s) \leftarrow W_{proc} * \sum_{c \in \mathcal{C}} s.n_c$
  $C_{mem}(s) \leftarrow W_{mem} * \sum_{c \in \mathcal{C}} C_m(j) * s.n_c$
  $C_{ress}(s) \leftarrow C_{proc}(s) + C_{mem}(s)$
  for each action $a \in A(s)$ **do**
    $C_{ajust}(a) \leftarrow \sum_{c \in \mathcal{C}} C_{asg}(c) * 1_{a(c)>0} + \sum_{c \in \mathcal{C}} C_{rel}(c) * 1_{a(c)<0}$
  end for
end for
initialiser s
repeat
  générer un nombre aléatoire *"rand"* entre 0 et 1
  if $rand < \epsilon(iteration)$ **then**
    choisir une action $a \in A(s)$ aléatoirement
  else
    choisir une action $a \in A(s)$ avec $\hat{Q}^i$ le plus élevé
  end if
  recevoir les plans d'exécution parallèles des requêtés soumises
  exécuter l'action a
  avertir le gestionnaire d'allocation global (Figure 11, flèche (a)), le placement puis l'ordonnancement sont lancés,
  Obtenir $C_{stor}(s)$ et $C_{pen}(s)$ estimé à partir du manager (Figure 11, flèche (b)),
  $R(s,a) \leftarrow C_{ress}(s) + C_{ajust}(a) + C_{pen}(s) + C_{stor}(s)$
  observer s' (le nouvel état)
  for chaque action $a' \in A(s')$ **do**
    $K \leftarrow \sum_{j \in Agents, j \neq i} k^j(s',a')$
    $\hat{Q}_g^i(s',a') \leftarrow \frac{\sum_{j \in Agents, j \neq i} k^j(s',a') * \hat{Q}_l^j(s',a')}{K}$
    $\hat{Q}^i(s',a') \leftarrow \frac{W_l * k^i(s',a') * \hat{Q}_l^i(s',a') + W_g * K * (nbAgents-1) * \hat{Q}_g^i(s',a')}{W_l * k^i(s',a') + W_g * K * (nbAgents-1)}$
  end for
  mettre à jour $\hat{Q}_l^i(s,a)$:
  $\hat{Q}_l^i(s,a) \leftarrow \hat{Q}_l^i(s,a) + \eta * (R(s,a) + \gamma * \max_{a'} \hat{Q}_l^i(s',a') - \hat{Q}_l^i(s,a))$
  $k^i(s,a) \leftarrow k^i(s,a) + 1$
  partager $\hat{Q}_l^i(s,a)$ et $k^i(s,a)$ avec les autres agents
  $s \leftarrow s'$
  $iteration \leftarrow iteration + 1$
  attendre la prochaine itération
until le système s'arrête

Algorithm 7 Q-learning Sarsa parallèle pour le dimensionnement automatique (Agent i)

initialiser tous les $\hat{Q}_l^i(s,a)$ d'une façon arbitraire
initialiser tous les $k^i(s,a)$ à 1
for chaque état $s \in S$ **do**
  $C_{proc}(s) \leftarrow W_{proc} * \sum_{c \in \mathcal{C}} s.n_c$
  $C_{mem}(s) \leftarrow W_{mem} * \sum_{c \in \mathcal{C}} C_m(j) * s.n_c$
  $C_{ress}(s) \leftarrow C_{proc}(s) + C_{mem}(s)$
  for each action $a \in A(s)$ **do**
    $C_{ajust}(a) \leftarrow \sum_{c \in \mathcal{C}} C_{asg}(c) * 1_{a(c)>0} + \sum_{c \in \mathcal{C}} C_{rel}(c) * 1_{a(c)<0}$
  end for
end for
initialiser s
choisir une action $a \in A(s)$ aléatoirement
repeat
  recevoir les plans d'exécution parallèles des requêtés soumises
  exécuter l'action a
  avertir le gestionnaire d'allocation global (Figure 11, flèche (a)), le placement puis l'ordonnancement sont lancés,
  Obtenir $C_{stor}(s)$ et $C_{pen}(s)$ estimé à partir du manager (Figure 11, flèche (b)),
  $R(s,a) \leftarrow C_{ress}(s) + C_{ajust}(a) + C_{pen}(s) + C_{stor}(s)$
  observer s' (le nouvel état)
  for chaque action $a' \in A(s')$ **do**
    $K \leftarrow \sum_{j \in Agents, j \neq i} k^j(s',a')$
    $\hat{Q}_g^i(s',a') \leftarrow \frac{\sum_{j \in Agents, j \neq i} k^j(s',a') * \hat{Q}_l^j(s',a')}{K}$
    $\hat{Q}^i(s',a') \leftarrow \frac{W_l * k^i(s',a') * \hat{Q}_l^i(s',a') + W_g * K * (nbAgents-1) * \hat{Q}_g^i(s',a')}{W_l * k^i(s',a') + W_g * K * (nbAgents-1)}$
  end for
  générer un nombre aléatoire *"rand"* entre 0 et 1
  if $rand < \epsilon(iteration)$ **then**
    choisir une action $a' \in A(s')$ aléatoirement
  else
    choisir une action $a' \in A(s')$ avec $\hat{Q}^i$ le plus élevé
  end if
  mettre à jour $\hat{Q}_l^i(s,a)$:
  $\hat{Q}_l^i(s,a) \leftarrow \hat{Q}_l^i(s,a) + \eta * (R(s,a) + \gamma * \hat{Q}_l^i(s',a') - \hat{Q}_l^i(s,a))$
  $k^i(s,a) \leftarrow k^i(s,a) + 1$
  partager $\hat{Q}_l^i(s,a)$ et $k^i(s,a)$ avec les autres agents
  $s \leftarrow s'$
  $a \leftarrow a'$
  $iteration \leftarrow iteration + 1$
  attendre la prochaine itération
until le système s'arrête

d'agents parallèles qui partagent leurs expériences. Chaque agent i prend ses décisions en utilisant une mesure $\hat{Q}^i$ composée de sa propre expérience locale $\hat{Q}^i_l$ et de l'expérience globale $\hat{Q}^i_g$ que les autres agents partagent avec lui.

Algorithme 6 (resp. Algorithme 7) illustre l'algorithme Q-learning classique (resp. Sarsa) parallèle appliqué à notre problème et exécuté par un agent i parmi un ensemble d'agents. $k^i(s, a)$ est le nombre de fois que l'action a a été prise suite à une visite à l'état s. L'agent peut donner plus d'importance à sa propre expérience et ainsi, dans ce cas, W_l et W_g sont choisis de sorte que $W_l > W_g$. L'algorithme 7 est exécuté par le gestionnaire de capacité (représenté sur la Figure 11).

4.5 Conclusion

Dans ce chapitre, nous avons présenté notre méthode de dimensionnement automatique. Cette dernière est basée sur l'apprentissage par renfoncement parallèle. Le modèle de coûts et l'allocation de ressources statique sont utilisés pour estimer la disponibilité des ressources et les pénalités d'une fenêtre temporelle future. Ces estimations sont injectées dans le modèle d'apprentissage pour améliorer la capacité d'apprentissage. Une comparaison expérimentale de notre méthode avec une méthode existante a montré l'avantage d'introduire le modèle de coûts et la sortie de l'allocation de statique dans le modèle d'apprentissage. Le chapitre suivant est consacré à l'évaluation des performances.

Chapitre 5

Évaluation des performances

Contenu

5.1 Introduction

Dans les deux chapitres précédents, nous avons présenté nos contributions de thèse, à savoir : la méthode d'allocation de ressources statique, la méthode réallocation (Chapitre 3) et la méthode de dimensionnement automatique (Chapitre 4). Dans ce chapitre, nous présentons l'évaluation des performances de l'ensemble des méthodes.

Nous avons choisi d'évaluer nos méthodes par simulation. La simulation utilise un programme informatique afin de donner un aperçu du comportement d'un système réel. La simulation est utilisée quand le système réel n'existe pas encore ou qu'il existe déjà mais que les expérimentations réelles sont coûteuses en terme de temps et/ou argent (Law et al., 2000; McHaney, 2009). La simulation a de nombreux avantages. Elle permet d'examiner l'impact d'une nouvelle idée. Cette dernière est implémentée dans le système réel seulement si les résultats de la simulation sont satisfaisants. La simulation permet aussi de détecter des erreurs de conception et les corriger avant l'implémentation dans le système réel. La détection et la correction des erreurs à la phase de simulation sont plus simples et moins coûteuses qu'à la phase de l'implémentation réelle. La simulation assure également une rapidité d'analyse. Après que la solution est conçue, il est possible de lancer des simulations à une vitesse supérieure que la vitesse de fonctionnement réel. Une simulation qui dure quelques secondes ou minutes peut représenter un fonctionnement réel qui dure des heures, des jours ou même des mois. Donc plusieurs scénarios et configurations peuvent être testés en un temps raisonnable. Enfin, la simulation permet de mettre en place des scénarios de test personnalisables et qui peuvent être répétés, ce qui est indispensable quand nous disposons d'un ensemble de méthodes à comparer. Le système réel peut être très complexe pour créer de tels scénarios.

La mise en oeuvre de l'évaluation des performances est présentée dans la section 5.2. Nous avons développé un simulateur de l'environnement cloud considéré. Le simulateur a été codé en Java.

Les formulations PLNE ont été implémentées avec l'outil GNU Linear Programming Kit [1] (GLPK). Les graphiques ont été créées avec la bibliothèque Matplotlib de Python. Les requêtes quasi-SQL ont été récupérées à partir du banc d'essai TPC-H (Kim et al., 2008). La base de données est générée avec l'outil Hive-testbench [2]. Il s'agit d'un générateur de données qui permet de tester des requêtes Apache Hive à grande échelle. Les requêtes considérées ont été analysées sur Hive pour connaître le plan d'exécution séquentiel et parallèle ainsi que les estimations initiales du nombre de tuples et taille de données intermédiaires des différents opérateurs.

L'évaluation des performances des méthodes proposées (allocation de ressources statique, réallocation de ressources et dimensionnement automatique) est présentée respec-

1. https ://www.gnu.org/software/glpk/
2. https ://github.com/hortonworks/ Hive-testbench

tivement dans les sections 5.3, 5.4 et 5.5. Pour chaque méthode, nous expliquons les méthodes avec lesquelles nous nous sommes comparés, les métriques de comparaison, les paramètres et scénario de simulation et l'analyse des résultats.

5.2 Mise en oeuvre de l'évaluation des performances

Nous avons effectué des simulations pour évaluer nos contributions de thèse. Le simulateur est constitué de plusieurs modules - Figure 14. Il a été développée en Java sous l'IDE Eclipse [3]. Chaque module contient un ensemble de classes Java. Les requêtes sont récupérées du banc d'essai TPC-H. La structure des plans d'exécution parallèles des requêtes est définie en observant leur génération sous Hive. L'exécution des requêtes est simulée. Leurs temps d'exécution et coût monétaire sont calculés en se basant sur les formules du modèle de coûts et le modèle économique présentés dans le chapitre 3. Les méthodes que nous avons proposé et celles avec lesquelles nous nous sommes comparés sont entièrement codées en Java. L'outil GLPK est utilisé pour la résolution des formulations PLNE de l'allocation de ressources statique. Dans la suite de cette section, nous allons donner des détails par rapport aux modules du simulateur et la génération des plans d'exécution parallèles utilisés pour la simulation.

5.2.1 Description des modules du simulateur

Un projet en Java est organisé en modules. La liste des modules de notre simulateur est la suivante :

— **Le module *Infrastructure* :** contient les éléments de l'infrastructure matérielle considérée dans la thèse (section 3.2.2). Les principales classes sont : *Cloud, MachinePhysique, MachineVirtuelle* et *RessourceLogique*. Dans une simulation, nous disposons d'un objet de la classe *Cloud*. Cet objet est constitué d'une liste d'objets de la classe *MachinePhysique*. Chaque objet *MachinePhysique* contient des objets de la classe *MachineVirtuelle*, dont chacune dispose d'une liste d'objets de la classe *RessourceLogique*.

— **Le module *Modèles* :** contient les différents modèles, notamment : *ModeleCouts* (implémente les formules du modèle de coûts - section 3.2.3), *ModeleEconomique*

3. https ://www.eclipse.org/

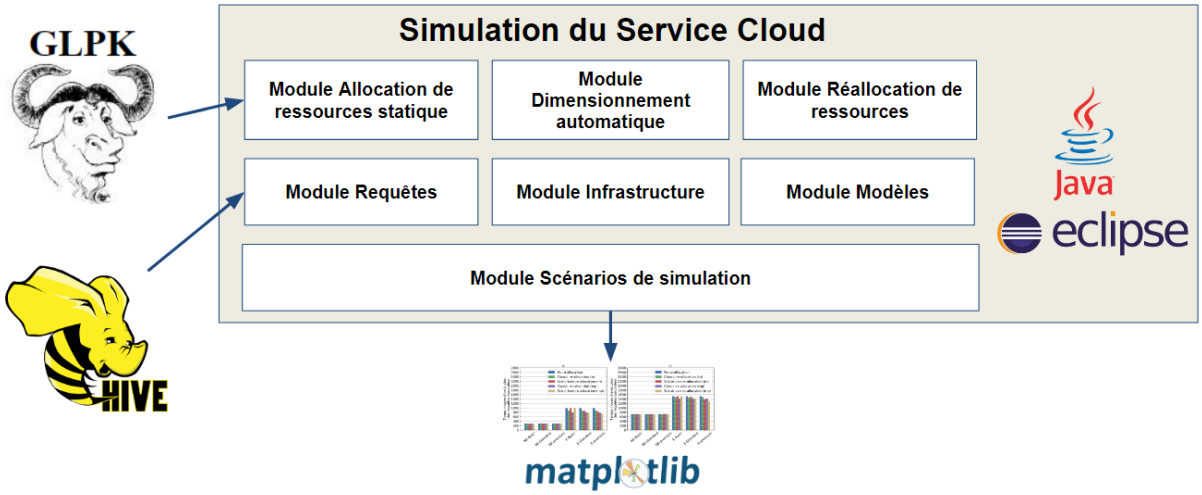


FIGURE 14 – Architecture du simulateur

(implémente les formules du modèle économique - section 3.2.4), *ModeleCommunication* (implémente les formules qui calculent le temps consacré à la communication entre tâches pour une requête ou un ensemble de requêtes) et *ModeleMemoire* (implémente les formules qui calculent le besoin mémoire minimal et maximal pour une opération donnée en fonction de la taille des données manipulées).

— **Le module *Requêtes* :** contient les classes liées à la représentation des plans d'exécution parallèles des requêtes soumises par les locataires. Les principales classes sont : *PlanExecution*, *Stade*, *Job*, *Operation*, *Tache*, *Locataire* et *Benchmark*. Le paradigme d'exécution des requêtes est présenté dans la section 3.2.1. La manière avec laquelle les plans d'exécution parallèles sont générés dans notre simulation est décrite par la suite dans la section 5.2.2.

— **Le module *Allocation de ressources statique* :** implémente des méthodes d'allocation de ressources statique (la méthode proposée dans la thèse et les méthodes avec lesquelles nous nous sommes comparés). Les principales classes sont : *PLNE2phases_Placement*, *PLNE2phases_Ordonnancement*, *PLNE1Phase*, *G_BRT*, *G_MPT* et *G_MPM*. Les méthodes d'allocation de ressources statique avec lesquelles nous nous sommes comparés (G_BRT, G_MPT et G_MPM) sont présentées par la suite dans la section 5.3.1.

— **Le module *Réallocation de ressources* :** implémente des méthodes d'allocation de réallocation de ressources (la méthode proposée dans la thèse et la méthode avec lesquelles nous nous sommes comparées) ainsi que la génération des erreurs

d'estimation. Les principales classes sont : *CollecteurPotentiel*, *CollecteStatistiques*, *ReAllocation*, *SimulerErreurEstimation*. La méthode de réallocation de ressources avec laquelle nous nous sommes comparés est présentée par la suite dans la section 5.4.1. La génération des erreurs d'estimation est décrite dans la section 5.4.2.

— **Le module *Dimensionnement automatique* :** implémente les éléments décrivant un PDM et les méthodes de dimensionnement automatique (la méthode proposée dans la thèse et la méthode avec laquelle nous nous sommes comparées). Les principales classes sont : *Etat*, *Action*, *Transition*, *Agent*, *PDM_Classique* et *PDM_Sarsa*. La méthode de dimensionnement automatique avec laquelle nous nous sommes comparés est présentée dans la suite par la section 5.5.1.

— **Le module *Scénarios de simulation* :** contient une classe pour chaque scénario de simulation. Ces classes contiennent la méthode *main* (la méthode principale d'un programme Java). Dans un scénario, l'exécution d'un ensemble de requêtes est simulée afin de mesurer une ou plusieurs métriques (notamment le temps d'exécution des requêtes et le coût monétaire). Les scénarios de simulation implémentés sont présentés dans les sections 5.3.2, 5.4.2 et 5.5.2.

5.2.2 Génération des plans d'exécution parallèles

Nous rappelons que chaque requête soumise par un locataire est transformée en un plan d'exécution parallèle par l'optimiseur de requêtes, ensuite l'allocation de ressources est effectuée. Dans nos contributions, nous nous intéressons principalement à l'allocation de ressources et non pas la génération des plans d'exécution parallèles. Ces derniers sont donc considérés comme une entrée pour nos simulations.

Afin d'obtenir des plans d'exécutions parallèles, nous avons d'abord utilisé l'outil Hive-testbench pour générer une base de données. Hive-testbench est un générateur de données qui permet de tester des requêtes Hive à grande échelle. Hive (Thusoo et al., 2010) est une solution construite sur Hadoop permettant l'interrogation et la manipulation de données avec un langage déclaratif quasi-SQL : HiveQL. Les requêtes sont exécutées sous Hadoop (donc le plan est parallèle). La base de données générée contient 8 tables : *customer*, *employe*, *lineitem*, *nation*, *orders*, *part*, *region* et *supplier*. Cette base peut être interrogée avec les requêtes TPC-H fournies par l'outil Hive-testbench. Nous rajoutons le mot-clé

```
EXPLAIN SELECT l_orderkey, sum(l_extendedprice * (1 - l_discount))
       as revenue, o_orderdate, o_shippriority
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING'
      and c_custkey = o_custkey
      and l_orderkey = o_orderkey
      and o_orderdate < '1995-03-22'
      and l_shipdate > '1995-03-22'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, o_orderdate
LIMIT 10;
```

FIGURE 15 – Exemple de requête quasi-SQL avec le mot-clé EXPLAIN

```
--------------------------------------------------------------------------------
        VERTICES      STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
--------------------------------------------------------------------------------
Map 1 ..........   SUCCEEDED     25         25        0        0       0       0
Map 2 ..........   SUCCEEDED      2          2        0        0       0       0
Map 3 ..........   SUCCEEDED     23         23        0        0       0       0
Reducer 4 ......   SUCCEEDED     18         18        0        0       0       0
Reducer 5 ......   SUCCEEDED      1          1        0        0       0       0
--------------------------------------------------------------------------------
VERTICES: 05/05  [==========================>>] 100%  ELAPSED TIME: 159.04 s
--------------------------------------------------------------------------------
```

FIGURE 16 – Exécution d'une requête quasi-SQL sur Hive

EXPLAIN [4] à ces requêtes afin d'afficher le plan d'exécution et aussi une estimation de nombre de tuples et taille de données intermédiaires. En lançant l'exécution de la requête, le degré de parallélisme est affiché sur l'écran. Après avoir capturé toutes les informations, nous avons créé des objets Java qui décrivent les plans d'exécution des requêtes considérées pour la simulation.

La Figure 15 montre une requête quasi-SQL à laquelle nous avons rajouté le mot-clé *EXPLAIN*. L'annexe D montre le plan d'exécution affiché. On peut trouver les dépendances entre les stades, les opérations de chaque stade, le nombre de tuples et la taille des données intermédiaires. La Figure 16 montre l'exécution de la requête, on peut trouver le degré de parallélisme de chaque stage (colonne *Total*).

4. https ://cwiki.apache.org/confluence/display/Hive/LanguageManual+Explain

5.3 Évaluation de la méthode d'allocation de ressources statique

Cette section est consacrée à l'évaluation des performances de la méthode d'allocation de ressources statique présentée dans la section 3.3 du chapitre 3.

5.3.1 Méthodes comparées et métriques

Notre méthode d'allocation statique (est qui sera notée par la suite **PLNE2P**) est d'abord comparée à trois heuristiques présentées dans (Kllapi et al., 2011). Les heuristiques (notées **G-BRT**, **G-MPT** et **G-MPM**) suivent le même algorithme générique gourmand mais diffèrent dans le critère de l'ordre du choix des tâches à placer ainsi que les ressources cibles. Dans chaque itération :

— **G-BRT** place la tâche avec le temps d'exécution maximum dans la ressource qui minimise l'écart-type de l'utilisation des ressources.

— **G-MPT** place la tâche avec le temps d'exécution maximum dans la ressource qui minimise le temps de d'exécution.

— **G-MPM** place la tâche avec la taille des résultats maximale dans la ressource qui minimise le coût monétaire.

Dans une deuxième étape, nous comparons notre méthode PLNE en deux phases (**PLNE2P**) avec une méthode PLNE en une phase (**PLNE1P**) conçue pour montrer les avantages d'adopter une approche en deux phases. Dans la méthode **PLNE1P**, il y a une seule formulation PLNE qui traite à la fois le placement et l'ordonnancement.

Les métriques considérées pour la comparaison sont le coût monétaire des pénalités (formule (3.5)), le coût monétaire de l'infrastructure (formule (3.6)) et le coût d'allocation (i.e. le temps d'exécution de la méthode d'allocation de ressources).

5.3.2 Paramètres et scénario de simulation

Nous considérons l'arrivée de requêtes simples ($<$ 6 jobs par requête) et complexes ($\geq$ 6 jobs par requête). Dans la simulation, nous considérons deux types de MV. Une MV de **type1** contient 32 CPU et 8Go de RAM, son prix par heure d'utilisation est de 1.5$. Une MV de **type2** contient 16 CPU et 4Go de RAM, son prix par heure d'utilisation

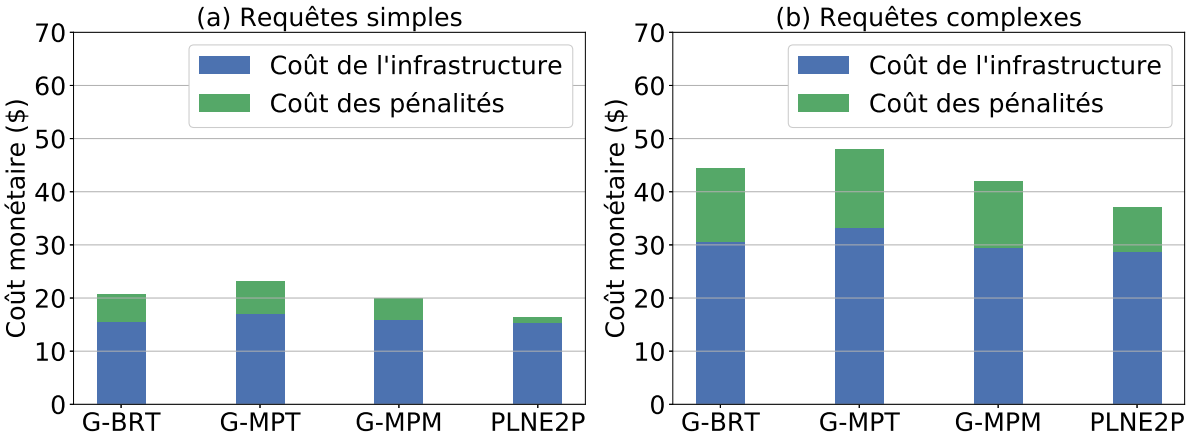


FIGURE 17 – Coût monétaire pour les requêtes simples et complexes (G-BRT, G-MPT, G-MPM, PLNE2P)

est de 0.75$. Chaque tâche contient de 16 à 40 tâches Map (resp. Reduce). La taille de chaque bloc Map est de 256 ou 512 Mo. Le taux initial de disponibilité des ressources est généré de manière aléatoire. Nous avons effectué des expérimentations avec différents taux d'arrivée des requêtes (2, 3 et 4 requêtes par unité de temps).

5.3.3 Analyse des résultats

En premier lieu, nous comparons les méthodes PLNE2P, G-MPM, G-BRT et G-MPT. La Figure 17 représente le coût monétaire moyen par unité de temps pour différents types de requêtes et taux d'arrivée. Les résultats montrent que G-MPM et PLNE2P sont moins coûteux que G-BRT et G-MPT. Ces deux dernières méthodes permettent d'équilibrer la charge et de réduire le temps d'exécution, mais elles ne suffisent pas à réduire les coûts monétaires. En effet, lorsque nous avons un ensemble de requêtes à placer et ordonner, et que nous voulons réduire les coûts monétaires, nous devons d'abord commencer par les requêtes les plus restrictives en terme de délai et poids de pénalité et non les requêtes qui minimisent le temps d'exécution global. G-MPM gère les coûts monétaires mais utilise une méthode gourmande dans laquelle une partie de la solution est déterminée à chaque étape de l'algorithme. Cette partie est déterminée avec les informations disponibles dans l'étape en cours et sans tenir compte de toutes les configurations possibles de placement et d'ordonnancement. Cela peut donner lieu à des choix qui peuvent sembler intéressants compte tenu de l'information disponible à l'étape où le choix a été fait, mais il s'avérera que ce n'est pas un bon choix plus tard.

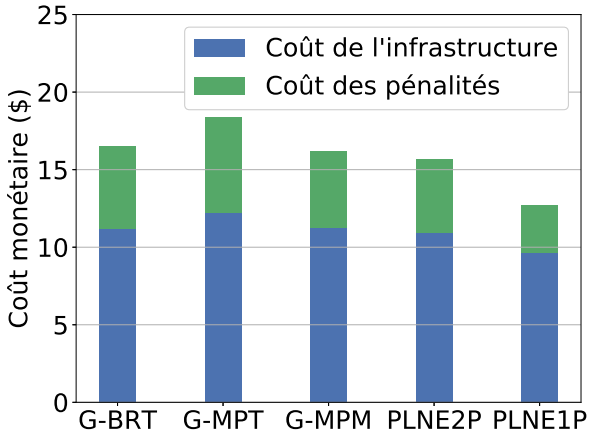


FIGURE 18 – Coût monétaire pour les requêtes simples (G-BRT, G-MPT, G-MPM, PLNE2P, PLNE1P)

En deuxième lieu, nous introduisons aussi la méthode PLNE1P. Les résultats de Figure 18 montrent que la méthode PLNE1P a évidemment le meilleur coût monétaire. En effet, si les problèmes de placement et d'ordonnancement sont traités en même temps, l'espace de recherche est considérablement plus grand. Elle est donc susceptible de trouver une meilleure solution en termes de coût monétaire. Par contre, la gestion simultanée du placement et d'ordonnancement constitue un problème plus complexe.

Le Tableau 5.3.1 illustre les temps d'allocation moyens, minimaux et maximaux des différentes méthodes. Étant donné la nature complexe du PLNE1P, son temps d'exécution est très long et déraisonnable dans la pratique. Bien que l'PLNE2P est plus lente que les méthodes gourmandes, son temps d'allocation reste raisonnable et nettement meilleur que l'PLNE1P.

Tableau 5.3.1 – Coût d'allocation (secondes)

	moyenne	minimum	maximum
G-BRT	0.020	0.017	0.052
G-MPT	0.223	0.178	0.401
G-MPM	0.228	0.176	0.483
PLNE2P	2.272	0.931	19.405
PLNE1P	376.043	54.763	1201.742

Tableau 5.4.1 – Paramètres du modèle économique

		Basic	**Standard**	**Premium**
Requêtes simples (*qs*)	DL_{qs}	435 sec		
	PW_{qs}	1*10^-5	50*10^-5	100*10^-5
	PR_{qs}	125 cent	400 cent	800 cent
requêtes complexes (*qc*)	DL_{qc}	900 sec		
	PW_{qc}	1*10^-5	50*10^-5	100*10^-5
	PR_{qc}	100 cent	500 cent	1000 cent
coût ressource logique		10/3600 cent/sec		
coût disque		10/1024/(30*24*3600) cent/Mo/sec		
coût réseau		10/(1024*1024) cent/Mo		

5.4 Évaluation de la méthode de réallocation de ressources

Cette section est consacrée à l'évaluation des performances de la méthode de réallocation de ressources présentée dans la section 3.4 du chapitre 3.

5.4.1 Méthodes comparées et métriques

Afin de montrer l'avantage de tenir compte des SLAs, la méthode de réallocation de ressources proposée (**SLA-driven re-allocation**) est comparée avec une méthode qui place les collecteurs de statistiques en considérant que les requêtes ont le même niveau de priorité (**classical re-allocation**), comme c'est le cas du placement de collecteurs dans (Kabra and DeWitt, 1998; Agarwal et al., 2012; Bruno et al., 2013). Dans la méthode avec laquelle nous nous comparons, le temps total de collecte par requête ($q.tempsTotalCollect$) est calculé en se basant uniquement sur un seuil fixe par rapport au temps total de la requête et non pas les dates limites et les poids des pénalités. Nous considérons dans la comparaison le cas où on effectue uniquement la réallocation mémoire (**m**) ainsi que le cas où on effectue aussi le re-placement (**mp**). Notre méthode est aussi comparée avec l'allocation statique sans réallocation (**no re-allocation**).

Les métriques considérées pour la comparaison sont le temps d'exécution des requêtes (voir le modèle de performance de la section 3.2.3), le coût monétaire des pénalités (formule (3.5)), le coût monétaire de l'infrastructure (formule (3.6)), le bénéfice (formule (3.7)) et la mémoire allouée par opérateur ($o.memory$ dans l'algorithme 3).

Tableau 5.4.2 – Paramètres système

Paramètre	**Valeur utilisée**
db_l (bande passante du disque local)	100 Mo/sec
db_{sfd} (bande passante du SFD)	80 Mo/sec
cpu (Vitesse de traitement CPU)	100 MIPS
Taille de la page	8192 Octets

5.4.2 Paramètres et scénario de simulation

Nous considérons deux types de requêtes : simples et complexes. Les requêtes simples sont constituées de deux jointures alors que les requêtes complexes sont constituées de sept jointures. Nous utilisons un facteur d'échelle égal à 100 pour la génération des données. Le facteur d'échelle se traduit approximativement en gigaoctets. Un facteur d'échelle de 100 correspond à environ 100 gigaoctets. Avec le facteur d'échelle choisi, nous avons des requêtes simples constituées de 5 stades : M1 (2 tâches), M2 (25 tâches), M3 (23 tâches), R1 (18 tâches), R2 (1 tâche). Les requêtes complexes contiennent 10 stades : M1 (1 tâche), M2 (1 tâche), M3 (9 tâches), M4 (1 tâche), M5 (1 tâche), M6 (12 tâches), M7 (25 tâches), M8 (24 tâches), R1 (1 tâche), R2 (1 tâche).

Dans la simulation, nous supposons qu'il y a trois types de locataires : *basic*, *standard* et *premium*. Le Tableau 5.4.1 présente les paramètres du modèle économique considéré, notamment : les dates limites (DL), le poids des pénalités (PW), le prix des requêtes (PR), le coût des ressources logiques, le coût disque et le coût réseau. L'unité de temps est la seconde. Les locataires de type premium sont les plus exigeants en terme de respect des dates limites et il s'agit des requêtes les plus chères. Les locataires de type *basic* sont les moins exigeants en terme de respect des dates limites et il s'agit des requêtes les moins chères. Le type *standard* est un compromis entre les locataires *premium* et *basic*. Les paramètres système utilisés sont présentés dans le Tableau 5.4.2.

Dans chaque expérimentation trois requêtes sont lancées au même moment : une requête *basic*, une requête *standard* et une requête *premium*. Nous supposons que nous disposons d'assez de ressources logiques pour que toutes les requêtes soient lancées en même temps. Nous supposons également que les MVs du cloud n'ont pas la même taille de mémoire disponible au lancement de la simulation. Nous considérons deux scénarios expérimentaux : (1) pas d'erreur d'estimation -NE- et (2) avec des erreurs d'estimation -E-. Le deuxième scénario est décomposé en trois sous scénarios : (2.1) une erreur dans

une requête *basic*, (2.2) une erreur dans une requête *standard* et (2.3) une erreur dans une requête *premium*. L'emplacement de l'erreur dans la requête choisie, la statistique concernée et l'impact sont définis avec une fonction aléatoire. Nous supposons, en cas d'erreur, que le nombre de tuples (resp. taille des relations) réel est supérieure au nombre (resp. à la taille) estimé pendant l'allocation statique. Chaque expérimentation est répétée plusieurs fois (25) afin d'obtenir différentes possibilités aléatoires. Les différentes métriques présentées par la suite dans les résultats sont la moyenne des métriques mesurées dans les 25 tests.

5.4.3 Analyse des résultats

Les Figures 19 et 20 présentent les résultats obtenus pour les requêtes simples et complexes, en terme de temps d'exécution des requêtes, coût monétaire et bénéfice.

Sur la Figure 19, nous remarquons que dans le cas où il n'y a pas d'erreurs d'estimation (NE-), le temps d'exécution moyen se situe autour de 290 secondes pour les requêtes simples (resp. 720 secondes pour les requêtes complexes). Dans le cas où il y a erreur (E-), les performances se dégradent. Le temps d'exécution moyen d'une requête se situe entre 735 secondes et 980 secondes pour les requêtes simples (resp. 1300 secondes et 1520 secondes pour les requêtes complexes). Le temps le plus élevé est constaté dans le cas où il n'y a pas de réallocation. En effet, le plan statique est maintenu et l'allocation n'est jamais mise à jour pour la prise en compte de l'erreur. Pour les requêtes de type *basic*, la re-alloaction classique donne un meilleur temps d'exécution comparé à la réallocation dirigée par SLA. En effet, vu que les locataires *basic* sont peu exigeants en terme du respect des dates limites, l'approche dirigée par SLAs place un nombre réduit de collecteur de statistiques et donc il est peu probable de détecter l'erreur. Pour les requêtes de type *standard*, la re-alloaction classique donne un temps d'exécution similaire à la réallocation dirigée par SLA. En effet, d'un côté la re-alloaction classique considère que tous les types de locataires ont le même niveau de priorité, d'un autre côté les locataires standard ont un niveau moyen d'exigence, ces deux aspects ont fait en sorte que le nombre de collecteurs rajoutés est approximativement le même pour la méthode classique et dirigée par SLAs. Pour les requêtes de type *premium*, la re-alloaction dirigée par SLAs donne un meilleur temps d'exécution comparé à la réallocation classique. En effet, vu que les locataires *premium* sont très exigeants en terme du respect des dates limites, l'approche dirigée par

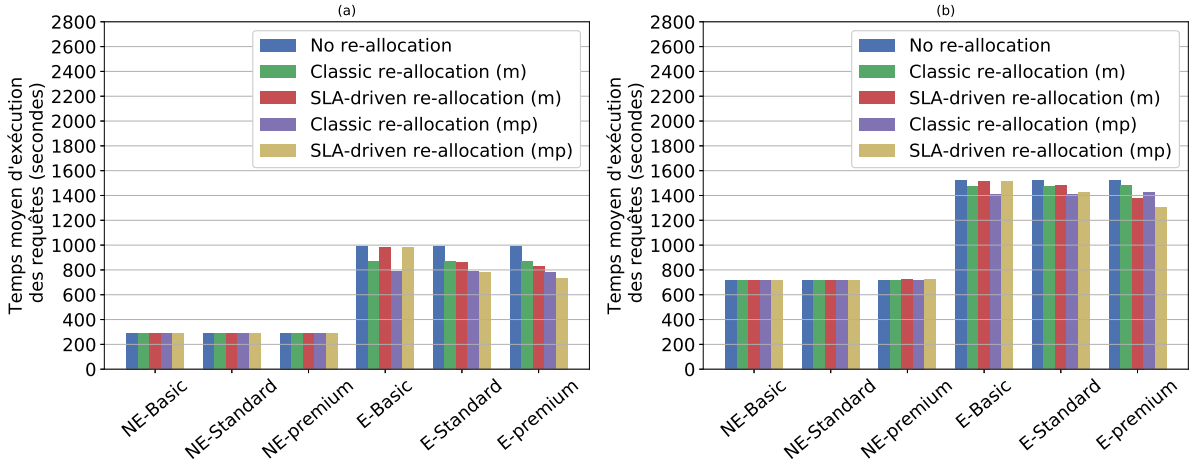


FIGURE 19 – Temps d'exécution pour les requêtes simples (a) et complexes (b)

SLA place un nombre important de collecteurs de statistiques et donc il est très probable de détecter l'erreur et déclencher le processus de la réallocation de ressources.

Les Figures 20-a et 20-c illustrent le coût monétaire de l'utilisation de l'infrastructure et le coût des pénalités moyen par requête. Le coût des pénalités diffère selon le type du locataire et la méthode de réallocation. Dans le cas des locataires *basic*, les pénalités sont plus importantes dans la méthode dirigée par les SLAs mais ça reste négligeable vu que le poids des pénalités est réduit pour les locataires *basic*. Dans le cas des locataires standard, le coût des pénalités est le même pour la méthode dirigée par SLAs et méthode classique. En effet, comme expliqué dans le paragraphe précèdent, le temps d'exécution de requêtes est pratiquement le même. Dans le cas des locataires *premium*, les pénalités sont moins importantes dans la méthode dirigée par les SLAs par rapport à la méthode classique, et cette fois l'écart n'est pas négligeable vu que le poids des pénalités pour les locataires *premium* sont élevés. Les Figures 20-b et 20-d montrent le bénéfice du fournisseur Cloud. Nous constatons que globalement la méthode dirigée par SLAs garantit le meilleur bénéfice, étant donné qu'elle favorise les locataires exigent dans le processus du placement de collecteurs.

Nous remarquons également dans les résultats de la Figure 20 que le re-placement (mp) donne de meilleurs résultats que la réallocation mémoire locale (m). En effet, dans les nouvelles ressources suite au re-placement nous disposons plus de mémoire disponible par rapport aux ressources initiales ce qui améliore les performances.

Les Tableaux 5.4.3 et 5.4.4 montrent la mémoire moyenne allouée, suite à la réallocation, par tâche des différents stades (M :Map, R :Reduce) et opérateurs (S :Scan, F :Filter,

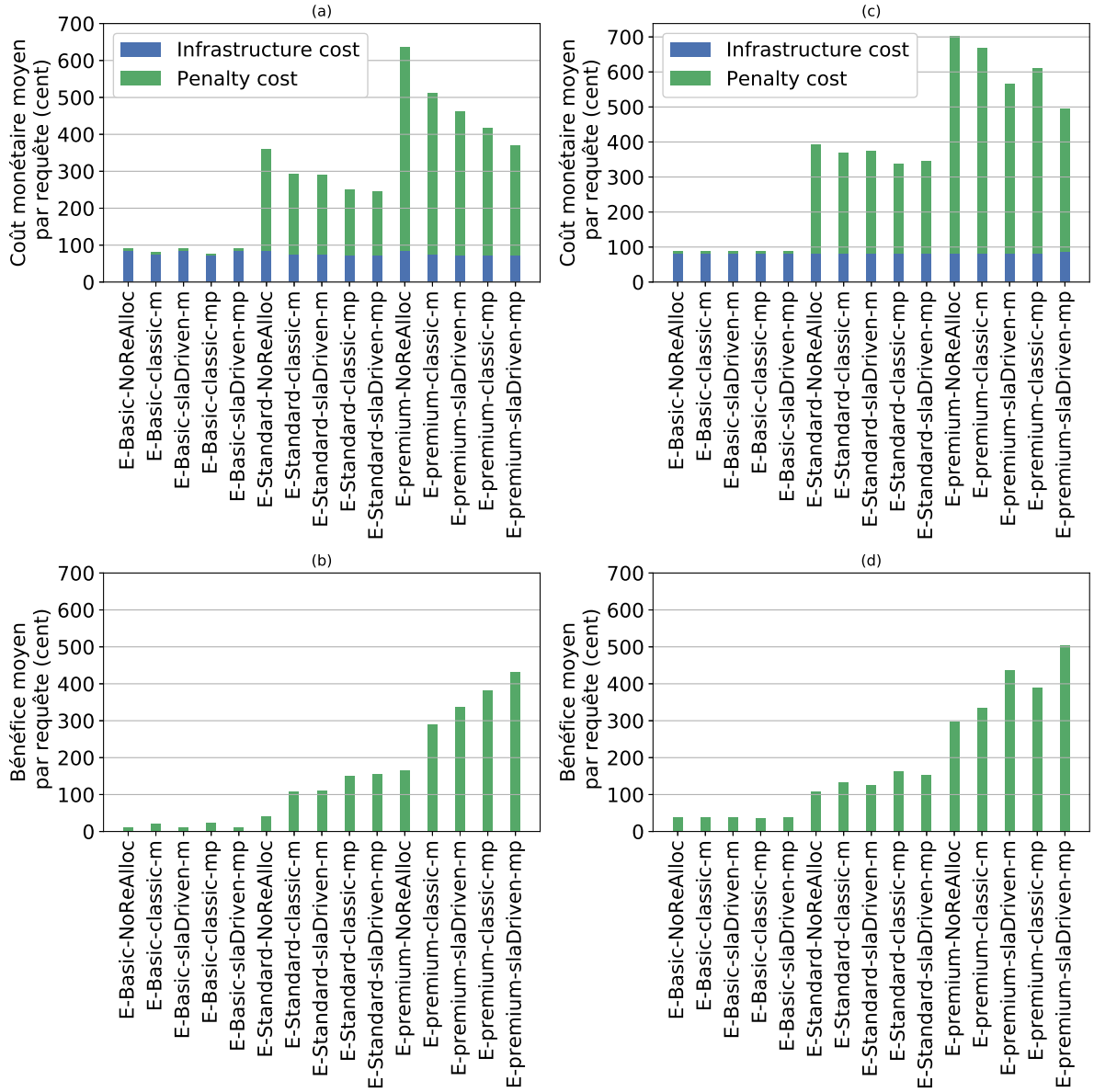


FIGURE 20 – Coût monétaire et bénéfice pour les requêtes simples ((a),(b)) et complexes ((c),(d))

Tableau 5.4.3 – Mémoire moyenne allouée (pages) par tâche pour les requêtes simples

		M1	M2	M3			R1		R2
		SFP	SFJ	SFPJ	A	R	A	R	LW
Allocation Statique		1	8	3853	25779	1	24089	1	1
SLA-Driven Re-allocation	**basic**	1	8	3853	25779	1	24089	1	1
	standard	1	8	168609	728739	1	754290	1	1
	premium	1	8	204341	997921	1	930604	1	1
Classical Re-allocation	**basic**	1	8	161750	742831	1	695254	1	1
	standard	1	8	161750	682831	1	725254	1	1
	premium	1	8	168609	758739	1	707106	1	1

Tableau 5.4.4 – Mémoire moyenne allouée (pages) par tâche pour les requêtes complexes

		M1	M2	M3	M4	M5	M6	M7	
		SFP	SF	SF	SFJ	SFJ	SFP	SFJ	J
Static Allocation		1	1	1	2	2	1	917	2
SLA-Driven Re-allocation	basic	1	1	1	2	2	1	917	2
	standard	1	1	1	2	2	1	917	2
	premium	1	1	1	2	2	1	917	2
Classic Re-allocation	basic	1	1	1	2	2	1	917	2
	standard	1	1	1	2	2	1	917	2
	premium	1	1	1	2	2	1	917	2

Tableau 5.4.5 – Mémoire moyenne allouée (pages) par tâche pour les requêtes complexes (suite)

		M8					R1		R2
		SFJ	J	J	A	Pa	A	Pa	PW
Static Allocation		6	7783	658	5407	1	6455	1	1
SLA-Driven Re-allocation	basic	6	7783	658	5407	1	6455	1	1
	standard	6	7783	47302	406093	1	415035	1	1
	premium	6	7783	108120	1037051	1	1179010	1	1
Classic Re-allocation	basic	6	7783	56424	504159	1	529329	1	1
	standard	6	7783	56424	504159	1	529329	1	1
	premium	6	7783	47646	509796	1	519457	1	1

P :Projection, J :Join, A :Agregation, R :Replication, W :Write) en nombre de pages. Dans le cas de la réallocation dirigée par SLA, la mémoire allouée pour les requêtes *premium* est supérieure à celle des requêtes *basic* et standard. Les requêtes *basic* ont une allocation proche à l'allocation initiale étant donné que les erreurs sont rarement détectées. Dans le cas de la réallocation classique, la mémoire allouée des différents types de locataires est proche et dans la même échelle que les locataires standard de la méthode dirigée par SLA.

5.5 Évaluation de la méthode de dimensionnement automatique

Cette section est consacrée à l'évaluation des performances de la méthode de dimensionnement automatique présentée dans le chapitre 4.

5.5.1 Méthodes comparées et métriques

D'abord, nous comparons notre méthode qui utilise le modèle de coûts et le plan d'allocation statique pour estimer la disponibilité future des ressources, les pénalités et l'utilisation du disque (notée **SQLCloudRL**) avec une méthode de dimensionnement automatique qui fonctionne indépendamment du modèle de coûts et l'allocation statique (Dutreilh et al., 2011; Barrett et al., 2013) (notée **BasicCloudRL**). Nous avons choisi de ne pas comparer notre méthode de dimensionnement automatique avec des méthodes basées sur des seuils. La différence entre l'approche basée sur l'apprentissage par renforcement et l'approche basée sur les seuils est discutée dans le chapitre état de l'art. L'avantage de l'apprentissage par renforcement est le fait qu'il est indépendant de l'intervention humaine. Si nous comparons notre travail avec une méthode basée sur des seuils, nous devrons définir manuellement les valeurs des seuils. Mettre en place une comparaison équitable n'est pas évident. Nous avons donc comparé notre travail avec une méthode de littérature également basée sur l'apprentissage par renforcement. La particularité de notre travail est la dépendance entre le dimensionnement automatique et l'allocation de ressources statique. En effet, comme expliqué dans la section 4.4 du chapitre 4, les sorties de l'allocation statique sont utilisées pour donner une représentation plus précise des états du PDM et de la fonction récompense. La méthode avec laquelle nous nous comparons utilise également l'algorithme d'apprentissage Q-learning mais suppose que le dimensionnement automatique est effectué indépendamment des sorties de l'allocation statique. Ensuite, nous montrons l'impact du partage d'expérience. Nous supposons que certains agents exploitent l'expérience partagée (**share**) et d'autres non (**no share**). Enfin, nous comparons les deux variantes de l'algorithme Q-learning (**Q-learning classique** et **Sarsa**).

Les métriques considérées sont le coût monétaire de la formule 4.5 et la durée de l'allocation élastique (temps de dimensionnement automatique + temps de placement + temps d'ordonnancement).

5.5.2 Paramètres et scénario de simulation

Dans la simulation, nous exécutons les algorithmes 6 et 7 pour un grand nombre d'itérations (jusqu'à 70000 itérations). Une itération est définie comme une exécution complète des lignes de 13 à 36 de l'algorithme 6 (lignes de 14 à 38 de l'algorithme 7).

Au début de chaque itération, nous supposons l'arrivée d'un certain nombre de requêtes représentées par leurs plans d'exécution parallèles. Le nombre de requêtes reçues suit une distribution aléatoire qui dépend de l'heure de la journée. Le nombre de demandes reçues le jour est plus important que le nombre de demandes reçues la nuit. Le nombre maximum de requêtes prises en compte est de 540 requêtes/heure. Il est important de mentionner qu'une requête commencée à l'itération i peut se terminer plus tard (i.e. à l'itération $i+j$, $j > 0$). A chaque itération, une décision de dimensionnement est prise.

Nous avons mis $\eta = 0.5$ et $\alpha = 0.8$. Nous rappelons que les algorithmes 6 et 7 utilisent le paramètre ϵ pour faire le compromis entre le choix des bonnes décisions et l'exploration de l'espace d'états. Afin de tester des différents scénarios, nous proposons trois politiques :

— politique 1 (trois étapes) :

$$\epsilon(iteration) = \begin{cases} 0.999, \text{ quand} \\ 0 \leq iteration < endStep1 \\ 0.4, \text{ quand} \\ endStep1 \leq iteration < endStep2 \\ 0.2, \text{ quand} \\ endStep2 \leq iteration < endOfSimul \end{cases}$$

— politique 2 (deux étapes) :

$$\epsilon(iteration) = \begin{cases} 0.999, \text{ quand} \\ 0 \leq iteration < endStep1 \\ 0.3, \text{ quand} \\ endStep1 \leq iteration < endOfSimul \end{cases}$$

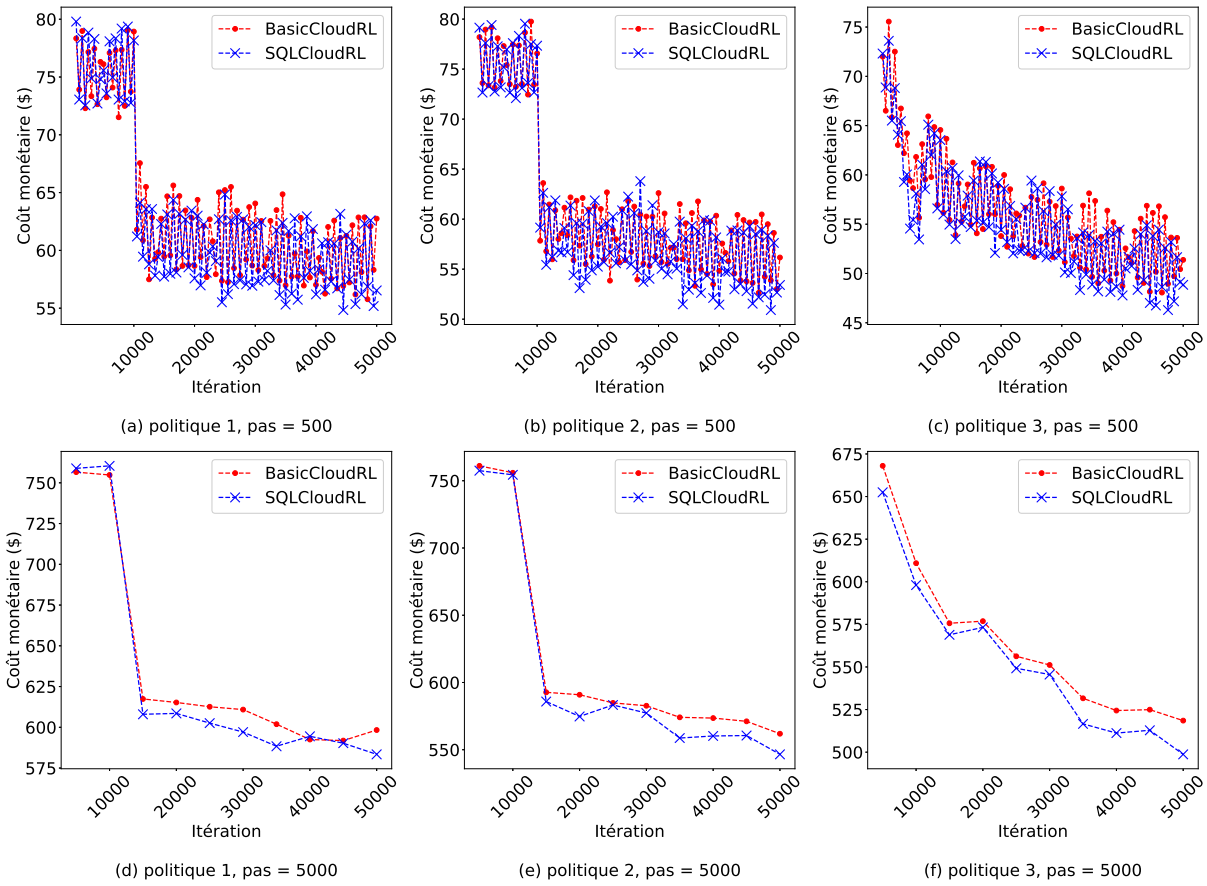
— politique 3 (une étape) :

$$\epsilon(iteration) = \begin{cases} 0.1, \quad \text{quand } 0 \leq iteration < endOfSimul \end{cases}$$

5.5.3 Analyse des résultats

5.5.3.1 Comparaison de notre méthode et une méthode d'apprentissage par renforcement de base

Dans la Figure 21 le coût monétaire est cumulatif sur un intervalle d'itérations. La valeur du "pas" indique la largeur de cet intervalle. Trois agents sont considérés dans chaque



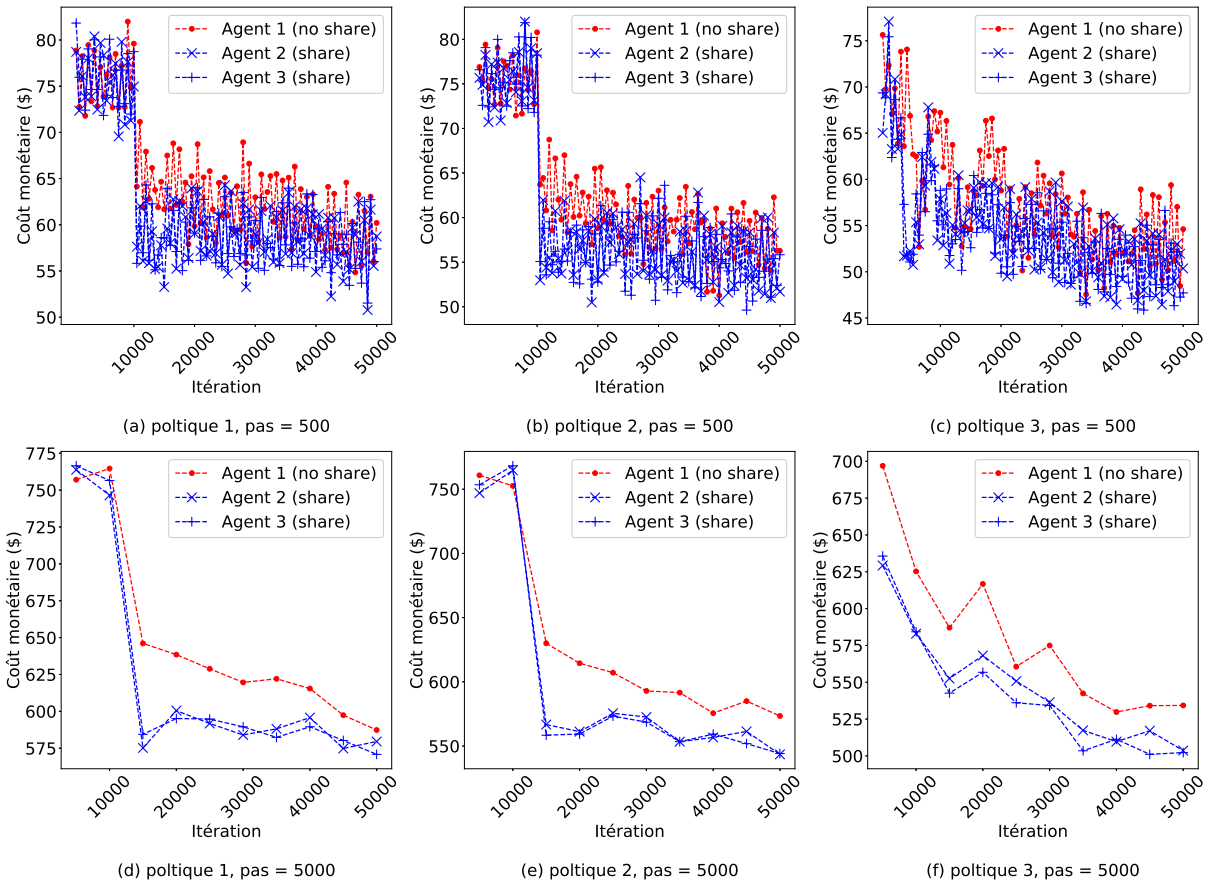
(a) politique 1, pas = 500 (b) politique 2, pas = 500 (c) politique 3, pas = 500

(d) politique 1, pas = 5000 (e) politique 2, pas = 5000 (f) politique 3, pas = 5000

FIGURE 21 – Évolution du coût monétaire dans le temps (SQLCloudRL VS. BasicCloudRL)

test et les valeurs données sur l'axe des y correspondent au coût monétaire moyen par unité de temps des trois agents. Lors de l'itération 1, chaque agent n'a aucune connaissance de l'environnement alors le coût monétaire est élevé au début. L'agent explore l'environnement, stocke ses expériences et les exploite dans la prise de décision ce qui fait que le coût monétaire diminue avec le temps.

D'une part, les décisions de dimensionnement pour les politiques 1 et 2 sont presque aléatoires au début ($\epsilon(iteration) = 0.999$ lorsque $0 \leq iteration < 10000$). Les coûts restent donc stables et élevés pendant cette période. Les décisions aléatoires permettent à l'agent d'explorer plus de possibilités afin que l'apprentissage soit rapide. Dès que nous changeons la valeur de ϵ à $iteration = 10000$, l'agent commence à utiliser l'expérience stockée pour prendre les décisions de dimensionnement afin que les coûts diminuent. La politique 3, par contre, utilise une approche progressive. L'agent fait un compromis entre l'exploration et l'optimisation dès le début ($\epsilon(iteration) = 0.1\ \forall iteration$). Le coût diminue donc moins brutalement que celui des politiques 1 et 2.



(a) poltique 1, pas = 500 (b) poltique 2, pas = 500 (c) poltique 3, pas = 500

(d) poltique 1, pas = 5000 (e) poltique 2, pas = 5000 (f) poltique 3, pas = 5000

FIGURE 22 – Évolution du coût monétaire dans le temps (No share VS. share)

La même figure montre l'avantage de notre solution par rapport à la méthode existante en termes de coût monétaire. L'avantage de notre proposition est dû à la prise en compte de l'estimation de la disponibilité de ressources et des pénalités. La formulation de la baseline peut être utile pour les requêtes courtes (applications Web) mais pas suffisante pour les requêtes longues des applications d'interrogation de base de données. Pour les requêtes longues, la prise en compte des estimations permet une définition plus précise de l'état du PDM. Le gain en termes de coût monétaire devient très important dans un environnement avec des dizaines, voire des centaines d'agents et après un grand nombre d'itérations.

5.5.3.2 Impact du partage d'expérience

Nous considérons trois agents qui travaillent en parallèle et partagent leurs expériences. Les agents 2 et 3 utilisent l'expérience partagée (partage) tandis que l'agent 1 n'utilise que sa propre expérience (pas de partage). Le Tableau 5.5.1 et la Figure 22 montrent les résultats. Nous notons les mêmes remarques que l'expérience précédente concernant la

Tableau 5.5.1 – Coût monétaire moyen ($) par itération pour le Q-learning sans (agent 1) et avec (agent 2 et 3) le partage d'expérience (phase 1 : $0 \leq iteration < 10000$, phase 2 : $10000 \leq iteration < 20000$, phase 3 : $20000 \leq iteration < 50000$)

	politique 1		
	phase 1	phase 2	phase 3
Pas de partage - coût	0.1522	0.1285	0.1224
Partage - coût	0.1517	0.1178	0.1170
Gain	5.4%		
	politique 2		
	phase 1	phase 2	phase 3
Pas de partage - coût	0.1513	0.1245	0.1175
Partage - coût	0.1517	0.1123	0.1119
Gain	6.1%		
	politique 3		
	phase 1	phase 2	phase 3
Pas de partage - coût	0.1322	0.1204	0.1092
Partage - coût	0.1216	0.1110	0.1037
Gain	5.8%		

tendance générale des courbes. En outre, les coûts des agents qui bénéficient de l'expérience partagée (agents 2 et 3) sont inférieurs à ceux des agents qui n'en bénéficient pas (agent 1). En effet, le partage d'expérience permet aux agents 2 et 3 d'apprendre plus vite qu'un agent 1.

Ensuite, nous varions le nombre d'agents et observons l'évolution du coût monétaire et de la durée d'allocation élastique : dimensionnement automatique + placement + ordonnancement (Figure 23). D'une part, le coût monétaire diminue avec l'augmentation du nombre d'agents. En effet, plus il y a d'agents partageant leur expérience, plus l'apprentis-

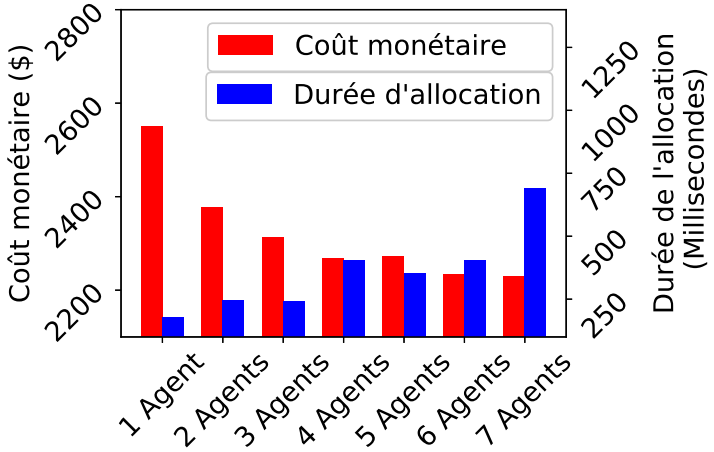
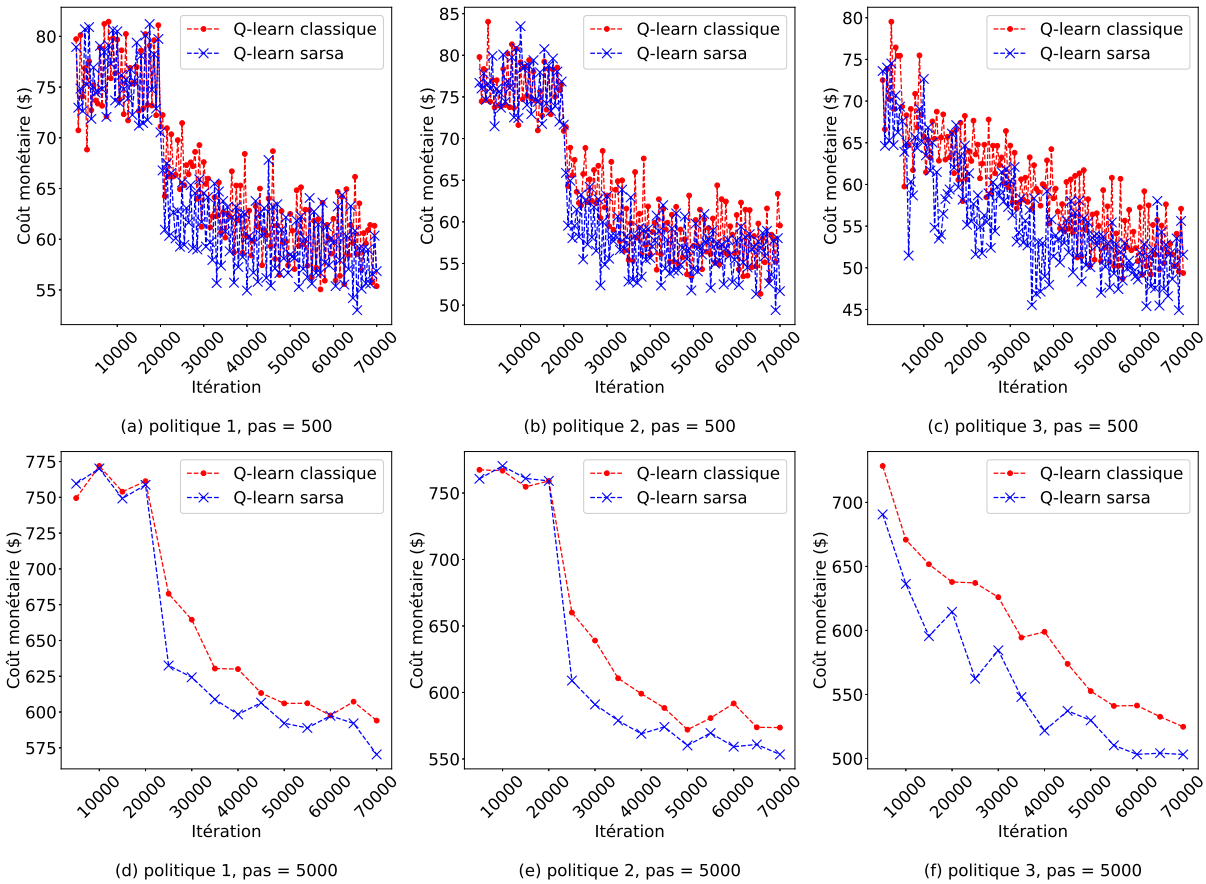


FIGURE 23 – Coût monétaire par rapport au temps moyen d'allocation

FIGURE 24 – Evolution du coût monétaire dans le temps (Q-learning classique VS. Sarsa)



(a) politique 1, pas = 500 (b) politique 2, pas = 500 (c) politique 3, pas = 500

(d) politique 1, pas = 5000 (e) politique 2, pas = 5000 (f) politique 3, pas = 5000

sage est rapide et donc le dimensionnement automatique se fait de manière plus efficace. D'autre part, l'augmentation du nombre d'agents implique plus de messages échangés et donc plus de temps d'allocation. Pour limiter la durée d'allocation lorsque le nombre d'agents est très important, il est possible de regrouper les agents en clusters afin que chaque agent ne partage son expérience qu'avec les agents de son cluster. Une autre solution n'est pas d'échanger à chaque itération mais plutôt après un certain nombre d'itérations.

5.5.3.3 Comparaison de l'algorithme d'apprentissage Q-learning classique et de la variante Sarsa

La différence entre l'algorithme Q-learning classique et Sarsa est la façon de mettre à jour les valeurs de $\hat{Q}(s,a)$. Q-learning standard utilise la meilleure action a' de l'état suivant s' pour mettre à jour $\hat{Q}(s,a)$ tandis que Sarsa choisit d'abord une action a' en utilisant la politique courante et revient ensuite pour mettre à jour $\hat{Q}(s,a)$. Les deux

Tableau 5.5.2 – Coût monétaire moyen ($) par itération pour l'apprentissage Q standard et Sarsa (phase 1 : $0 \leq iteration < 20000$, phase 2 : $20000 \leq iteration < 40000$, phase 3 : $40000 \leq iteration < 70000$)

	politique 1		
	phase 1	phase 2	phase 3
QL classique - coût	0.1518	0.1304	0.1208
QL sarsa - coût	0.1519	0.1232	0.1182
	politique 2		
	phase 1	phase 2	phase 3
QL classique - coût	0.1522	0.1256	0.1163
QL sarsa - coût	0.1525	0.1176	0.1133
	politique 3		
	phase 1	phase 2	phase 3
QL classique - coût	0.1344	0.1228	0.1089
QL sarsa - coût	0.1269	0.1108	0.1029

algorithmes combinent l'exploration et l'optimisation dans la prise de décision mais le Q-learning classique considère que l'agent prend toujours la politique optimale lors de la mise à jour des valeurs de $\hat{Q}$ alors que Sarsa considère le fait que la politique réelle combine l'exploration et l'optimisation (c'est-à-dire que Sarsa permet à l'agent de savoir que certaines de ses décisions sont aléatoires). L'expérience de Sarsa est donc plus précise que le Q-learning classique. Nous ne considérons qu'un seul agent dans cette expérience. Le Tableau 5.5.2 et la Figure 24 confirment que Sarsa est plus efficace que l'apprentissage Q-learning classique en termes de coût monétaire.

5.6 Conclusion

Ce chapitre a été consacré à l'évaluation des performances de l'ensemble des méthodes proposées. Nous avons d'abord présenté le simulateur développé et la démarche suivie pour la génération des plans d'exécution parallèles utilisés. Ensuite, nous avons présenté l'évaluation des performances des méthodes proposées : allocation de ressources statique, réallocation de ressources et dimensionnement automatique.

Notre méthode d'allocation de ressources statique PLNE2P garantit moins de coût monétaire comparé aux méthodes G-MPM, G-BRT et G-MPT. En effet, G-BRT et G-MPT équilibrent les charges et réduisent le temps d'exécution des requêtes, mais elles ne suffisent pas à réduire les coûts monétaires. G-MPM prend en compte les coûts monétaires, mais utilise une méthode gourmande dans laquelle une partie de la solution

est déterminée à chaque étape en se basant sur les informations disponibles et non pas l'ensemble des configurations possibles. La méthode PLNE1P donne un meilleur coût monétaire comparé à la méthode PLNE2P. Par contre, la gestion simultanée du placement et d'ordonnancement constitue un problème complexe. PLNE2P est plus lente que les méthodes gourmandes mais son temps d'allocation est raisonnable comparé à PLNE1P.

Notre méthode de réallocation dirigée par les SLAs garantit un meilleur bénéfice pour le fournisseur comparé à la méthode classique. Ceci est dû au fait qu'elle prend en compte le contrat établi avec le locataire et favorise la détection des erreurs dans les requêtes avec un coût de pénalités élevé. La méthode de réallocation classique constitue une bonne solution dans le cas où il s'agit principalement de minimiser le temps d'exécution des requêtes. Par contre, dans un environnement cloud, le but est de satisfaire les besoins des différents locataires et maximiser le bénéfice du fournisseur du cloud.

Les expériences réalisées par rapport à notre méthode de dimensionnement automatique concernent trois aspects :

— D'abord, l'apprentissage par renforcement et son applicabilité au problème du dimensionnement automatique pour les requêtes d'interrogation de bases de données. Une comparaison de notre méthode et d'une méthode existante a montré l'avantage d'utiliser le modèle de coûts et l'allocation de ressources statique pour la définition du PDM. En effet, ce dernier devient plus précis et le coût monétaire est plus faible.

— Ensuite, la partage d'expérience entre plusieurs agents. Nous avons observé que l'exploitation de l'expérience partagée permet aux agents d'apprendre plus rapidement et de réduire les coûts, mais que le temps d'allocation élastique est plus important en raison de la communication entre agents.

— Enfin, la performance de l'apprentissage de deux algorithmes Q-Learning. La variante Sarsa est plus efficace que le Q-learning classique. Cela peut s'expliquer par le fait que Sarsa est basée sur des calculs plus précis de $\hat{Q}$. En effet, Sarsa permet à l'agent d'apprendre que certaines de ses décisions sont aléatoires.

Après avoir évalué les performances de nos méthodes, nous terminons ce document dans le chapitre suivant avec une synthèse de nos travaux de recherche et en proposant des voies possibles pour des travaux futurs.

Chapitre 6

Conclusion générale

Contenu

Ce chapitre clôture le document en donnant une synthèse des contributions apportées par cette thèse et en proposant des pistes pour étendre les travaux. La synthèse des contributions est présentée dans la section 6.1. Les perspectives sont détaillées par la suite dans la section 6.2.

6.1 Synthèse des contributions

Les contributions présentées dans ce document s'inscrivent dans le contexte d'allocation de ressources élastique dans le cloud. Nous nous sommes en particulier focalisés sur l'interrogation parallèle de bases de données. Nous avons proposé une stratégie d'allocation de ressources (constituée d'une méthode d'allocation statique et une méthode de réallocation) ainsi qu'une méthode de dimensionnement automatique pour l'exécution des requêtes sur des architectures parallèles.

Dans notre ***première contribution***, nous avons développé une méthode d'allocation de ressources statique en deux phases. La première phase représente le placement alors que la deuxième phase représente l'ordonnancement. Le placement consiste à affecter des tâches aux ressources logiques. L'ordonnancement consiste à définir l'ordre d'exécution des tâches qui tient compte dépendances entre les tâches d'une même requête. La méthode d'allocation de ressources statique proposée a pour but d'assurer un bon compromis entre le bénéfice du fournisseur et la satisfaction de ses locataires, et en même temps garantir un coût d'allocation raisonnable. Chaque phase de la méthode est representée par un modèle PLNE et résolue avec l'outil GLPK. Le fait que le problème d'allocation est réparti en

deux phases permet de réduite le temps d'exécution d'allocation.

La méthode d'allocation de ressources statique en deux phases a été comparée à des méthodes gloutons existantes : G-BRT, G-MPT et G-MPM (Kllapi et al., 2011) et une méthode PLNE monophase. Les résultats ont montré que notre méthode ainsi que G-MPM donnent lieu à moins de coût monétaire que les méthodes G-BRT et G-MPT. Ces deux dernières méthodes cherchent principalement à réduire le temps d'exécution d'une manière globale. Par contre, quand nous voulons réduire les coûts monétaires, nous devons d'abord commencer par les requêtes les plus exigeantes en terme de délai et poids de pénalité. La méthode G-MPM prend en compte les coûts monétaires mais utilise une approche gloutons. Une partie de la solution est déterminée à chaque étape. Cette partie est déterminée grâce aux informations disponibles dans l'étape actuelle de l'algorithme sans tenir compte de toutes les configurations possibles. Cela peut donner lieu à des choix qui semblent bons mais il s'avérera que ce n'est pas le cas plus tard. La méthode PLNE monophase a le meilleur coût monétaire car l'espace de recherche est considérablement grand. Par contre, son temps d'exécution est très long et non raisonnable.

Nous nous sommes intéressés dans la ***deuxième contribution*** aux erreurs d'estimation. Afin de détecter les erreurs, des collecteurs de statistiques sont introduits dans différents endroits du plan d'exécution de la requête. Une réallocation de ressources est déclenchée si une erreur avec des conséquences significatives est détectée. La réallocation consiste à modifier l'allocation de ressources dynamiquement pendant l'exécution de la requête. Les collecteurs pertinents sont choisis en se basant sur la nature des SLAs. Les locataires les plus exigeants sont favorisés dans l'étape du choix de collecteurs de statistiques.

La méthode de réallocation dirigée par les SLAs a été comparée à une méthode classique qui considère que les requêtes ont le même niveau de priorité (Kabra and DeWitt, 1998; Agarwal et al., 2012; Bruno et al., 2013). Le coût des pénalités diffère selon le type du locataire (*basic*, *standard* ou *premium*). Les pénalités pour les requêtes *basic* plus élevées dans la méthode dirigée par les SLAs mais c'est négligeable car le poids des pénalités est réduit pour ces requêtes. Les pénalités pour les requêtes *standard* sont les mêmes pour les deux méthodes. Les pénalités pour les requêtes *premium* sont plus importantes dans la méthode classique et l'écart avec notre méthode n'est pas négligeable étant donné que le poids des pénalités est élevé. Nous avons constaté que la méthode dirigée par SLAs

garantit le meilleur bénéfice vu qu'elle favorise les locataires exigeants dans le processus du placement de collecteur.

La ***troisième contribution*** est une méthode de dimensionnement automatique de ressources qui vise à répondre aux spécificités de l'interrogation de bases de données. Le dimensionnement automatique consiste à choisir le moment pour ajouter ou supprimer des ressources attribuées au service en fonction de la charge. La méthode est basée sur l'apprentissage par renforcement parallèle. Elle estime la disponibilité des ressources et les pénalités à payer dans une fenêtre temporelle future pour donner une description précise du modèle d'apprentissage. Ces estimations sont calculées en se basant sur le modèle de coûts et l'allocation de ressources statique. L'expérience de chaque agent est partagée avec les autres.

La méthode est comparée avec une méthode de dimensionnement automatique qui ne se base pas sur le modèle de coûts et l'allocation statique (Dutreilh et al., 2011; Barrett et al., 2013). La comparaison a montré l'avantage d'utiliser le modèle de coûts et l'allocation de ressources statique. Le PDM devient plus précis et le coût monétaire est plus faible. Ensuite, nous avons comparé le cas où l'expérience est partagée avec le cas où il n'y a pas de partage d'expérience. Les résultats ont montré que l'exploitation de l'expérience partagée permet aux agents d'apprendre plus rapidement. Par contre, le coût d'allocation élastique est plus important, à cause, de la communication entre les agents.

6.2 Perspectives

Les travaux rapportés dans cette thèse pourraient être étendus selon plusieurs directions. Dans cette section, nous discutons certaines de ces directions :

1. ***Intégrer d'autres imprévus :*** La réallocation de ressources est utilisée quand l'exécution de requêtes rencontre des imprévus. Dans notre travail, l'imprévu considéré était des erreurs d'estimations par rapport de la taille des données intermédiaires. D'autres imprévus peuvent être intégrés à notre travail comme les pannes des ressources ou l'annulation de l'exécution de requêtes par le locataire. Les pannes des ressources ont un impact sur le temps d'exécution des requêtes. Elles nécessitent de faire un choix entre déplacer les tâches et les données ou bien attendre la reprise de la ressource en panne. Le travail que nous avons réalisé suppose qu'il y a une

seule copie des données manipulées par les requêtes. La réplication de données peut être utilisée pour assurer une meilleure tolérance aux pannes mais le coût monétaire supplémentaire qu'elle engendre doit être considéré dans la fonction objectif du problème.

2. ***Combiner l'apprentissage par renforcement avec des techniques d'apprentissage profond :*** le travail présenté dans ce document est basée sur l'apprentissage par renforcement via les algorithmes Q-learning classique et Sarsa. L'apprentissage par renforcement peut être combiné avec des techniques de deep learning (Mnih et al., 2015; Silver et al., 2016; Evans and Gao, 2016). Cette combinaison, appelée apprentissage par renforcement profond, est mieux adaptée pour traiter des espaces d'états avec des dimensions très grandes (François-Lavet et al., 2018). En effet, l'approche nécessite de mémoriser seulement les valeurs des paramètres d'un réseau de neurones et non pas la valeur de tous les $\hat{Q}$ comme c'est le cas dans le Q-learning classique et Sarsa. Mao et al. (2016) affirment que l'apprentissage par renforcement profond est bien adapté aux problèmes de gestion des ressources. Des travaux récents ont exploré la possibilité d'appliquer cette approche pour la gestion de ressources cloud (Liu et al., 2017; Cheng et al., 2018). Il serait intéressant d'intégrer des algorithmes d'apprentissage profond dans le travail que nous présentons dans ce document afin de gérer efficacement des espaces d'états très grands.
3. ***Considérer l'historique de l'utilisation du service par les locataires :*** d'autres aspects peuvent être intégrés aux méthodes présentées dans ce document. Les requêtes dans notre travail sont caractérisées par leur date limite et le poids de pénalités. Nos méthodes cherchent à minimiser les coûts monétaires du fournisseur en se basant principalement sur ces deux caractéristiques. Il serait intéressant d'introduire aussi l'historique de l'utilisation du service par les locataires. Par exemple, les requêtes des locataires qui ont l'habitude d'utiliser le service très récemment sont favorisées dans l'allocation de ressources même s'ils ne sont pas de type *premium*. Ceci semble ne pas être avantageux pour le bénéfice du fournisseur à court terme mais le départ de ces locataires a des conséquences négatives sur le long terme.
4. ***Explorer d'autres possibilités et scénarios dans l'évaluation des performances :***

   — Le nombre maximal d'agents que nous avons considérés est 7. Il serait inté-

ressant d'effectuer des tests avec un nombre plus d'importants (des dizaines d'agents). Dans ce cas, il faut adopter une stratégie pour réduire le coût d'allocation élastique. Par exemple, répartir les agents en sous-ensembles indépendants (deux agents se partagent leurs expériences uniquement s'ils sont dans le même sous-ensemble) ou diminuer la fréquence de partage d'expérience.

— Dans notre processus d'allocation de ressources élastique, il y a une étape de choix de la méthode d'allocation de ressource statique. Dans nos expérimentations le choix de la méthode est fait manuellement. Il serait intéressant d'intégrer un mécanisme qui automatise le choix de la méthode en se basant sur la complexité du problème (nombre de requêtes, leur taille, nombre de ressources...).

Bibliographie

Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I., and Zhou, J. (2012). Reoptimizing data parallel computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 281–294.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., and Merle, P. (2017). Elasticity in cloud computing : state of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2) :430–447.

Alpaydin, E. (2014). *Introduction to machine learning*. MIT press.

Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., and Xu, M. (2007). Web services agreement specification (ws-agreement). In *Open grid forum*, volume 128, page 216.

Angelou, E., Papailiou, N., Konstantinou, I., Tsoumakos, D., and Koziris, N. (2012). Automatic scaling of selective sparql joins using the tiramola system. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*, page 1. ACM.

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. (2015). Spark sql : Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM.

Audet, C., Savard, G., and Zghal, W. (2007). New branch-and-cut algorithm for bilevel linear programming. *Journal of Optimization Theory and Applications*, 134(2) :353–370.

Azumah, K. K., Sørensen, L. T., and Tadayoni, R. (2018). Hybrid cloud service selection strategies : A qualitative meta-analysis. In *2018 IEEE 7th International Conference on Adaptive Science & Technology (ICAST)*, pages 1–8. IEEE.

Balas, E., Ceria, S., and Cornuéjols, G. (1993). A lift-and-project cutting plane algorithm for mixed 0–1 programs. *Mathematical programming*, 58(1-3) :295–324.

Barrett, E., Howley, E., and Duggan, J. (2013). Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation : Practice and Experience*, 25(12) :1656–1674.

Bégin, M.-E., Jones, B., Casey, J., Laure, E., Grey, F., Loomis, C., and Kubli, R. (2008). An egee comparative study : Grids and clouds-evolution or revolution. *EGEE III project Report*, 30 :1–33.

Bonneau, S. (1999). *Placement de requete (s) sql sur une architecture parallele a memoire distribuee : du statique au dynamique*. PhD thesis, Toulouse 3.

Bouganim, L., Kapitskaia, O., and Valduriez, P. (1998). Memory-adaptive scheduling for large query execution. In *Proceedings of the seventh international conference on Information and knowledge management*, pages 105–115. ACM.

Bragg, R. (2008). Cloud computing : When computers really rule. *Tech News World*, 12(12) :2009.

Bruno, N., Jain, S., and Zhou, J. (2013). Continuous cloud-scale query optimization and processing. *Proceedings of the VLDB Endowment*, 6(11) :961–972.

Chang, V. (2015). Towards a big data system disaster recovery in a private cloud. *Ad Hoc Networks*, 35 :65–82.

Chauhan, A., Fontama, V., Hart, M., Tok, W.-H., and Woody, B. (2014). *Introducing Microsoft Azure HDInsight*. Microsoft press.

Chen, D.-S., Batson, R. G., and Dang, Y. (2010). Applied integer programming. *Hoboken, NJ*.

Chen, X., Hao, Q. Y., Jin, Y., and Zhang, W. C. (2015). Database query in a share-nothing database architecture. US Patent 9,146,959.

Cheng, D., Zhou, X., Lama, P., Wu, J., and Jiang, C. (2017). Cross-platform resource scheduling for spark and mapreduce on yarn. *IEEE Transactions on Computers*, 66(8) :1341–1353.

Cheng, M., Li, J., and Nazarian, S. (2018). Drl-cloud : Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, pages 129–134. IEEE Press.

Chieu, T. C., Mohindra, A., and Karve, A. A. (2011). Scalability and performance of web applications in a compute cloud. In *e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on*, pages 317–323. IEEE.

Chun, S.-H. and Choi, B.-S. (2014). Service models and pricing schemes for cloud computing. *Cluster Computing*, 17(2) :529–535.

Date, C. J. and Darwen, H. (1987). *A Guide to the SQL Standard*, volume 3. Addison-Wesley New York.

de Haaff, B. (2008). Cloud computing-the jargon is back. *Cloud Computing Journal, August*.

Dean, J. and Ghemawat, S. (2008). Mapreduce : simplified data processing on large clusters. *Communications of the ACM*, 51(1) :107–113.

Dutreilh, X., Kirgizov, S., Melekhova, O., Malenfant, J., Rivierre, N., and Truck, I. (2011). Using reinforcement learning for autonomic resource allocation in clouds : towards a fully automated workflow. In *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, pages 67–74.

Evans, R. and Gao, J. (2016). Deepmind ai reduces google data centre cooling bill by 40%. *DeepMind blog*, 20.

Farokhi, S., Jamshidi, P., Lakew, E. B., Brandic, I., and Elmroth, E. (2016). A hybrid cloud controller for vertical memory elasticity : A control-theoretic approach. *Future Generation Computer Systems*, 65 :57–72.

Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al. (2018). Azure accelerated networking : Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 51–66.

Floratou, A., Minhas, U. F., and Özcan, F. (2014). Sql-on-hadoop : full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment*, 7(12) :1295–1306.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J., et al. (2018). An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4) :219–354.

Gandhi, A., Thota, S., Dube, P., Kochut, A., and Zhang, L. (2016). Autoscaling for hadoop clusters. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on*, pages 109–118. IEEE.

Garcia-Molina, H. (2008). *Database systems : the complete book*. Pearson Education India.

Gardarin, G. (2003). *Bases de données*. Editions Eyrolles.

Garfinkel, S. (1999). *Architects of the information society : 35 years of the Laboratory for Computer Science at MIT*. MIT press.

Garofalakis, M. N. and Ioannidis, Y. E. (1996). Multi-dimensional resource scheduling for parallel queries. In *ACM SIGMOD Record*, volume 25, pages 365–376. ACM.

Garofalakis, M. N. and Ioannidis, Y. E. (1997). Parallel query scheduling and optimization with time-and space-shared resources. *SORT*, 1(T2) :T3.

Geelan, J. et al. (2009). Twenty-one experts define cloud computing. *Cloud Computing Journal*, 4 :1–5.

Ghanbari, H., Simmons, B., Litoiu, M., and Iszlai, G. (2011). Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723. IEEE.

Griebler, D., Vogel, A., Maron, C. A., Maliszewski, A. M., Schepke, C., and Fernandes, L. G. (2018). Performance of data mining, media, and financial applications under private cloud conditions. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00450–00456. IEEE.

Gruenheid, A., Omiecinski, E., and Mark, L. (2011). Query optimization using column statistics in hive. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, pages 97–105. ACM.

Hale, M. and Egerstedty, M. (2015). Differentially private cloud-based multi-agent optimization with constraints. In *2015 American Control Conference (ACC)*, pages 1235–1240. IEEE.

Hameurlain, A. (1996). Traitement parallèle dans les bases de données relationnelles.

Han, R., Guo, L., Ghanem, M. M., and Guo, Y. (2012). Lightweight resource scaling for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 644–651. IEEE.

Hasan, M. Z., Magana, E., Clemm, A., Tucker, L., and Gudreddi, S. L. D. (2012). Integrated and autonomic cloud resource scaling. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1327–1334. IEEE.

Hua, K. A., Lee, C., and Peir, J.-K. (1991). Interconnecting shared-everything systems for efficient parallel query processing. In *[1991] Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 262–270. IEEE.

Huang, C.-W., Shih, C.-C., Hu, W.-H., Lin, B.-T., and Cheng, C.-W. (2013). The improvement of auto-scaling mechanism for distributed database-a case study for mongodb. In *Network Operations and Management Symposium (APNOMS), 2013 15th Asia-Pacific*, pages 1–3. IEEE.

Ibrahim, A. A. Z. A., Varrette, S., and Bouvry, P. (2018). On verifying and assuring the cloud sla by evaluating the performance of saas web services across multi-cloud providers. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 69–70. IEEE.

Ioannidis, Y. E. and Christodoulakis, S. (1991). *On the propagation of errors in the size of join results*, volume 20. ACM.

Iranpour, E. and Sharifian, S. (2018). A distributed load balancing and admission control algorithm based on fuzzy type-2 and game theory for large-scale saas cloud architectures. *Future Generation Computer Systems*, 86 :81–98.

Jarke, M. and Koch, J. (1984). Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2) :111–152.

Jha, S., Merzky, A., and Fox, G. (2009). Using clouds to provide grids with higher levels of abstraction and explicit support for usage modes. *Concurrency and computation : Practice and Experience*, 21(8) :1087–1108.

Jin, L.-j., Machiraju, V., and Sahai, A. (2002). Analysis on service level agreement of web services. *HP June*, page 19.

JoSEP, A. D., KAtz, R., KonWinSKi, A., Gunho, L., PAttERSon, D., and RABKin, A. (2010). A view of cloud computing. *Communications of the ACM*, 53(4).

Kabra, N. and DeWitt, D. J. (1998). Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD Record*, volume 27, pages 106–117. ACM.

Karthikeyan, S. A. (2018). Introduction to azure iaas. In *Practical Microsoft Azure IaaS*, pages 1–38. Springer.

Khatua, S., Ghosh, A., and Mukherjee, N. (2010). Optimizing the utilization of virtual resources in cloud environment. In *Virtual Environments Human-Computer Interfaces and Measurement Systems (VECIMS), 2010 IEEE International Conference on*, pages 82–87. IEEE.

Kim, K., Jeon, K., Han, H., Kim, S.-g., Jung, H., and Yeom, H. Y. (2008). Mrbench : A benchmark for mapreduce framework. In *2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 11–18. IEEE.

Kllapi, H., Sitaridi, E., Tsangaris, M. M., and Ioannidis, Y. (2011). Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 289–300. ACM.

Konstantinou, I., Angelou, E., Tsoumakos, D., Boumpouka, C., Koziris, N., and Sioutas, S. (2012). Tiramola : elastic nosql provisioning through a cloud management platform. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 725–728. ACM.

Kouki, Y. (2013). *Approche dirigée par les contrats de niveaux de service pour la gestion de l'élasticité du" nuage"*. PhD thesis, Nantes, Ecole des Mines.

Kouki, Y. and Ledoux, T. (2012). Sla-driven capacity planning for cloud applications. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 135–140.

Kretchmar, R. M. (2002). Parallel reinforcement learning. In *The 6th World Conference on Systemics, Cybernetics, and Informatics*.

Law, A. M., Kelton, W. D., and Kelton, W. D. (2000). *Simulation modeling and analysis*, volume 3. McGraw-Hill New York.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-bound methods : A survey. *Operations research*, 14(4) :699–719.

Liu, N., Li, Z., Xu, J., Xu, Z., Lin, S., Qiu, Q., Tang, J., and Wang, Y. (2017). A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 372–382. IEEE.

Ludwig, H., Keller, A., Dan, A., King, R. P., and Franck, R. (2003). Web service level agreement (wsla) language specification. *Ibm corporation*, pages 815–824.

Maarouf, A., Marzouk, A., and Haqiq, A. (2015). Practical modeling of the sla life cycle in cloud computing. In *2015 15th International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 52–58. IEEE.

Malawski, M., Juve, G., Deelman, E., and Nabrzyski, J. (2015). Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Computer Systems*, 48 :1–18.

Mao, H., Alizadeh, M., Menache, I., and Kandula, S. (2016). Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM.

Mazrekaj, A., Shabani, I., and Sejdiu, B. (2016). Pricing schemes in cloud computing : an overview. *International Journal of Advanced Computer Science and Applications*, 7(2) :80–86.

McHaney, R. (2009). *Understanding computer simulation*. Bookboon.

Mehta, M. and DeWitt, D. J. (1997). Data placement in shared-nothing parallel database systems. *The VLDB Journal—The International Journal on Very Large Data Bases*, 6(1) :53–72.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing.

Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing.

Mills, T. C. and Mills, T. C. (1991). *Time series techniques for economists*. Cambridge University Press.

Mitschang, B. (1995). Query processing in database systems.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540) :529.

Nag, B. and DeWitt, D. J. (1998). Memory allocation strategies for complex decision support queries. In *Conference on Information and Knowledge Management : Proceedings of the seventh international conference on Information and knowledge management (CIKM)*, volume 2, pages 116–123. Citeseer.

Naskos, A., Gounaris, A., and Katsaros, P. (2017). Cost-aware horizontal scaling of nosql databases using probabilistic model checking. *Cluster Computing*, 20(3) :2687–2701.

Naskos, A., Gounaris, A., and Konstantinou, I. (2018). Elton : A cloud resource scaling-out manager for nosql databases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1641–1644. IEEE.

Naskos, A., Stachtiari, E., Gounaris, A., Katsaros, P., Tsoumakos, D., Konstantinou, I., and Sioutas, S. (2015). Dependable horizontal scaling based on probabilistic model checking. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 31–40. IEEE.

Nassar, M. (2018). Support cloud sla establishment using mde. *Cloud Computing and Big Data : Technologies, Applications and Security*, 49 :167.

Padberg, M. and Rinaldi, G. (1987). Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1) :1–7.

Pahl, C. (2015). Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3) :24–31.

Parkhill, D. F. (1966). Challenge of the computer utility.

Persico, V., Pescapé, A., Picariello, A., and Sperlí, G. (2018). Benchmarking big data architectures for social networks data processing using public cloud platforms. *Future Generation Computer Systems*, 89 :98–109.

Pietri, I., Chronis, Y., and Ioannidis, Y. (2019). Fairness in dataflow scheduling in the cloud. *Information Systems*, 83 :118 – 125.

Pokahr, A. and Braubach, L. (2016). Elastic component-based applications in paas clouds. *Concurrency and Computation : Practice and Experience*, 28(4) :1368–1384.

Ranganathan, P., Gharachorloo, K., Adve, S. V., and Barroso, L. A. (1998). Performance of database workloads on shared-memory systems with out-of-order processors. In *ACM SIGPLAN Notices*, volume 33, pages 307–318. ACM.

Rao, J., Bu, X., Xu, C.-Z., and Wang, K. (2011). A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 45–54. IEEE.

Rao, J., Bu, X., Xu, C.-Z., Wang, L., and Yin, G. (2009). Vconf : a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing*, pages 137–146. ACM.

Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A., and Curino, C. (2015). Apache tez : A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369. ACM.

Salagnon, C. (1994). *Ordonnancement et placement dans les s. GBD Paralleles*. PhD thesis, Nice.

Santana, G. A. A. (2016). *CCNA Cloud CLDFND 210-451 Official Cert Guide*. Cisco Press.

Shapiro, J. F. (1968). Group theoretic algorithms for the integer programming problem ii : Extension to a general algorithm. *Operations Research*, 16(5) :928–947.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587) :484.

Simmons, B., Ghanbari, H., Litoiu, M., and Iszlai, G. (2011). Managing a saas application in the cloud using paas policy sets and a strategy-tree. In *Proceedings of the 7th International Conference on Network and Services Management*, pages 343–347. International Federation for Information Processing.

Stavrinides, G. L. and Karatza, H. D. (2016). Scheduling different types of applications in a saas cloud. In *Proceedings of the 6th International Symposium on Business Modeling and Software Design (BMSD'16)*, pages 144–151.

Stavrinides, G. L. and Karatza, H. D. (2019). Performance evaluation of a saas cloud under different levels of workload computational demand variability and tardiness bounds. *Simulation Modelling Practice and Theory*, 91 :1–12.

Talebian, H., Gani, A., Sookhak, M., Abdelatif, A. A., Yousafzai, A., Vasilakos, A. V., and Yu, F. R. (2019). Optimizing virtual machine placement in iaas data centers : taxonomy, review and open issues. *Cluster Computing*, pages 1–42.

Tan, Z. and Babu, S. (2016). Tempo : robust and self-tuning resource management in multi-tenant parallel databases. *Proceedings of the VLDB Endowment*, 9(10) :720–731.

Tesauro, G., Jong, N. K., Das, R., and Bennani, M. N. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*, pages 65–73. IEEE.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., and Murthy, R. (2010). Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE.

Tsoumakos, D., Konstantinou, I., Boumpouka, C., Sioutas, S., and Koziris, N. (2013). Automated, elastic resource provisioning for nosql clusters using tiramola. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 34–41. IEEE.

Valduriez, P. (1993). Parallel database systems : Open problems and new issues. *Distributed and parallel Databases*, 1(2) :137–165.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache hadoop yarn : Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM.

Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., and Bao, X. (2017). Amazon aurora : Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM.

Verma, A., Cherkasova, L., and Campbell, R. H. (2011). Aria : automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM.

Wan, M., Wu, C., Wang, J., Qiu, Y., Xin, L., Mullender, S., Mühleisen, H., Scheers, B., Zhang, Y., Nes, N., et al. (2016). Column store for gwac : A high-cadence, high-density, large-scale astronomical light curve pipeline and distributed shared-nothing database. *Publications of the Astronomical Society of the Pacific*, 128(969) :114501.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4) :279–292.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. PhD thesis, King's College, Cambridge.

White, T. (2012). *Hadoop : The definitive guide*. " O'Reilly Media, Inc.".

Xue, Y., Xue, K., Gai, N., Hong, J., Wei, D. S., and Hong, P. (2019). An attribute-based controlled collaborative access control scheme for public cloud storage. *IEEE Transactions on Information Forensics and Security*.

Yangui, S., Ravindran, P., Bibani, O., Glitho, R. H., Hadj-Alouane, N. B., Morrow, M. J., and Polakos, P. A. (2016). A platform as-a-service for hybrid cloud/fog environments. In *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–7. IEEE.

Yao, Y., Gao, H., Wang, J., Sheng, B., and Mi, N. (2019). New scheduling algorithms for improving performance and resource utilization in hadoop yarn clusters. *IEEE Transactions on Cloud Computing*.

Yin, S., Hameurlain, A., and Morvan, F. (2018). Sla definition for multi-tenant dbms and its impact on query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 30(11) :2213–2226.

Yu, B., Mitchell, J. E., and Pang, J.-S. (2019). Solving linear programs with complementarity constraints using branch-and-cut. *Mathematical Programming Computation*, 11(2) :267–310.

Yu, P. S. and Cornell, D. W. (1993). Buffer management based on return on consumption in a multi-query environment. *The VLDB Journal—The International Journal on Very Large Data Bases*, 2(1) :1–38.

Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. (2010a). Delay scheduling : a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010b). Spark : Cluster computing with working sets. *HotCloud*, 10(10-10) :95.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing : state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1) :7–18.

Zhao, Y., Calheiros, R. N., Bailey, J., and Sinnott, R. (2016). Sla-based profit optimization for resource management of big data analytics-as-a-service platforms in cloud computing environments. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 432–441. IEEE.

Zhong, C. and Yuan, X. (2019). Intelligent elastic scheduling algorithms for paas cloud platform based on load prediction. In *2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, pages 1500–1503. IEEE.

Zhou, Z., Abawajy, J., Chowdhury, M., Hu, Z., Li, K., Cheng, H., Alelaiwi, A. A., and Li, F. (2018). Minimizing sla violation and power consumption in cloud data centers using adaptive energy-aware algorithms. *Future Generation Computer Systems*, 86 :836–850.

Annexe A

Modèle de coûts pour l'exécution sans-pipeline

Le temps d'exécution estimé (ET_{stat}) d'une requête q, dont le plan est représenté par le modèle d'exécution sans-pipeline, est calculé comme suit dans le cas idéal où les différents phases Map et Reduce sont exécutés l'une après l'autre sans chevauchement avec d'autres requêtes qui partagent les mêmes ressources :

$$ET_{stat}(q) = \sum_{j \in \mathcal{J}_q} et_{stat}(j) \tag{A.1}$$

Avec $\mathcal{J}_q$ est l'ensemble des jobs de la requête q. $et_{stat}(j)$ est le temps d'exécution estimé du job j. En cas de chevauchement avec d'autres requêtes qui partagent les mêmes ressources, le calcul de $ET_{stat}(q)$ doit tenir compte de l'ordonnancement imposé par l'allocation de ressources.

Le temps d'exécution estimé d'un job j est calculé comme suit :

$$et_{stat}(j) = T_{lect-sfd}(j) + T_{map}(j) + T_{rep}(j) + T_{tr}(j) + T_{reduce}(j) + T_{ecrt-sfd}(j) \tag{A.2}$$

$T_{lect-sfd}(j)$ est le temps estimé pour lire les données en entrée du job j à partir du système de fichiers distribué (SFD). $T_{ecrt-sfd}(j)$ est le temps estimé pour écrire les données en sortie du job j sur le système de fichiers distribué (SFD).

$$T_{lect-sfd}(j) = (TDEM_j / pdm_j) / db_{sfd} \tag{A.3}$$

$$T_{ecrt-sfd}(j) = (TDSR_j / pdr_j) / db_{sfd} \tag{A.4}$$

Avec $TDEM_j$ est la taille des données en entrée du job j. $TDSR_j$ est la taille des données en sortie du job j. pdm_j est le degré de parallélisme de la phase map du job

j (i.e. le nombre de tâches dans la phase map). pdr_j est le degré de parallélisme de la phase reduce du job j (i.e. le nombre de tâches dans la phase reduce). db_{sfd} est la bande passante du SFD.

$T_{map}(j)$ (resp. $T_{reduce}(j)$ est le temps estimé des opérateurs exécutés par la phase map (resp. la phase reduce) du job j. Il dépend donc de la nature des opérateurs qui constituent la phase map (resp. reduce), mais aussi de l'algorithme utilisé.

$T_{rep}(j)$ est le temps estimé pour répartir les données en sortie de la phase map :

$$T_{rep}(j) = t_{hash} * (NBSM_j/pdm_j) \tag{A.5}$$

Avec t_{hash} est le temps pour appliquer une fonction de hachage sur un tuple. $NBSM_j$ est le nombre de tuples en sortie de la phase map.

$T_{tr}(j)$ est le temps estimé pour transférer les données entre la phase map et la phase reduce :

$$T_{tr}(j) = ((TDSM_j/(pdm_j * pdr_j))/nb + nd) * max(pdm_j; pdr_j) \tag{A.6}$$

Avec $TDSM_j$ est la taille des données en sortie de la phase map du job j. nb est la bande passante du réseau. nd est la latence du réseau.

Annexe B

Exemple de calcul de T_{op} pour une requête TPC-H

Nous présentons dans cette annexe le détail de calcul de $T_{op}(s)$ pour les stades de la requête de la Figure 7. Pour les stades qui contiennent des jointures ou des agrégations, nous considérons le cas où on applique un algorithme one-pass et aussi two-pass. Les paramètres utilisés sont décrits dans les Tableaux 3.2.1, 3.2.2 et 3.2.3.

Le stade M1 : (1) lire R_{M1} à partir du SFD, (2) appliquer la sélection, (3) appliquer la projection.

$$\begin{aligned} &T_{op}(M1) = (\\ &(||R_{M1}||/pd_{M1})/db_{sfd} + \ \textit{// (1)} \\ &(|R_{M1}|/pd_{M1}) * t_{filter} + \ \textit{// (2)} \\ &\sigma(|R_{M1}|/pd_{M1}) * t_{project}) \ \textit{// (3)} \end{aligned} \tag{B.1}$$

Le stade M2 (jointure one-pass) : (1) lire R_{M2} à partir du disque local, (2) exécuter la phase *build*, (3) lire S_{M2} à partir du SFD, (4) appliquer la sélection, (5) exécuter la

phase *prob*.

$$
\begin{aligned}
&T_{op}(M2) = (\\
&||R_{M2}||/db_l + \ \textit{// (1)}\\
&|R_{M2}| * t_{hash} + \ \textit{// (2)}\\
&(||S_{M2}||/pd_{M2})/db_{sfd} + \ \textit{// (3)}\\
&(|S_{M2}|/pd_{M2}) * t_{filter} + \ \textit{// (4)}\\
&(\sigma(|S_{M2}|)/pd_{M2}) * (t_{hash} + t_{search}) +\\
&(\psi(|R_{M2}| * \sigma(|S_{M2}|))/pd_{M2}) * t_{join}) \ \textit{// (5)}
\end{aligned}
\tag{B.2}
$$

Le stade M2 (jointure two-pass) : (1) lire R_{M2} à partir du disque local, (2) partitionner R_{M2}, (3) écrire R_{M2} sur le disque local, (4) lire S_{M2} à partir du SFD, (5) appliquer la sélection, (6) partitionner S_{M2}, (7) écrire S_{M2} sur le disque local, (8) lire R_{M2} à partir du disque local, (9) exécuter la phase *build*, (10) lire S_{M2} à partir du disque local, (11) exécuter la phase *prob*.

$$
\begin{aligned}
&T_{op}(M2) = (\\
&||R_{M2}||/db_l + \ \textit{// (1)}\\
&|R_{M2}| * t_{hash} + \ \textit{// (2)}\\
&((1-f) * ||R_{M2}||)/db_l + \ \textit{// (3)}\\
&(||S_{M2}||/pd_{M2})/db_{sfd} + \ \textit{// (4)}\\
&(|S_{M2}|/pd_{M2}) * t_{filter} + \ \textit{// (5)}\\
&(\sigma(|S_{M2}|)/pd_{M2}) * t_{hash} + \ \textit{// (6)}\\
&(((1-f) * \sigma(||S_{M2}||))/pd_{M2})/db_l + \ \textit{// (7)}\\
&((1-f) * ||R_{M2}||)/db_l + \ \textit{// (8)}\\
&|R_{M2}| * t_{hash} + \ \textit{// (9)}\\
&(((1-f) * \sigma(||S_{M2}||))/pd_{M2})/db_l + \ \textit{// (10)}\\
&(\sigma(|S_{M2}|)/pd_{M2}) * (t_{hash} + t_{search}) +\\
&(\psi(|R_{M2}| * \sigma(|S_{M2}|))/pd_{M2}) * t_{join}) \ \textit{// (11)}
\end{aligned}
\tag{B.3}
$$

Le stade M3 (jointure one-pass) : (1) lire R_{M3} à partir du disque local, (2) exécuter la phase *build*, (3) lire S_{M3} à partir du SFD, (4) appliquer la sélection, (5) appliquer la projection, (6) exécuter la phase *probe*, (7) partitionner les données, (8) écrire sur le disque local, (9) lire du disque local, (10) exécuter l'agrégation.

$$
\begin{aligned}
&T_{op}(M3) = (\\
&||R_{M3}||/db_l + \ \textit{// (1)}\\
&|R_{M3}| * t_{hash} + \ \textit{// (2)}\\
&(||S_{M3}||/pd_{M3})/db_{sfd} + \ \textit{// (3)}\\
&(|S_{M3}|/pd_{M3}) * t_{filter} + \ \textit{// (4)}\\
&(\sigma(|S_{M3}|)/pd_{M3}) * t_{project} + \ \textit{// (5)}\\
&(\sigma(|S_{M3}|)/pd_{M3}) * (t_{hash} + t_{search}) +\\
&(\psi(|R_{M3}| * \sigma(|S_{M3}|))/pd_{M3}) * t_{join} + \ \textit{// (6)}\\
&(\psi(|R_{M3}| * \sigma(|S_{M3}|))/pd_{M3}) * t_{hash} + \ \textit{// (7)}\\
&\psi(||R_{M3}|| * \pi(\sigma(||S_{M3}||))/pd_{M3})/db_l + \ \textit{// (8)}\\
&\psi(||R_{M3}|| * \pi(\sigma(||S_{M3}||))/pd_{M3})/db_l + \ \textit{// (9)}\\
&(\psi(|R_{M3}| * \sigma(|S_{M3}|))/pd_{M3}) * (t_{hash} + t_{search} + t_{agg})) \ \textit{// (10)}
\end{aligned}
\tag{B.4}
$$

Le stade M3 (jointur two-pass) : (1) lire R_{M3} à partir du disque local, (2) partitionner R_{M3}, (3) écrire R_{M3} sur le disque local, (4) lire S_{M3} à partir du SFD, (5) appliquer la sélection, (6) appliquer la proejction, (7) partitionner S_{M3}, (8) écrire S_{M3} sur le disque local, (9) lire R_{M3} à partir du disque local, (10) exécuter la phase *build*, (11) lire S_{M3} à partir du SFD, (12) exécuter la phase *probe*, (13) partitionner les données, (14) écrire les données sur le disque local, (15) lire les données à partir du disque local, (16) exécuter

l'agrégation.

$$
\begin{aligned}
&T_{op}(M3) = (\\
&||R_{M3}||/db_l + \ \textit{// (1)}\\
&|R_{M3}| * t_{hash} + \ \textit{// (2)}\\
&((1-f) * ||R_{M3}||)/db_l + \ \textit{// (3)}\\
&(||S_{M3}||/pd_{M3})/db_{sfd} + \ \textit{// (4)}\\
&(|S_{M3}|/pd_{M3}) * t_{filter} + \ \textit{// (5)}\\
&(\sigma(|S_{M3}|) * t_{project} + \ \textit{// (6)}\\
&(\sigma(|S_{M3}|)/pd_{M3}) * t_{hash} + \ \textit{// (7)}\\
&(((1-f) * \pi(\sigma(||S_{M3}||)))/pd_{M3})/db_l + \ \textit{// (8)}\\
&((1-f) * ||R_{M3}||)/db_l + \ \textit{// (9)}\\
&|R_{M3}| * t_{hash} + \ \textit{// (10)}\\
&(((1-f) * \pi(\sigma(||S_{M3}||)))/pd_{M3})/db_l + \ \textit{// (11)}\\
&(\sigma(|S_{M3}|)/pd_{M3}) * (t_{hash} + t_{search}) +\\
&((\psi(|R_{M3}| * \sigma(|S_{M3}|))/pd_{M3}) * t_{join}) + \ \textit{// (12)}\\
&((\psi(|R_{M3}| * \sigma(|S_{M3}|))/pd_{M3}) * t_{hash} + \ \textit{// (13)}\\
&((\psi(||R_{M3}|| * \pi(\sigma(||S_{M3}||)))/pd_{M3})/db_l + \ \textit{// (14)}\\
&(\psi(||R_{M3}|| * \pi(\sigma(||S_{M3}||)))/pd_{M3})/db_l + \ \textit{// (15)}\\
&(|R_{M3}|/pd_{M3}) * (t_{hash} + t_{search} + t_{agg})) \ \textit{// (16)}
\end{aligned}
\tag{B.5}
$$

Le stade R1 (agrégation one-pass) : (1) lire R_{R1} à partir du disque local, (2) exécuter l'agrégation.

$$
\begin{aligned}
&T_{op}(R1) = (\\
&(||R_{R1}||/pd_{R1})/db_l + \ \textit{// (1)}\\
&(|R_{R1}|/pd_{R1}) * (t_{hash} + t_{search} + t_{agg})) \ \textit{// (2)}
\end{aligned}
\tag{B.6}
$$

Le stade R1 (agrégation two-pass) : (1) lire R_{R1} à partir du disque local, (2) partitionner les données, (3) écrire les données sur le disque local, (4) lire les données à

partir du disque local, (5) exécuter l'agrégation.

$$
\begin{aligned}
&T_{op}(R1) = (\\
&(||R_{R1}||/pd_{R1})/db_l + \ \textit{// (1)}\\
&(||R_{R1}||/pd_{R1}) * t_{hash} + \ \textit{// (2)}\\
&(((1-f) * ||R_{R1}||)/pd_{R1})/db_l + \ \textit{// (3)}\\
&(((1-f) * ||R_{R1}||)/pd_{R1})/db_l + \ \textit{// (4)}\\
&(|R_{R1}|/pd_{R1}) * (t_{hash} + t_{search} + t_{agg}) \ \textit{// (5)}
\end{aligned}
\tag{B.7}
$$

Le stade R2 : (1) lire à partir du disque local, (2) écrire sur le SFD.

$$
\begin{aligned}
&T_{op}(R2) = (\\
&limit(R_{R2})/db_l + \ \textit{// (1)}\\
&limit(R_{R2})/db_{sfd}) \ \textit{// (2)}
\end{aligned}
\tag{B.8}
$$

Annexe C

Formulation PLNE pour le modèle d'exécution sans-pipeline

La formulation PLNE (contraintes et fonction objective linéaires) du placement et d'ordonnancement sans-pipeline est présentée dans ce qui suit. Les Tableaux C.0.1, C.0.2 et C.0.3 définissent respectivement les ensembles, les paramètres et les variables utilisés.

C.1 Placement (1^{ere} phase)

Nous introduisons les familles de variables x et y :

— $x_{i,m,a}$ indique si la tâche map $m \in \mathcal{M}_i$ du job $i \in \mathcal{J}$ est placée sur la ressource map $a \in \mathcal{A}$ $(= 1)$, ou non $(= 0)$.

— $y_{i,r,b}$ indique si la tâche reduce $r \in \mathcal{R}_i$ du job $i \in \mathcal{J}$ est placée sur la ressource reduce $b \in \mathcal{B}$ $(= 1)$, ou non $(= 0)$.

Il s'agit donc de variables binaires :

Tableau C.0.1 – Notation utilisée pour les ensembles de la formulation PLNE sans-pipeline

Ensembles	
$\mathcal{J}$	Ensemble des jobs de toutes les requêtes soumises
$\mathcal{M}_i$	Ensemble de tâches map du job i
$\mathcal{R}_j$	Ensemble tâches reduce du job j
$\mathcal{A}$	Ensemble de ressources map
$\mathcal{B}$	Ensemble de ressources reduce
$\mathcal{F}$	Ensemble des jobs finaux des requêtes soumises

Tableau C.0.2 – Notation utilisée pour les paramètres de la formulation PLNE sans-pipeline

paramètres	
$C^M_{mem}(i)$	Quantité mémoire minimale pour une une tâche map du job $i \in \mathcal{J}$
$C^R_{mem}(i)$	Quantité mémoire minimale pour une une tâche reduce du job $i \in \mathcal{J}$
$C_{mem}(a)$	Quantité mémoire minimale garantit par la ressource $a \in \mathcal{A} \cup \mathcal{B}$
$Fm_{a,t}$	Indique si la ressource map $a \in \mathcal{A}$ est initialement disponible à l'instant t $(= 1)$, ou non $(= 0)$
$Fr_{b,t}$	Indique si la ressource reduce $b \in \mathcal{B}$ est initialement disponible à l'instant t $(= 1)$, ou non $(= 0)$
$Q_{i,j}$	Quantité de données transférées entre une tâche du job $i \in \mathcal{J}$ et une tâche du job $j \in \mathcal{J}$
Tm_i	Temps de réponse local d'une tâche map du job $i \in \mathcal{J}$
Tr_i	Temps de réponse local d'une tâche reduce du job $i \in \mathcal{J}$
T	Nombre de fenêtres temporelles futures prises en compte
$Dist(a, b)$	Distance entre la ressource a et b
W_{proc}	Poids d'utilisation du processeur
W_{mem}	Poids d'utilisation de la mémoire
W_{stor}	Poids associé au stockage des résultats intermédiaires
W_{com}	Poids de la communication
W_{rep}	Poids de l'équilibrage de charge
$C_{map}(a)$	Coût d'une ressource map $a \in \mathcal{A}$, $C_{map}(a) = W_{proc} + W_{mem} * C_{mem}(a)$
$C_{reduce}(b)$	Coût d'une ressource reduce $b \in \mathcal{B}, C_{reduce}(b) = W_{proc} + W_{mem} * C_{mem}(b)$
$C_{com}(a, b)$	Coût de communication, $C_{com}(a, b) = W_{com} * Dist(a, b)$
D_i	Date limite de la requête à laquelle appartient le job $i \in \mathcal{J}$
W_i	Poids de pénalité du job $i \in \mathcal{F}$
q^m_i	Quantité estimée de données générées des tâches map du job $i \in \mathcal{J}$
q^r_i	Quantité estimée de données générées des tâches reduce du job $i \in \mathcal{J}$
Am_{im}	Indique la ressource sur laquelle la tâche Map $m \in \mathbb{M}_i$ du job $i \in \mathbb{J}$ a été placée après la phase de placement
Ar_{ir}	Indique la ressource sur laquelle la tâche reduce $r \in \mathbb{R}_i$ du job $i \in \mathbb{J}$ a été placée après la phase de placement
$O_{i,j}$	Indique l'ordre entre les jobs. Si $O_{i,j} = 1$ alors le job $j \in \mathbb{J}$ ne peut pas commencer avant la fin du job $i \in \mathbb{J}$

Tableau C.0.3 – Notation utilisée pour les variables de la formulation PLNE sans-pipeline

Variables du placement	
$x_{i,m,a}$	Indique si la tâche map $m \in \mathcal{M}_i$ du job $i \in \mathcal{J}$ est placée sur la ressource map $a \in \mathcal{A}$ $(= 1)$, ou non $(= 0)$
$y_{i,r,b}$	Indique si la tâche reduce $r \in \mathcal{R}_i$ du job $i \in \mathcal{J}$ est placée sur la ressource reduce $b \in \mathcal{B}$ $(= 1)$, ou non $(= 0)$
$z_{a,b}$	la quantité maximale de données transférées entre les tâches placées sur la ressource a et les tâches placées sur la ressource b
α	variable fictive utilisée dans la contrainte (C.10)
β	variable fictive utilisée dans la contrainte (C.11)
Variables de l'ordonnancement	
$v_{i,m,t}$	Indique si la tâche map $m \in \mathcal{M}_i$ du job $i \in \mathcal{J}$ a commencé à ou avant l'instant t $(= 1)$, ou non $(= 0)$
$w_{i,r,t}$	Indique si la tâche reduce $r \in \mathcal{R}_i$ du job $i \in \mathcal{J}$ a commencé à ou avant l'instant t $(= 1)$, ou non $(= 0)$
$u_{i,m,t}$	Indique si les résultats de la tâche $m \in \mathcal{M}_i \cup \mathcal{R}_i$ du job $i \in \mathcal{J}$ sont stockés sur disque à l'instant $t \in \{0, ..., T\}$ $(= 1)$ ou non $(= 0)$
γ	Famille de variables fictives utilisées dans les contraintes (C.31) et (C.32)

$$x_{i,m,a} \in \{0,1\},\ \forall i \in \mathcal{J}, m \in \mathcal{M}_i, a \in \mathcal{A} \tag{C.1}$$

$$y_{j,r,b} \in \{0,1\},\ \forall j \in \mathcal{J}, r \in \mathcal{R}_j, b \in \mathcal{B} \tag{C.2}$$

Plusieurs tâches peuvent être affectées à la même ressource. L'exclusivité d'exécution est alors assurée dans le temps avec le modèle d'ordonnancement qui sera présenté dans la section C.2. La quantité mémoire minimale C^M_{mem} pour une tâche map ne doit pas dépasser la quantité mémoire minimale garantit C_{mem} par la ressource choisie (équation C.3). De même, pour les tâches reduce (équations C.4) :

$$C^M_{mem}(i) * x_{i,m,a} \leq C_{mem}(a),\ \forall i \in \mathcal{J}, m \in \mathcal{M}_i, a \in \mathcal{A} \tag{C.3}$$

$$C^R_{mem}(i) * y_{i,r,b} \leq C_{mem}(b),\ \forall i \in \mathcal{J}, r \in \mathcal{R}_i, b \in \mathcal{B} \tag{C.4}$$

Chaque tâche est placée dans une seule et unique ressource :

$$\sum_{a \in \mathcal{A}} x_{i,m,a} = 1, \ \forall i \in \mathcal{J}, m \in \mathcal{M}_i \tag{C.5}$$

$$\sum_{b \in \mathcal{B}} y_{i,r,b} = 1, \ \forall i \in \mathcal{J}, r \in \mathcal{R}_i \tag{C.6}$$

Afin d'assurer le parallélisme inter-job, deux tâches map (ou reduce) appartenant au même job ne peuvent pas être placées dans la même ressource :

$$\sum_{m \in \mathcal{M}_i} x_{i,m,a} \leq 1, \ \forall i \in \mathcal{J}, a \in \mathcal{A} \tag{C.7}$$

$$\sum_{r \in \mathcal{R}_i} y_{i,r,b} \leq 1, \ \forall i \in \mathcal{J}, b \in \mathcal{B} \tag{C.8}$$

Notre formulation PLNE du placement gère l'équilibrage de charge. Afin d'assurer une répartition équitable des tâches entre les ressources, nous proposons de minimiser le nombre maximum de fenêtres de temps allouées dans chaque ressource, ce qui évite d'allouer trop de tâches à une ressource et peu à une autre. Pour modéliser cela de manière linéaire, nous introduisons deux variables α et β. Nous ajoutons les contraintes suivantes. La fonction objectif présentée par la suite inclut α et β comme variables à minimiser.

$$\alpha, \beta \in \{0, 1....., T\} \tag{C.9}$$

$$\sum_{i \in \mathcal{J}} \sum_{m \in \mathcal{M}_i} Tm_i * x_{i,m,a} + \sum_{t<T} (1 - Fm_{a,t}) \leq \alpha, \ \forall a \in \mathcal{A} \tag{C.10}$$

$$\sum_{i \in \mathcal{J}} \sum_{r \in \mathcal{R}_i} Tr_i * x_{i,r,b} + \sum_{t<T} (1 - Fr_{b,t}) \leq \beta, \ \forall b \in \mathcal{B} \tag{C.11}$$

Avec $Fm_{a,t} = 1$ si la ressource Map a est initialement disponible à l'instant t, $= 0$ sinon, $Fr_{b,t} = 1$ si la ressource Reduce b est initialement disponible à l'instant t, $= 0$ sinon, Tm_i est le temps de réponse local des tâches Map du job i, Tr_i est le temps de réponse local d'une tâche Reduce du job i.

$z_{a,b}$ est la quantité maximale de données transférées entre les tâches placées sur la ressource a et les tâches placées sur la ressource b. La famille de variables z satisfait la condition suivante :

$$z_{a,b} \in \{0, 1...., UpperBound(z)\},\ \forall a \in \mathcal{A}, b \in \mathcal{B} \tag{C.12}$$

$$x_{i,m,a} = 1 \text{ and } y_{j,r,b} = 1 \Rightarrow z_{a,b} \geq Q_{i,j}$$
$$\forall i, j \in \mathcal{J}, m \in \mathcal{M}_i, r \in \mathcal{R}_j, a \in \mathcal{A}, b \in \mathcal{B}, Q_{i,j} > 0 \tag{C.13}$$

$Q_{i,j}$ est la quantité de données transférées entre une tâche du job i et une tâche du job j. Cette condition peut être exprimée linéairement comme suit :

$$Q_{i,j} * x_{i,m,a} + Q_{i,j} * y_{j,r,b} - z_{a,b} \leq Q_{i,j},$$
$$\forall i, j \in \mathcal{J}, m \in \mathcal{M}_i, r \in \mathcal{R}_j, a \in \mathcal{A}, b \in \mathcal{B}, Q_{i,j} > 0 \tag{C.14}$$

La fonction d'objectif du modèle de placement comprend les coûts des ressources et réseau. W_{proc} (poids d'utilisation du processeur) et W_{mem} (poids d'utilisation de la mémoire) sont déduits selon l'influence du processeur et la mémoire sur le coût monétaire :

$$\begin{aligned} f = & \sum_{i \in \mathcal{J}} \sum_{m \in \mathcal{M}_i} \sum_{a \in \mathcal{A}} C_{map}(a) * Tm_i * x_{i,m,a} \\ & + \sum_{i \in \mathcal{J}} \sum_{r \in \mathcal{R}_i} \sum_{b \in \mathcal{B}} C_{reduce}(b) * Tr_i * y_{i,r,b} \\ & \sum_{a \in \mathcal{A}} \sum_{b \in \mathcal{B}} C_{com}(a,b) * z_{a,b} + W_{rep} * (\alpha + \beta) \end{aligned} \tag{C.15}$$

Avec W_{rep} est le poids de l'équilibrage de charge, $C_{com}(a,b) = W_{com} * Dist(a,b)$, $C_{map}(a) = W_{proc} + W_{mem} * C_{mem}(a)$ et $C_{reduce}(b) = W_{proc} + W_{mem} * C_{mem}(b)$, $Dist(a,b)$ est la distance entre la ressource a et b, W_{com} est le poids de la communication. La formulation du problème à résoudre est :

$$\begin{aligned} &\text{minimize} && f \\ &\text{subject to} && (C.1), (C.2), (C.3), (C.4), (C.5), (C.6), (C.7), (C.8), \\ & && (C.9), (C.10), (C.11), (C.12), (C.14) \end{aligned}$$

La solution du problème est trouvée avec le logiciel GLPK.

C.2 Ordonnancement (2^{eme} phase)

La configuration optimale du modèle de placement précédent est considérée comme une entrée pour le modèle d'ordonnancement que nous présentons dans cette section. Nous recherchons maintenant les fenêtres de temps allouées à chaque tâche. Les variables suivantes sont introduites :

— $v_{i,m,t} = 1$ si la tâche Map m du job i a commencé à ou avant le temps t ; $= 0$ sinon.

— $w_{i,r,t} = 1$ si la tâche Reduce r du job i a commencé à ou avant le temps t ; $= 0$ sinon.

Nous avons donc :

$$v_{i,m,t} \in \{0,1\},\ \forall i \in \mathcal{J}, m \in \mathcal{M}_i, t < T \tag{C.16}$$

$$w_{i,r,t} \in \{0,1\},\ \forall i \in \mathcal{J}, r \in \mathcal{R}_i, t < T \tag{C.17}$$

$$v_{i,m,t} \leq v_{i,m,t+1},\ \forall i \in \mathcal{J}, m \in \mathcal{M}_i, t < T \tag{C.18}$$

$$w_{i,r,t} \leq w_{i,r,t+1},\ \forall i \in \mathcal{J}, r \in \mathcal{R}_i, t < T \tag{C.19}$$

Pour un job donné, les tâches reduce ne peuvent pas commencer avant la fin des tâches map. Cette contrainte peut être exprimée de façon linéaire :

$$v_{i,m,t-Tm_i} \geq w_{i,r,t},\ \forall i \in \mathcal{J}, m \in \mathcal{M}_i, r \in \mathcal{R}_i, t - Tm_i \geq 1 \tag{C.20}$$

$$w_{i,r,t} \leq 0,\ \forall i \in \mathcal{J}, m \in \mathcal{M}_i, t - Tm_i < 1 \tag{C.21}$$

Une ressource ne peut pas contenir plus d'une tâche à la fois (contrainte d'exclusivité). De la définition des familles de variables v et w et sachant qu'une tâche ne peut être interrompue avant sa fin, on peut le déduire :

— $v_{i,m,t} - v_{i,m,t-Tm_i} = 1$ si la tâche Map m du job i utilise la ressource a à l'instant t ; $= 0$ sinon.

— $w_{i,r,t} - w_{i,r,t-Tr_i} = 1$ si la tâche Reduce r du job i utilise la ressource b à l'instant t ; $= 0$ sinon.

Am_{im} (ou Ar_{ir}) indique la ressource sur laquelle la tâche Map m (resp. tâche Reduce r) du job i a été placée après la phase de placement. La formulation linéaire de la contrainte d'exclusivité est donc la suivante :

$$\sum_{i\in\mathcal{J}} \sum_{\substack{m\in\mathcal{M}_i \\ Am_{im}=a}} \sum_{t-Tm_i\geq 1} (v_{i,m,t} - v_{i,m,t-Tm_i})$$

$$+\sum_{i\in\mathcal{J}} \sum_{\substack{m\in\mathcal{M}_i \\ Am_{im}=a}} \sum_{t-Tm_i<1} v_{i,m,t} \leq Fm_{a,t},\ \forall a \in \mathcal{A}, t < T \tag{C.22}$$

$$\sum_{i\in\mathcal{J}} \sum_{\substack{r\in\mathcal{R}_i \\ Ar_{ir}=b}} \sum_{t-Tr_i\geq 1} (w_{i,r,t} - w_{i,r,t-Tr_i})$$

$$+\sum_{i\in\mathcal{J}} \sum_{\substack{r\in\mathcal{R}_i \\ Ar_{ir}=b}} \sum_{t-Tr_i<1} w_{i,r,t} \leq Fr_{b,t},\ \forall b \in \mathcal{B}, t < T \tag{C.23}$$

La famille de paramètres O indique la priorité entre les jobs. Si $O_{i,j} = 1$ alors le job j ne peut pas commencer avant la fin du job i. Nous proposons la formulation linéaire suivante pour la contrainte de priorité :

$$v_{j,m,t} - w_{i,r,t-Tr_i} \leq 1 - O_{i,j},$$

$$\forall i,j \in \mathcal{J}, m \in \mathcal{M}_j, r \in \mathcal{R}_i, t - Tr_i \geq 1 \tag{C.24}$$

$$v_{j,m,t} \leq 1 - O_{i,j},\ \forall i,j \in \mathcal{J}, m \in \mathcal{M}_j, t - Tr_i < 1 \tag{C.25}$$

Les résultats intermédiaires d'une tâche sont maintenus sur l'espace de stockage local jusqu'à ce que les tâches successives commencent. La variable $u_{i,m,t}$ indique si les résultats de la tâche $m \in \mathcal{M}_i \cup \mathcal{R}_i$ du job $i \in \mathcal{J}$ sont stockés sur le disque à l'instant t ou non $(= 0)$:

$$v_{i,m,t} \in \{0,1\},\ \forall i \in \mathcal{J}, m \in \mathcal{M}_i \cup \mathcal{R}_i, t < T \tag{C.26}$$

$$u_{i,m,t}v_{i,m,t} = 1 \text{ and } w_{i,r,t} = 0 \Rightarrow u_{i,m,t} = 1$$

$$\forall i \in \mathcal{J}, m \in \mathcal{M}_i, r \in \mathcal{R}_i, t < T \tag{C.27}$$

$$w_{i,r,t} = 1 \text{ and } v_{j,m,t} = 0 \Rightarrow u_{i,r,t} = 1$$

$$\forall i,j \in \mathcal{J}, m \in \mathcal{M}_i, r \in \mathcal{R}_i, t < T, O_{i,j} = 1 \tag{C.28}$$

Les contraintes (C.27) et (C.28) peuvent être exprimées linéairement comme suit :

$$v_{i,m,t} + (1 - w_{i,r,t}) - u_{i,m,t} \leq 1,$$
$$\forall i \in \mathcal{J}, m \in \mathcal{M}_i, r \in \mathcal{R}_i, t < T \tag{C.29}$$
$$O_{i,j} * w_{i,r,t} + O_{i,j} * (1 - v_{j,m,t}) - u_{i,r,t} \leq 1,$$
$$\forall i, j \in \mathcal{J}, m \in \mathcal{M}_j, r \in \mathcal{R}_i, t < T, O_{i,j} = 1 \tag{C.30}$$

Les paramètre économique qui influe sur l'ordonnancement des tâches est le coût des pénalités et disque. Un poids de pénalité W_i est associé à chaque fenêtre de temps d'exécution après la date limite. Par exemple, le fournisseur peut proposer deux offres : (1) basic et (2) premium. Il est clair que la violation des SLAs de l'offre (2) est moins tolérée que l'offre (1). Ainsi, si un job i est de l'offre (1) et que le job j est de l'offre (2) alors $W_j > W_i$. Chaque requête a une date limite, qui est spécifiée dans le SLA. L'accumulation des pénalités commence à partir du moment où l'exécution des requêtes dépasse la date limite. La fonction objective à minimiser est la suivante :

$$g = \sum_{i \in \mathcal{F}} \sum_{D_i - T r_i < t < T} W_i * \max_{r \in \mathcal{R}_i} (1 - w_{i,r,t}) + W_{stor} * (\sum_{i \in \mathcal{J}} \sum_{m \in \mathcal{M}_i} \sum_{t < T} q_i^m * u_{i,m,t} + \sum_{i \in \mathcal{J}} \sum_{r \in \mathcal{R}_i} \sum_{t < T} q_i^r * u_{i,r,t})$$

Cette fonction objectif n'est pas linéaire. Pour avoir une forme linéaire, nous introduisons une famille de variables γ telle que :

$$\gamma_{i,t} \in \{0, 1\}, \ \forall i \in \mathcal{J}, t < T \tag{C.31}$$
$$1 - w_{i,r,t} \leq \gamma_{i,t} \ \forall i \in \mathcal{J}, r \in \mathcal{R}_i, t < T \tag{C.32}$$

La fonction de l'objectif peut être formulée linéairement comme suit :

$$g' = \sum_{i \in \mathcal{F}} \sum_{D_i - T r_i < t < T} W_i * \gamma_{i,t} + W_{stor} * (\sum_{i \in \mathcal{J}} \sum_{m \in \mathcal{M}_i} \sum_{t < T} q_i^m * u_{i,m,t} + \sum_{i \in \mathcal{J}} \sum_{r \in \mathcal{R}_i} \sum_{t < T} q_i^r * u_{i,r,t})$$

Le problème à résoudre est :

$$\begin{aligned} &\text{minimize} && g' \\ &\text{subject to} && (C.16), (C.17), (C.18), (C.19), (C.20), (C.21), (C.22), (C.23), \\ & && (C.24), (C.25), (C.26), (C.29), (C.30), (C.31), (C.32) \end{aligned}$$

La solution du problème est trouvée avec le logiciel GLPK.

Annexe D

Exemple de plan d'exécution sous Hive

Nous présentons dans cette annexe un exemple du plan d'exécution affiché quand le mot-clé *EXPLAIN* est ajouté à une requête quasi-SQL sous Hive. La requête considérée est celle de la Figure 15 du chapitre 5. Les lignes 4 à 9 décrivent les dépendances entre les stades. Nous disposons de 5 stades : *Map 1*, *Map 2*, *Map 3*, *Reducer 4*, *Reducer 5*. Le reste du fichier décrit le détail de chaque stades, notamment : la liste des opérations, le nombre de tuples et la taille des données intermédiaires.

```
1  OK
2  Plan optimized by CBO.
3
4  Vertex dependency in root stage
5  Map 1 <- Map 2 (BROADCAST_EDGE)
6  Map 3 <- Map 1 (BROADCAST_EDGE)
7  Reducer 4 <- Map 3 (SIMPLE_EDGE)
8  Reducer 5 <- Reducer 4 (SIMPLE_EDGE)
9
10 Stage-0
11    Fetch Operator
12       limit:10
13       Stage-1
14          Reducer 5
15          File Output Operator [FS_28]
16             compressed:false
17             Statistics:Num rows: 10 Data size: 1140 Basic stats: COMPLETE Column stats:
18             COMPLETE
19             table:{"input format:":"org.apache.hadoop.mapred.TextInputFormat","output
20             format:":"org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat",
21             "serde:":"org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe"}
22             Limit [LIM_27]
23                Number of rows:10
24                Statistics:Num rows: 10 Data size: 1140 Basic stats: COMPLETE Column
25                stats: COMPLETE
26                Select Operator [SEL_26]
27                |  outputColumnNames:["_col0","_col1","_col2","_col3"]
28                |  Statistics:Num rows: 41541220 Data size: 4735699080 Basic stats:
29                COMPLETE Column stats: COMPLETE
30                |<-Reducer 4 [SIMPLE_EDGE]
31                   Reduce Output Operator [RS_25]
32                      key expressions:_col1 (type: double), _col2 (type: string)
33                      sort order:-+
34                      Statistics:Num rows: 41541220 Data size: 4735699080 Basic stats:
35                      COMPLETE Column stats: COMPLETE
36                      value expressions:_col0 (type: bigint), _col3 (type: int)
```

```
Select Operator [SEL_24]
   outputColumnNames:["_col0","_col1","_col2","_col3"]
   Statistics:Num rows: 41541220 Data size: 4735699080 Basic stats:
   COMPLETE Column stats: COMPLETE
   Group By Operator [GBY_23]
   |  aggregations:["sum(VALUE._col0)"]
   |  keys:KEY._col0 (type: bigint), KEY._col1 (type: string),
   |   KEY._col2 (type: int)
   |  outputColumnNames:["_col0","_col1","_col2","_col3"]
   |  Statistics:Num rows: 41541220 Data size: 4735699080 Basic
   stats: COMPLETE Column stats: COMPLETE
   |<-Map 3 [SIMPLE_EDGE] vectorized
      Reduce Output Operator [RS_22]
         key expressions:_col0 (type: bigint), _col1 (type:
         string), _col2 (type: int)
         Map-reduce partition columns:_col0 (type: bigint), _col1
         (type: string), _col2 (type: int)
         sort order:+++
         Statistics:Num rows: 41541220 Data size: 4735699080 Basic
         stats: COMPLETE Column stats: COMPLETE
         value expressions:_col3 (type: double)
         Group By Operator [OP_52]
            aggregations:["sum(_col3)"]
            keys:_col0 (type: bigint), _col1 (type: string), _col2
            (type: int)
            outputColumnNames:["_col0","_col1","_col2","_col3"]
            Statistics:Num rows: 41541220 Data size: 4735699080
            Basic stats: COMPLETE Column stats: COMPLETE
            Select Operator [OP_51]
               outputColumnNames:["_col0","_col1","_col2","_col3"]
               Statistics:Num rows: 41541220 Data size: 5068028840
               Basic stats: COMPLETE Column stats: COMPLETE
               Map Join Operator [MAPJOIN_50]
               |  condition map:[{"":"Inner Join 0 to 1"}]
               |  HybridGraceHashJoin:true
               |  keys:{"Map 3":"_col0 (type: bigint)","Map 1":"_col0
                  (type: bigint)"}
               |  outputColumnNames:["_col2","_col3","_col6","_col7"
                  ,"_col8"]
               |  Statistics:Num rows: 41541220 Data size: 5068028840
                  Basic stats: COMPLETE Column stats: COMPLETE
               |<-Map 1 [BROADCAST_EDGE] vectorized
               |  Reduce Output Operator [RS_47]
               |     key expressions:_col0 (type: bigint)
               |     Map-reduce partition columns:_col0 (type: bigint)
               |     sort order:+
               |     Statistics:Num rows: 7142858 Data size:
                     757142948 Basic stats: COMPLETE Column stats: COMPLETE
               |     value expressions:_col2 (type: string),
                     _col3 (type: int)
               |     Map Join Operator [MAPJOIN_46]
               |     |  condition map:[{"":"Inner Join 0 to 1"}]
               |     |  HybridGraceHashJoin:true
               |     |  keys:{"Map 2":"_col0 (type: bigint)","Map 1"
                        :"_col1 (type: bigint)"}
               |     |  outputColumnNames:["_col0","_col2","_col3"]
               |     |  Statistics:Num rows: 7142858 Data size:
                        757142948 Basic stats: COMPLETE Column stats: COMPLETE
               |     |<-Map 2 [BROADCAST_EDGE] vectorized
               |     |  Reduce Output Operator [RS_43]
               |     |     key expressions:_col0 (type: bigint)
               |     |     Map-reduce partition columns:_col0
                           (type: bigint)
               |     |     sort order:+
               |     |     Statistics:Num rows: 2142857 Data size:
                           17142856 Basic stats: COMPLETE Column stats:
                           COMPLETE
               |     |     Select Operator [OP_42]
               |     |        outputColumnNames:["_col0"]
               |     |        Statistics:Num rows: 2142857 Data
                              size: 17142856 Basic stats: COMPLETE Column
                              stats: COMPLETE
```

```
|     |        Filter Operator [FIL_41]
|     |           predicate:((c_mktsegment = 'BUILDING')
                    and c_custkey is not null) (type: boolean)
|     |           Statistics:Num rows: 2142857 Data
                    size: 216428557 Basic stats: COMPLETE
                    Column stats: COMPLETE
|     |           TableScan [TS_3]
|     |              alias:customer
|     |              Statistics:Num rows: 15000000
                       Data size: 8730004039 Basic stats:
                       COMPLETE Column stats: COMPLETE
|     |<-Select Operator [OP_45]
|           outputColumnNames:["_col0","_col1",
            "_col2","_col3"]
|           Statistics:Num rows: 50000000 Data size:
            5700000000 Basic stats: COMPLETE Column
            stats: COMPLETE
|           Filter Operator [FIL_44]
|              predicate:((o_orderdate < '1995-03-22')
               and o_custkey is not null and o_orderkey
               is not null) (type: boolean)
|              Statistics:Num rows: 50000000 Data size:
               5700000000 Basic stats: COMPLETE Column
               stats: COMPLETE
|              TableScan [TS_0]
|                 alias:orders
|                 Statistics:Num rows: 150000000 Data
                    size: 79500000000 Basic stats:
                    COMPLETE Column stats: COMPLETE
|<-Select Operator [OP_49]
      outputColumnNames:["_col0","_col1","_col2"]
      Statistics:Num rows: 200012634 Data size: 7142858
      Basic stats: COMPLETE Column stats: COMPLETE
      Filter Operator [FIL_48]
         predicate:((l_shipdate > '1995-03-22') and
         l_orderkey is not null) (type: boolean)
         Statistics:Num rows: 200012634 Data size:
         23601490812 Basic stats: COMPLETE Column stats: COMPLETE
         TableScan [TS_6]
            alias:lineitem
            Statistics:Num rows: 600037902 Data size:
            483318264443 Basic stats: COMPLETE Column
            stats: COMPLETE
```