Collège Doctoral
Université Sud de France
Montpellier - Perpignan

# THÈSE

Pour obtenir le grade de
Docteur

Délivré par l'**Université Montpellier II**

Préparée au sein de l'école doctorale **I2S***
Et de l'unité de recherche **UMR 5506**

Spécialité: **Informatique**

Présentée par **Miguel Liroz Gistau**
miguel.liroz_gistau@inria.fr

# Partitionnement dans les Systèmes de Gestion de Données Parallèles

Soutenue le 17/12/2013 devant le jury composé de :

| | |
|---|---|
| Pascal Molli, Professeur, Université de Nantes | Rapporteur |
| Abdelkader Hameurlain, Professeur, Université Toulouse III | Rapporteur |
| Marta Patiño-Martínez, Professeur, U. Polytechnique de Madrid | Examinatrice |
| Reza Akbarinia, Chargé de Recherche, INRIA, Montpellier | Encadrant |
| Esther Pacitti, Professeur, Université de Montpellier 2 | Directrice |
| Patrick Valduriez, Directeur de Recherche, INRIA, Montpellier | Directeur |

UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

* **I2S:** École doctorale Information Structures Systèmes

# Résumé

Au cours des dernières années, le volume des données qui sont capturées et générées a explosé. Les progrès des technologies informatiques, qui fournissent du stockage à bas prix et une très forte puissance de calcul, ont permis aux organisations d'exécuter des analyses complexes de leurs données et d'en extraire des connaissances précieuses. Cette tendance a été très importante non seulement pour l'industrie, mais a également pour la science, où les meilleures instruments et les simulations les plus complexes ont besoin d'une gestion efficace des quantités énormes de données.

Le parallélisme est une technique fondamentale dans la gestion de données extrêmement volumineuses car il tire parti de l'utilisation simultanée de plusieurs ressources informatiques. Pour profiter du calcul parallèle, nous avons besoin de techniques de partitionnement de données efficaces, qui sont en charge de la division de l'ensemble des données en plusieurs partitions et leur attribution aux nœuds de calculs. Le partitionnement de données est un problème complexe, car il doit prendre en compte des questions différentes et souvent contradictoires telles que la localité des données, la répartition de charge et la maximisation du parallélisme.

Dans cette thèse, nous étudions le problème de partitionnement de données, en particulier dans les bases de données parallèles scientifiques qui sont continuellement en croissance. Nous étudions également ces partitionnements dans le cadre MapReduce.

Dans le premier cas, nous considérons le partitionnement de très grandes bases de données dans lesquelles des nouveaux éléments sont ajoutés en permanence, avec pour exemple une application aux données astronomiques. Les approches existantes sont limitées à cause de la complexité de la charge de travail et l'ajout en continu de nouvelles données limitent l'utilisation d'approches traditionnelles. Nous proposons deux algorithmes de partitionnement dynamique qui attribuent les nouvelles données aux partitions en utilisant une technique basée sur l'affinité. Nos algorithmes permettent d'obtenir de très bons partitionnements des données en un temps d'exécution réduit comparé aux approches traditionnelles.

Nous étudions également comment améliorer la performance du framework MapReduce en utilisant des techniques de partitionnement de données. En particulier, nous sommes intéressés par le partitionnement efficient de données d'entrée

avec l'objectif de réduire la quantité de données qui devront être transférées dans la phase intermédiaire, connu aussi comme « shuffle ». Nous concevons et mettons en œuvre une stratégie qui, en capturant les relations entre les tuples d'entrée et les clés intermédiaires, obtient un partitionnement efficace qui peut être utilisé pour réduire de manière significative le surcharge de communications dans MapReduce.

## Titre en français

*Partitionnement dans les Systèmes de Gestion de Données Parallèles*

## Mots-clés

- Partitionnement de données
- Systèmes parallèles
- Bases de données parallèles
- MapReduce

# Abstract

During the last years, the volume of data that is captured and generated has exploded. Advances in computer technologies, which provide cheap storage and increased computing capabilities, have allowed organizations to perform complex analysis on this data and to extract valuable knowledge from it. This trend has been very important not only for industry, but has also had a significant impact on science, where enhanced instruments and more complex simulations call for an efficient management of huge quantities of data.

Parallel computing is a fundamental technique in the management of large quantities of data as it leverages on the concurrent utilization of multiple computing resources. To take advantage of parallel computing, we need efficient data partitioning techniques which are in charge of dividing the whole data and assigning the partitions to the processing nodes. Data partitioning is a complex problem, as it has to consider different and often contradicting issues, such as data locality, load balancing and maximizing parallelism.

In this thesis, we study the problem of data partitioning, particularly in scientific parallel databases that are continuously growing and in the MapReduce framework.

In the case of scientific databases, we consider data partitioning in very large databases in which new data is appended continuously to the database, e.g. astronomical applications. Existing approaches are limited since the complexity of the workload and continuous appends restrict the applicability of traditional approaches. We propose two partitioning algorithms that dynamically partition new data elements by a technique based on data affinity. Our algorithms enable us to obtain very good data partitions in a low execution time compared to traditional approaches.

We also study how to improve the performance of MapReduce framework using data partitioning techniques. In particular, we are interested in efficient data partitioning of the input datasets to reduce the amount of data that has to be transferred in the shuffle phase. We design and implement a strategy which, by capturing the relationships between input tuples and intermediate keys, obtains an efficient partitioning that can be used to reduce significantly the MapReduce's communication overhead.

## Title in English

*Data Partitioning in Parallel Data Management Systems*

## Keywords

- Data partitioning
- Parallel Systems
- Parallel Databases
- MapReduce

## Equipe de Recherche

Zenith Team, INRIA & LIRMM

## Laboratoire

LIRMM - Laboratoire d'Informatique, Robotique et Micro-électronique de Montpellier

## Adresse

Université Montpellier 2
UMR 5506 - LIRMM
CC477
161 rue Ada
34095 Montpellier Cedex 5 - France

# Résumé étendu

## Introduction

Les quantités de données qui sont captées ou générées par des dispositifs informatiques modernes ont augmenté de façon exponentielle au cours de ces dernières années. Des nombreuses nouvelles sources de données ont été développées et sont devenues omniprésentes. Il s'agit notamment des réseaux sociaux (et de l'Internet en général), des web logs, des capteurs, des équiêments GPS, des systèmes de commerce électroniques, des instruments scientifiques, etc. Le progrès dans la capacité des systèmes informatiques, qui fournissent du stockage à bas prix et une très forte puissance de calcul, conduit les organisations à garder toutes leurs données, même anciennes, et d'effectuer des analyses complexes afin d'en extraire des connaissances, qui peuvent être exploitées comme aide dans le processus de prise de décisions.

*Big Data* est l'expression utilisée pour désigner les données qui, en raison de leur taille et de leur complexité, sont difficiles à manipuler avec des outils traditionnels de traitement et de gestion de données. Le big data se caractérise par trois dimensions (aussi appelés les trois Vs) : 1) le volume, qui fait référence aux très grandes quantités de données générées, 2) la vitesse, qui représente les taux élevés avec lesquels les flux de données rentrent dans les organisations, et enfin 3) la variété, qui se réfère aux nombreux types et formats de données différents dans lesquels les données sont produites.

L'un des domaines où le besoin de nouvelles technologies de gestion de données extrêmement volumineuses est devenu essentiel est la science. Les améliorations dans la précision des instruments d'observation et l'accroissement de la complexité des modèles de simulation ont multiplié les quantités de données qui doivent être analysées. En outre, les données scientifiques sont complexes et présentent une grande variété de types et formats, y compris des données multidimensionnelles, des graphes, des séquences, etc., qui posent des défis supplémentaires dans leur gestion. Des exemples de projets scientifiques qui entrent dans cette catégorie sont l'expérience ATLAS, une expérience de physique des particules réalisée à l'accélérateur de particules ou Large Hadron Collider (LHC) du CERN et qui produit 1 Po de données chaque année ; ou les catalogues astronomiques qui sont générés à partir d'observations régulières du ciel par les modernes télescopes optiques. Le

Sloan Digital Sky Survey (SDSS), qui a débuté en 2000, a été le premier effort de ce type et a déjà recueilli aujourd'hui plus de 140 To de données. Les projets à venir, comme le Dark Energy Survey (DES) ou le Grand Synoptic Survey Telescope (LSST) vont produire beaucoup plus de données, ce qui va augmenter encore la pression sur leurs systèmes de gestion des données.

Lorsqu'on traite de grandes quantités de données, le calcul parallèle est l'une des techniques fondamentales à utiliser, car il permet de tirer parti de l'utilisation simultanée de plusieurs ressources informatiques afin d'accélérer (speed-up) les calculs ou de maintenir les temps de réponse du système (scale-up), lorsque la taille des données d'entrée et la charge de travail augmentent. Inévitablement, le calcul massivement parallèle est une solution importante à la fois dans l'industrie et dans la recherche lors de l'élaboration de nouvelles techniques pour le traitement des données à grande échelle. Par exemple, le framework MapReduce, qui offre la distribution automatique des donnés, la parallélisation et la tolérance aux pannes d'une manière transparente, est devenu l'une des standards en matière d'analyse de données à grande échelle.

## Le problème du partitionement des données

Le calcul parallèle exige que chaque nœud qui participe au traitement obtienne une partie du travail. Ceci est réalisé par une approche en deux étapes : le partitionnement des données (ou fragmentation) pour diviser l'ensemble de données, et le placement des données (ou la répartition) pour assigner des fragments à des nœuds du système.

Si les données d'entrée sont initialement stockées hors du système parallèle, elles doivent être divisées et transférées à chacun des nœuds participants. Cependant, dans de nombreux cas, les données sont déjà stockées dans les mêmes nœuds qui exécutent le programme afin d'éviter la surcharge de transferts des données, appliquant ainsi le principe qui dit que « déplacer le calcul est moins cher que déplacer les données », ce qui maximise la localité de référence. L'exemple majeur de ce principe concerne les bases de données parallèles [97], où chaque nœud est affecté à une partie des données et un processeur de requêtes détermine les nœuds qui participent au traitement d'une requête. Un autre exemple concerne les systèmes de fichiers distribués où les différents fichiers et même les différents fragments dans lesquels un fichier est divisé sont répartis sur tous les nœuds. Quand un calcul est soumis, le système essaie d'exécuter les unités de travail dans les mêmes nœuds qui stockent leurs données d'entrée.

Un problème important lié à la façon dont les données sont partitionnées et attribuées aux nœuds est la répartition de charge, car si les nœuds sont affectés à différentes charges de travail, les avantages du calcul parallèle diminuent parce que les ressources peuvent être sous-utilisées, et la durée totale d'exécution devient celle du nœud le plus lent. L'un des défis liés à ce problème est que,

dans de nombreux cas, la localité des données et la répartition de charge sont contradictoires et un compromis doit être trouvé.

Il y a beaucoup de stratégies différentes pour partitionner et allouer un ensemble de données et même l'objectif d'optimisation peut changer. Le choix d'une approche particulière dépendrait de nombreux facteurs tels que les caractéristiques des programmes, l'architecture du système, les capacités du réseau, etc.

Dans cette thèse, nous étudions le problème de partitionnement de données dans les systèmes parallèles dans deux contextes différents : (1) les bases de données scientifiques qui augmentent de façon continue avec l'arrivée de nouvelles données et (2) le framework MapReduce. Dans les deux cas, nous proposons des approches automatiques, qui sont accomplies de manière transparente pour les utilisateurs, afin de les libérer du problème de partitionnement complexe.

Par rapport à (1), nous considérons le partitionnement de très grandes bases de données dans lesquelles des nouveaux éléments sont ajoutés en permanence. Ainsi, le développement de stratégies de partitionnement de données efficaces est l'une des principales exigences pour obtenir de bons résultats. En particulier, ce problème est plus difficile dans le cas de certaines bases de données scientifiques, tels que les catalogues astronomiques. La complexité du schéma limite l'applicabilité des approches automatiques traditionnelles basées sur les techniques de partitionnement basiques. Le haut dynamisme rend l'utilisation d'approches basées sur les graphes impraticable, car ils nécessitent de considérer l'ensemble des données afin de trouver un bon schéma de partitionnement. En conséquence, nous avons besoin d'une approche qui soit capable de capturer les relations entre les données, comme dans le partitionnement basé sur les graphes, mais d'une manière dynamique, c'est-à-dire, en ne considérant que les éléments qui sont ajoutés à chaque fois.

Par rapport à (2), nous étudions comment le partitionnement des données affecte la performance de travaux (*jobs*) MapReduce. Un travail MapReduce est détaillé par deux fonctions définies par l'utilisateur, appelées map et reduce, qui consomment et produisent des paires clé-valeur. Ces fonctions sont exécutées en parallèle par plusieurs tâches qui sont responsables du traitement d'un fragment des données. Les tâches map consomment des paires clé- valeur d'entrée et produisent des paires intermédiaires. Celles-ci sont alors triées et regroupées par clés, puis livrées aux tâches reduce. Le framework MapReduce est responsable du partitionnement, du tri et du transfert des paires des tâches map aux tâches reduce. Ce processus est connu sous le nom de la phase *shuffle*. La réduction de la quantité de données qui sont transférées dans cette phase est très importante pour les performances, car elle augmente la localité des données dans les tâches reduce, et diminue ainsi la surcharge des exécutions. Dans la littérature, plusieurs optimisations ont été proposées avec l'objet de réduire le transfert de données entre les tâches map et reduce. Néanmoins, toutes ces approches sont limitées par la façon dont les paires clé-valeur intermédiaires sont réparties sur les sorties de la

phase map. Ce problème peut être résolu si les données d'entrée sont divisées de sorte que les paires intermédiaires qui partagent la même clé sont produites par la même tâche map.

# État de l'art

## Parallélisme

Le calcul parallèle dénote l'utilisation simultanée de plusieurs ressources de calcul afin d'effectuer un calcul. La tâche d'origine est divisée en multiples petites sous-tâches, qui sont assignées aux ressources informatiques dans le système et exécutées en parallèle. Les ressources informatiques peuvent être processeurs ou cœurs dans une seule machine multiprocesseur/multi-cœur ou représenter ordinateurs autonomes reliées par un réseau dans un cluster ou une grille. L'objectif principal d'un système parallèle est l'amélioration du temps de réponse des programmes. Cette amélioration peut être mesurée par le *speed-up*, qui quantifie combien plus rapide est le système parallèle par rapport à une exécution séquentielle du programme. Idéalement, l'accélération doit être linéal au nombre de ressources informatiques utilisées ; cependant, dans des nombreux programmes il y a certaines parties qui ne peuvent pas être exécutées en parallèle ou doit attendre à la finalisation d'autres parties, par exemple, à cause de dépendances. En conséquence, il y a un point à partir duquel l'ajout de nouvelles ressources dans le système n'augmente pas sa performance.

Une autre façon de mesurer l'amélioration de la performance du calcul parallèle est le *scale-up*, qui est utilisé pour déterminer la capacité d'un système de maintenir la performance quand les ressources et la taille des données et la charge de travail sont augmentés proportionnellement. Cette mesure quantifie le scalabilité du système. Idéalement, le temps d'exécution resterait constant quand le nombre de ressources et la taille des données et la charge de travail augmente à la même proportion.

Les techniques les plus importants utilisés dans le calcul parallèle sont les suivantes :

- **Partitionnement de données et allocation :** Quand un programme est exécuté en plusieurs ressources de calcul, chacun d'entre eux doit être alloué une partie différente du travail. Certains calculs simplement assignent différents paramètres pour chacun des nœuds, qui génèrent l'entrée par eux-mêmes. Mais dans des nombreux cas, l'entrée se compose d'un grand ensemble de données qui doit être divisé et attribué aux nœuds. Ceci est réalisé dans une procédure en deux étapes : le partitionnement de données ou fragmentation (division de l'ensemble de données) et le placement ou allocation (attribution des fragments aux nœuds du système). Le partitionnement des données et l'allocation sont spécialement importantes dans les

architectures shared-nothing, où chaque nœud dispose des disques durs attachés qui sont accessibles beaucoup plus rapide que les disques attachés à d'autres nœuds. Dans des nombreux cas, les données sont stockées dans les mêmes ordinateurs qui exécutent le calcul afin d'éviter la surcharge de transferts des données maximisent ce qui est connu comme *localité des données*.

- **Répartition de charge :** Dans un système parallèle, il est fondamental que les nœuds reçoivent la même quantité de travail (ou un travail proportionnel à leur puissance de calcul, si elle est différente), afin d'éviter la surcharge de certains d'entre eux tandis que d'autres restent inactifs. Ceci est connu comme la répartition de charge et a un impact significatif sur l'utilisation des ressources, le débit et le temps de réponse. La répartition de charge est étroitement liée avec le partitionnement des données et l'allocation, même si elle se rapporte également à l'ordonnancement (*scheduling*), et fréquemment entre en conflit avec la localité des données.
- **Replication :** Il consiste à stocker plusieurs copies des mêmes données dans différents nœuds. Il est utilisé dans les systèmes parallèles à la fois pour améliorer la disponibilité et la performance. La disponibilité est augmentée car même si certains des nœuds tombe en panne, les données sont toujours accessibles en récupérant les exemplaires conservés dans d'autres nœuds. La performance peut être améliorée par : 1) l'exécution de programmes qui accèdent aux mêmes données dans différents nœuds, augmentant ainsi le parallélisme ; 2) l'amélioration de la répartition de charge en répliquant les *hotspots* à différents nœuds, et 3) l'augmentation de la localité des données.
- **Tolérance aux pannes :** Deux concepts sont au cœur de la tolérance aux pannes : la fiabilité et la disponibilité. La *fiabilité* est la capacité d'un système à fonctionner sans faute pendant une période de temps, considèrent une faute comme un écart de la spécification du système. Il est normalement utilisé pour les éléments qui ne peuvent pas être réparés. La *disponibilité* est la fraction de temps qu'un système est opérationnel. Il se réfère toujours à des systèmes qui peuvent être réparés et mesure leur capacité à tolérer les pannes de leurs éléments.
  Un système parallèle peut être construit de telle sorte que les pannes individuelles de certains de ses éléments ne rendent pas le système indisponible. La principale technique utilisée pour obtenir la tolérance aux pannes est la *redondance*. La capacité de faire face aux pannes comporte trois capacités : la détection des pannes lorsqu'elles se produisissent ; la redirection des requêtes affectées du nœud en panne à un autre de manière transparente (*failover*) et la réintroduction des répliques en panne dans le système quand ils sont à nouveau disponibles (*récupération*).

## Partitionement dans les bases de données

Un système de base de données parallèle [97] combine les techniques de gestion de bases de données et les techniques de traitement des données en parallèle afin de stocker de gros volumes de données et fournir des niveaux acceptables de performance et de disponibilité. Le partitionnement de données est utilisé pour diviser les éléments de base de données et les allouer aux nœuds qui participent au traitement des requêtes. Le parallélisme peut être obtenu par l'exécution simultanée de plusieurs requêtes et/ou par l'exécution en parallèle d'une seule requête sur plusieurs nœuds. Selon l'application, l'objectif du partitionnement est différent. Par exemple, lorsque de petites requêtes sont exécutées, le partitionnement essaye de délimiter leur exécution dans un ou quelques nœuds afin d'éviter la surcharge liée à l'exécution distribuée. D'autre part, lorsque l'application se compose de longues requêtes, l'objectif du partitionnement est de répandre les éléments accédés par les requêtes dans tous les nœuds d'une manière équilibrée afin que le temps de réponse total soit minimisé. Dans le premier cas, des techniques de *clustering* sont utilisées, en essayant de placer des objets qui sont fréquemment consultés conjointement dans les mêmes partitions. Dans le second cas, l'objectif est à l'opposé, et le terme utilisé fréquemment est *declustering*.

Considérant une base de données relationnelle (avec des relations contenant des tuples), les relations peuvent être partitionnées de deux façons : horizontalement et verticalement. Dans le premier cas, chaque fragment est attribué un sous-ensemble des tuples. Dans la seconde, les attributs de la relation sont divisés en plusieurs groupes. Chaque fragment contient les attributs composant la clé primaire ainsi que les attributs de l'un des groupes. Le partitionnement vertical est généralement utilisé pour la conception physique de base de données. Le partitionnement horizontal, d'autre part, est utilisé dans tous les cas et est la principale forme de partitionnement utilisée pour diviser l'ensemble des données dans une base de données parallèle.

Trois techniques de base ont été largement utilisées dans la littérature pour partitionner les relations et affecter les tuples aux nœuds dans une base de données parallèle : *partitionnement round-robin*, qui attribue les tuples aux nœuds de façon séquentielle ; le *partitionnement par hachage*, qui applique une fonction de hachage sur les attributs de partitionnement pour obtenir le nœud de destination ; et le *partitionnement par intervalle*, qui définit des intervalles de valeurs des attributs de partitionnement et les attribue aux différents nœuds.

Les techniques de base fonctionnent bien dans des nombreux cas et sont encore utilisées dans des nombreuses applications, mais dès que la charge de travail devient plus complexe, la conception d'un partitionnement efficace devient plus difficile, car beaucoup de possibilités doivent être envisagées. Plusieurs outils de partitionnement ont été conçus afin de partitionner automatiquement la base de données de la manière la plus efficace possible, étant donné un modèle de charge

de travail [34]. Les choix sont basés sur les techniques de partitionnement de base, principalement par hachage et par intervalle, et sélectionnent les attributs de partitionnement qui sont les plus adéquats.

Les techniques de partitionnement basées sur des graphes [41, 90, 84] représentent une autre alternative pour le partitionnement automatique des données qui peut fonctionner avec n'importe quel schéma et quelle que soit la complexité des requêtes de la charge de travail, car seules les relations entre les éléments sont utilisées. La charge de travail est modélisée sous forme d'un graphe, où les sommets représentent les éléments et les arêtes sont pondérées en fonction de la similitude entre deux données. La similitude est calculée par l'addition de la fréquence totale des requêtes qui accèdent les deux éléments à la fois. Un autre modèle est également utilisé dans lequel la charge de travail est représentée comme un hypergraphe, où les arêtes peuvent connecter plus de deux sommets en même temps. Dans ce cas, chaque requête génère une arête reliant tous les éléments accédés, par opposition au modèle de graphe, où une requête accédant à $n$ éléments génère $n(n-1)$ arêtes.

## MapReduce

MapReduce désigne à la fois le modèle de programmation et le framework initialement développé par Google [43] pour le traitement parallèle des données à grande échelle. Les utilisateurs n'ont qu'à fournir deux fonctions, appelées map et reduce, et le système gère toutes les questions liées à la parallélisation, la tolérance aux pannes, la distribution des données et la répartition de charge.

Les fonctions map et reduce sont définies sur des paires clé-valeur. La fonction map consomme des paires clé-valeur d'entrée et produit une liste (éventuellement vide) de paires clé-valeur intermédiaires. Puis, le framework regroupe les paires intermédiaires par clé et délivre chaque groupe (la clé et toutes les valeurs associées) à la fonction reduce, qui produit à son tour une liste (éventuellement vide) de paires clé- valeur de sortie.

Le framework MapReduce exécute des programmes en parallèle dans un cluster shared-nothing. Il y a deux types de processus, les *workers*, qui exécutent les tâches map et reduce, et le *master*, qui est en charge de contrôler l'exécution des workers. Habituellement, les données d'entrée et de sortie sont stockées dans un système de fichiers distribué, par exemple Google File System [60], qui s'exécute dans les mêmes nœuds où les travaux MapReduce sont exécutés.

Dans un travail MapReduce, l'entrée est divisée en $M$ parties (*splits*), qui sont consommées par $M$ tâches map, une par partie. La sortie des tâches map est partitionnée selon la clé intermédiaire dans $R$ fragments en utilisant une fonction de partitionnement, par défaut ($hash(k_2) \mod R$), qui sont ensuite traités par $R$ tâches reduce.

Quand un travail est lancé, le master partitionne l'entrée en $M$ fragments. Les

tâches map et reduce sont alors attribuées aux travailleurs (workers) dès qu'ils deviennent inoccupés, premièrement les tâches map, puis les tâches reduce, une fois que toutes les tâches map sont finies. La sortie des tâches map est découpée en $R$ fragments selon la clé intermédiaire et stockée sur les disques durs locaux des workers. Les tâches reduce prennent ces sorties et les trient par la clé de sorte que toutes les valeurs d'une clé intermédiaire donnée sont traitées conjointement par la fonction reduce. Une fois que toutes les tâches map et reduce ont terminé, l'utilisateur est averti. La phase intermédiaire d'un travail MapReduce, lorsque les clés intermédiaires sont partitionnées, triées et transférées vers les nœuds qui exécutent les tâches reduce est connu comme le *shuffle*.

MapReduce est utilisé en combinaison avec un système de fichiers distribué, de sorte que les données sont déjà stockées dans les mêmes nœuds qui effectuent le calcul. Dans Google, le Google File System (GFS) [60] est employé. Par défaut, il divise automatiquement les fichiers en blocs de 64 Mo, qui sont ensuite répliquées (en général 3 fois) et stockés sur des machines différentes.

Dans la version MapReduce de Google, une correspondance entre les blocs de fichiers dans GFS et les splits dans l'entré des tâches map est établie. Le master essaie de planifier des tâches map dans les mêmes machines stockant le bloc correspondant. Si ce n'est pas possible, une réplique proche est choisie (par exemple, dans le même *rack*). L'objectif est d'économiser la bande passante du cluster, car la plupart des données d'entrée sont lues localement sans transfert de réseau.

Néanmoins, lors de la planification des tâches reduce, la localité des données n'est pas du tout prise en compte. En conséquence, selon le volume de données produites dans les tâches map et les caractéristiques du réseau (la bande passante, la topologie du réseau), le shuffle peut prendre un temps considérable pour finir [127, 98].

Il y a eu quelques travaux qui ont essayé d'améliorer la localité des données et de réduire la surcharge produit par les transferts dans le shuffle. Dans [111], un système dit de *pre-shuffling* est proposé afin de réduire les transferts de données dans le shuffle. Un ordonnanceur modifié vérifie les fragments d'entrée avant que la phase de map commence et prédit le reducer où les paires clé-valeur sont réparties. Ensuite, les données sont affectées à une tâche map près du reducer attendu. De même, dans [67], les tâches reduce sont assignées aux nœuds qui minimisent les transferts sur le réseau entre les nœuds et les racks. Toutefois, dans ce cas, la décision est prise en temps de programmation de tâches reduce. La limitation des ces approches est que, même si toutes les paires intermédiaires sont produites dans le même nœud, la fonction de partitionnement peut les forcer à être séparés en plusieurs tâches reduce, réduisant ainsi les possibilités de programmation basées sur la localité des données. Dans [74], ce problème est résolu en attribuant les clés intermédiaires des reducers au moment de la programmation. Le nombre de tâches reduce doit être égal au nombre de nœuds. Quand toutes les

paires intermédiaires ont été produites, l'algorithme d'ordonnancement attribue les paires intermédiaires aux tâches reduce avec un algorithme glouton. Cet algorithme intègre à la fois la localité des données et la répartition de charge, de sorte que la localité des données ne produise pas de tâches reduce avec beaucoup plus de travail. Cependant, il dépend encore de la distribution des clés intermédiaires dans les sorties des tâches map. Si les paires intermédiaires avec la même clé sont uniformément produites dans tous les travailleurs, les gains possibles de cette approche ne sont pas significatifs.

## Partitionnement dynamique pour des bases de données en croissance continue

Dans beaucoup de domaines scientifiques les analyses des données nécessaires pour obtenir des enseignements impliquent la gestion et le traitement d'énormes quantités d'informations. Ceci est le résultat d'une explosion dans la quantité de données produites par les nouveaux instruments scientifiques beaucoup plus précises et des modèles de simulation plus complexes. L'un de ces domaines est l'astronomie, où les télescopes modernes équipés de caméras très puissantes génèrent de grandes quantités de données provenant des observations régulières du ciel. Nous nous concentrons sur le Dark Energy Survey (DES), un projet auquel nous avons collaboré avec nos collègues du LNCC (Rio de Janeiro), dont le but est d'aider à la découverte de la nature de l'énergie sombre. Un catalogue composé de grandes tables avec des milliards de tuples et des centaines d'attributs (correspondant aux dimensions, principalement des nombres réels en double précision) stocke les objets découverts et est continuellement augmenté avec des nouvelles observations effectuées. Les scientifiques du monde entier peuvent accéder à la base de données avec des requêtes qui peuvent contenir un nombre considérable d'attributs.

Le volume des données que ces applications contiennent pose des défis importants pour la gestion de données. En particulier, des solutions efficaces sont nécessaires pour partitionner et distribuer les données dans plusieurs serveurs, notamment un cluster, afin d'optimiser l'exécution des requêtes, spécialement les petites requêtes accédant à une petite partie de l'ensemble de données. Une stratégie de partitionnement efficace aurait comme objectif la minimisation du nombre de fragments qui sont accédés dans l'exécution d'une requête, réduisant ainsi les surcharges liées à l'exécution distribuée. Les approches traditionnelles de partitionnement horizontal, comme le partitionnement par hachage ou par intervalle, sont incapables de saisir les schémas d'accès complexes présents dans les applications de calcul scientifique, en particulier parce que ces applications utilisent généralement des relations compliquées, y compris des opérations mathématiques, sur un grand ensemble de colonnes, et sont difficiles à être prédéfinis a priori. Les

approches basées sur le partitionnement de graphes peuvent bien fonctionner avec ces requêtes complexes, car elles peuvent travailler avec n'importe quel schéma et quelle que soit la complexité des requêtes de la charge de travail. Cependant, elles ont besoin de faire un recalcul complet du partitionnement lorsque l'ensemble de données change, ce qui peut devenir prohibitif dans notre scénario. En outre, le modèle basé sur des graphes ne tient pas compte du placement des données précédentes, et beaucoup de transferts de données peuvent avoir lieu afin d'appliquer le nouveau partitionnement.

Dans cette partie, nous nous intéressons au partitionnement dynamique de grandes bases de données qui sont agrandies continuellement. Après avoir modélisé le problème de partitionnement des données en ensembles de données dynamiques, nous proposons deux algorithmes dynamiques basés sur la charge de travail, appelés *DynPart* et *DynPartGroup*, qui adaptent efficacement le partitionnement à l'arrivée de nouveaux éléments de données. Ces algorithmes sont conçus en se basant sur une heuristique que nous avons développée en tenant compte de l'affinité entre les nouvelles données et les requêtes et les fragments. Contrairement aux algorithmes basés sur la charge de travail statique, le temps d'exécution de nos algorithmes ne dépend pas de la taille totale de la base de données, mais seulement de celle des nouvelles données, ce qui les rend appropriés pour les bases de données en croissance constante.

Compte tenu d'un partitionnement $\pi(D)$, on définit une mesure de *l'efficacité du partitionnement* comme le rapport entre le nombre de fragments qu'une requête doit accéder ($rel(q, \pi(D))$) et le nombre minimum de fragments qui pourraient être employées pour y répondre ($minfr(q, \pi(D))$) :

$$eff(q, \pi(D)) = \frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|}$$

À partir de cette mesure, nous générons une heuristique, que nous appelons *affinité*, qui est défini entre les nouveaux éléments de données qui arrivent et les fragments qui forment le partitionnement :

$$aff(d, F) = - \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q) \frac{minfr(q, \pi(D))}{|rel(q, \pi(D))| \, (|rel(q, \pi(D))| + 1)}$$

où $comp(q, d)$ exprime que $d$ est compatible avec la requête $q$ et $f(q)$ représente sa fréquence dans la charge de travail.

Nous proposons une première version de l'algorithme, appelé *DynPart*, qui traite les nouveaux éléments de données un par un et sélectionne le fragment ayant une affinité maximale comme destination. Des contraintes de déséquilibre (soit sur la taille des fragments ou sur leur charge) sont appliquées quand les allocations sans déterminées.

Une deuxième version de l'algorithme, appelé *DynPartGroup* est également

proposée. Dans ce cas, une étape de regroupement préliminaire est effectuée sur les nouveaux éléments. Les éléments équivalents (appartenant à la réponse des mêmes requêtes) sont regroupés et attribués comme une seule unité à l'un des fragments. Cette stratégie réduit le nombre de calculs d'affinité et évite la scission des groupes en raison du déséquilibre, un scénario qui dégraderait l'efficacité du partitionnement.

Nous avons validé notre approche en implémentant les algorithmes de partitionnement et les exécutant sur des données réelles provenant du catalogue Sloan Sky Digital Survey (SDSS), car les données du DES ne sont pas encore disponibles. Nous avons utilisé les requêtes SDSS SkyServer SQL query log comme charge de travail de l'application. Nous avons comparé le temps d'exécution de nos algorithmes à celle d'une approche statique basée sur le partitionnement de graphes. Les résultats montrent que, dès la taille de la base de données augmente, le temps d'exécution de l'algorithme statique augmente de manière significative, par contre, celle de nos algorithmes reste stable. Ils présentent en outre l'avantage que nos algorithmes, bien que basés sur une approche heuristique, ne dégradent pas considérablement l'efficacité du partitionnement. Les expériences dévoilent que dans le cas d'ensembles de données dans lesquels il y a une forte corrélation entre les nouveaux éléments de données, l'algorithme *DynPartGroup* maintient un très bon comportement. Ils indiquent également que cet algorithme n'est pas très affecté par le déséquilibre dans la taille ou la charge des fragments.

## Partitionnement des données pour minimiser les transferts de données dans MapReduce

MapReduce [43] s'est imposé comme l'une des alternatives les plus populaires pour le traitement des grands ensembles de données grâce à la simplicité de son modèle de programmation et sa gestion automatique de l'exécution en parallèle dans les clusters. Il divise le calcul en deux phases principales, à savoir, le map et reduce, qui à leur tour sont effectués par plusieurs tâches qui traitent les données en parallèle. Entre les deux, il y a une phase intermédiaire, appelée shuffle, où les données produites par la phase map sont ordonnées, regroupées et transférées vers les machines en charge de l'exécution de la phase reduce.

MapReduce applique le principe qui stipule que « déplacer le calcul est moins cher que déplacer les données » et essaie ainsi de planifier les tâches map dans des machines proches des données d'entrée qu'ils traitent, afin de maximiser la localité des données. La localité des données est souhaitable, car elle réduit la quantité de données transférées via le réseau, ce qui réduit la consommation d'énergie ainsi que le trafic réseau des centres de données. Cependant, dans l'ordonnancement des tâches reduce, la localité des données n'est pas du tout prise en compte. Certaines travaux [127, 98] ont démontré que les transferts à travers le réseau peuvent

poser une surcharge considérable dans l'exécution des travaux MapReduce. En conséquence, plusieurs optimisations ont été proposées afin de réduire les transferts de données entre les mappers et les reducers. Nous avons couvert certaines des propositions, qui vont de la planification intelligente des tâches reduce [111, 67] à l'attribution dynamique des clés intermédiaires aux tâches reduce au moment de la planification [74]. Néanmoins, toutes ces approches sont limitées par la façon dont les paires clé-valeur intermédiaires sont réparties sur les sorties de la phase map. Si les données associées à une clé intermédiaire donnée sont présentes dans toutes les sorties des tâches map, les paires dans tout l'ensemble des nœuds sauf un doivent encore être transférées à travers le réseau.

Dans cette thèse, nous proposons une technique, appelée *MR-part*, qui vise à minimiser les données transférées entre les mappers et les reducers dans la phase de shuffle de MapReduce. Pour ce faire, il rend compte des relations entre les tuples d'entrée et les clés intermédiaires en surveillant l'exécution d'un ensemble des travaux MapReduce qui sont représentatifs de la charge de travail. Ensuite, en fonction des relations capturées, il partitionne les fichiers d'entrée et attribue les tuples d'entrée aux fragments appropriés de telle manière que les travaux MapReduce ultérieurs tireront pleinement parti de la localité des données dans la phase reduce. Afin de caractériser la charge de travail, nous injectons une composante de surveillance (*monitoring*) dans le famework MapReduce qui produit les métadonnées requises. Ensuite, un autre élément, qui est exécuté hors ligne, combine les informations saisies pour tous les travaux MapReduce de la charge de travail et partitionne les données d'entrée en conséquence. Nous avons modélisé la charge de travail en utilisant un hypergraphe, auquel on applique un algorithme de coupe minimal des graphes pour attribuer les tuples aux fragments d'entrée.

Nous avons implémenté un prototype de *MR-part* et l'avons intégré dans Hadoop. Le composant du monitoring comme celui d'ordonnancement sont incorporés à Hadoop, alors que les composants de repartitionnement sont exécutés hors ligne et en dehors du framework Hadoop. Chaque fois que possible, nous avons défini une interface générale et fourni une ou plusieurs implémentations. Cela permet de tester plusieurs stratégies avec un minimum d'effort et rend l'entretien et l'extension du prototype simple. De cette façon, des stratégies alternatives peuvent être envisagées pour le suivi de la charge de travail, mais encore plus utile est la possibilité de définir des algorithmes d'ordonnancement spécifiques. Les nouvelles implémentations de surveillance ou de planification peuvent être encapsulées dans un fichier `jar` et chargés dynamiquement sans avoir à modifier le code du framework Hadoop.

Nous avons évalué notre prototype dans Grid5000, une infrastructure à grande échelle composée de différents sites avec plusieurs clusters. Nous avons utilisé le benchmark TPC-H et exécuté ses requêtes sur Hadoop native, la version modifiée de Hadoop incorporant l'attribution dynamique des clés intermédiaires [74] et *MR-Part*. Les résultats montrent que, sans le repartitionnement des données

d'entrée, peu de gain est obtenu en ce qui concerne le volume de données transférées dans la phase shuffle. Cependant, *MR-part* réduit les transferts de données à moins de 10% du total pour différents types de requêtes et différentes tailles du cluster. Cette réduction a un impact significatif sur le temps total d'exécution lorsque la bande passante du réseau est limitée, compte tenu que le temps de shuffle est considérablement réduit.

# Conclusions

Cette thèse s'inscrit dans le contexte des applications big data, en utilisant le parallélisme à grande échelle pour le traitement et la gestion efficace de grands ensembles de données. En particulier, nous nous sommes concentrés sur le problème de partitionnement des données, ce qui est fondamental pour obtenir un traitement parallèle et donc d'améliorer les performances des applications qui traitent de grands ensembles de données. Nous avons abordé deux problèmes en particulier :

- Nous avons abordé le problème de partitionnement automatique dans les grandes bases de données scientifiques où des nouveaux éléments sont insérés dès que de nouvelles mesures sont effectuées. Nous avons identifié les principales limitations des approches existantes, essentiellement : 1) l'inefficacité des approches automatiques basés sur les techniques de base pour gérer la complexité des applications scientifiques, et 2) le long temps d'exécution des algorithmes de partitionnement basés sur les graphes. Comme solution, nous avons proposé deux algorithmes, *DynPart* et *DynPartGroup* qui allouent dynamiquement les nouveaux éléments en fonction de l'affinité qu'ils ont avec les fragments actuels dans le partitionnement. Le temps d'exécution de nos algorithmes reste constant même si la taille de la base de données augmente, tandis que celle des approches basées sur des graphes augmente. Les résultats révèlent également que dans nos algorithmes, l'efficacité du partitionnement est préservée lorsque la base de données croît en taille.
- Nous avons étudié comment le partitionnement des données affecte la performance des travaux MapReduce et proposé une approche en vue de réduire la quantité de données transférées via le réseau dans la phase shuffle. Il est basée sur le partitionnement des données d'entrée, qui est effectué après une phase de surveillance (monitoring) où les relations entre les tuples d'entrée et les clés intermédiaires sont capturées. Cette répartition peut être utilisée pour réduire les transferts via le réseau dans la phase shuffle, ce qui se traduit par des réductions significatives sur le temps d'exécution lorsque la bande passante est limitée.

# Contents

# Chapter 1

# Introduction

The amount of data that is captured or generated by modern computing devices has augmented exponentially over the last years. Numerous novel input data sources have been developed and become omnipresent. These include social networks (and the Internet in general), web logs, sensors, GPS devices, trading systems, scientific instruments, etc. The enhanced capacity of computer systems, which provides vast storage and increased computing capabilities, has led organizations to keep their data and perform complex analysis in order to extract knowledge from it, allowing them to obtain insightful understanding of their business, which would, in turn, help them in the decision making process.

*Big Data* is the term that has been coined to refer to the data that, because of its size and complexity, is difficult to be handled by traditional data management and processing tools. It is commonly characterized by three dimensions (also called the *three Vs*): 1) *Volume*, which refers to the increased quantity of generated data; 2) *Velocity*, which denotes the high rate at which data flows into organizations; and 3) *Variety*, which refers to the many different data types and formats in which data is produced.

One of the domains where the need for new big data management technologies have become critical is science. The improvements in the precision of observational instruments and the increased complexity of simulation models has multiplied the quantity of data that has to be analyzed. Moreover, scientific data is complex and presents a wide variety of types and formats, including multidimensional data, graphs, sequences, etc., which pose additional challenges for data management. Examples of scientific projects that fall into this category include the ATLAS experiment, a particle physics experiment at CERN's Large Hadron Collider (LHC) that produces 1 petabyte of data each year; or the astronomical catalogs that are generated from regular observations of the sky by novel optical telescopes. The Sloan Digital Sky Survey (SDSS), which was started in 2000, has been the first effort of this type and has already collected more than 140 TB of data to the present. Future projects, such as the Dark Energy Survey (DES) or the Large Synoptic Survey Telescope (LSST) will produce much larger quantities

of data, thereby putting much pressure on their data management systems.

When dealing with large amounts of data, *parallel computing* is one of the fundamental techniques to be used, as it leverages the concurrent utilization of multiple computing resources in order to accelerate (speed-up) the computations or to maintain the response times of the system (scale-up) as the input data size increases. Indeed, massive parallel computing has been a major solution in both industry and research when developing new techniques for big data processing. For example, the MapReduce framework, which provides for automatic distribution, parallelization and fault-tolerance in a transparent way, has become one of the standards in big data analysis.

## 1.1 Data Partitioning Problem

For a program to be executed concurrently using several processing nodes in a parallel computer system (e.g. a cluster), the work has to be divided and assigned to each one of them. This usually requires a given input dataset to be divided and assigned to each of the nodes. This is carried out in a two-step process: data partitioning (or fragmentation) to divide the dataset, and data placement (or allocation) to assign the fragments to the system's nodes.

If the original input data is stored at a single node or outside the parallel system, then it has to be divided and transferred to each of the participating nodes. However, in many cases, the data is stored at the same nodes that execute the program in order to avoid the overhead of data transfers, thus applying the principle "moving computation is cheaper than moving data", which fosters locality of reference to the data. The paradigmatic example of this principle is *distributed and parallel databases*, where each node is assigned a partition of the database and a query processor determines the nodes that participate in the processing of a query. Another example is a distributed file system where different files and even different fragments of the same file are spread over all the nodes. When a program is executed, the system executes as much as possible each unit of work at the same node that stores its input.

An important issue related to how data is partitioned and assigned to the nodes is load balancing, since if nodes are assigned a different amount of work, the benefits of parallel computing decrease because the resources are under-utilized, and the response time becomes that of the slowest node. One of the challenges related to this problem is that in many cases data locality and load balancing are contradictory and a compromise has to be found.

There are many different strategies to partition and allocate a dataset and even the optimization goal may change. The choice of a particular approach depends on many factors such as the the characteristics of the programs, the architecture of the system, the network capabilities, etc.

In this thesis, we study the problem of data partitioning in parallel systems in two different contexts: (1) in scientific databases that are continuously growing and (2) in the MapReduce framework. In both cases, we propose automatic approaches, which are performed transparently to the users, in order to free them from the burden of complex partitioning.

Wrt (1), we consider applications with very large databases, where data items are continuously appended. Thus, the development of efficient data partitioning is one of the main requirements to yield good performance. In particular, this problem is harder in the case of some scientific databases, such as astronomical catalogs. The complexity of the schema limits the applicability of traditional automatic approaches based on the basic partitioning techniques. The high dynamicity makes the usage of graph-based approaches impractical, as they require to consider the whole dataset in order to come up with a good partitioning scheme. So, we need an approach that is able to capture the relationships between tuples, as in graph-based partitioning, but in a dynamic way, i.e., only considering the data elements that are appended at each time.

Wrt (2), we study how data partitioning affects the performance of MapReduce computations. A MapReduce job is specified by two user defined functions, called map and reduce, which consume and produce key-value pairs. These functions are executed in parallel by several tasks which are responsible for processing a fragment of the data. Map tasks consume input key-value pairs and produce intermediate pairs. These should be sorted and grouped by key and then delivered to the reduce tasks. The MapReduce framework is responsible of partitioning, sorting and transferring the pairs from map to reduce tasks. This process is known as the shuffle phase. Reducing data transfer in MapReduce's shuffle phase is very important for performance because it increases data locality of reduce tasks, and thus decreases the overhead of job executions. In the literature, several optimizations have been proposed to reduce data transfer between map and reduce tasks. Nevertheless, all these approaches are limited by how intermediate key-value pairs are distributed over map outputs. This issue can be solved if the input data is partitioned so that the intermediate pairs that share the same key are produced by the same map task.

## 1.2 Contributions

The main contributions of this thesis are:

**A dynamic workload-driven partitioning approach for continuously growing databases.** In this work, we propose *DynPart* and *DynPartGroup*, two dynamic partitioning algorithms for continuously growing databases. These algorithms efficiently adapt the data partitioning to the arrival of new data ele-

ments by taking into account the affinity of new data with queries and fragments. In contrast to existing static approaches, our approach offers constant execution time, no matter the size of the database, while obtaining very good partitioning efficiency. We validate our solution through experimentation over real-world data; the results show its effectiveness.

**An automatic repartitioning strategy that reduces significantly data transfer in MapReduce job execution.** In this work, we address the problem of high data transfers in MapReduce, and propose a technique that repartitions tuples of the input datasets, and thereby optimizes the distribution of key-values over mappers, and increases the data locality in reduce tasks. Our approach captures the relationships between input tuples and intermediate keys by monitoring the execution of a set of MapReduce jobs which are representative of the workload. Then, based on those relationships, it assigns input tuples to the appropriate chunks. With this data repartitioning and a smart scheduling of reducer tasks, our approach significantly contributes to the reduction of transferred data between mappers and reducers in job executions. We evaluate our approach through experimentation in a Hadoop deployment on top of Grid5000 using standard benchmarks. The results show high reduction in data transfer during the shuffle phase compared to Native Hadoop.

## 1.3 Publications

- M. Liroz-Gistau, R. Akbarinia, E. Pacitti, F. Porto, and P. Valduriez, « Dynamic workload-based partitioning algorithms for continuously growing databases », *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS Journal)*, Lecture Notes in Computer Science, vol. 8320, 2013, To appear

- M. Liroz-Gistau, R. Akbarinia, E. Pacitti, F. Porto, and P. Valduriez, « Dynamic workload based partitioning for large-scale databases », in *Database and Expert Systems Applications (DEXA 2012)*, ser. Lecture Notes in Computer Science, vol. 7447, 2012, pp. 183–190

- M. Liroz-Gistau, R. Akbarinia, D. Agrawal, E. Pacitti, and P. Valduriez, « Data partitioning for minimizing transferred data in MapReduce », in *Data Management in Cloud, Grid and P2P Systems (Globe 2013)*, ser. Lecture Notes in Computer Science, vol. 8059, 2013, pp. 1–12

- M. Liroz-Gistau, R. Akbarinia, E. Pacitti, F. Porto, and P. Valduriez, « Dynpart: dynamic partitioning for large-scale databases », in *28èmes Journées Bases de Données Avancées (BDA 2012)*, Clermont-Ferrand, Oct.

2012

- M. Liroz-Gistau, R. Akbarinia, D. Agrawal, E. Pacitti, and P. Valduriez, « MR-Part : minimizing data transfers between mappers and reducers in MapReduce », in *29èmes Journées Bases de Données Avancées (BDA 2013)*, Nantes, Oct. 2013

## 1.4 Organization of the Thesis

The rest of the thesis is organized as follows.

**Chapter 2: State of the Art.** In this chapter, we review the state of the art. It is divided in three sections: In Section 2.1, we give a general overview of parallel computing and introduce the main architectures, computing paradigms and techniques. Section 2.2 focuses on data partitioning in parallel databases, which is the context of the work presented in Chapter 3. Finally, Section 2.3 describes the MapReduce framework, which is used in Chapters 4 and 5 and presents its limitations and the most important solutions to overcome them.

**Chapter 3: DynPart and DynPartGroup.** In this chapter, we deal with the problem of automatic workload-based partitioning in large databases where the data is continuously appended. We formally define the problem in Section 3.2 and propose two algorithms: *DynPart*, which is specified in Section 3.3, and an extension called *DynPartGroup*, which is described in Section 3.4. We evaluate the algorithms experimentally and compare them with a static graph-based partitioning approach in Section 3.5. Finally, we compare our proposal with related work and discuss possible extensions.

**Chapter 4: MR-Part.** This chapter addresses the problem of data transfers in MapReduce shuffle's phase and proposes a new strategy, called *MRPart*, in order to minimize the amount of data that is transferred through the network. Section 4.2 formalizes MapReduce job execution and presents the problem definition. Then, Section 4.3 explains the strategy and depicts the designed algorithms. These algorithms are assessed and compared to native Hadoop and previous reduce locality-aware approaches by experimental evaluation. The results show a significant reduction in the amount of data transferred in the shuffle phase, which turns into a reduction of the response time, specially when the network bandwidth is limited.

**Chapter 5: MR-Part Prototype.** In this chapter, we describe our implementation of the *MRPart* strategy proposed in Chapter 4 and its integration within the MapReduce framework. We also discuss how *MRPart* can be extended to include new ideas and variations of the algorithms. The implementation is divided in three main parts: Section 5.2 describes the system in

charge of workload monitoring, Section 5.3 describes the programs involved in combining workload information and performing the repartitioning of the input files and Section 5.4 gives the modifications that have been made in the MapReduce framework in order to alter the scheduling of reduce tasks.

# Chapter 2

# State of the Art

In this chapter we first present an overview of parallel computing, the main parallel architectures and computing paradigms. Then, in Section 2.2 we focus on the data partitioning problem in parallel and distributed databases. We describe the main works on the topic by classifying them in three groups: basic techniques, workload-based approaches and graph-based approaches. Section 2.3 addresses MapReduce, which is a major framework for parallel processing. We present its operation in detail and identify the main limitations and proposals that have been presented in order to overcome them. Finally, in Section 2.4 we relate the state of the art to our own work, which is presented in the next chapters of the thesis.

## 2.1 Overview Of Parallel Computing

### 2.1.1 What is parallel computing?

Parallel computing denotes the concurrent utilization of multiple computing resources in order to perform a computation. The original task is divided into multiple smaller tasks that are assigned to the computing resources in the system and executed in parallel. Computing resources may be just CPUs or cores in a single multi-processor/multi-core machine or represent autonomous computers linked through a network in a cluster or grid. Parallel computing has been applied to many problems including numerical simulations of complex systems, for scientific applications (e.g., global climate, astrophysical or earthquake modeling) and industrial purposes (e.g., crash simulation); web search; transaction processing; decision support, etc.

The main goal of a parallel system is to improve the response time of program execution. This improvement can be measured by the *speed-up*, $S_p$, which quantifies how faster is the parallel system in comparison with a sequential execution of the program:

$$S_p = \frac{T_1}{T_p}, \tag{2.1}$$

where $p$ denotes the number of processors in the system and $T_i$ the time it takes to execute the program on $i$ processors.

Ideally the speed-up should be $p$ (*linear speed-up*). However, in many program executions, there are some parts that cannot be executed in parallel or should wait for other parts to finish, e.g., because of dependencies. As a consequence, there is a point at which the addition of new resources in the system does not increase performance. The *Amdahl's law* [16] expresses this idea. It states that the speed-up of a given program is limited by the fraction that should be executed sequentially. Let $B$ be the fraction of the program that should be executed sequentially. Then, the speed-up can be expressed as

$$S_p = \frac{p}{B \cdot p + (1 - B)}. \tag{2.2}$$

Note that even the result of this expression is difficult to obtain because of start-up costs, communication costs, interference between resources, skew, etc.

An alternative way of measuring the performance improvement of parallel computation is the *scale-up*, which is used to determine how well a system sustains performance as both the number of nodes and the data size and load are increased proportionally. This measure quantifies the scalability of the system. Ideally, execution time would remain constant as the number of resources and the data size and load increase proportionally.

A parallel system provides the following advantages [97]:

**High performance:** As computing resources execute in parallel, the execution time of programs can be reduced significantly. Moreover, systems may be able to process bigger datasets or serve more users in reasonable times.

**High availability:** Introducing redundancy may allow the system to increase its availability and fault-tolerance. For instance, data replication is an effective technique to support *failover*, that is, readdressing the computations that were proceeding at the failed nodes to other resources that are available at that time.

**Extensibility:** Parallel systems should make it easy to accommodate larger datasets or more users. Some alternatives, such as cloud computing are even able to adapt dynamically to changes in the load by provisioning or deprovisioning resources automatically, an ability known as *elasticity*.

### 2.1.2 Architectures

The way in which hardware elements, e.g. main memory, disks and processors, are linked through an interconnection network gives rise to three main architectures: shared-memory, shared-disk and shared-nothing.

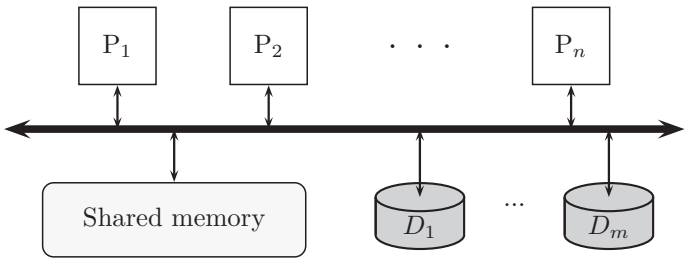



**Figure 2.1:** Shared memory architecture

### Shared-Memory

A shared-memory system consists of a single RAM main memory that can be accessed by any processor through a fast interconnection network, e.g. a high speed bus or a cross-bar switch. Processors can also access any given disk as required (see Figure 2.1).

Shared-memory systems have an easy programming model since there is a single view of data. Processes can communicate with each other at the speed of memory by writing to the same locations.

An important disadvantage of such systems is that interconnections are expensive as they require complex hardware. Moreover, extensibility is limited as the network may become a bottleneck when there are conflicting accesses to the shared memory. Finally, the memory represents a single point of failure, which may limit availability.

Symmetric multiprocessors (SMP) are a specific case of shared-memory systems where all processors are identical. Processors are interconnected using a bus, a crossbar switch or an on-chip mesh network. While buses and crossbar switches impose a bottleneck on the communication network, mesh architectures scale better and are able to support a higher number of processors.

When each processor has its own memory but all the memory is distributed and virtually shared, access times depend on whether the address belongs to the local memory or to the memory attached to another processor. This type of memory access is called Non-Uniform Memory Access (NUMA), and provides a simple model of shared-memory that allows the system to better scale, at the expense of longer accesses to remote memory.

### Shared-Disk

In a shared-disk architecture all disks can be accessed by any processor through an interconnection network, but processors have separate and private main memory blocks (see Figure 2.2). A global cache consistency mechanism guarantees that accesses to the same data are done in an ordered fashion. Typically this is

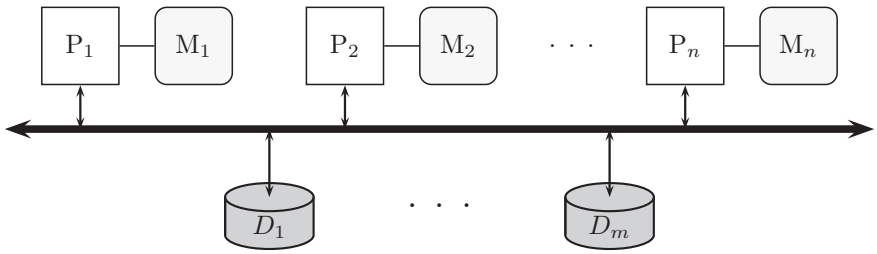



**Figure 2.2:** Shared disk architecture

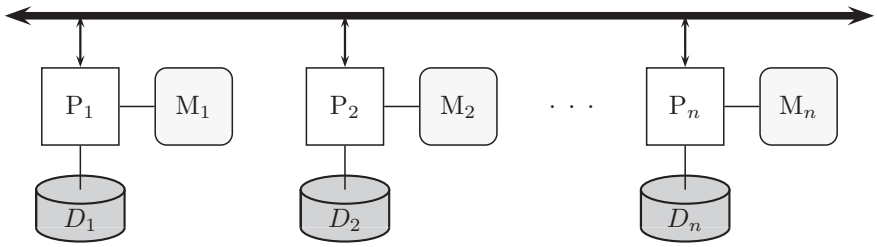



**Figure 2.3:** Shared nothing architecture

implemented by means of a distributed lock manager.

Shared-disk systems are less costly than shared-memory systems given that the cost of the interconnection is notably lower as standard bus technologies can be used. They are also easy to extend since data accesses can usually confined to local memory, reducing interferences. Availability is also higher, as disks and/or memory faults are localized and do not accept other resources. As a downside, programs are more complex because consistency has to be taken into account and they may suffer from performance problems.

### Shared-Nothing

In a shared-nothing architecture, each processor has its own private memory and disk. In fact, each node is an autonomous computer that can function independently and is connected to the other nodes through a network (see Figure 2.3). Shared-nothing systems are even less costly than shared-disk systems, as no special interconnection is required to access the disks. Additionally, the system is more extensible than shared-disk systems. Finally, high availability can be obtained using replication.

Nevertheless, shared-nothing systems are much more complex than shared-memory and shared-disks. All distribution functions have to be implemented by the application, and load balancing has to be carefully managed as it is affected by data partitioning.

#### Clusters

A cluster is a set of connected computers that share resources and behave as if they were a single computer. The nodes of a cluster are typically made of off-the-shelf components (low-cost, low-performance commodity computers) which are better in terms of cost/performance ratio than high-performance resources and allow the system to be extended easily. However, for High Performance Computing (HPC), clusters can also be made of expensive high-end computers connected through high-speed networks.

The nodes in a cluster are usually identical (homogeneous cluster) but can also consist of computers with different characteristics (heterogeneous cluster). Interconnections are usually implemented through local networks, although more advanced technologies, such as Myrinet or Infiniband can also be employed, typically in HPC clusters. In any case, all the elements of a cluster are geographically concentrated. Clusters can use either a shared-nothing or a shared-disk architecture. In the second case, two main technologies have been introduced to share the disks: network-attached storage (NAS), which consists of network appliances with one or more hard drives specifically built to provide access to storage through file-based protocols, and storage-area network (SAN), which uses dedicated networks to provide direct access to block level data storage. NAS architecture makes use of the LAN network and is easy to deploy and maintain and of relatively low cost. However the performance is low and does not scale very well. On the other hand, SANs usually employ Fibre Channel, which make them faster and more scalable but much more expensive.

### 2.1.3 Computing Paradigms

By leveraging on the ideas of cluster computing, several distributed computing paradigms have been proposed with different features and typically used in different domains.

#### Grid Computing

Grid computing is a parallel computing paradigm in which a large number of resources, typically heterogeneous and geographically dispersed, are coupled together to perform large tasks. A middleware is in charge of coordinating the resources and offering the user the possibility of submitting tasks and asking for different levels of quality of service. It is difficult to give an exact definition of a Grid, since there are many different implementations and the concept has evolved over time. Ian Foster, one of the inventors of the concept, gave a checklist of requirements of a grid system [55]:

- It *coordinates resources that are not subject to centralized control*. Resources are spread in several administrative domains and the system needs to ad-

dress security, policy, payment or membership issues, among others.

- It *uses standard, open, general-purpose protocols and interfaces.* A grid system employs multi-purpose protocols to implement all its attributions. It is important that these protocols are open and ideally they should be standard to improve interoperability.
- It *delivers nontrivial qualities of service.* Resources are coordinated in a way they can offer various qualities of service such as response time, throughput, availability and security in order to respond to complex user demands.

The grid provides coordinated resource sharing and problem solving in dynamic, multi-institutional Virtual Organizations (VO). By VO, we refer to a set of individuals, organizations or companies that agree on a set of rules for resource sharing. This sharing not only refers to data but to direct access to computers or software. It is highly controlled as both resource providers and consumers specify which are the conditions under which this sharing is performed.

Grid services and protocols are organized in the following layers [56]:

**Grid Fabric:** Provides the resources that are shared in the Grid, e.g., computational resources, storage, catalogs, network resources, etc. Resources can refer to actual physical elements or denote logical entities, for instance a distributed file system. Resources should provide both enquiry and resource management mechanisms to be exploited by higher level layers.

**Connectivity:** This layer defines communication (transport, routing, naming) and authentication protocols.

**Resource:** Defines the protocols for negotiation, initiation, monitoring, control, accounting and payment of operations on individual shared resources. We distinguish two type of protocols: information protocols, which are employed to retrieve information about the structure and state of resources; and management protocols, which are used to negotiate access to those resources.

**Collective:** As opposed to the resource layer, this layer coordinates the access to collections of resources and is responsible of many tasks, such as resource discovery, co-allocation and scheduling, monitoring, data replication, etc.

**Application:** This layer comprises the user applications, that are built by accessing the services provided by lower level layers.

#### Case study: Grid5000

Grid5000 [64] is a french nation wide infrastructure for large scale and distributed computing research created in 2003 and mainly funded by INRIA, with the participation of 19 laboratories. It includes 9 geographically distributed sites that are composed of one to many clusters with computers of different characteristics. Currently, Grid5000 consists of a total of more than 7000 cores. On each

site, the nodes connected to the same switch enjoy 1Gbps point-to-point links. Links between switches and connections between sites (that use the RENATER network) have 10Gbps. High-speed networks, namely InfiniBand and Myrinet, are also available for some nodes. Users can reserve a set of nodes and submit jobs through OAR [31], the resource manager that handles node allocation in Grid5000. Users are also allowed to deploy specific environments as desired with Kadeploy [75]. Grid5000 is the platform used for large-scale experiments in this thesis.

### Cloud Computing

Cloud computing is the latest trend in parallel computing and has been subject of much attention. The main idea is to offer both software and hardware services on demand over the Internet in a pay-as-you go basis [22]. Cloud computing is an evolution and combination of several existing models, including service oriented architectures (SOA), utility computing, cluster computing, virtualization, autonomous computing and grid computing [97]. However, it provides a new set of features:

- The illusion of infinite resources available on demand.
- The elimination of users commitment allowing them to scale-up (add more resources) or down (release resources) depending on their needs.
- The introduction of the pay-as-you-go model on a short-term basis, letting users to acquire and release both computational and storage resources for short periods of time.

The main challenge of cloud computing from the technical point of view is to support very large scale infrastructures with many users and resources in a cost-effective way. This is the reason why the main cloud providers are web industry giants such as Amazon, Google or Microsoft which have the ability to build big data centers and profit from economies of scale.

Cloud computing provides benefits for infrastructure providers and users [87]. Providers can obtain improved server utilization and reliability through virtualization and reduce energy consumption by moving data centers to cheaper energy locations. Users can use any type of specialized resource without the need to install and maintain it, adapt to workload changes by scaling up and down and deploy easily their systems using specific virtual machines; which gives the user the illusion of isolated resource access.

Cloud services can be provided at three levels:

**Infrastructure as a Service (IaaS):** provides on-demand resources, typically computing power, network and storage as services. More complex resources such as firewalls, IP addresses, load balancers, VLANs or software bundles are also offered. Resources are usually virtualized and the user is allowed to scale up and down as needed. This allows the system to allocate multiple

applications on the same physical server as virtual machines. The provider charges fees in a pay-as-you go model depending on the amount and type of resources and the time they are used. For instance, Amazon EC2 [10] offers 18 different instances with different hardware characteristics (main memory, architecture, number of CPUs and cores, storage, etc), operating systems and installed software at different prices per hour. Data transfers are also charged per GB and different storage options are also available in a usage basis like Amazon S3 [13] or Amazon EBS [9]. Other examples of IaaS providers are Nimbus [92], Eucalyptus [93], OpenNebula [95], Google Compute Engine [63] and HP Cloud [72].

**Platform as a Service (PaaS):** delivers a high-level computing platform targeted for software developers as a service. The maintenance, load-balancing and scalability is delivered by the service provider so that developers need only to focus on their solutions and not on the underlying hardware. The platform usually includes an operating system, a programming and execution environment and a database. Examples of PaaS are Google App Engine [61], Windows Azure [129] and Amazon RDS [12].

**Software as a Service (SaaS):** provides software applications as a service so that the user directly uses the applications deployed on the cloud infrastructure. The user just needs a client, e.g., the web browser, and everything is installed and managed in the Cloud. Billing is typically done in a per-use basis, normally through subscriptions. Applications range from simple ones such as email and calendar to complex software like CRM (Customer Relationship Management). There are plenty of service providers, the most notorious being Google App [62], Amazon Web Services [15] and Salesforce [107].

**Deployment Models.** The cloud computing model described so far, which brings access to resources to the general public through Internet is know as *public cloud*. When private institutions make use of technologies and practices of cloud computing for their own data centers, these are denoted as *private clouds*. As opposed to public clouds, in private clouds, the operator has explicit knowledge of all the users and can define its own policies and employ traditional security mechanisms. Sometimes, public clouds provide certain level of isolation between users in what is called *virtual private cloud* (VPC). The isolation is attained by means of private IP networks and VLANs and some form of encryption such as VPNs. An example of such a service is Amazon Virtual Private Clouds [14].

In a similar way than in the Grid Computing, different institutions (VOs) owning private clouds can federate to form a *community cloud* (also known as federated cloud) and share their resources. Furthermore, an organization with a private cloud can use resources of a public cloud, typically to respond to surge requirements, in what is called *cloudbursting*. This type of deployment is denoted

as *hybrid cloud*. These combinations are envisioned as possible evolutions of the Grid architectures to be used in scientific computing [87].

### 2.1.4 Parallel Databases

Very large database applications, e.g., e-commerce, data-warehousing, data mining, require the use of parallel systems in order to store large volumes of data and provide acceptable levels of performance and availability. Two types of applications are distinguished: On-Line Transaction Processing (OLTP), which typically generates a high number of concurrent transactions and On-Line Analytical Processing (OLAP), which produces big complex queries.

A parallel database system combines database management and parallel processing techniques to attain its objectives. The way in which it is designed varies in different implementations: from a straightforward transformation of existing DBMS to more complex blending of parallel computing and database techniques. In any case, they consist of a client-server architecture with three subsystems: a *session manager*, which connects and disconnects with the other subsystems; a *request manager*, which receives requests from the user, manages them, e.g., compiles the queries, and triggers their execution; and a *data manager*, which provides the necessary low-level functions to execute compiled queries [24].

The relational data model is used, which offers good opportunities for data-based parallelism [48]. Relational queries consist of operators applied to database relations that produce new relations, so they can be composed in parallel data-flow graphs. The parallelism can be obtained in several ways [97]:

- **Inter-query parallelism:** consists of the parallel execution of multiple queries generated by concurrent transactions. A better throughput can be obtained.
- **Intra-query parallelism:** consists of the execution of a single query in parallel in order to obtain better response times. Two complementary strategies can be used:
  - **Intra-operator parallelism:** where each operator is divided into a set of sub-operators that are applied to a fragment of the data.
  - **Inter-operator parallelism:** consists of the parallel execution of several operators of a given query. When several operators are chained in a producer-consumer flow we apply *pipeline parallelism*. When there is no dependency relation between the operators, we apply *independent parallelism*.

### 2.1.5 Techniques

In order to implement and execute computations in a parallel system,many techniques have been studied, developed and applied in many systems. In this

section, we briefly describe the most important ones.

### Data Partitioning and Allocation

When a program is executed in several computational resources, each of them has to be assigned a different portion of the work. Some computations just assign different parameters to each of the nodes, which generate the input by themselves. But in many cases, the input consists of a large dataset that has to be divided and assigned to the nodes. This is carried out in a two-step process: data partitioning or fragmentation (to divide the dataset) and data placement or allocation (to assign the fragments to the system's nodes). Data partitioning and allocation are specially important in shared-nothing architectures, where each node has attached disks which are accessed much faster than the disks attached to other nodes.

If the original input data is stored in a single site or outside the parallel system, it has to be divided and transferred to the participating computers. However, in many cases, the data is stored in the same computers that execute the program in order to avoid the overhead of data transfers, thus applying the principle stating that "moving computation is cheaper than moving data", which aims for *data locality*. The paradigmatic examples of this idea are distributed and parallel databases, where each node is assigned a partition of the database and there is a mechanism that determines the nodes that must participate in the processing of a query [97]. Another important example of this principle outside databases is the use of distributed file systems where different files and even different fragments of the same file are spread over all the computers of the system [32, 60, 110, 128]. When a program is executed, each of the units of work in which it is divided is attempted to be executed at the same node that stores its input [43].

There are many different strategies to partition and allocate a dataset. The selection of a specific strategy depends on many factors, such as the type of programs being executed, the information available about their distribution and evolution over time (workload information), the data structure and layout, etc. In Section 2.2, we give a more detailed description of these strategies and their convenience.

### Load Balancing

In a parallel system, it is fundamental that nodes are given a similar amount of work (or a work proportional to their computational power if it differs) in order to avoid the overload of some of them while others remain idle. This is known as load balancing and has a significant impact on resource usage, throughput and response time. The opposite situation, where a node or a set of nodes have to deal with larger data sets or more complex computations is known as *skew*. Load balancing is tightly connected with data partitioning and allocation, although

it also relates to scheduling, and it frequently conflicts with data locality. An illustrating example occurs when a set of popular fragments, called *hotspots*, are assigned to the same node. In order to preserve data locality, a lot of computations have to be performed at that node, thus generating an important skew and hurting load balancing.

Load balancing can be pursued within a single parallel program or by considering several programs that are executed in parallel. In the first case, the response time of the program is determined by the longest unit of work. Load balancing reduces it by evening the loads of all nodes in order to reduce response times. In the second case, a better throughput can be obtained by executing different programs at different nodes in parallel.

There are many different approaches to obtain load balancing and they depend on the type of program and partitioning used. In general, we can divide these approaches in two types: static and dynamic (or adaptive). When the workload is static and known in advance, data allocation and scheduling may be carried out so that the expected load at each node is similar (by also taking into account other issues such as data locality). On the other hand, if the workload is unknown and/or changes over time, a dynamic approach should be used [46].

### Replication

*Data replication* is used in parallel systems both to improve both availability and performance. Availability is increased because even if some of the nodes fail, data can still be accessed by retrieving one of the copies stored at other nodes. Performance can be improved by: 1) executing programs that access the same data at different nodes, thus increasing parallelism; 2) enhancing load balancing by replicating hotspots at different nodes; and 3) increasing data locality.

Nevertheless, data replication comes at a price. Whenever a data element is modified, the changes have to be propagated to the other replicas. Depending on the application requirements, this synchronization may be more or less complex and a different replication protocol should be used. In general, the way in which replication is carried out depends on the following factors:

- **Consistency model:** The consistency model defines the rules under which writes and reads are ordered in the execution. This is fundamental for atomic operations, such as transactions. The strongest levels of consistency are *linearizability* [71] and *serializability* [25] but they can be relaxed to improve performance, e.g., eventual consistency [125].
- **Where updates are performed:** *Centralized* protocols require that all updates are first performed on a master copy, while *distributed* ones allow updates to be executed on any replica. In centralized systems, the master replicas can all be stored at the same site (*single master*) or spread among the sites (*primary copy*).

- **How updates are propagated:** Updates can be propagated within the context of a transaction (*eager replication*) or after the transaction has committed (*lazy replication*). In the last case, some form of conflict resolution may be necessary in order to resynchronize the copies.

There is another form of replication, called *computational replication*, which is also used to improve performance and reliability. In this case, the same task is executed several times at different nodes. In that way, the program does not need to wait for tasks on faulty or slow nodes and response time is reduced. This technique has been employed for performance reasons in peer-to-peer communities [105], the Grid [88, 113] and MapReduce [43], in the form of speculative task execution. In volunteer computing, where malicious participants may introduce erroneous results, computational replication can also be used to check the correctness of the outputs by employing majority voting mechanisms between the task replicas [109].

### Fault Tolerance

Two concepts are central to fault tolerance: reliability and availability. *Reliability* is the capability of a system to operate without failures for a period of time, defining failures as deviations over the system specification. It is normally used for elements that cannot be repaired. Formally. it is defined as a probability [97]:

$$R(t) = P\{0 \text{ failures in time } [0,t) \mid \text{no failures at } t = 0\} \tag{2.3}$$

*Availability* is the fraction of time that a given system is operational, although it can also be seen as the probability of the system to be operational at a given time. Typically it is a decimal measure, e.g., 0.9999, although sometimes a metric called *nines* is used instead, which denotes the number of nines in the decimal metric. It always refers to systems that can be repaired and measures their ability to tolerate failures on their elements.

In a distributed or parallel system, there are different types of failures:

- **Program failures:** These are the failures caused by erroneous conditions in the program execution or inadequate input data.
- **Node/System failures:** Either with a hardware or a software origin, the effect is that the content of the main memory is lost.
- **Media failures:** They are failures produced in secondary storage devices containing data relevant to the system operation. It is assumed that the contents of the disks are accessible any more.
- **Communication failures:** Refer to failures in communications between the nodes of the system. Network protocols are responsible of dealing with some of the possible failures, e.g., error correction and ordering of messages, but other failures, such as line failures are not solved by them. An special case is when the network fails in such a way that nodes are isolated in two

or more disjoint groups. This is called *network partition*.

A parallel system can be built so that individual failures of some of its elements do not make the system unavailable. The main technique used to obtain fault tolerance is *redundancy*, that can refer to the inclusion of duplicated hardware elements, e.g, an additional power supply, or software mechanisms, such as replication as explained before. It is also important that the adequate recovery mechanisms are integrated into the system design, so that when transient failures are solved, the system can return to an appropriate state.

The first requirement to deal with failures is to be able to detect them when they occur. This ability can be provided by a specific component such as the communication layer, e.g., TCP, or a group communication system [36], or implemented as a component of the replication logic. Once the failure has been detected, a *failover* mechanism is applied. It consists in redirecting the affected requests from the failed node to another one in a transparent way to the user. This process may be more complex if the master is the node that failed. Moreover, depending of the consistency model, failover may require the abortion of ongoing transactions. Finally, the *recovery* mechanism is in charge of reintroducing failed replicas in the system by transferring the missed state updates to the recovering nodes.

## 2.2 Data Partitioning in Databases

### 2.2.1 Definition

Data partitioning consists in splitting a given dataset into a set of fragments. This technique is needed in distributed and parallel systems, as usually the data has to be divided and assigned to the participating nodes.

In database systems, the use of the relational model makes data partitioning practical. A partitioning $\pi(R) = \{F_1, ..., F_n\}$ over relation $R$ is said to be correct if it satisfies the following properties [97]:

- **Completeness:** Every data item that appears in $R$ should appear in at least one of the fragments $F_i$.
- **Reconstruction:** There should be an operator $\nabla$ that allows the original relation to be rebuilt from the fragments, $R = \nabla F_i,\ \forall F_i \in \pi(R)$
- **Disjointness:** Each item of $R$ should appear in at most one of the fragments.

### 2.2.2 Types

A given relation can be partitioned in two ways: horizontally or vertically. If both strategies are used, i.e., a relation is first horizontally partitioned and then vertically or vice versa, we are talking about *hybrid partitioning*.

**Horizontal partitioning.** It consists of dividing relation $R$ along its tuples, so that each fragment is assigned a subset of the tuples. The reconstruction operator $\nabla$ is the union, i.e., $R = \bigcup F_i,\ \forall F_i \in \pi(R)$.

Two types of horizontal partitioning are distinguished: *primary horizontal partitioning*, which is done using predicates over a given relation and *derived horizontal partitioning*, which is performed by propagating the partitioning over the attributes of another relation in a link, where the link is defined by an equijoin operation [97].

In the first case, the partitioning is done by applying the selection operator over a set of predicates. In this way, fragment $F_i$ is obtained by applying the fragmentation predicate $p_i$ over $R$, $F_i = \sigma_{p_i}(R)$,. The way in which those predicates are selected depends on the global conceptual schema of the database, in particular the links between relations; and on the application, mainly the queries that are executed and their relative frequencies. For derived partitioning, if there is a link between relations $R$ and $S$, the fragments on $S$ are obtained from the fragments on $R$ by applying the semijoin operator: $S_i = S \ltimes R_i$. When there are more than one link into relation $S$, a choice must be made regarding derived partitioning. It should be done depending on the join characteristics and the frequency of its use.

**Vertical partitioning.** It consists of dividing $R(A), A = a_1, ..., a_m$, so that each of the fragments $F_i(A_i)$ only contains a subset of the attributes of $R$. The reconstruction operator is the join on the primary key of $R$, i.e., $R = \bowtie_{PK} F_i,\ \forall F_i \in \pi(R)$, where $PK \subseteq A$ is the primary key of $R$. As a consequence, the attributes of the primary key should be replicated in all the fragments. The attributes that are accessed together are privileged to be on the same fragment because a join, which is a costly operation, is required to answer queries that access attributes in several fragments.

With a concept similar to vertical partitioning, *column-oriented databases* store data along columns instead of rows. The basic approach (and seminal paper) is the Decomposition Storage Model (DSM) [40] where a relation $R(A_1, ..., A_n)$ is partitioned vertically in $n$ fragments, each corresponding to a binary relation with a tuple identifier and one attribute. The main goal is to reduce I/O bandwidth as only the attributes needed in the query are accessed on disk, as in vertical partitioning. However, the column-oriented databases provide further advantages related to the data layout and the query execution optimizations that go beyond that of reduced I/O. In fact, the use of analogous vertical partitioning approaches in row-oriented databases do not obtain the performance gains of column-oriented databases [1].

Although an old concept, column-oriented systems have received a lot of attention recently, because of their superior performance for OLAP which features ad-hoc queries. Notorious systems include MonetDB [27], which follows a pure

DSM model, C-Store [116] which groups and optionally replicate attributes in projections or Vertica [124] and Sybase IQ [108], as examples of commercial systems.

### 2.2.3 Optimization Objectives

Data partitioning can be expressed as an optimization problem. Two opposite optimization goals can be used when designing a partitioning strategy:

- **Clustering:** The goal is to place data elements that are frequently accessed together in the same fragment(s). If a query accesses a small number of elements, data clustering will require a small number of fragments to answer it. This is the only suitable strategy for vertical partitioning, since fragment combination is expensive, as it requires to execute joins. It is also the strategy to be used in horizontal partitioning when the overhead of distributed execution is high with respect to query response time.
- **Declustering:** The goal is to spread the elements a query must access in every fragment in the most balanced way. This strategy is used when the objective is to optimize intra-query parallelism. In the execution of a query and under the same computation capabilities at all nodes, the fragment with the higher number of relevant data items will determine the response time.

### 2.2.4 Techniques

#### Basic techniques

Three basic techniques have been extensively used in the literature to partition and assign tuples to nodes in a parallel database:

- **Round-robin partitioning:** The tuples of a relation are assigned to each node in a round-robin fashion, i.e., if there are $n$ nodes, tuple $i$ is assigned to node ($i \mod n$). It guarantees uniform data distribution. A scan of a relation can be performed in parallel at all nodes, but direct access to tuples based on predicates needs that all nodes be accessed.
- **Hash-based partitioning:** A hash function is applied over a given attribute or set of attribute's values (partitioning attributes), whose result determines the node in which that tuple is stored. Queries with exact-match selection predicates on the partitioning attributes can be executed at just one node, while the rest of the queries have to access all the nodes.
- **Range-based partitioning:** Tuples are assigned to the nodes depending on the value intervals of some attributes (partitioning attributes). Exact-match queries can be executed at just one node, the same way as in hash-based partitioning. Range-queries can also be executed at just one node (or a few if the query range is big).

These techniques have been early adopted in parallel databases. Notorious examples include The Gamma Database Machine Project [47], which offered the three possibilities (round-robin being the default one), Bubba [39], which used either a hash function or ranges that are defined based on the frequency of access rather than the size; or Teradata [49]. Due to their simplicity, they are still being used in modern storage systems, such as cloud key-value stores. For example, Google's BigTable [33] uses range partitioning on the key to split the tables into small fragments called *tablets*, which are then allocated based on load balancing. Yahoo's PNUTS [38] allows to use both range-based partitioning (in ordered tables) and hash-based partitioning (in hash tables) to produce the tablets. Like BigTable, its number is much bigger than the number of nodes to allow better load balancing. Amazon's Dynamo [45] and Cassandra [86], among others, use consistent hashing [80] to partition and allocate the rows. In this technique, the output of the hash function is treated as a circular space. Nodes are given a random id and are responsible for the rows with hash values (resulting from applying the hash function) between that id and the id of the next node in the circle.

In a parallel query execution, there is an initial step that creates and initializes processes and communications. Its execution time is proportional to the degree of parallelism. Thus, for simple queries, if the degree of parallelism is high, this phase can dominate the response time. As a consequence, the number of nodes involved in the execution of a query should be adjusted depending on its complexity. The round-robin approach leaves no place to control this aspect and queries must always be redirected to all nodes. On the other hand, in the case of hash-based and range-based partitioning, some types of queries can be localized in one or a few nodes. Exact-match queries on the partitioning attributes are directed to just one node in both approaches. In the case of range-based predicates on those attributes, range-based partitioning is able to localize the execution at a few number of nodes (depending on the range size). However, this is not always advisable, as mentioned before. Hash-based partitioning needs to access all nodes when executing range-based selection queries, giving no control in the degree of parallelism. A *Hybrid Partitioning Strategy* has been proposed in [59] that provides the best of both strategies and is able to adapt the degree of parallelism to optimize query response time. This approach creates several small ranges and then allocates them in a round robin fashion. Queries with small ranges or exact matches can be executed at one node. As the size of the range increases, a higher number of nodes would be involved.

From the basic strategies, only the round-robin approach guarantees uniform data distribution. However this is not the only source of skew. In [126], five different types of partitioning skew are distinguished. Attribute value skew (AVS) describes the variation on the number of tuples for different values of a given attribute, while Tuple Placement Skew (TPS) refers to the variation on the number

of tuples assigned to each node. AVS, which is inherent to the dataset, can be the cause of the TPS both in hash-based and range-based partitioning. Selectivity Skew (SS) is introduced when the selectivity of predicates varies on each node. Finally, all strategies are vulnerable to Redistribution Skew (RS), which occurs when data is redistributed between two operators, and to the Join Product Skew (JPS), that arises when the join selectivity varies among nodes.

The round-robin approach needs no parameter, so it can be applied automatically. However, both hash-based and range-based partitioning require a set of attributes to be selected as the partitioning attributes. The choice should depend on the workload; in particular, in the attributes used in the predicates of the queries. Depending on the workload, the selection of those attributes can be quite complex, which is why some automatic partitioning strategies have been designed. The range-base approach also requires a strategy to select the range boundaries in order to produce balanced partitions. This task may not be trivial due to AVS.

#### Workload-based techniques

When the workload of an application gets complex, the selection of a partitioning strategy may become extremely difficult for the database administrator. This has motivated the emergence of tuning advisors that recommend and simulate the impact of different database configurations on performance [34]. Notorious examples of such tools are Database Engine Tuning Advisor (DTA) [3] on Microsoft SQL Server, DB2 Design Advisor [122] in IBM's DB2 Universal Database and SQL Access Advisor [42] in Oracle 10g.

DTA was the outcome of the AutoAdmin project, which started in 1996. The first versions were interested in the automatic selection of indices in order to improve the performance of the system under a given workload. They provide the concept of an hypothetical ("what-if") index, that allows DBAs to analyze the impact of the addition of a given set of indices in the database [35]. This tool was later enhanced to include the possible design alternatives in the creation of materialized views [4] and the use of horizontal and vertical partitioning [5]. The final architecture includes four steps. First, the search space is reduced by *pruning*, which discards the table sets (where materialized views could be created) and column sets (from which indices and partitioning keys could be obtained) that are not relevant to any query. Second, in the *candidate selection* step, the set of alternatives are further reduced by considering their cost-benefit. The result is a (near-) optimal configuration for the workload, but often with important costs in terms of storage and update overheads. Because of that, there is another *merging* step in which the alternatives are combined in order to produce cheaper candidates that can benefit several queries at the same time. These new alternatives are added to those produced in the second step. Finally,

in the *enumeration* step, using the workload and the set of candidates produced in the second and third phases, a search algorithm is employed in order to obtain the subset of candidates with the smallest total cost for the given workload. Due to the complexity of the search space, heuristics have to be employed in this procedure.

DB2 advisor also started by working with indices [122] and included other possible design alternatives such as materialized query tables (MQT), multi-dimensional clustering (MDC) and partitioning [133]. Two types of approaches to obtain the set of design alternatives to apply are identified: the *iterative* approach, which selects each feature one at a time (indexing, partitioning, etc.) and the *integrated* approach, where search is performed in the combined search space, which is the approach of DTA. In DB2 advisor, the relationships among features are categorized. For strong dependencies, an integrated approach is used. For the cases where there are no dependencies or they are weak, an iterative approach is employed. DB2 Advisor has three components: IM, which recommends MQTs and indices, P, which is responsible of partitioning and C, which recommends MDCs.

Recent works have also proposed automatic partitioning adapted to the workload but their main focus is how to partition the database in multiple nodes and not physical design. In [91] a partitioning advisor is designed to improve the performance of complex queries in Massive Parallel Processing (MPP) systems. One of the main goals of the proposed system is to minimize the data movement operations when data needs to be re-partitioned in some query executions. Their approach is tightly coupled with the query optimizer, as it uses the same cost model. They reduce the search space by inferring lower bounds on partial partitioning configurations, which allows the search algorithm to apply branch and bound techniques [83] more efficiently, i.e., discard whole branchs of the solution tree early in the search. Tables can be either replicated or partitioned using a hash function on a single column. The possible columns used in the partitioning are selected before the search among the ones referenced in equi-joins and in group-bys.

In [100], the targeted system is an enterprise-class OLTP application, which includes stored procedures, load-balancing constraints under variable skews and complex schemas and deployments. They implement the solution in H-store [79], a main-memory DBMS deployed in a shared-nothing architecture. The objective of the partitioning is to reduce the number of distributed transactions while moderating the effect of temporal skew. Their search algorithm, called Horticulture, is provided with the application schema, the stored procedure definitions and a trace of the workload. Using an adaptation of the large-neighborhood search technique [6], they explore the search space in order to obtain the best solution. The design alternatives are: 1) horizontal partitioning on one or multiple attributes using either hash-based or range-based partitioning; 2) whole table replication;

or 3) a replication of a secondary-index on a set of attributes.

Automatic database partitioning has also received some attention in scientific databases. For instance, in [99] the authors propose AutoPart, an algorithm that automatically finds the best partitioning alternatives for physical design in large-scale scientific databases, where indexes and materialized views are less appropriate because of the data volume and the continuous insertion of new data in the database. Two types of partitioning are used: categorical and vertical partitioning. Categorical partitioning horizontally partitions a table using categorical attributes, i.e., attributes with a small number of discrete values, typically used to identify classes of objects. The algorithm starts by fragmenting the tables by identifying the categorical attributes. Then, on each of the partitions, atomic fragments are generated for vertical partitioning. An atomic fragment is a subset of the attributes that is always accessed atomically, i.e., there is no query accessing a subset of those attributes. Those fragments are later combined into composite fragments with a greedy algorithm until no gains in the workload cost can be obtained.

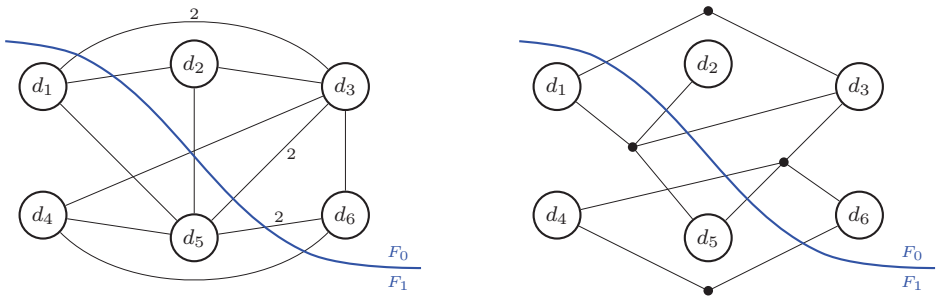
### Graph-based techniques

Graphs have been commonly used to represent data dependencies in a computation, and then graph partitioning algorithms to divide the work between the different nodes of a parallel system [69]. Examples of graph usage include parallelization of matrix-vector multiplications, neural net simulations, particle calculations, VLSI layout design and database partitioning.

In database systems, graph-based techniques have been used both for declustering [90, 84] and clustering [41]. As opposed to other partitioning approaches, e.g., the ones mentioned so far, graph-based techniques can work with any schema and independent of the complexity of the queries in the workload [41], as only relations between data items are used.

Liu et al. [90] address the problem of declustering a database in a single node with multiple disks, so that queries can be fully parallelized, i.e., data is read from multiple disks in parallel. The core idea is the concept of similarity, which represents the likelihood that two data items will be accessed together as part of query execution. A model of the workload is built using a graph $G = (V, E)$, called Weighted Similarity Graph (WSJ), where vertices represent data items and edges are weighted depending on the similarity between two data items, i.e, $w(d_i, d_j) = f$ denote that queries accessing data items both $d_i$ and $d_j$ accounts for a total frequency of $f$. The WSJ is then split into $k$ independent components (as much as disks in the system) by means of a max-cut graph partitioning algorithm. The optimization goal of this algorithm is to maximize the weight of the edges that are crossed by the cut. Constraints on the size or frequency of access of the components can also be incorporated into the model.

$D = \{d_1, d_2, d_3, d_4, d_5, d_6\}$, $Q = \{q_1, q_2, q_3, q_4\}$
$q_1 = \{d_1, d_2, d_3, d_5\}, q_2 = \{d_3, d_4, d_5, d_6\}, q_3 = \{d_1, d_3\}, q_4 = \{d_5, d_6\}$




(a) (b)

**Figure 2.4:** Graph models for data partitioning: (a) graph, (b) hypergraph

As pointed out in [69], the graph model is not completely appropriate for all applications and other related models may be used instead. In [84], the idea is developed and instead of using a graph to represent the workload, an hypergraph $H = (V, E)$ is built. The objective remains the same: declustering a database for efficient execution of queries in a single-node system with multiple disks. Vertices also represent data items but, in this case, an hyperedge $e = \{d_{e_1}, ..., d_{e_n}\} \subseteq V$, which is a set of vertices, is added for each of the queries in the workload. Therefore, if a query $q$ accesses data items $\{d_{q_1}, ..., d_{q_n}\}$, while a single hyperedge containing all the accessed items is added in the hypegraph model, a clique of $n$ vertices, with $\frac{n(n-1)}{2}$ edges, needs to be included in the graph model. The partitioning is obtained by applying a modified version of the hypergraph partitioning problem, where the optimization goal is to minimize the maximum weight each hyperedge has on a partition. The idea is that the partition with the highest weight will represent the node with the highest amount of work in the query processing and hence determine the query response time. When the number of elements accessed in a query is equal or less than the number of partitions, the algorithm's optimization goal is exactly the same as the standard hypergraph partitioning problem, i.e., minimize the weighted sum of conductivities of hyperedges.

Figure 2.4 compares both graph models. In the example, the database consists of 5 data items and 4 queries (frequency is not considered for simplicity). In Figure 2.4(a), the workload is modeled using the method proposed in [90]: for each query, a clique linking all the accessed data item is added to the graph (the edge weight is increased if it already existed). In our example, the obtained partitioning has a cut value of 11, corresponding to the sum of the weights of the edges traversing the cut. In Figure 2.4(b) an hypergraph is used as in [84]. In

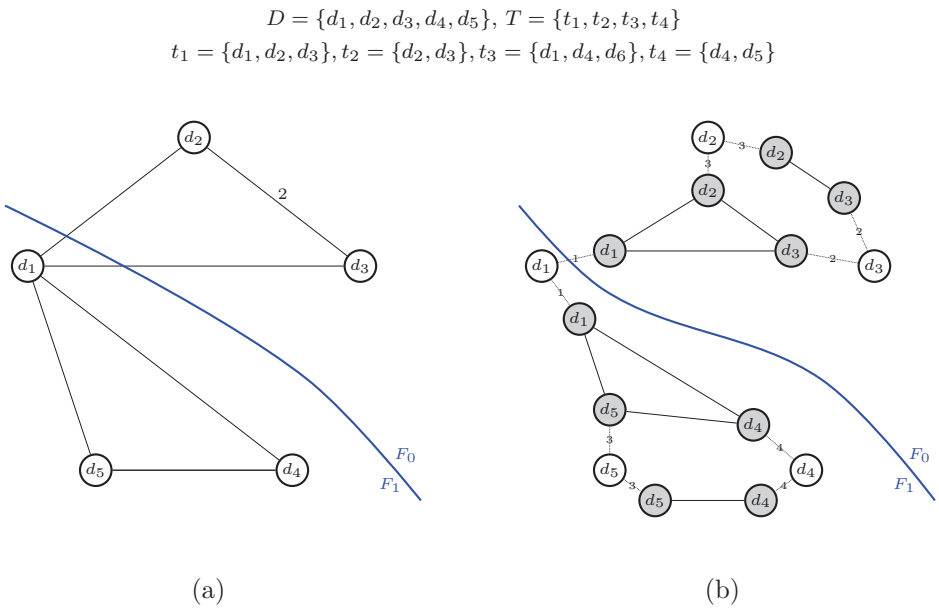



**Figure 2.5:** Graph model used in Schism (a) without replication, (b) with replication

this case, each query generates an hyperedge connecting all the accessed elements. The cut value used for the optimization goal would be 6, corresponding to the sum of the maximum weights each edge cut by the partitioning has on a given fragment.

Curino et al. [41] propose Schism, whose objective of the partitioning is different. The targeted system is an OLTP application, and the goal is to reduce the number of distributed transactions, that is, maximize the number of transactions that are executed at a single node. A simple graph model is used, but in this case the graph is built so that replication alternatives are considered (see Figure 2.5). Instead of queries, edges are built from transactions, i.e., when two tuples are co-accessed in a transaction, the frequency of the query is added to the weight of the edge that links the vertices representing those tuples. The possibility of replication is modeled by exploding the vertex associated with a tuple into $n + 1$ vertices in a star-shape schema, $n$ being the number of transactions accessing the tuple. The cost of replication is considered by weighting the edges between the new created vertices and the center of the star. Since, the partitioning objective is clustering, the algorithm that is used to partition the graph is a min-cut partitioning algorithm.

An explanation phase is added to the system to have a compact model of the tuple assignments to partitions, as an alternative of a fine-grained, tuple-to-partition index. The result is a set of rules that assign the values of some of the tuple attributes to specific partitions. A decision tree classifier is used for that matter. Frequently used attributes are selected and their correlation with

assigned partitions analyzed. Several other optimizations are also envisioned to obtain scalability on the graph partitioning algorithm, including transaction-level sampling, tuple-level sampling, relevance filtering, etc.

Constrained k-way (hyper-) graph partitioning is known to be NP-hard. As a result, several heuristics have been used and implemented, mainly iterative-improvement heuristics such as the ones proposed by Kernighan-Lin (KL) [82] and Fiduccia-Mattheyses (FM) [53]. The idea is the following. Starting from an initial bi-partition, iteratively move the vertices that offer the maximum gain in the cut-metric from one partition to the other. Even negative gains are allowed to provide the algorithm with a hill-climbing ability. Multilevel partitioning [70] is widely used, which consists of three phases: 1) *coarsening phase*, where the graph is transformed into a sequence of smaller graphs; 2) *partitioning phase*, where a partitioning of the graph is carried out; and 3) *uncoarsening phase*, where the partitioning obtained over the coarsened graph is projected back to the original graph. K-way partitioning is usually performed by applying recursive bisection, where two-way partitioning is recursively applied until the desired number of partitions is obtained. However, direct k-way partitioning methods also exist.

There exist several software packages devoted to graph, hypergraph and mesh partitioning, both as stand-alone programs and libraries that can be integrated into other applications. Notorious examples include METIS [81] (and its alternatives for parallel partitioning, ParMETIS; and hypegraph partitioning, hMETIS); PaToH [121], devoted to the partitioning of hypergraphs; Scotch [57], which provides sequential and parallel versions for graph partitioning, and a sequential version for mesh and hypergraph partitioning; and Zoltan [26], which provides sequential and parallel versions of graph, hypergraph and mesh partitioning. Scotch and Zoltan also offer other related tools for dynamic load balancing, graph coloring, etc. In general, they allow the user to control many parameters, including the algorithms used in many of the sub-phases, imbalance constraints, optimization goals, etc. Moreover, they work with different graph formats, with and without weights in both vertices and edges and allow to specify some constraints, like fixed vertices.

## 2.3 MapReduce

In this section, we describe MapReduce, its implementation and several proposed improvements. Section 2.3.1 describes MapReduce's programming model and operation. More details about the implementation are given in Section 2.3.2. Finally, Section 2.3.3 identifies MapReduce's limitations and presents the most important works aiming at their improvement.

### 2.3.1 Overview

MapReduce denotes both the programming model and the framework originally developed by Google [43] for parallel processing of large scale datasets. Users only need to provide two functions, called map and reduce, and the framework handles all the issues related to parallelization, fault-tolerance, data distribution and load balancing. Although the programming model by itself is not new, as it is inspired by the map and reduce primitives of functional languages such as Lisp, it made a tremendous impact as it allows programmers to implement scalable and fault-tolerant versions of their applications in a simple way.

MapReduce has received a lot of attention both in research and industry. An open-source implementation of the framework, called Apache Hadoop [19] has become extremely popular. There are other implementations of MapReduce, such as Amazon Elastic MapReduce [11], Aster MapReduce Appliance [23] and Greenplum MapReduce [103].

#### Programming Model

MapReduce programs are expressed by means of two functions:

***map*:** consumes input key-value pairs and produces for each of them a (possibly empty) list of intermediate key-value pairs. Formally:

$$map : (\mathcal{K}_1, \mathcal{V}_1) \rightarrow list(\mathcal{K}_2, \mathcal{V}_2)$$

***reduce*:** receives an intermediate key and all its intermediate values and produces a (possibly empty) list of output key value pairs. Formally:

$$reduce : (\mathcal{K}_2, list(\mathcal{V}_2)) \rightarrow list(\mathcal{K}_3, \mathcal{V}_3)$$

The execution of the map and reduce functions over different pairs can be done in parallel. The framework is responsible of sorting and grouping intermediate pairs with the same key for the reduce function execution.

**Example: WordCount.** An example that is frequently used to illustrate MapReduce is the `wordcount` example. This program counts the occurrences of words in a given file. The implementation of this program in MapReduce is shown in Algorithm 1. The map function just divides a given line into words, and produces a pair (*word*, 1). The reduce function receives for each word all the produced counts and adds them before emitting the final count. In the following example, we illustrate the behavior of map and reduce functions:

*map*(1321, “Darkness there, and nothing more.”) →
⟨ (“Darkness”, 1), (“there”, 1), (“and”, 1), (“nothing”, 1), (“more”, 1) ⟩

*reduce*(“more”, (1,1,1,1,1,1,1,1)) →
⟨ (“more”, 8) ⟩

**Algorithm 1:** Wordcount in MapReduce

**Types**:
$\mathcal{K}_1$ : long; $\mathcal{V}_1$ : text
$\mathcal{K}_2, \mathcal{K}_3$ : text; $\mathcal{V}_2, \mathcal{V}_3$ : int

```
1 map( offset : K1, line : V1 )
2     foreach word ∈ line do
3         emit (word, 1)

4 reduce( word : K2, counts : list(V2) )
5     sum ← 0
6     foreach c ∈ counts do
7         sum ← sum + c
8     emit (word, sum)
```

### Architecture and Basic Operation

The MapReduce framework executes programs in parallel in a shared-nothing cluster (see Figure 2.6). There are two types of processes, the *workers*, which execute map and reduce tasks and the *master*, which is responsible of controlling the workers' execution. Usually, input and output data are stored in a distributed file system, e.g., GFS [60], which executes at the same nodes where MapReduce jobs are run.

In a MapReduce job, the input is partitioned into $M$ splits, which are consumed by $M$ map tasks, one per split. The map tasks output is partitioned by the intermediate key into $R$ fragments using a partitioning function, by default ($hash(k_2) \mod R$), which are then processed by $R$ reduce tasks.

When a job is launched, the master partitions the input into $M$ fragments. Map and reduce tasks are then assigned to workers as they become idle, first the map tasks and then the reduce tasks, once all map tasks are finished. The output of the map tasks is partitioned into $R$ fragments by the intermediate key and stored in the local disks of the workers. Reduce tasks fetch these outputs and sort them by key so that all the values of a given intermediate key are processed together by the reduce function. Once all map and reduce tasks have finished, the user is notified.

In this case, the input is divided into 3 splits and consumed by map tasks $m_0$, $m_1$ and $m_2$, respectively. After reading the split, the corresponding input key-value pairs are passed to the map function. The map function executes the

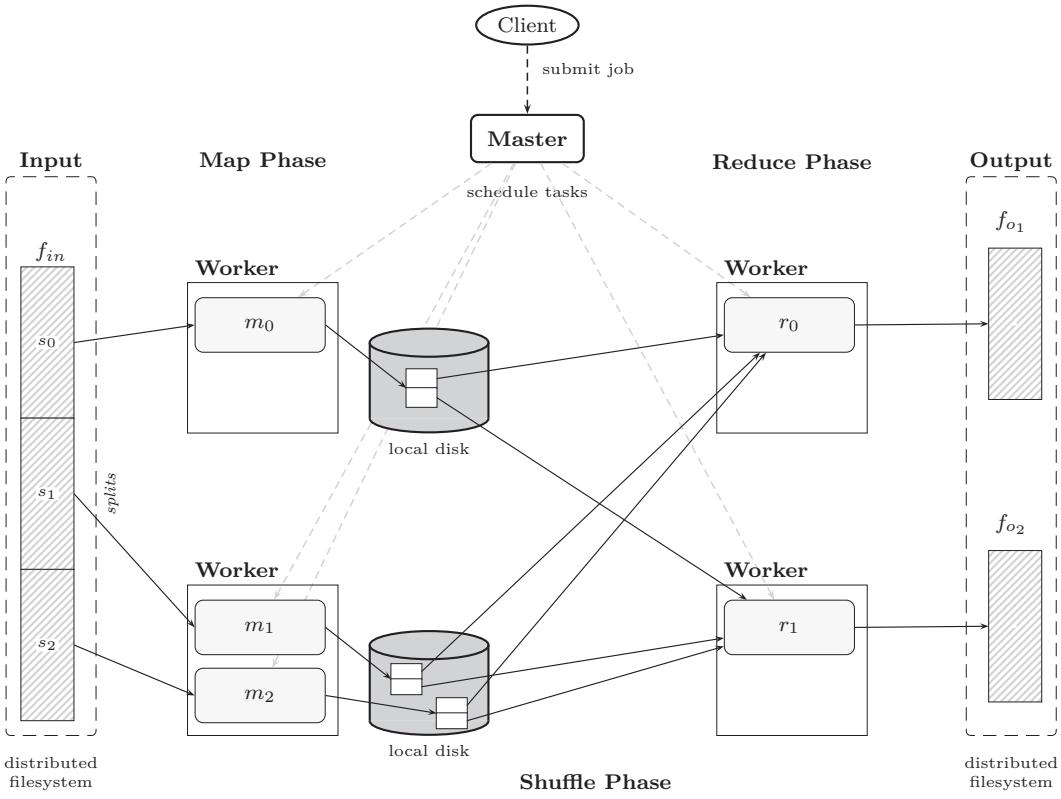



**Figure 2.6:** Architectural view of a MapReduce job execution

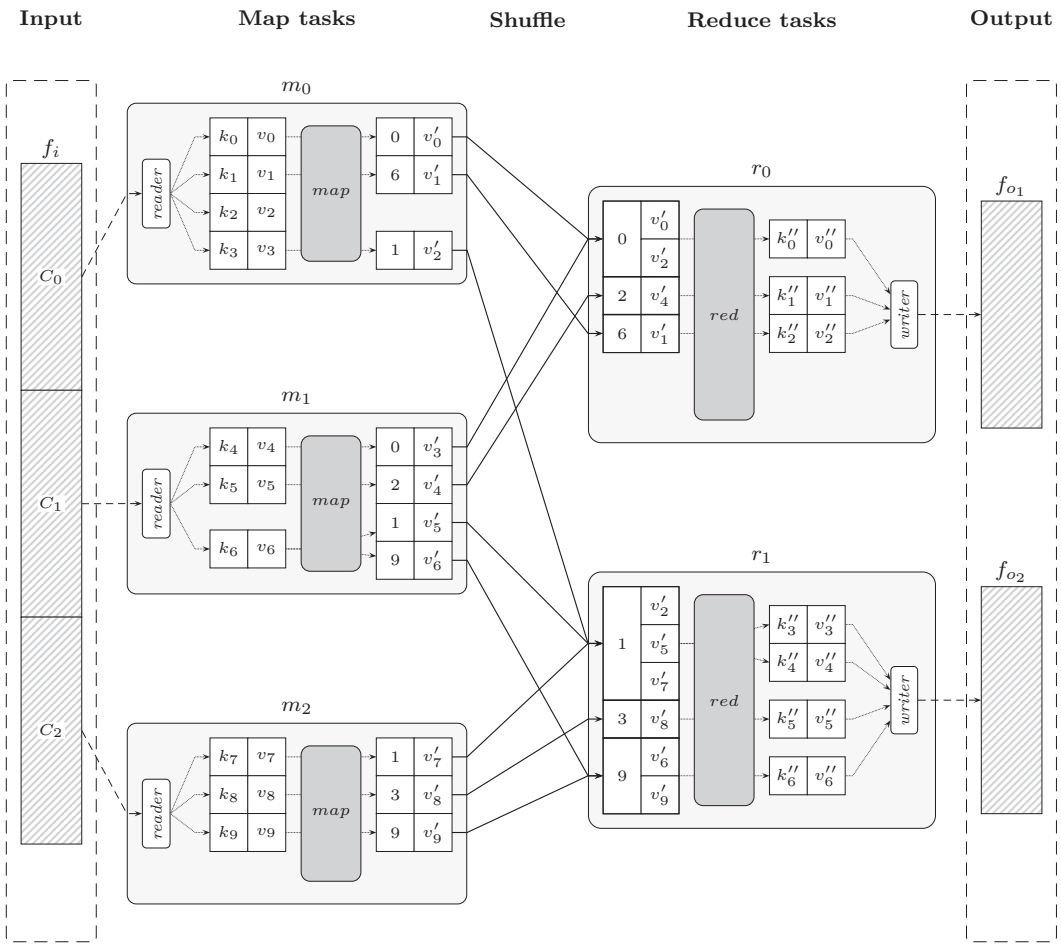



**Figure 2.7:** Logical view of a MapReduce job execution

program provided by the user and generates a set of intermediate pairs. This pairs are partitioned and sent to the corresponding reduce, where they are grouped by key. In the example, the default partitioning function is used, so even keys are sent to reduce task $r_0$ and odd keys to $r_1$. The key and the set of intermediate values are then passed to the reduce function, which in turn produces a set of output key-value pairs, which are written back to the file system.

### Data Locality

MapReduce is used in combination with a distributed file system, so that data is already stored in the same nodes that perform the computation. In Google, the Google File System (GFS) [60] is employed. By default, it automatically splits files into 64MB blocks, which are then replicated (typically 3 times) and stored in different machines.

In Google's MapReduce, a correspondence between file blocks in GFS and

splits in the map tasks input is established. The master tries to schedule map tasks in the same machines storing the corresponding block. If this is not possible, a close replica is chosen (for instance, in the same rack). The objective is to save cluster bandwidth, as most input data is read locally without any network transfer.

### Shuffle

*Shuffle* is the name given to the intermediate phase of MapReduce, when intermediate keys are partitioned, sorted and transferred to the nodes executing the reduce tasks. Depending on the volume of data produced in the map tasks and the network characteristics (bandwidth, network topology), this phase may take a considerable amount of time [127, 98].

The shuffle phase consists of the following steps, which are shown in Figure 2.8:

1. **Buffering:** The intermediate key-value pairs produced in the map tasks are buffered in memory and periodically written to disk. In Hadoop there are thresholds both on the size and the number of pairs that determine when to flush this information.
2. **Spilling:** To flush the buffer, the intermediate pairs are written to disk in a file called *spill*. In each spill, the key-value pairs are partitioned into $R$ fragments which are sorted and then stored. Notice that $R$ is the number of reduce tasks.
3. **Merging spills:** All spill files generated by the same map tasks are merged into $R$ files, one per reduce task, before communicating the success to the master.
4. **Copying map outputs to reduce tasks:** The master forwards the information about the location of the map tasks outputs to the reduce tasks, which then fetches this information through remote procedure calls (HTTP protocol in Hadoop).
5. **Merging map outputs:** The files coming from different map tasks are merged and sorted by intermediate key. Then, the workers iterate over the intermediate key-value pairs. For each unique key found, a worker passes the key and the set of values to the reduce function defined by the user.

MapReduce also provides the possibility of optimizing the shuffle phase by providing an additional function, called *combiner*:

$$combine : (\mathcal{K}_2, list(\mathcal{K}_2, \mathcal{V}_2)) \rightarrow list(\mathcal{K}_2, \mathcal{V}_2)$$

This function is employed to reduce the amount of I/O when flushing the buffer to disk and to reduce the amount of data transferred through the network. It may be applied in steps 2 and 3. When the reduce function is commutative

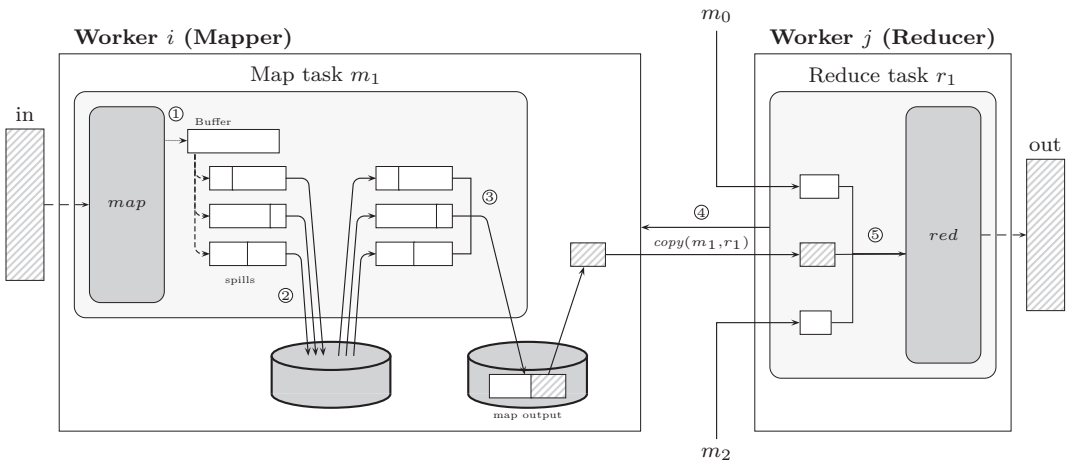



**Figure 2.8:** Shuffle phase in Hadoop

and associative, which is a common scenario, the same function can be employed for both combine and reduce operations.

### Fault-tolerance

In big data centers, where MapReduce jobs are executed, failures are the norm rather than the exception. In order to deal with failures, MapReduce incorporates a powerful fault-tolerance mechanism to deal with and recover from those failures.

The failure of workers is controlled by periodic communication between the master and each of the workers. When no message is received from a worker for a given amount of time, it is considered as failed. All map tasks finished in that worker and all map and reduce tasks that were in progress are marked as uncompleted and can be scheduled again. Note that completed map tasks need to be re-executed because their output is needed by the reduced tasks and is only kept in the worker's local disk. When the new map task is finished, reduce tasks should be informed in order to retrieve the data from the new assigned worker instead of from the failed one.

The failure of the master in some implementations like Hadoop is not supported. In this case, all running jobs are aborted. The main reason is that, provided that there is a single master, the probability of its failure is unlikely. In any case, in Google's implementation, the master makes periodic checkpoints to GFS. If it fails, a new master can be started from the last stored checkpoint.

MapReduce is also capable of dealing with very slow workers, called *stragglers*. When a job is near the end, i.e., all the tasks have been completed or are in progress, replicas of the tasks in progress are also scheduled. This is known as *speculative execution*. In that way, if a task in progress is too slow because it is being executed in a straggler, the backup task may complete before and reduce

the total response time of the job. In [43], an example is given where this approach improves job response time by 44%.

Finally, MapReduce also supports failures in the processing of individual records. This can be originated from bugs in the program or a malformed input which only affects a small part of the data. Whenever a failure is detected in the processing of a record, its sequence number is sent to the master along with the error information. If more than a given threshold of failures are detected in the same record, new instances of that task will be instructed to skip it, thus allowing the computation to ignore this particular case and continue with the rest of the computation.

### 2.3.2 Implementation Details

#### Hadoop and Other Implementations

Apache Hadoop [19] is the most popular implementation of MapReduce. Its core provides an open-source implementation of Google's MapReduce [43] and Google File System [60], denoted respectively Hadoop MapReduce and Hadoop Distributed File System (HDFS) [112]. Nevertheless, this only represents the kernel of Hadoop's platform, which also includes other projects such as Hadoop YARN [20], a framework for job scheduling and resource management; Pig [94], a high-level dataflow language; Hive [120], a data-warehousing infrastructure with an SQL-like interface; HBase [21], a distributed database that mimics Google's BigTable [33], ZooKeeper [73], a coordination service which shares the same goals as Google's Chubby [30] but uses another protocol, etc.

Hadoop MapReduce is very close to the details given in Google's paper [43], only differing in the terminology, e.g., the master is known as the *jobtracker* and the workers as *tasktrackers*, and small details, e.g., reduce tasks fetch the map outputs using the HTTP protocol instead of remote procedure calls. Of course, since Google's paper is not exhaustive, many engineering decisions have been taken on their own and many enhancements have been included.

Yet, there are many other implementations of MapReduce like Amazon Elastic MapReduce [11], which runs on top of Amazon EC2 [10] and Amazon S3 [13]; Greenplum's [103], which is integrated with their RDBMS in a single parallel dataflow engine; or Aster MapReduce Appliance [23], which is closely tied to SQL, among others. Many other products have also incorporated the possibility of plugging Hadoop MapReduce (or other implementations for that matter) into their big data analytic engines, e.g., [58, 96, 117].

#### Input and Output

MapReduce supports different formats and sources for input and output data, as required by the user. Although reading data from files in a distributed file

system, e.g., GFS in Google's implementation, is the most common usage, it can also be read from databases, data structures in main memory, etc. The input interface needs to implement how to divide the input data into several splits, one per map, and how to obtain key-value pairs from those splits.

In Hadoop, the `InputFormat` interface is used for that matter. It is composed of two methods:

- `getSplits()`: this method defines the set of splits in which the input is divided. It receives the number of desired splits as an hint and produces a set of `InputSplit` instances with the information about the split, which at least includes their length and locations. This information is used by the scheduler to assign map tasks to workers.
- `getRecordReader()`: returns an implementation of the `RecordReader` interface. This class is responsible of parsing the input data and generating the key-value pairs that are passed to the map function.

The user is allowed to define its own implementations of the input readers, but some classes are provided for the most common situations. For instance, implementations for reading data from files (`FileInputFormat`) and databases (`DBInputFormat`) are provided. In the first case, several formats are supported, mainly:

- **Text files**: The input is given in text files. By default Hadoop uses `TextInputFormat`, which generates a pair for each file line in which the key refers to its offset and the value contains the text of the line. Other implementations are also supplied, for instance `KeyValueTextInputFormat`, where each line contains a key and a value separated by a tab.
- **Sequence files**: The input is given binary files which contain a set of serialized key-value pairs. They support compression as part of the format, allowing the usage of arbitrary types and are generally more efficient. In many cases, the user just needs to provide mechanisms to serialize and deserialize the used types.

Note that splits are generated from the HDFS chunks. When key-value pairs cross the boundaries of the chunks, the `InputFormat` is able to obtain the rest of the pair's data by performing remote reads on the next chunk.

Hadoop also allows to get the input data from different sources and formats at the same time, using `MultipleInputs`. A specific format is assigned to each of the given input paths.

In the same way, different formats can be used and defined for the output, and the user can define the format of the job output. The default is `TextOutputFormat`, in which each key-value pair is written as a text line, where the key and the value are separated by a tab.

#### Scheduling

In Hadoop, scheduling is done in two steps: first a job needs to be selected to run and then a task within the job is assigned to one of the tasktrackers. Early versions of Hadoop included a FIFO scheduler for clusters in which jobs are executed in submission order and employ the whole cluster to perform the computations. Later, priority was added to the scheduler, but without preemption, thus jobs still have to wait for executing jobs to finish. From version 0.19, new schedulers were added to the system and can be chosen by the administrator. For instance, in the *Fair Scheduler*, jobs are grouped into pools and assigned a fair share of the cluster capacity. The whole scheduling architecture has been changed in version 2 with the inclusion of Yarn [20].

Each tasktracker in the cluster has a set of slots that can run map and reduce tasks, which can be configured by the administrator (the rule of thumb is to define as many slots as cores in the node). Periodically, the tasktracker sends heartbeat messages to the jobtracker, so that the latter can detect when failures occur. In the message, the tasktracker also reports the number of idle slots. The jobtracker selects a job and a task within the job. For map tasks, it first tries to assign local tasks, i.e., tasks whose input is replicated on the node, or rack task if local task assignment is not possible. In the case of reduce tasks, only one reduce task is scheduled at a time with no locality constraints. In both cases, if there are failed tasks, they have priority over the other tasks. Finally, if there are no tasks waiting to be scheduled, speculative tasks can be considered.

### 2.3.3 Limitations and Main improvements

There has been an intense debate around MapReduce and its comparison with parallel databases. Pavlo et al. [101] compare the performance of Hadoop with an unrevealed parallel database management system (DBMS) and Vertica, concluding that the performance of MapReduce was 2 to 50 times slower except for the case of data loading. Later, both parts have the opportunity to explain themselves in the Communications of the ACM [44, 115]. Anderson et al. [18] also criticized data intensive scalable computing (DISC) in general for having poor efficiency, and particularly focused on MapReduce as it the most of popular of such systems.

Among the critiques that MapReduce has received, the following are the most serious ones:

- It has no high-level language and schema support and this forces the developers to incorporate all that functionality into their programs. The need to parse the data every time it is read may also add a significant overhead. Nonetheless, in [76] it is argued that the main responsible of this overhead is immutable decoding, which requires extensive CPU utilization, as the framework needs to create a new object for each key-value pair in

the input.

- It does not incorporate compression, and when it used in an ad-hoc way, it offers no performance gains [101]. However, parallel DBMS can particularly benefit from it, reducing total I/O.
- It does not include traditional database optimizations such as indexes or column-based layouts, which can significantly reduce the total amount of data that has to be read in order to answer a query.
- It has large start-up and scheduling costs. As opposed to parallel DBMS, where a query plan is first designed and sent to all participant nodes, MapReduce employs online scheduling, which requires the interchange of control messages between the computers, which incurs an important overhead. Moreover, the lack of a predefined pipeline excludes some performance optimizations used in parallel DBMS, such as reducing the amount of transferred data or pipeline parallelism.
- The shuffle phase is inefficient. The materialization of the intermediate data in $M \times R$ files and the usage of a pull model may put a lot of pressure on the disk, which has to perform a lot of seeks [101]. Moreover, the sort-merge scheme used to group the keys is not always needed and more efficient strategies may be used in some cases [76].

Dean et al. [44] pointed out that some of the critiques are actually misconceptions about the system. MapReduce is a specialized tool for performing possibly complex analysis in heterogeneous systems, were data may be stored in many formats. The programming model is also focused on their specific use cases, as complicated transformations are easier to express in MapReduce than in SQL. Some of the mechanisms that are claimed to be lacking, such as indices, can actually be used in the system, and the critiques about the performance made assumptions about the way the programs and the framework are implemented, e.g., using text formats, start-up costs. Other issues related to performance, such as materialization of intermediate results, are actually a choice aiming at providing fault-tolerance for job executions. Nonetheless, some of the limitations are still to be addressed and a lot of research effort have focused on such task.

In the rest of the section, we explain in detail some proposals that have been presented in order to overcome MapReduce's limitations and incorporate additional functionalities.

### Extended Programming Model

Sawzall [102] is a scripting language proposed by Google that is executed over a MapReduce framework and simplifies the implementation of analytical applications. It also consists of two phases, a first phase that consumes single records and emits values to external aggregators, and an aggregator phase. These phases match map and reduce phases of a MapReduce job. Protocol buffers are

used to specify a schema on the input records, as each type is defined through a data definition language (DDL) file.

Several other declarative languages have been also proposed to be used on top of the MapReduce framework. A notorious example is Apache Pig [94], an environment built on top of Hadoop that uses a high-level language to express MapReduce queries. Jobs are expressed as a sequence of operations that perform single, high-level relational algebra-style transformations such as filter, join, group by, etc. An optimizer is in charge of building a logical query plan that is compiled into a chain of MapReduce jobs. Pig supports a nested data model and offers a set of pre-defined user-defined functions (UDFs) for the most common operations. Another example is Hive [120], a data warehousing solution built also on top of Hadoop. It defines a declarative language similar to SQL called HiveQL. The data model organizes data into tables, partitions and buckets. All the meta information is stored in a system catalog. Queries are compiled into directed acyclic graphs (DAGs) of MapReduce jobs and query optimizations are also employed in the process. Similar to Hive, SCOPE [132] defines a SQL-like declarative scripting language that incorporates C# expressions. Queries are transformed into job plans that are executed in Microsoft's distributed computing platform, called Cosmos, which is similar to MapReduce. Clydesdale [78] is a research prototype built on top of Hadoop that focuses on workloads for which the data model fits a star schema, i.e., a fact table and several dimension tables. It implements several optimizations for this specific type of queries and will provide a SQL parser and compiler in order to transform them into the corresponding MapReduce jobs.

MapReduce's programming model has also been extended beyond the inclusion of a declarative language interface. For instance, Hadoop Online Prototype (HOP) [37], a modified version of Hadoop, is able to pipeline data between map and reduce tasks and between different jobs. This allows the modified framework to support online aggregation and continuous queries. The pipeline parallelism between map and reduce tasks is achieved by sending the spills produced when the map output buffer is full, directly to the corresponding reduce tasks, provided they can keep up with the rate. Pipeline between jobs is carried out by sending directly the output of the reduce tasks to the map tasks of the next job in the chain, avoiding the expensive writes in HDFS. HaLoop [29, 28] extends the original Hadoop framework in order to support iterative programs. MapReduce API is modified to express iterative jobs. This allows the system to apply some optimizations like inter-iteration locality, by which map and reduce tasks accessing the same data in different iterations are assigned to the same nodes. HaLoop takes advantage of caching and indexing of loop-invariant data or caching the reducers output data in order to accelerate the evaluation of fixpoint termination conditions.

### Database Optimizations

As a consequence of Pavlo et al.'s paper [101] pointing out the weaknesses of MapReduce when compared to parallel databases, many works started to incorporate database techniques into the MapReduce framework. HadoopDB [2] was one of the first proposals in this direction. It consists of a hybrid approach where the data is stored in a set of databases that are connected using Hadoop. The databases replace the data nodes in the distributed file system and allow the system to fetch the data more efficiently, as database optimizations are used in the process. The interface between the different systems is handled by the SMS (SQL to MapReduce to SQL) planner, which receives queries in SQL and translates them into query plans by extending Hive [120].

Hadoop++ [50] takes another approach. The execution plan is mapped to a physical query execution plan and all the operators identified. In this way, many other functions apart from map are reduce are identified and formalized including 10 UDFs. In Hadoop++, two main database-inspired optimizations are incorporated and evaluated. The first one, called Trojan index, consists in incorporating an index on each of the chunks, thus reducing the amount of reads that need to be performed. The second, called Trojan join, is aimed for efficient join processing. The idea is to co-partition data at load time by applying the same partitioning function on the join attributes of the two relations involved in the join and to store the groups with the same key in the same chunk. An extension of Hadoop++ enriches this framework by including Trojan data layouts [77], which organize each chunk into attribute groups that depend on the workload. In that way, attributes that are frequently accessed together in the workload are assigned to the same group. By incorporating an index into each of the chunks only the needed groups are accessed and I/O operations are reduced. They compare their approach to other traditional approaches such as horizontal, columnar and PAX (a data organization model that group all the values of an attribute within each cache page [8]) layouts and show that the proposed approach is more efficient.

In [68], as in [77], the three possible traditional data placement structures, namely horizontal row store, vertical row store and PAX, are used and studied in the context of MapReduce. They propose RCFile, a data placement structure for MapReduce-based data warehouses that is based on PAX. Each chunk is divided into several row groups. In each row group, values for the same attribute are stored contiguously. Then, the contents are compressed, optionally using different algorithms for different columns. Although RCFiles and Trojan data layouts use related approaches and claim to work better than the alternatives, they are not well compared. In [77] the PAX approach is used in the comparison, but is not implemented through row stores and uses data compression as in [68]. On the other hand, column groups are used in RCFile's evaluation, but their are not used in a PAX-like approach. In [54], RCFile's weaknesses are pointed out:

namely that I/O elimination is limited by HDFS and file system's prefetching mechanism and that it imposes a storage overhead due to metadata. As an alternative, they propose to partition the dataset into several splits and store the data from each column in a different file. In order to guarantee data locality when accessing multiple columns, the HDFS block placement policy is modified in order to automatically co-locate the column files of each split. An additional advantage of this approach is that the addition of columns to the dataset is straightforward, as it only requires to add a new file for each of the splits.

CoHadoop [52] shares the same objective as Trojan joins [50], that is, to co-locate the related data in the same nodes. However, the strategy is similar to that of [54]. While Hadoop++ is a static approach, which needs to reorganize the data for co-location in the same job, CoHadoop allows to incrementally co-locate files as new data arrives. This is carried out by modifying the HDFS placement policy and including the concept of write affinity, in an analogous way as in [54]. Data from different files is partitioned using the same partitioning function. The set of created chunks are co-located by assigning the same locator, which will force HDFS to store them in the same nodes.

### Shuffle Optimizations

The shuffle phase of a MapReduce job may incur an important part of total execution time [127, 98]. In [127], through simulation, it is shown that the network topology has a significant impact on the shuffle overhead. Four topologies are studied: star, double rack, tree and DCell. Double rack, followed by tree are the worst case scenarios, as they comprise network links that are shared by multiple communications. Unfortunately, the typical topology in MapReduce clusters is a generalization of double rack, as there are several racks where computers are connected in a star schema, but communications between nodes in different racks have to pass through a shared interconnect. In [98], reduce locality is pointed out as a significant issue in MapReduce jobs, showing through examples that a bad reduce locality can multiply response time by four.

By default, MapReduce uses hash partitioning to distribute intermediate keys among reduce tasks. In a configuration with $N$ nodes, this implies a reduce data locality of only $\frac{1}{N}$, e.g., just for 10 nodes 90% of the intermediate data is transferred through the network. As a result, there have been some works which have tried to improve data locality and decrease the overhead of the shuffle phase. In [111], a pre-shuffling scheme is proposed to reduce data transfers in the shuffle phase. A modified scheduler looks over the input splits before the map phase begins and predicts the reducer the key-value pairs are partitioned into. Then, the data is assigned to a map task near the expected future reducer. Similarly, in [67], reduce tasks are assigned to the nodes that reduce the network transfers among nodes and racks. However, in this case, the decision is made at reduce

scheduling time. There is a tradeoff on when to start the scheduling reduce tasks. As more map tasks have finished, the information about data locality is more accurate, however, the possible benefits of parallelizing data transfer and map execution are lost (early shuffling). A parameter is incorporated into the scheduler that accounts for that tradeoff. It has to be chosen statically for each application.

The limitations of the mentioned approaches is that even if all intermediate pairs are produced in the same node, the partitioning function may force them to be separated into several reduce tasks, reducing the possibilities of locality-aware scheduling. In [74], this problem is addressed by assigning intermediate keys to reducers at scheduling time. The number of reduce tasks is set to the number of nodes. Then, when all intermediate pairs have been produced, the scheduling algorithm assign intermediate keys to reduce tasks with a greedy algorithm. This algorithm incorporates both data locality and load balancing awareness, so that enforcing data locality does not produce skewed reduce tasks. However, it still depends in the distribution of intermediate keys in the map outputs. If intermediate pairs with the same key are uniformly produced at all workers, the possible gains of this approach are not significant.

#### Load balancing

Skew may have a significant impact on the response time of MapReduce jobs, both in map tasks, since the reduce function cannot start until all maps have finished, and in reduce tasks, as the response time is determined by the last finished reduce task. The original implementation of MapReduce tries to overcome the skew produced by stragglers by means of speculative execution of tasks [43]. Some works have tried to improve this approach by considering some other issues. For instance, in [131], the way in which speculative tasks are scheduled is modified in order to account for workers with different characteristics (heterogeneous clusters) and overcome some of the limitations of the original MapReduce's approach. The outcome is a new scheduling algorithm called Longest Approximate Time to End (LATE). This algorithm selects the task estimated to finish the latest for speculative execution. The termination time is estimated by taking into account the processing rate and the quantity of work that remains to be done. Moreover, the worker selected to execute this task need to be one of the fastest nodes, which is guaranteed by establishing a threshold on the processing speed with respect to the average speed.

Divergences on the performance of workers is not the only source of skew, as shown in [89]. Many phenomena follow Zipfian distributions, where a few elements are extremely common while there is a long tail of rare elements, and this also holds in MapReduce computations. Thus, other sources of skew have handled.

In Mantri [17], a more complex approach is used. Three types of skew are identified by analyzing the jobs executed in a production cluster: 1) data skew, due to unequal assignment of work to tasks; 2) cross-rack traffic, which makes some tasks waste a lot of time in order to fetch the input data; and 3) bad and busy machines, which is the case treated in [43, 131]. Depending on the cause, a different measure is taken: ranging from the speculative approach taken in previous approaches, to network-aware placement, scheduling of larger task first or replicating output to avoid re-computation in case of failures.

There are still other sources of skew that are not solved by Mantri or speculative execution. In [85], four additional cases are identified: In the map phase, skew may be produced due to 1) expensive records, which require more CPU or memory, or 2) heterogeneous maps. In the reduce phase, 3) the partitioning function may create uneven partitions or 4) some of the created groups be expensive, i.e., contain a lot of values. The proposed solution identifies the stragglers and repartitions the remaining unprocessed data into several fragments, which are forwarded to other workers to be processed there. Straggler detection is postponed until there are no tasks to occupy idle workers. If the estimated remaining time is at least twice the overhead of repartitioning, the procedure is triggered; otherwise, the task continues normally. Range-based repartitioning based on a scanning of the remaining data is used to determine the partitioning intervals.

Other approaches focus on the skew in reduce execution. In [74] individual intermediate keys are assigned to reduce tasks at scheduling time. A greedy algorithm assigns the keys to the nodes based on both data locality and fairness. For this, the map tasks collect information about the frequency of the intermediate keys, which is sent to the master and aggregated. Once all map tasks have finished, all the collected information is used by the greedy algorithm.

Gufler et al. [65] improve this approach in two ways: the complexity of the reduce function is incorporated into the cost and a threshold on the number of keys for which statistics are collected is introduced. The cost of executing the reduce function may be super-linear to the size of the groups in the reduce task.Therefore, even for the same number of values, groups of different sizes may require different times [66]. For instance, if the reduce function has a complexity in the order of $O(n^2)$, a reduce task with one group of $2n$ values will take more time to execute than a reduce task with two groups of just $n$ values. As a consequence, the size of the groups is taken into account in the cost model. And, provided that big groups contribute a lot more to the total cost, the statistics collected on each map phase about the cardinality of each intermediate key can be approximated by only taking into account the groups bigger than a given threshold and estimating the size of the rest.

In [130], a sampling MapReduce job is executed in advance to obtain an estimation of the key's frequency distribution. This information is used to create a partitioning scheme that is later used in the execution of the original job.

This strategy is also used in the Pig framework [94] to adjust the partitioning function when executing some operators like sort. A sampling approach has also been incorporated into the Oracle Loader for Hadoop (OLH) in [104]. The map output is sampled by applying the map function to samples of the input data. The size of the sample is calculated dynamically, i.e., the sampling process is finished when the desired quality is obtained according to a model. Then, keys are classified into three types: large, medium and small. The framework chops large keys into several medium keys that are smaller than the average reduce load. Chopping is only allowed if the reduce function is distributive. If it is distributive over the union, hash based chopping is used; if it is only distributive over the concatenation, range-based partitioning has to be used instead. Medium keys are assigned to reducers using a greedy bin-packing algorithm. This assignments is passed to map tasks through a partition file. Small keys are assigned to reducers via hashing.

In [123] an adaptive solution is proposed in order to overcome the skew both in the map and reduce phases. In order to achieve that, mappers are allowed to asynchronously communicate through a transactional, distributed meta-data store. This allows them to have a global view of the system and make coordinated optimization decisions. Map phase skew is handled by allowing map tasks to dynamically load input splits. This strategy eliminates the overhead of map tasks start-up and allows to have smaller input splits that are better for load balancing. Map tasks also sample the map outputs and update a global view in the meta-data store, allowing to adapt the intermediate key partitioning in order to deal with reduce skew.

## 2.4 Conclusions

As shown in this chapter, data partitioning is a fundamental technique for parallel computing, both in parallel databases and massively parallel processing frameworks such as MapReduce. It is used in combination with other techniques also covered in this chapter, such as replication and load balancing.

In parallel databases, data partitioning is used in order to divide the data collections into fragments and assign them to the nodes that participate in the processing of queries. Parallelism can be obtained with the concurrent execution of several queries and/or with the parallel execution of a single query at several nodes. Depending on the application, the objective of the partitioning is different. For instance, when small queries are executed, the partitioning tries to confine their execution at one or a few nodes in order to avoid the overhead of distributed execution. On the other hand, when the application consists of long running queries, the objective of the partitioning is to spread the data items accessed by queries at all the nodes in a balanced way so that total response time is minimized.

As the workload becomes more complex, the design of an efficient partitioning becomes more difficult, as many alternatives need be considered. Several partitioning advisors have been designed in order to automatically partition the database in the most efficient way, given a workload model. They make use of the basic partitioning techniques, mainly hash-based and range-based partitioning, and select the partitioning attributes that are more adequate. Graph-based partitioning techniques represent another alternative for automatic data partitioning which can work with any schema and regardless of the complexity of the queries in the workload.

In MapReduce data partitioning is employed in order to split the input data and assign the fragments to the tasks that execute in parallel the map phase of the job. Afterwards, intermediate data is again repartitioned and transferred to the reduce tasks, which execute the second phase of the MapReduce job. While data locality is taken into account in the execution of the map phase, it is not considered in the reduce phase. Several works have focused on overcoming this limitation, as well as other inefficiencies of the original MapReduce framework. However, some scenarios are yet to be optimized.

We identified two main limitations of the current works related to data partitioning. In this thesis, we address these limitations:

- **Partitioning in continuously growing scientific databases:** We consider scientific databases composed of tables with a large number of attributes and complex access patterns where new data items are inserted as new observations are performed. Automatic approaches based on the basic techniques, namely hash-based and range-based partitioning, are not capable to deal with the complex queries of scientific applications. Moreover, the search space is huge as a lot of attributes should be considered. Graph-based approaches, not being affected by the schema complexity, can be able to capture tuple relations. However, they require a whole repartitioning each time new data items are appended. As a consequence, we need a partitioning approach that is able to efficiently adapt the partitions to the continuous arrival of new data items. We cover this problem in Chapter 3.
- **Expensive data transfers in MapReduce's shuffle phase:** As shown in Section 2.3.3, the shuffle phase in MapReduce may produce a significant overhead on job execution because of expensive network transfers. Some works have tried to overcome this problem by modifying the way in which the MapReduce framework schedules reduce tasks and how intermediate keys are assigned to those tasks. Nevertheless, if the intermediate keys are generated uniformly on the map outputs, the gains of those approaches are not considerable. In Chapter 4, we propose a strategy to repartition input data in order to take full advantage of intelligent scheduling of intermediate keys and reduce network transfers in MapReduce execution.

# Chapter 3

# Dynamic Partitioning for Continuously Growing Databases

In this chapter, we address the problem of dynamic data partitioning in large databases where data is continuously appended. As a motivating example, we illustrate the problem with astronomical catalogs, composed of tables with a large number of attributes which are accessed through complex queries and which grow as new observations are performed. Traditional automatic partitioning approaches are limited due to the number of attributes and workload complexity; and graph-based approaches, which are able to capture those relationships, require the recomputation of the partitioning from scratch each time new data is inserted into the database. To overcome those limitations, we propose two partitioning algorithms that dynamically assign new arriving data elements using the affinity of new data with queries and fragments. The problem is formalized in Section 3.2 and the algorithms described in Sections 3.3 and 3.4. Then, we evaluate the proposed algorithms and compare them to traditional graph-based approaches to show their effectiveness.

## 3.1 Motivation and Overview of the Proposal

We are witnessing the proliferation of applications that have to deal with huge amounts of data. The major software companies, such as Google, Amazon, Microsoft or Facebook have adapted their architectures in order to support the enormous quantity of information that they have to manage. Scientific applications are also struggling with those kinds of scenarios and significant research efforts are directed to deal with it [7]. An example of these applications is the management of astronomical catalogs; for instance those generated by the Dark Energy Survey (DES) [118] project with which we are collaborating. In this project, huge tables with billions of tuples and hundreds of attributes (corresponding to dimensions, mainly double precision real numbers) store the collected sky data. Data

is appended to the catalog database as new observations are performed and the resulting database size is estimated to reach 100TB very soon. Scientists around the globe can access the database with queries that may contain a considerable number of attributes.

The volume of data that such applications hold poses important challenges for data management. In particular, efficient solutions are needed to partition and distribute the data in multiple servers, e.g., in a cluster, in order to optimize query execution, specially small queries accessing a small part of the dataset. An efficient partitioning scheme would try to minimize the number of fragments that are accessed in the execution of a query, thus minimizing the overhead of the distributed execution. Vertical partitioning solutions may be useful for physical design at each node, but fail to provide an efficient distributed partitioning, in particular for applications with high dimensional queries, where joins would have to be executed by transferring data between nodes. Traditional horizontal partitioning approaches, such as hashing or range-based partitioning, are unable to capture the complex access patterns present in scientific computing applications, especially because these applications usually make use of complicated relations, including mathematical operations, over a big set of columns, and are difficult to be predefined a priori.

One solution is to use partitioning techniques based on the workload. Among them, graph-based partitioning is an effective approach for that purpose as it is independent of the schema and workload complexity [41]. As explained in Section 2.2.4, a graph (or hypergraph) representing the relationships between queries and data elements is built and the problem is reduced to that of minimum k-way cut problem. However, this method requires to process the entire graph in order to obtain the partitioning. This strategy works well for static applications, but scenarios where new data is inserted to the database continuously, which is the most common case for scientific computing, introduce an important problem. Each time a new set of data is appended, the partitioning should be redone from scratch, and as the size of the database grows, the execution time of such operation may become prohibitive. Moreover, since the graph model does not take into account previous data placements, a lot of data transfers may have to take place to enforce the new partitioning.

In this chapter, we are interested in dynamic partitioning of large databases that grow continuously. After modeling the problem of data partitioning in dynamic datasets, we propose two dynamic workload-based algorithms, called *DynPart* and *DynPartGroup*, that efficiently adapt the partitioning to the arrival of new data elements. Our algorithms are designed based on a heuristic that we developed by taking into account the affinity of new data with queries and fragments. In contrast to the static workload-based algorithms, the execution time of our algorithms do not depend on the total size of the database, but only on that of the new data and this makes them appropriate for continuously growing

databases.

We validated our solutions through experimentation over real-world data sets. The results show that they obtain high performance gains in terms of partitioning execution time compared to one of the most efficient static partitioning algorithms. We also compared both algorithms and concluded that the grouping strategy of *DynPartGroup* obtains better partitioning efficiencies and performs better, specially in scenarios with high correlation between new data items and strict imbalance constraints. *DynPartGroup* is a variation of *DynPart*, which groups data items before calculating fragment affinities. This strategy adapts better for the situations where there is high correlation on the new data items and the imbalance constraints (maximum allowed imbalance) are strict, and offers an improved performance.

## 3.2 Problem Definition

In this section, we state the problem we are addressing and specify our assumptions. We start by defining the problem of static partitioning, and then extend it for a dynamic situation where the database can evolve over time.

### 3.2.1 Static Partitioning

The static partitioning is done over a set of *data items* and for a *workload*. Let $D = \{d_1, ..., d_n\}$ be the set of data items. The workload consists of a set of queries $W = \{q_1, ..., q_m\}$. We use $q(D) \subseteq D$ to denote the set of data items that a query $q$ accesses when applied to the data set $D$. Given a data item $d \in D$, we say that it is *compatible* with a query $q$, denoted as $comp(q, d)$, if $d \in q(D)$. Queries are associated with a relative frequency $f : W \rightarrow [0, 1]$, such that $\sum_{q \in W} f(q) = 1$.

Partitioning of a data set is defined as follows.

**Definition 3.2.1.** Partitioning of a data set $D$ consists of dividing the data of $D$ into a set of fragments, $\pi(D) = \{F_1, ..., F_p\}$, such that there is no intersection between the fragments, $\forall i \neq j : F_i \cap F_j = \emptyset$, and the union of all fragments is equal to $D$, i.e., $\bigcup_{i=1}^{p} F_i = D$.

Let $q(F)$ denote the set of data items in fragment $F$ that are compatible with $q$. Given a partitioning $\pi(D)$, the set of *relevant fragments* of a query $q$, denoted as $rel(q, \pi(D))$, is the set of fragments that contain some data accessed by $q$, i.e., $rel(q, \pi(D)) = \{F \in \pi(D) : q(F) \neq \emptyset\}$.

To avoid a high imbalance on the size of the fragments, we use an *imbalance factor*, denoted by $\epsilon_s$. The size of the fragments at each time should satisfy the following condition: $|F| \leq \left\lceil \frac{|D|}{|\pi(D)|}(1 + \epsilon_s) \right\rceil$.

In this chapter, we are interested in minimizing the number of query accesses to fragments. Note that the minimum number of relevant fragments of a query $q$

is $minfr(q, \pi(D)) = \left\lceil \frac{|q(D)|}{(|D| \,/\, |\pi(D)|)(1+\epsilon_s)} \right\rceil$. We define the *efficiency of a partitioning* for a workload based on its efficiency for queries. Intuitively, the *efficiency of a partitioning for a query* represents the ratio between the minimum number of relevant fragments of $q$ and the number of fragments that are actually accessed under the given partitioning:

**Definition 3.2.2.** Given a query $q$, then the efficiency of a partitioning $\pi(D)$ for $q$, denoted as $eff(q, \pi(D))$ is computed as:

$$eff(q, \pi(D)) = \frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|} \tag{3.1}$$

When the number of accessed fragments is equal to the minimum possible, i.e., $minfr(q, \pi(D))$, the efficiency is 1.

Using $eff(q, \pi(D))$, we define the efficiency of a partitioning $\pi(D)$ for a workload $W$ as follows.

**Definition 3.2.3.** The efficiency of a partitioning $\pi(D)$ for a workload $W$, denoted as $eff(W, \pi(D))$, is equal to the sum of the efficiencies of partitioning $\pi(D)$ for all queries in $W$ multiplied by their relative frequencies. In other words,

$$eff(W, \pi(D)) = \sum_{q \in W} f(q) \times eff(q, \pi(D)) \tag{3.2}$$

Given a set of data items $D$ and a workload $W$, the goal of static partitioning is to find a partitioning $\pi(D)$ such that $eff(W, \pi(D))$ is maximized.

### 3.2.2 Dynamic Partitioning

Let us assume now that the data set $D$ grows over time. For a given time $t$, we denote the set of data items of $D$ at $t$ as $D(t)$ [1].

During the application execution, there are some events, namely *data insertions*, by which new data items are inserted into $D$. These events in the model correspond to the appending of the tuples corresponding to new observations in the DES catalog. No changes in the schema are involved. Let $T_{\text{ev}} = (t_1, \ldots, t_m)$ be the sequence of time points corresponding to those events. Note that between two consecutive time points $t_i$, $t_{i+1}$, $D$ remains constant. In this chapter, we assume that the workload is stable and neither the queries nor their frequencies change. However, the queries may access new data items as the data set grows.

Let us now define the problem of dynamic partitioning as follows. Let $T_{\text{ev}} = (t_1, \ldots, t_m)$ be the sequence of time points corresponding to data insertion events; $D(t_1), \ldots, D(t_m)$ be the set of data items at $t_1, \ldots, t_m$ respectively; and $W$ be a

1. We confine this formulation to this subsection for the sake of simplicity, so that, in the next sections, when we use $D$ we mean $D(t_i)$.

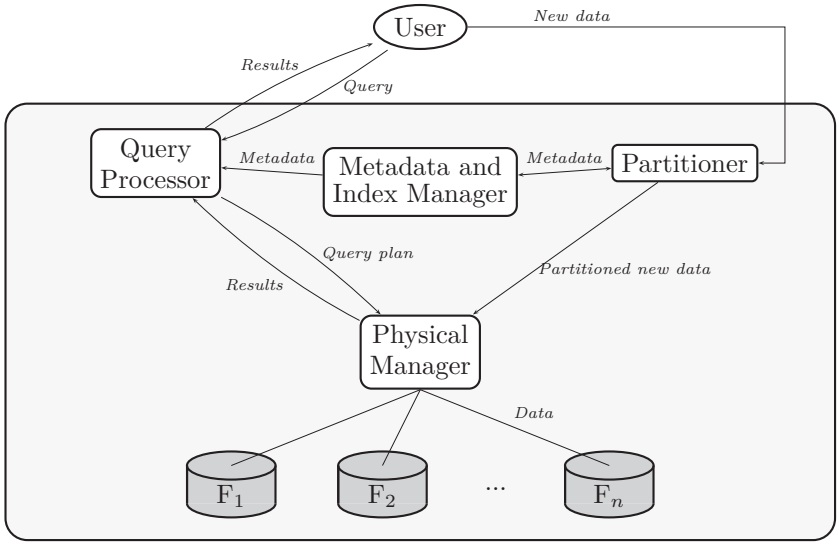



**Figure 3.1:** System architecture

given workload. Note that, as we only consider data insertions, if $t_i < t_j$ then $D(t_i) \subset D(t_j)\ \forall t_i, t_j \in T_{ev}$.

The goal is to find a set of partitionings $\pi(D(t_1)), ..., \pi(D(t_m))$ for data sets $D(t_1), \ldots, D(t_m)$ respectively, such that the sum of the efficiencies of the partitionings for $W$ are maximized. In other words, our objective is as follows:

$$maximize\left(\sum_{q \in W} (f(q) \times \mathit{eff}(q, \pi(D(t))))\right) \forall t \in T_{ev}$$

## 3.3 Affinity Based Dynamic Partitioning

In this section, we propose an algorithm, called *DynPart*, that deals with dynamic partitioning of data sets. It is based on a principle that we developed using the partitioning efficiency measure described in the previous section.

### 3.3.1 System Overview

In this chapter, our proposal mainly focuses on how the data is partitioned in fragments. Here, we provide an overview of a system architecture taking advantage of our partitioning approach. The components of this architecture are as follows (see Figure 3.1):

- **Query processor:** It parses the user queries, accesses the metadata and index manager, prepares an optimized execution plan and sends it to the physical manager to retrieve the data from fragments.
- **Metadata and Index Manager:** Stores metadata about the partitioning, and also indexes the location of the data items in the fragments.

- **Physical Manager:** It is in charge of storing/retrieving data to/from fragments.
- **Partitioner:** It holds the data items until a given number of items is inserted. Then, it obtains the necessary metadata and executes the partitioning algorithm. Finally, it transfers the data items to the corresponding fragments and informs the metadata and index manager about the modifications in the fragments. This component may also be contacted to include in the query results the corresponding data items in new added data.

We assume a shared nothing architecture composed of data nodes containing a physical data manager that stores one or several fragments at each node, and dedicated nodes for other components. We use a shared nothing architecture as it is the most common one since it is cheaper and can be scaled easily when required. The query processor and the metadata and index manager are preferred to be executed at the same node (nodes) to avoid communication overhead, as the query processor always has to access the index.

### 3.3.2 Principle

Let $d$ be a new inserted data item. We can express the efficiency of the new dataset as:

$$\mathit{eff}(W, \pi(D \cup \{d\})) = \mathit{eff}(W, \pi(D)) + \Delta \tag{3.3}$$

Let assume that $F$ is the fragment selected to insert $d$. The efficiency will remain the same for all queries but those which now have to access $F$ in order to retrieve $d$ but did not before. Hence, we can calculate $\Delta$ as [2]:

$$\Delta \approx \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\left(\mathit{eff}(q, \pi(D \cup \{d\})) - \mathit{eff}(q, \pi(D))\right) \tag{3.4}$$

$$= \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\left(\frac{\mathit{minfr}(q, \pi(D))}{|\mathit{rel}(q, \pi(D))| + 1} - \frac{\mathit{minfr}(q, \pi(D))}{|\mathit{rel}(q, \pi(D))|}\right) \tag{3.5}$$

$$= - \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\frac{\mathit{minfr}(q, \pi(D))}{|\mathit{rel}(q, \pi(D))|\,(|\mathit{rel}(q, \pi(D))| + 1)} \tag{3.6}$$

where $q : q(F) = \emptyset \wedge \mathit{comp}(q, d)$ is the set of queries that will read $d$ but no other data items in $F$.

Based on this idea, we define *the affinity between the data d and fragment F*:

$$\mathit{aff}(d, F) = - \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\frac{\mathit{minfr}(q, \pi(D))}{|\mathit{rel}(q, \pi(D))|\,(|\mathit{rel}(q, \pi(D))| + 1)} \tag{3.7}$$

2. Note that this approximation is an equality in all cases except when the increment in $|q(D)|$ makes $\mathit{minfr}(q, \pi(D))$ to be increased by 1, which happens very rarely.

Using (3.7), we develop a heuristic algorithm that places the new data items in the fragments based on the maximization of the affinity between the data items and the fragments. Notice that the affinity is always negative or zero. In the first case, the efficiency of the partitioning will decrease, while in the second it will remain the same. Thus, the bigger the affinity of a data item to $F$ (closer to zero), the less the efficiency will degrade.

### 3.3.3 Algorithm

Our *DynPart* algorithm takes a set of new data items $D'$ as input and selects the best fragments to place them. For each new data item $d \in D'$, it proceeds as follows (see the pseudo-code in Algorithm 2). First, it finds the set of queries that are compatible with the data item. This can be done by executing the queries of $W$ on $D'$ or by comparing their predicates with every new data item. Then, for each compatible query $q$, *DynPart* finds the relevant fragments of $q$, and increases the fragments affinity using the expression in (3.7). Initially the affinity of fragments is set to zero.

**Algorithm 2:** *DynPart*

**Input**:
$D'$: Set of new data items
$\pi(D)$: Partitioning
**Result**:
$\pi(D \cup D')$: Partitioning including the new data items

```
1  begin
2      foreach d ∈ D' do
3          foreach q : comp(q, d) do
4              foreach F ∉ rel(q, π(D)) do
5                  if feasible(F) then
                       // aff(F) is initialized to 0
6
7                      aff(F) ← aff(F) − f(q) minfr(q,π(D)) / (|rel(q,π(D))|(|rel(q,π(D))|+1))
8          if ∃F ∈ π(D) : aff(F) > 0 then
9              dests ← arg max_{F∈π(D)} aff(F)
10         else
11             dests ← {F ∈ π(D) : feasible(F)}
12         F_dest ← select from arg max_{F∈dests} |F|
13         move d to F_dest
14         update metadata
```

After computing the affinity of the relevant fragments, *DynPart* has to choose

the best fragment for $d$. Not all of the fragments satisfy the imbalance constraints, thus we must only consider those that do meet the restrictions. We define the function *feasible*$(F)$ to determine whether a fragment can hold more data items or not:

$$feasible(F) = |F| + 1 \leq \left\lceil \frac{|D|}{|\pi(D)|}(1 + \epsilon_s) \right\rceil \tag{3.8}$$

Accordingly, *DynPart* selects from the set of feasible fragments the one with the highest affinity. If there are multiple fragments that have the highest affinity, then the smallest fragment is selected, in order to keep the partitioning as balanced as possible.

*DynPart* works over a set of new data items $D'$, instead of a single data item. This allows the system to perform bulk operations over a set of $n$ data items instead of executing $n$ times the same operations, which is in general more costly. Moreover, it gives the algorithm more flexibility in the application of the imbalance constraints and groups data insertions in each of the fragments.

Let $comp_{\text{avg}}$ be the average number of compatible queries per data item, and $rel_{\text{avg}}$ be the average number of relevant fragments per query. Then, the average execution time of the algorithm is $O(comp_{\text{avg}} \times rel_{\text{avg}} \times |D'|)$, where $|D'|$ is the number of new data items to be appended to the fragments. The complexity can be $O(|W| \times |\pi(D)| \times |D'|)$ in the worst case, e.g. when all queries are compatible to all new data and the partitioning has not been done well. However, in practice, the averages are usually much smaller than the worst case values. The reason is that the queries usually access a small portion of the data (not the whole set), thus the average number of compatible queries per data item is low. In any case, in order to reduce the number of queries, we may use a threshold on the frequency, so that only queries above that threshold are considered. In addition, the partitioning efficiency of our approach is good (see experimental results in the next section), so the average number of relevant fragments per query is low.

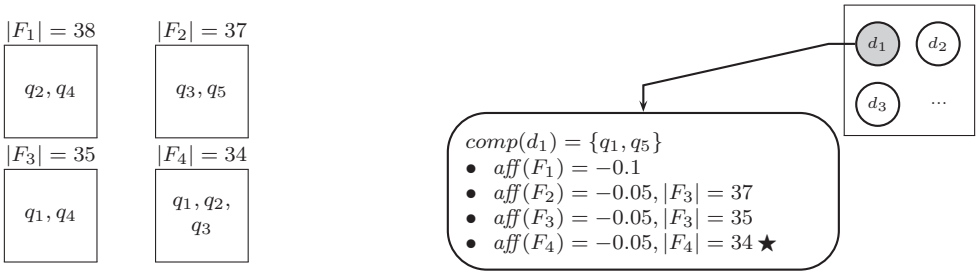
### 3.3.4 Example

Figure 3.2 illustrates the execution of the *DynPart* algorithm. Before its execution, the system is partitioned into 4 fragments, whose sizes are shown in the figure. The workload consists of 5 queries, which are represented inside the fragments they access. There are 16 new data items, $D' = \{d_1, ..., d_{16}\}$, that should be distributed over the fragments. The imbalance factor is $\epsilon_s = 0.05$, so resulting maximum size (taking into account new data items) is 42. We show the execution of the algorithm for some of the steps.

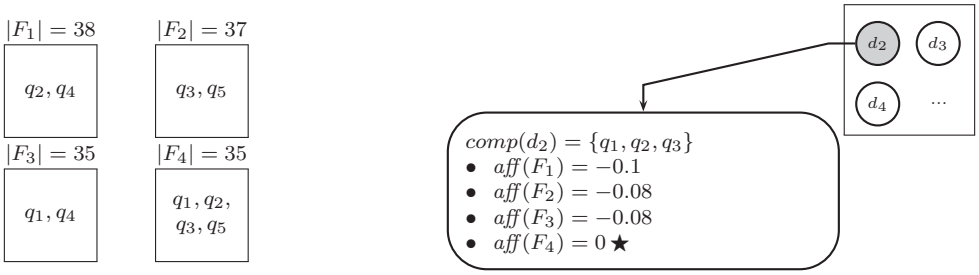
In Step 1 we show the insertion of data item $d_1$. The set of compatible queries is indicated in $comp(d_1)$. For each of these queries, the affinity of the relevant fragments is increased by the corresponding expression. As a consequence, $F_1$ has a total affinity of $-0.1$, resulting from the affinity expression applied to $q_1$

$D = \{d_1, ..., d_{16}\}$, $\epsilon_s = 0.05$, $W = \{q_1, q_2, q_3, q_4, q_5\}$,

$f(q_1) = 0.3$, $q_1(D) = \{d_1, d_2, d_3, d_4, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$
$f(q_2) = 0.2$, $q_2(D) = \{d_2, d_9, d_{10}, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$
$f(q_3) = 0.3$, $q_3(D) = \{d_2, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$
$f(q_4) = 0.1$, $q_4(D) = \{d_9, d_{10}\}$
$f(q_5) = 0.1$, $q_5(D) = \{d_1, d_3, d_4\}$

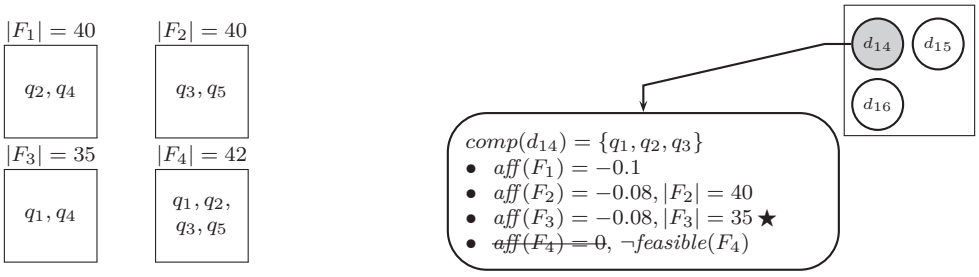



**Step 1**




**Step 2**

...




**Step 14**

**Figure 3.2:** Example of operation of the *DynPart* algorithm

and $q_5$; and $F_1$, $F_2$ and $F_3$ have an affinity of $-0.05$, resulting from the expression applied to $q_1$ for $F_2$ and $q_5$ for $F_3$ and $F_4$. The three fragments have the highest affinity, but $F_4$ is selected since it is the smallest fragment.

In Step 2, the processing for data item $d_2$ is depicted. Note that the information has been updated as a consequence of last move: the size of $F_4$ has been incremented by 1 and the set of accessing queries now include $q_5$, provided that $d_1$ is accessed by it. In this case, the highest affinity is that of fragment $F_4$, so it is selected and $d_2$ is moved to it.

The algorithm continues to execute as before until Step 14. In that case, the fragment with the highest affinity is $F_4$, but it can not be selected, as it would violate the imbalance constraint. As a consequence, the next fragment in terms of affinity is selected and data item $d_{14}$ is placed in fragment $F_3$.

### 3.3.5 Data Structures

Our algorithm needs to maintain information about the relevant fragments of each query, so that we can compute the affinity efficiently. Queries are assigned a unique identifier and stored on a hash table for efficient access. For each of them, we store the set of relevant fragments as a list, as they are always accessed sequentially, i.e., no random access. Space complexity is $O(|W| \times |\pi(D)|)$ in the worst case, but, as we have pointed out, the average number of relevant fragments stays low even when the number of fragments increases. For example, in our experiments, for 1024 fragments, the average number of relevant fragments do not exceed 18 in any scenario. We also need to store the set of queries for each of the new data items. Again, as this set is accessed sequentially, we keep a list of query identifiers.

Our algorithm needs to create a data structure for each new data item to store the affinity of the possible destination fragments. For this, there are several alternatives. One option is to keep an array of size $|\pi(D)|$ initialized to zero. Note that, as the actual number of possible destinations is much lower than the total number of fragments, we would waste a lot of space with zero-affinity entries. Therefore, we keep a hash table of fragments and only compute those for which the affinity is non-zero. With this method, access time will be maintained, while space requirements will be significantly reduced.

### 3.3.6 Dealing with Deletes and Updates

So far, we have only considered the case where data items are appended to the database. However, we could easily extend our approach to deal with deletions and updates. For a deletion, we only need to consider metadata maintenance. Whenever a data item $d$ is deleted, the size of the fragment where it was placed should be reduced by one. We would also have to check for all queries compatible

with $d$ whether they still have to access that fragment or not, and update their set of relevant fragments if necessary. An efficient way to do this is to keep the number of data items accessed by each query on every of its relevant fragments, i.e., $|q(F)| \forall F \in rel(q, \pi(D))$. Then, whenever $d$ is deleted from a fragment $F$, $|q(F)|$ would be reduced by 1. If the size reaches 0, then $F$ should be deleted from the set of relevant fragments.

The case of updating a data item can be considered as a deletion followed by an insertion. However, we can benefit from previous information, and only recalculate the compatibility of queries that are affected by the changes.

## 3.4 Dealing with Imbalance

In the algorithm presented in the previous section, new data items are treated individually even if they are highly correlated. As a consequence, the destination chosen for them may differ if at a given point the selected fragment reaches the maximum size constrained by the imbalance factor. The problem might be specially important when there are big groups of similar elements and/or the imbalance constraints are too restrictive. In this section we present a variation of the previous algorithm which tries to avoid such a situation by grouping similar elements together and taking a common decision for all of them.

### 3.4.1 Algorithm

The extended version of our algorithm, which we call *DynPartGroup*, starts by dividing the set of new data items $D'$ into a set of groups $G$ such that all members of each group are accessed exactly by the same set of queries. Thus, the members of each group share exactly the same affinity for each given fragment. If they are allocated to different fragments, the partitioning efficiency of each of the incident queries is likely to decrease. The construction of the groups is included in function CreateGroups(). A list of groups is built, where each group stores the set of composing tuples and the set of accessing queries. All items in a group are treated in the same way.

The algorithm (the pseudo-code is shown on Algorithm 3) first creates the groups and orders them by size in descending order, i.e., the biggest groups are considered before the smallest ones. The rationale is that, if we consider first the biggest groups, there is more free space on the fragments and the probability that all members of these groups fit on the same fragment is higher.

Once groups are ordered, an affinity value is calculated for each group, exactly in the same way it was done for individual data items in the basic algorithm. In this case, function $feasible(F, g)$ will return true if $F$ plus the data items of the

**Function** CreateGroups($D'$)

**Input**:
$D'$: Set of new data items
**Result**:
$G$: Set of group with of equivalent data items

```
1  begin
2      G ← emptyList()
3      foreach d ∈ D' do
4          qs = {q : comp(q, d)}
5          if ∃g ∈ G : g.qs = qs then
6              g.ts ← g.ts ∪ {d}
7          else
8              g_new.ts ← {d}
9              g_new.qs ← qs
10             G ← insert(G, g_new)
11     return G
```

group $g$ does not violate the imbalance factor, i,e:

$$feasible(F, g) = |F \cup g.ts| \leq \left\lceil \frac{|D|}{|\pi(D)|}(1 + \epsilon_s) \right\rceil \tag{3.9}$$

If there is no feasible destination for $F$, the group is split into two equal halves and the resulting groups are inserted back in the list in the corresponding positions so that the order is maintained. At some point, those groups would be considered again but, in this case, individually. Note that other splitting strategies may be envisioned, e.g., assigning only the elements that fit in the fragment with the highest affinity and considering the rest as a new group. However, this will be in detriment of other big groups that might have to be subsequently split, and they will not offer any gain regarding the partitioning efficiency, as the group would be split anyway.

Let us now analyze the complexity of the algorithm. We divide the analysis in two parts; first we analyze the group creation and ordering part, and then the rest of the algorithm. Function CreateGroups($D'$) has to go over all the elements in $D'$. Each of them has to be compared with existing groups to check if accessing queries match, which can be done by defining a hash function over the query sets. This function has a complexity of $O(|W|)$. As a result, the total complexity of group creation is $O(|D'| \times |W|)$. Let $|G|$ be the number of groups, then the complexity of group sorting is $O(|G| \times \log |G|)$. In the worst case, $|G| = |D'|$, but as we will see in the experimental section, the number of groups is usually much lower than that value.

**Algorithm 3:** *DynPartGroup*

**Input**:
$D'$: Set of new data items
$\pi(D)$: Partitioning
**Result**:
$\pi(D \cup D')$: Partitioning including the new data items

```
1  begin
2      G ← CreateGroups(D')
3      order G by |g.ts| in descending order
4      while G ≠ ∅ do
5          g ← first(G)
6          G ← G − {g}
7          foreach q ∈ g.qs do
8              foreach F ∉ rel(q, π(D)) do
9                  if feasible(F) then
                       // aff(F) is initialized to 0
10
11                     aff(F) ← aff(F) − f(q) minfr(q,π(D)) / (|rel(q,π(D))|(|rel(q,π(D))|+1))
12         if ∃F ∈ π(D) : aff(F) > 0 then
13             dests ← arg max_{F∈π(D)} aff(F)
14         else
15             dests ← {F ∈ π(D) : feasible(F)}
16         if dests ≠ ∅ then
17             F_dest ← select from arg min_{F∈dests} |F|
18             move d to F_dest
19             update metadata
20         else
21             split g into two equal sets g_1 and g_2
22             insert g_1 and g_2 in G maintaining G's order
```

The complexity of the rest of the algorithm is calculated in a similar way than in the basic algorithm. The main difference is the number of times the outer loop has to be executed. The worst case is the situation where there is a single group, the imbalance factor is near 0 and $|\pi(D)| \geq |D'|$. In that case, only one data item can be inserted on each fragment, and the group would have been split in $|D'|$ groups of size 1. This would cause $|D'| - 1$ splits and require $2 \times |D'| \in O(|D'|)$ executions of the outer loop, which would imply $O(|W| \times |\pi(D)| \times |D'|)$ affinity calculations, as in the basic algorithm.

The size of $|G|$ can vary throughout the execution, as each split increases its size by one. In the worst scenario explained above, its size will increase until reaching $|D'|$, point from which it will be consumed, as all groups would be of size 1. Assume that the ordered insertion on $G$ is executed on $O(\log |G|)$. Then, all the sequence of insertions would need $O(\log 1) + O(\log 2) + ... + O(\log |D'|) = O(\log |D'|!) = O(|D'| \log |D'|)$. Hence, the worst case complexity is $O(|W| \times |\pi(D)| \times |D'| + |D'| \log |D'|)$

However, that worst case is very rare as usually there are a higher number of groups, and the splits are uncommon. Thus, we can say that in the average case execution complexity of this part of the algorithm is $O(comp_{\text{avg}} \times rel_{\text{avg}} \times |G|)$.

### 3.4.2 Example

Figure 3.3 compares the assignments performed by the basic version of the algorithm (*DynPart*), and the algorithm we described above (*DynPartGroup*), in the same scenario as in the previous section. Compatible queries for all data items are shown in previous example but can also be inferred from the groupings shown in the top of the figure, i.e., all the data items in a group have the corresponding set of compatible queries. In the basic algorithm, data items are assumed to be processed in the order indicated in the subindex, i.e., first $d_1$, then $d_2$, etc. Finally, recall that an imbalance factor of 0.05 for a fragment of size 40 means that the maximum size of the fragment at the end of the execution is 42.

Figure 3.3(a) shows the final assignment performed by the extended algorithm. All the groups are assigned to a single fragment and the chosen fragments have always one of the highest affinities, so the allocations are optimal. In figure 3.3(b) the assignments resulting from the execution of the basic algorithm are depicted. Note that, in this case, groups $g_1$ and $g_2$ have to be split into different fragments. As a consequence, $q_1$, $q_3$ and $q_5$ increment the number of accessed fragments by 1 and $q_2$ by 2, thus decreasing partitioning efficiency. This is the consequence of fragment $F_4$ being at its maximum size in step 14, which prevents it to be selected in further phases of the algorithm.

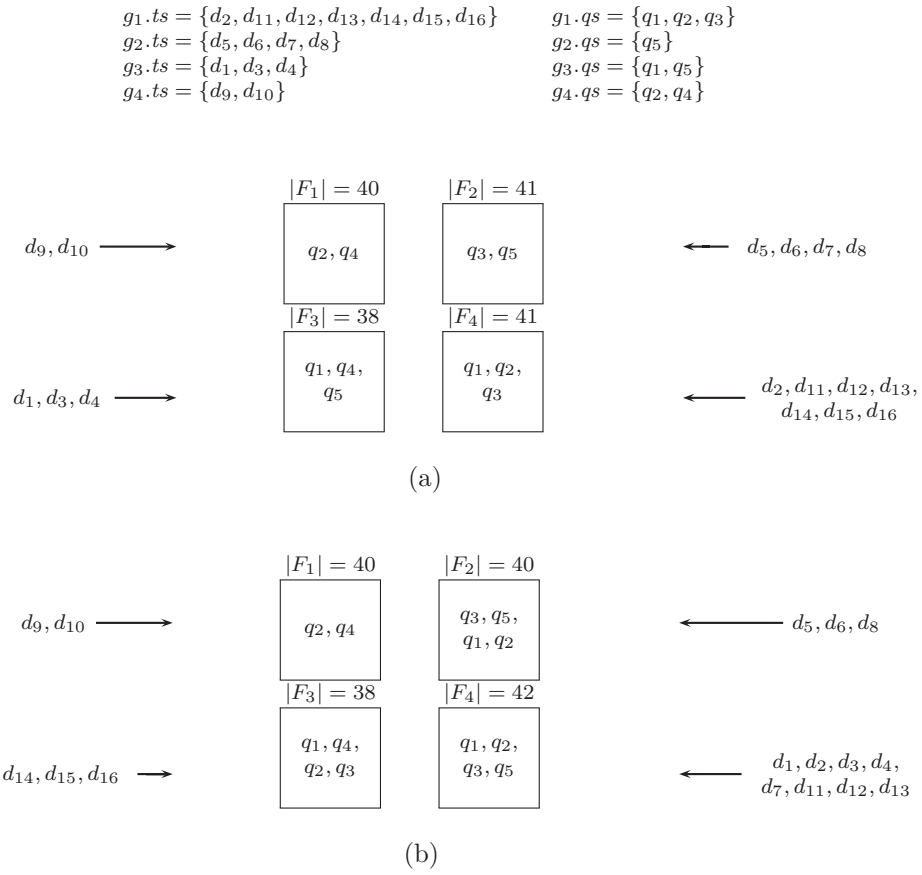



**Figure 3.3:** Example of execution of the distribution algorithms: a) algorithm *DynPartGroup*, b) algorithm *DynPart*

### 3.4.3 Balancing Fragments Based on Load

In Section 3.2, we modeled the problem of data partitioning using a size balancing constraint. Nonetheless, the problem may also be formalized if a load balancing constraint is required. Intuitively, with load we mean the number of accesses to the fragments.

Let us first define formally the load of a dataset as follows.

**Definition 3.4.1.** The load of a data set $D$, denoted $L(D)$ is defined as the sum of the frequencies of the queries accessing its data items:

$$L(D) = \sum_{q \in W} f(q) \times |q(D)| \tag{3.10}$$

Given this definition, we can reformulate the imbalance constraint in the following way: $L(F) \leq \frac{L(D)}{|\pi(D)|}(1 + \epsilon_l)$. As a result, the formula for the minimum number of fragments that should be accessed for a given query should be modified accordingly:

$$minfr(q, \pi(D)) = \left\lceil \frac{L(q(D))}{(L(D) \,/\, |\pi(D)|)(1 + \epsilon_l)} \right\rceil \tag{3.11}$$

Note that in the numerator we use $L(q(D))$ instead of $|q(D)|$ because we should take into account that items accessed by $q$ are also accessed by other queries that we have to consider.

To use this new imbalance constraint, our algorithms only need some minor modifications as follows. In Algorithm 2, in case of ties in the affinity measure, the least loaded fragment should be selected instead of the smallest one. Moreover, in Algorithm 3, groups should be ordered by load instead of by size. Furthermore, function *feasible* should be redefined as follows:

$$feasible(F, g) = L(F \cup g.ts) \leq \left\lceil \frac{L(D)}{|\pi(D)|}(1 + \epsilon_l) \right\rceil \tag{3.12}$$

## 3.5 Experimental Evaluation

To validate our dynamic partitioning algorithms, we conducted a thorough experimental evaluation over real-world data. In Section 3.5.1, we describe our experimental setup. In Section 3.5.2, we report on the execution time of our algorithms and compare them with a well known static workload-based algorithm. In Section 3.5.3, we study the effect of the heuristic, which we used in our algorithms, on partitioning efficiency. Finally, Section 3.5.4 studies how the imbalance factor and the correlation of new data affect the partitioning efficiency.

### 3.5.1 Set-up

For our experimental evaluation we used the data from the Sloan Digital Sky Survey catalog, Data Release 8 (DR8) [114], as it is being used in LIneA in Brazil [3]. It consists of a relational database with several observations for both stars and galaxies. We obtained a workload sample from the SDSS SkyServer SQL query log data, which stores the information about the real accesses performed by users. In total, the database comprises almost 350 million tuples, that take 1.2 TB of space. The query log consists of a total of 27000 queries, some of which are similar in the SQL form but produce different results, as they use different parameters.

All queries were executed on the database and the tuple ids accessed by each of them were recorded. Only tuples accessed by at least one query were considered. We simulated the insertions on the database by selecting a subset of the tuples as the initial state and appending the rest of the tuples in groups. We varied the following parameters: 1) the number of tuples inserted to the database on each execution of our algorithm, $|D'|$; 2) the number of fragments in which the database is partitioned, $|\pi(D)|$; 3) the imbalance factors, $\epsilon_s$ and $\epsilon_l$; and 4) the order of data items, so as to produce datasets with higher correlation between consecutive data items. On each of the experiments, the specific numbers are detailed.

All experiments were executed in a 3.0 GHz Intel Core 2 Duo E8400, running Ubuntu 11.10 64-bit with 4GB of memory.

### 3.5.2 Partitioning Time

In this section, we study the execution time (partitioning time) of the dynamic algorithms *DynPart* (DP in the figure) and *DynPartGroup* (DPG) and compare them with a static graph partitioning algorithm (SP). For the later, we use PaToH [4], an hyper-graph partitioner. Figure 3.4 shows the comparison of the partitioning time for 16 fragments and for data size balancing ($\epsilon_s = 0.15$) and load balancing ($\epsilon_l = 0.15$). We executed the dynamic algorithms with two values for $|D'|$: 500000 and 1 million tuples. Similar results are obtained for different values of $|\pi(D)|$. As the difference between execution times of the static and the dynamic algorithms is significant, we use a logarithmic scale for the y-axis in order to show the results. The results are depicted until a database size of 20 million tuples, as the memory requirements for the static partitioning are bigger than the total available memory for higher values. The dynamic algorithms, on the other hand, do not cause any problem as the memory footprint depends on $|D'|$, which is constant throughout the experiment.

3. Data from the DES project is still unavailable, so we have used data from SDSS, which is a similar, previous project

4. `http://bmi.osu.edu/~umit/software.html`

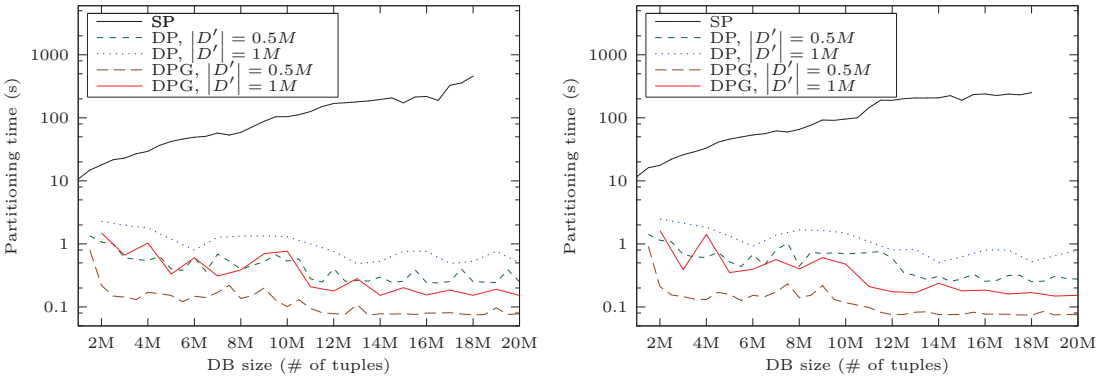



**Figure 3.4:** Comparison of partitioning times of the dynamic and graph-based partitioning algorithms as the DB size increases ($|\pi(D)| = 16$) for a) data size balancing ($\epsilon_s = 0.15$) and b) load balancing ($\epsilon_l = 0.15$).

As it can be seen, partitioning time increases for the graph partitioning algorithm as the size of the database increases, provided that the size of the graph increases accordingly. For the dynamic algorithms, on the other hand, the execution time stays at the same level, as it is always executed for the same number of data items. Some variation is observed since the features of the new items adapt differently to the partitioning. However the trend is constant.

In the figure, we can also observe that the execution times of the *DynPartGroup* algorithm are better that those of the basic algorithm. This is caused by the reduced number of affinity calculations, as we will show later.

We compared the execution of our algorithms for different sizes of $D'$. Figure 3.5 shows the average execution time of the *DynPart* and the *DynPartGroup* algorithms as $|D'|$ increases for different number of fragments and for both balancing strategies. As expected, the execution time is linearly related to the buffer size. Also, the higher number of fragments, the higher the execution time. This increase is not linear since the number of relevant fragments does not increase at the same pace. In fact, the number of relevant fragments does not exceed 8 for $|\pi(D)| = 256$ and 16 for $|\pi(D)| = 1024$. The difference on the execution time between the *DynPart* and the *DynPartGroup* algorithms is also noticeable.

In Figure 3.6, we represent the average execution times for the different stages of the dynamic algorithms corresponding to the same scenario of Figure 3.4. Both algorithms contain the following stages: calculate affinities, select max affinity and update metadata. The extended algorithm also contains two additional stages, namely create groups and sort groups. Finally, another phase is depicted, which represents the rest of the operations executed during the distribution but not associated to a particular algorithm.

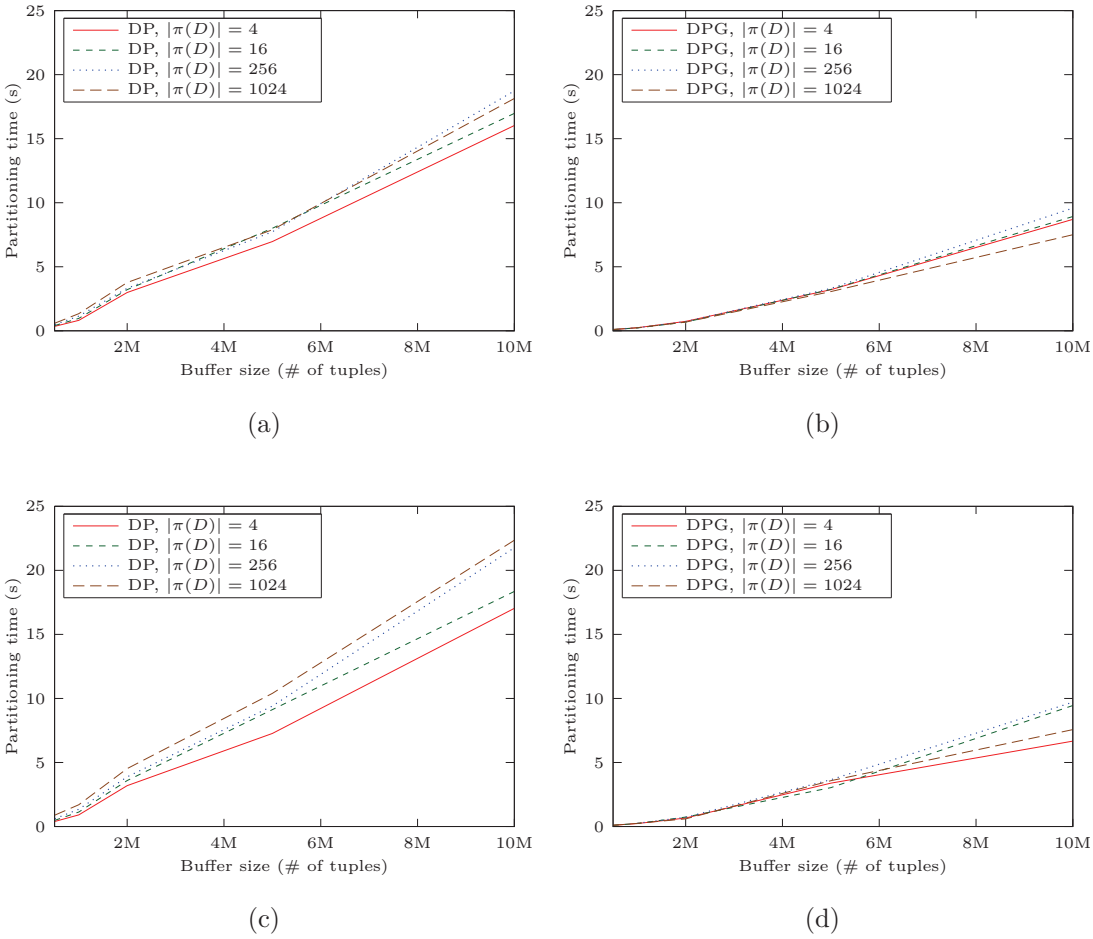



**Figure 3.5:** Partitioning time vs. $|D'|$ for a) *DynPart* and data size balancing ($\epsilon_s = 0.15$), b) *DynPartGroup* and data size balancing ($\epsilon_s = 0.15$), c) *DynPart* and load balancing ($\epsilon_l = 0.15$) and d) *DynPartGroup* and load balancing ($\epsilon_l = 0.15$).

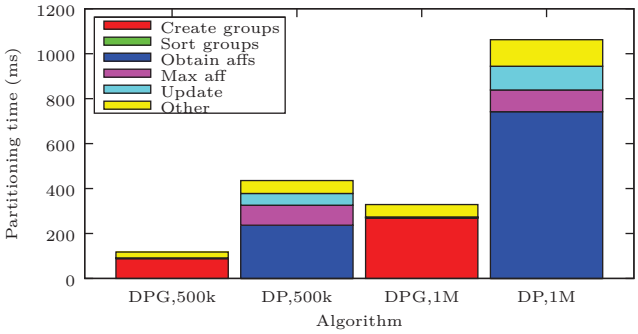
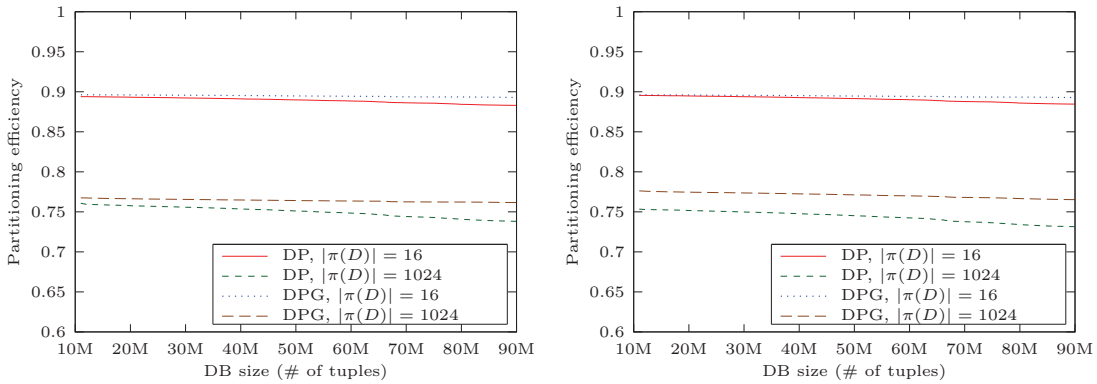



**Figure 3.6:** Comparison of dynamic algorithms' partitioning times (data size balancing with $\epsilon_s = 0.15$)




(a) (b)

**Figure 3.7:** Comparison of partitioning efficiency as the size of the DB grows ($|D'| = 1M$) for a) data imbalance and b) load imbalance

As we can observe in Figure 3.6, the distribution of execution times is completely different for both algorithms. The *DynPart* algorithm spends most of the time in the calculation of the affinities, although the time spent in the rest of the phases is also significant. On the other hand, *DynPartGroup* spends almost all the time in the creation of the groups, whereas the time spent in the rest of the stages is negligible. This can be explained by considering the number of groups created in average, 664 for $|D'| = 500k$ and 1360 for $|D'| = 1M$, which represent around 0.13% of the number of tuples. As a consequence, with *DynPartGroup* the time for computing affinities, selecting the best fragment, and updating the corresponding metadata is significantly reduced.

### 3.5.3 Partitioning Efficiency

One of the important issues to consider for the dynamic algorithms is how they affect the partitioning efficiency.

We executed the algorithms as the database is fed with new data after an initial partitioning using the graph-based partitioning approach. With $|D'| = 1\,\text{M}$, Figure 3.7 shows how the partitioning efficiency evolves as the database grows for different number of fragments, $|\pi(D)|$. Similar results were obtained for other configurations of $|D'|$. The efficiency decreases as the database grows, as expected, but this reduction is very small. For example, in the worst case, $|\pi(D)| = 1024$ and data size balancing, the partitioning efficiency decreases $2.82 \times 10^{-3}$ in average for each 10 million new tuples. The difference between *DynPart* and *DynPartGroup* is very small for small values of $|\pi(D)|$, but increases for higher values. In any case, it is below 5% for the worst case.

To evaluate the quality of our partitioning approach, in addition to the parti-

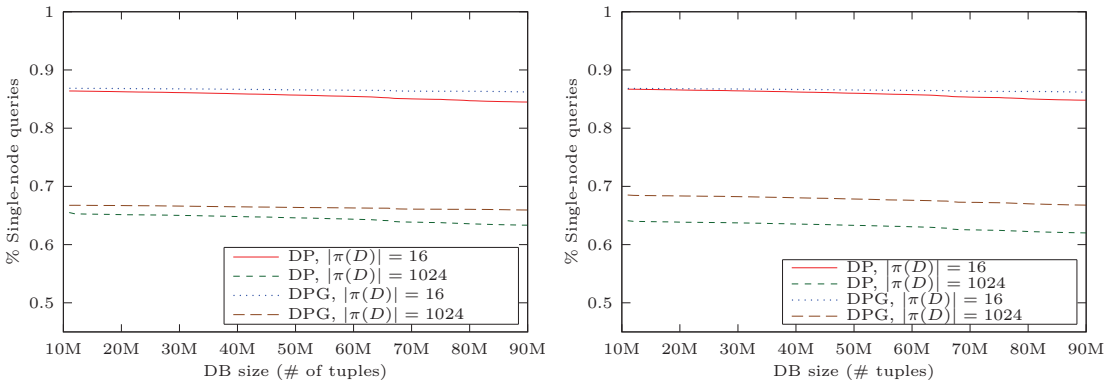



**Figure 3.8:** Comparison of percentage of single-node queries as the size of the DB grows ($|D'| = 1M$) for a) data imbalance and b) load imbalance

tioning efficiency metrics, as in [41, 100] we studied the percentage of single-node queries, which means the percentage of the queries that can be executed using the data of only one fragment. Figure 3.8 shows the results. As seen, when the number of fragments is small, the results are similar to what we reported for partitioning efficiency metrics. However, for higher number of nodes, the number of single-node queries is lower. The reason is that in these cases the partitions are smaller, so it is more difficult to confine all the results of a query in a single fragment

### 3.5.4 Effect of Imbalance Factor and Data Correlation

The imbalance factor ($\epsilon_s$ or $\epsilon_l$) may affect the efficiency as it constraints the flexibility of the algorithm in allocating new data items. The lower the imbalance factor, the less flexibility, which may imply that some data items are not placed in the optimal fragments because they are full. Figures 3.9(a) and 3.9(c) show the average partitioning efficiency for different values of $\epsilon_s$ and $\epsilon_l$, respectively. The efficiency decreases as the imbalance factor decreases, as expected, but it is much more noticeable for the *DynPart* algorithm.

To enrich our study, we have considered other scenarios by reordering the data so that correlated data items arrive together. In order to do that, we executed the *DynPart* algorithm over the initial data set and created the corresponding partitions. Then we reordered the data by placing on defined intervals data of only one of the fragments at a time. That way, we increase the correlation of new data ($D'$) on each execution of the algorithm.

Figures 3.9(b) and 3.9(d) show the same configuration as before but with a new ordering created by producing 8 fragments on the original data and placing

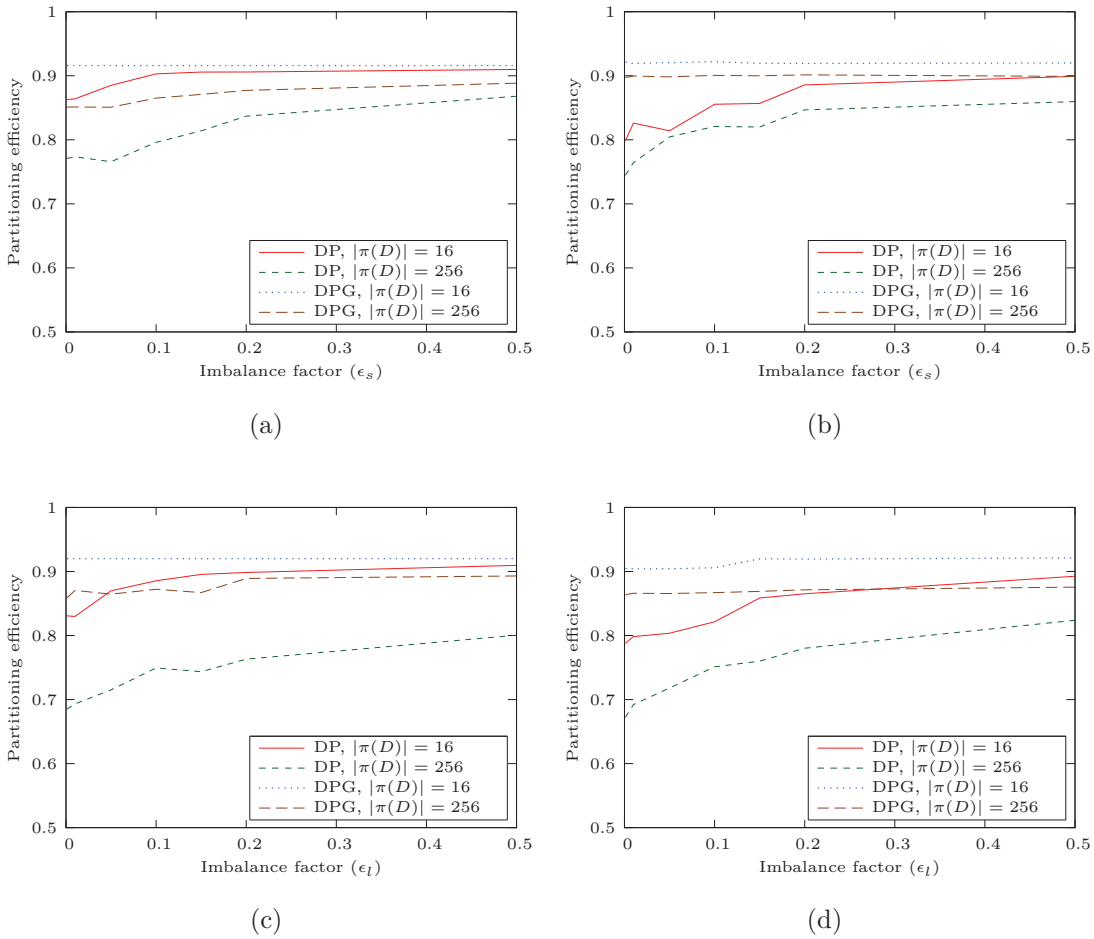
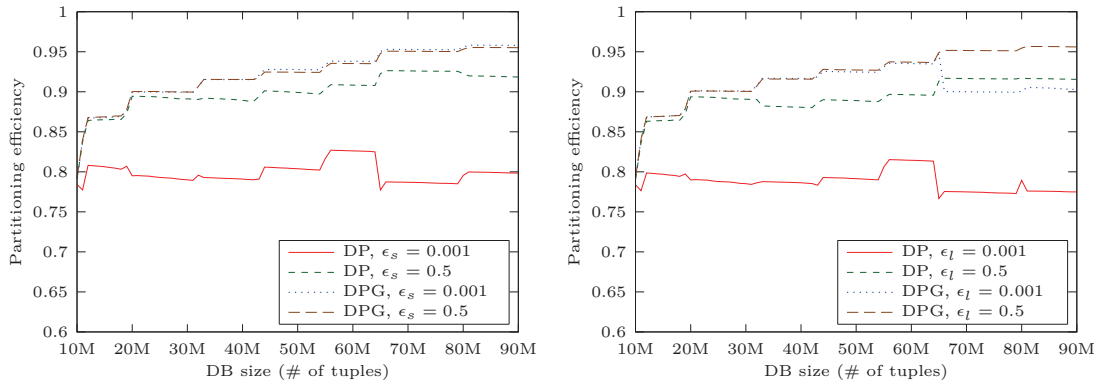



**Figure 3.9:** Partitioning efficiency vs. imbalance factor for a) original data set and data size balancing, b) reordered data set and data size balancing, c) original data set and load balancing and d) reordered data set and load balancing

items of one of those fragment in intervals of $10M$ [5]. As we see, in the case of correlated data, the impact of the imbalance factor is higher than in the previous scenario. Nevertheless, the *DynPartGroup* algorithm still shows good behavior for different values of $\epsilon_s$ and $\epsilon_l$.

Finally, in Figure 3.10 we show the evolution of the partitioning efficiency as the database grows for imbalance factors of 0.001 and 0.5, which represent both extremes on the studied values of $\epsilon_s$ and $\epsilon_l$. This confirms that higher correlations on the inserted data affect the resulting partitioning efficiency. At the beginning the efficiency is low, since all the inserted data is highly correlated and data items that should be allocated together have to be split because of imbalance

5. We have produced different reorderings and the experiments show similar results




(a) (b)

**Figure 3.10:** Partitioning efficiency for the reordered data ($|\pi(D)| = 16$) for a) data size balancing and b) load balancing

constraints. However, as new data items with different affinities are included and the imbalance is more flexible, the efficiency increases.

By comparing the behavior of both dynamic algorithms we can state that the *DynPartGroup* algorithm obtains better partitioning efficiencies consistently. The *DynPart* algorithm approaches *DynPartGroup* when the imbalance factor is high, but degrades as the imbalance constraints are stricter. This difference between the partitioning efficiency of the two algorithms is even higher for configurations with more number of fragments.

## 3.6 Related Work

Partitioning has been used both for declustering and clustering. In this chapter, we are interested in the later, as we are trying to reduce the number of query accesses to the fragments.

As we explained in Section 2.2.4, the basic and most popular techniques for database partitioning include 1) round-robin, which consists in assigning each tuple to a different fragment; 2) hash-partitioning, which applies a hash function of a predefined set of attributes; and 3) range-based partitioning, which splits data on ranges on a given set of attributes. These techniques are still being used, for instance in distributed key-value stores, as they work well in many scenarios. Dynamo [45] uses a modified version of hash-partitioning on the key and, as a consequence, only obtain single-site query executions when the query contains equality predicates on the key. In general, hash-based partitions are good for clustering only when the queries contain equality predicates on the partitioning attributes, which is not the case of our workload. BigTable [33] and PNUTS [38]

use range-based partitioning on the keys; which still is too simple for our reference queries. In general, the complexity of scientific workloads makes it hard to design a good partitioning strategy manually, so automatic partitioning is preferred [99].

In Section 2.2.4 we covered the different approaches employed in automatic database partitioning, including the systems proposed by major vendors such as Microsoft's SQL Server AutoAdmin [35, 5] and IBM's DB2 Database Advisor [106, 133]. Many of these works have focused on partitioning (both vertical and horizontally) as an element of physical design for a single-node, along with indexing and materialized views. For instance, in [5] a set of physical design alternatives (that includes partitioning) is generated. Then, in order to limit the search space they prune the set of candidates. Similar procedures are used in other works, such as AutoPart [99], which is focused on scientific workloads. In this case only vertical and categorical partitioning are considered and the resulting partitioning is also used for physical design at a single-node.

Some other proposals use analogous techniques to automatically generate partitions in distributed systems. The solution proposed in [106] uses a similar approach but with the goal of distributing the queries over all the nodes (data declustering). Automatic database partitioning for distributed databases has recently received further attention. In [91], data is partitioned automatically to optimize the execution of MPP systems. As a possible alternative they only consider hash-based partitioning over a single column. In [100], both hash and range-based partitioning on the most accessed attributes are considered for partitioning in OLTP systems. To find a near optimal solution, their approach explores a solution space by adapting the large-neighborhood search technique. However, this approach and most of the approaches mentioned above are not well suited for our underlying scientific applications that are characterized by complex workload predicates involving many attributes; and this significantly degrades the efficiency of those approaches

Graph-based approaches have been used to capture more complex relations between the workload and the data both for partitioning with the objective of declustering [90, 84] and clustering [41]. The most important works were covered in Section 2.2.4. Schism [41] is a recent system that partitions the data by building a graph containing the relations between queries and tuples. Data items are retrieved using an index or by means of predicate-based explanations, depending on the scenario. However, like other existing graph-based approaches, it is static and needs to redo the partitioning from scratch when the data changes. As we showed in this chapter, this approach does not perform well for growing databases, and a dynamic approach is hence required. Furthermore, as new produced partitioning schemes are not aware of previous ones, large amounts of data transfers may have to take place in order to apply the new data placements.

## 3.7 Conclusions

In this chapter, we proposed a pair of dynamic algorithms for partitioning continuously growing large databases. We modeled the partitioning problem for dynamic datasets and proposed a new heuristic to efficiently distribute new arriving data, based on the affinity it has with the different fragments in the application. We designed two alternatives, *DynPart*, the basic algorithm, and *DynPartGroup*, which better deals with strict imbalance constraints.

We validated our approach through implementation, and compared its execution time with that of a static graph-based partitioning approach. The formalism used in the construction of our algorithms is very related to that of the hypergraph, as data elements are equivalent to vertices and queries are equivalent to hyperedges (subsets of vertices). Imbalance constraints are also present in both models, as graph-partitioning algorithms incorporate them as part of their input.

The results show that, as the size of the database grows, the execution time of the static algorithm increases significantly, but that of our algorithms remains stable. This was expected, as our algorithm was designed to only consider the new elements when calculating the assignments. They also exhibit that, for the given dataset, our algorithms, although based on a heuristic approach, do not degrade partition efficiency considerably. As a future work, scenarios where the workload evolve over time and produce a degradation of the partitioning efficiency could also be envisageable. In those cases, efficient reorganization of data items is required by taking into account not only the efficiency but also the cost of the data transfers needed to attain the new data partitioning.

The experiments show that in the case of datasets where there is a high correlation between new data items, the *DynPartGroup* algorithm maintains a very good behavior. They also indicate that this algorithm is not highly affected by the imbalance of fragments' sizes. We also consider, as a possible strategy to explore, modified versions of the grouping algorithm, where the equivalence between data elements could be relaxed.

Overall, our experiments confirm that our dynamic partitioning strategy is able to efficiently deal with the data of our astronomic application. But, we believe that its utilization is not limited to this application, and it can be used for data partitioning in many other applications where the data items are appended continuously.

# Chapter 4

# Data Partitioning for Minimizing Data Transfers in MapReduce

In this chapter, we focus on the shuffle phase of MapReduce executions. We identify the overhead that this phase may pose in the MapReduce framework and pinpoint the limitations of the approaches that have tried to overcome it. We propose *MR-Part*, a repartitioning strategy that reduces the quantity of data that should be transferred through the network in the shuffle phase. The problem is formalized in Section 4.2 and the proposed approach explained in detail in Section 4.3. Then, its efficiency is assessed through experiments on the Grid5000 platform under varying parameters.

## 4.1 Motivations and Overview

MapReduce [43] has established itself as one of the most popular alternatives for big data processing due to its programming model simplicity and automatic management of parallel execution in clusters of machines. It divides the computation into two main phases, namely map and reduce, which in turn are carried out by several tasks that process the data in parallel. Between them, there is a phase, called shuffle, where the data produced by the map phase is ordered, partitioned and transferred to the appropriate machines executing the reduce phase.

MapReduce applies the principle of "moving computation towards data" and thus tries to schedule map tasks in MapReduce executions close to the input data they process, in order to maximize data locality. Data locality is desirable because it reduces the amount of data transferred through the network, and this reduces energy consumption as well as network traffic in data centers.

Recently, several optimizations have been proposed to reduce data transfers between mappers and reducers. For example, [67] and [98] try to reduce the amount of data transferred in the shuffle phase by scheduling reduce tasks close to the map tasks that produce their input. Ibrahim et al. [74] go even further

and dynamically partition intermediate keys in order to balance load among reduce tasks and decrease network transfers. Nevertheless, all these approaches are limited by how intermediate key-value pairs are distributed over map outputs. If the data associated to a given intermediate key is present in all map outputs, the pairs in all the nodes but one still have to be transferred.

In this chapter, we propose a technique, called *MR-Part*, that aims at minimizing the transferred data between mappers and reducers in the shuffle phase of MapReduce. *MR-Part* captures the relationships between input tuples and intermediate keys by monitoring the execution of a set of MapReduce jobs which are representative of the workload. Then, based on the captured relationships, it partitions the input files, and assigns input tuples to the appropriate fragments in such a way that subsequent MapReduce jobs following the modeled workload will take full advantage of data locality in the reduce phase. In order to characterize the workload, we inject a monitoring component in the MapReduce framework that produces the required metadata. Then, another component, which is executed offline, combines the information captured for all the MapReduce jobs of the workload and partitions the input data accordingly. We have modeled the workload by means of a hypergraph, to which we apply a min-cut k-way graph partitioning algorithm to assign the tuples to the input fragments.

We implemented *MR-Part* in Hadoop, and evaluated it through experimentation on top of Grid5000 using standard benchmarks. The results show significant reduction in data transfer during the shuffle phase compared to Native Hadoop. They also exhibit a significant reduction in response time when network bandwidth is limited.

## 4.2 Problem Definition

We are given a set of MapReduce jobs which are representative of the system workload, and a set of input files. We assume that future MapReduce jobs follow similar patterns as those of the representative workload, at least in the generation of intermediate keys.

The goal of our system is to automatically partition the input files so that the amount of data that is transferred through the network in the shuffle phase is minimized in future executions. We make no assumptions about the scheduling of map and reduce tasks, and only consider intelligent partitioning of intermediate keys to reduce tasks, e.g., as it is done in [74].

### 4.2.1 Input Dataset

We define the input data as a set of tuples $D = \{d_1, ..., d_n\}$, which is divided by the MapReduce framework into a set of chunks $C = \{C_1, ..., C_p\}$. The function that assigns tuples to chunks is the following:

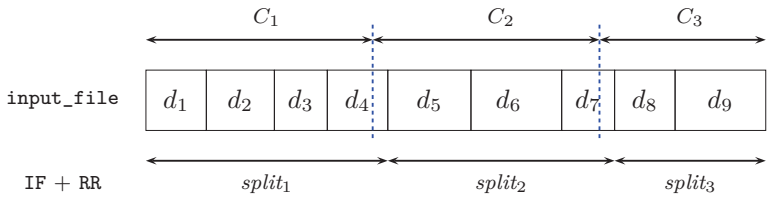



**Figure 4.1:** Input dataset and tuple assignments

$$loc : D \rightarrow C \tag{4.1}$$

Actually, before executing a job there is no physical assignment between tuples and chunks. We consider a tuple $d_i$ to be allocated to a chunk $C_j$ if it is generated by the `RecordReader` when parsing input split assigned to chunk $C_j$ by the `InputFormat`. As explained in Section 2.3.2, the `InputFormat` is the component of the MapReduce framework that generates input splits and assigns them to map tasks and the `RecordReader` is the component that parses those splits and produces input key-value pairs.

We consider a set of MapReduce jobs that are executed over the same input dataset and that use the same `InputFormat` and `RecordReader`. This guarantees that the key-value pairs that are fetched to the MapReduce jobs are exactly the same in all jobs. We also assume that the data is stored in a distributed file system like GFS or HDFS, where files are divided into chunks and allocated in the nodes of the system. Normally each file format is parsed by a single combination of `InputFormat` and `RecordReader`; therefore, using the same combination of components in all the jobs that read the same file is a common practice.

Let us illustrate the $loc()$ function with the example shown in Figure 4.1. In that scenario, there are three chunks and the assignments are $loc(d_1) = loc(d_2) = loc(d_3) = loc(d_4) = C_1$, $loc(d_5) = loc(d_6) = loc(d_7) = C_2$, etc. The bytes corresponding to tuples $d_4$ and $d_7$ cross chunk boundaries. We assign them to chunks $C_1$ and $C_2$ respectively as the `RecordReader` generates them when parsing the corresponding splits.

### 4.2.2 Transferred Data in Shuffle Phase

Let us formally define the transferred data which we want to minimize. Let $job_\alpha$ denote a MapReduce job. It is composed of $M_\alpha = \{m_1, ..., m_p\}$ map tasks and $R_\alpha = \{r_1, ..., r_q\}$ reduce tasks. We do not consider map or reduce tasks which fail or are the result of speculative execution and are not retained. We assume that each map task $m_i$ processes chunk $c_i$, for $i = 1, ..., p$.

Let $I_\alpha = \{ip_1, .., ip_m\}$ be the set of intermediate key-value pairs produced by the map phase, such that $map(d_j) = \{ip_{j_1}, ..., ip_{j_t}\}$. $key(ip_j)$ represents the

key of intermediate pair $ip_j$ and $size(ip_j)$ represents its total size in bytes. $K_\alpha$ is defined as the set of intermediate keys produced in the execution of $job_\alpha$, $K_\alpha = \bigcup_{ip \in I_\alpha} key(ip)$.

We define $output(m_i) \subseteq I_\alpha$ as the set of intermediate pairs produced by map task $m_i$, $output(m_i) = \bigcup_{d_j \in C_i} map(d_j)$. We also define $input(r_i) \subseteq I_\alpha$ as the set of intermediate pairs assigned to reduce task $r_i$. This assignment is controlled by the reduce partitioning function:

$$part : K_\alpha \rightarrow R_\alpha \tag{4.2}$$

Let $N = \{n_1, .., n_s\}$ be the set of machines that compose the cluster; $node(t)$ represents the machine where task $t$ is executed:

$$node : M_\alpha \cup R_\alpha \rightarrow N \tag{4.3}$$

The way in which this assignment is done depends on the scheduling algorithm, the properties of the job and the characteristics and behavior of the cluster where it is executed.

Now we distinguish between local and remote transfers of intermediate tuples. Let $ip_j$ be an intermediate key-value pair, produced in map task $m$, i.e., $ip_j \in output(m)$ and consumed by reduce task $r$, i.e., $ip_j \in input(r)$. We define $P_\alpha(ip_j) \in \{0, 1\}$ as a variable that indicates whether $ip_j$ is transferred or not through the network:

$$P_\alpha(ip_j) = \begin{cases} 0 & \text{if } node(m) = node(r), \\ 1 & \text{if } node(m) \neq node(r). \end{cases} \tag{4.4}$$

From function $P_\alpha$ we can derive the total amount of data that is transferred through the network in the execution of $job_\alpha$.

$$transfer(job_\alpha) = \sum_{ip_j \in I_\alpha} size(ip_j) P_\alpha(ip_j) \tag{4.5}$$

### 4.2.3 Problem Statement

Let $W = \{job_1, ..., job_w\}$ be the set of jobs in the workload sample. Our goal is to minimize the total amount of data transferred over the network in the shuffle phase of jobs involved in $W$:

$$minimize \left( \sum_{job_\alpha \in W} transfer(job_\alpha) \right)$$

by optimizing:

- the assignments of data items to chunks, i.e., *loc* function,

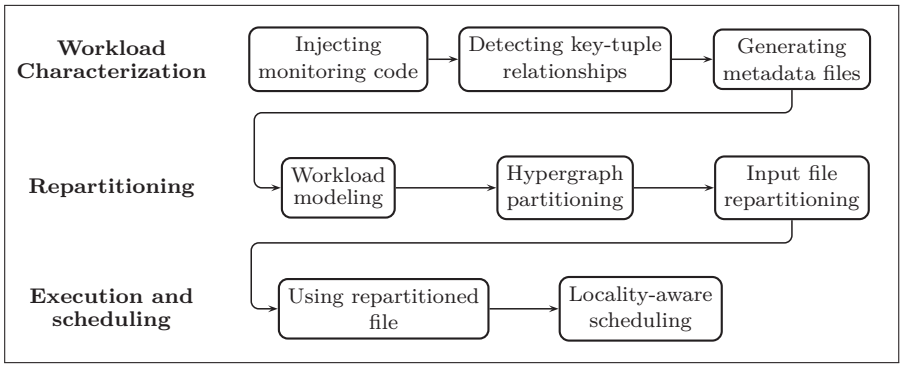



**Figure 4.2:** *MR-Part* workflow scheme

- the reduce partitioning function, i.e., *part* function and
- the scheduling algorithm, i.e., *node* function,

## 4.3 MR-Part

In this section, we propose *MR-Part*, a technique that by automatic partitioning of MapReduce input files allows Hadoop to take full advantage of locality-aware scheduling of intermediate keys, and to reduce significantly the amount of data transferred between map and reduce nodes during the shuffle phase. *MR-Part* proceeds in three main phases, as shown in Figure 4.2: 1) Workload characterization, in which information about the workload is obtained from the execution of MapReduce jobs, and then combined to create a model of the workload represented as a hypergraph; 2) Repartitioning, in which a graph partitioning algorithm is applied over the hypergraph produced in the first phase, and based on the results the input files are repartitioned; 3) Scheduling, that takes advantage of the input partitioning in further executions of MapReduce jobs, and by an intelligent assignment of intermediate keys to reduce tasks decreases the amount of data transferred in the shuffle phase. Phases 1 and 2 are executed offline over the model of the workload, so their cost is amortized over future job executions.

### 4.3.1 Workload Characterization

In order to minimize the amount of data transferred through the network between map and reduce tasks, *MR-Part* tries to perform the following actions: 1) grouping all input tuples producing a given intermediate key in the same chunk and 2) assigning the key to a reduce task executing at the same node.

The first action needs to find the relationship between input tuples and intermediate keys. With that information, tuples producing the same intermediate key are co-located in the same chunk.

### Monitoring

We inject a monitoring component in the MapReduce framework that monitors the execution of map tasks and captures the relationship between input tuples and intermediate keys. This component is transparent to the user program.

The development of the monitoring component is not straightforward because the map tasks receive entries of the form $(\mathcal{K}_1, \mathcal{V}_1)$, but with this information alone we are not able to uniquely identify the corresponding input tuples. However, if we always use the same `InputFormat` and `RecordReader` to read the file, we can uniquely identify an input tuple by a combination of its input file name, its chunk starting offset and the position of `RecordReader` when producing the input pairs for the map task. This is similar to the approach taken in MapReduce's bad records skipping mechanism, as it also needs to uniquely identify the tuples that produce the errors in the execution.

For each map task, the monitoring component produces a metadata file as follows. When a new input chunk is loaded, the monitoring component creates a new metadata file and writes the chunk information (file name and starting offset). Then, it initiates a record counter ($rc$). Whenever an input pair is read, the counter is incremented by one. Moreover, if an intermediate key $k$ is produced, it generates a pair $(k, rc)$. When the processing of the input chunk is finished, the monitoring component groups all key-counter pairs by their key, and for each key it stores an entry of the form $\langle k, \{rc_1, ..., rc_n\}\rangle$ in the metadata file.

### Combination

While executing a monitored job, all metadata is stored locally. Whenever a repartitioning is launched by the user, the information from different metadata files have to be combined in order to generate a hypergraph for each input file. The hypergraph is used for partitioning the tuples of an input file, and is generated using the metadata files created in the monitoring phase.

As we explained in Section 2.2.4, a hypergraph $H = (H_V, H_E)$ is a generalization of a graph in which each hyper edge $e \in H_E$ can connect more than two vertices. In fact, a hyper edge is a subset of vertices, $e \subseteq H_V$. In our model, vertices represent input tuples and hyper edges characterize tuples producing the same intermediate key in a job.

The pseudo-code for generating the hypergraph is shown in Algorithm 4. Initially the hypergraph is empty, and new vertices and edges are added to it as the metadata files are read. The metadata of each job is processed separately. For every job, our algorithm creates a data structure $T$, which stores for each intermediate key, the set of input tuples that produced the key. For every entry in the file, the algorithm generates the corresponding tuple ids and adds them to the entry in $T$ corresponding to the generated key. For easy id generation, we store in each metadata file, the number of input tuples processed for the associated

chunk, $n_i$. In order to produce unique ids from the record numbers we use the following function:

$$generateTupleID(c_i, rc) = \sum_{j=1}^{i-1} n_i + rc \tag{4.6}$$

After processing all metadata of a job, for every read tuple, our algorithm adds a vertex to the hypergraph (if it is not already there). Then, for each intermediate key, it adds a hyper edge containing the set of tuples that have produced the key.

**Algorithm 4:** Metadata combination

**Input**:
$F$: Input file
$W$: Set of jobs composing the workload
**Result**:
$H = (H_V, H_E)$: Hypergraph modeling the workload

```
1  begin
2      H_E ← ∅
3      H_V ← ∅
4      foreach job ∈ |W| do
5          T ← ∅
6          K ← ∅
7          foreach m_i ∈ M_job do
8              md_i ← getMetadata(m_i)
9              if F = getFile(md_i) then
10                 foreach ⟨k, {rc_1, ..., rc_n}⟩ ∈ md_i do
11                     {t_1.id, ..., t_n.id} ← generateTupleID(c_i, {rc_1, ..., rc_n})
12                     T[k] ← T[k] ∪ {t_1.id, ..., t_n.id}
13                     K ← K ∪ {k}
14         foreach intermediate key k ∈ K do
15             H_V ← H_V ∪ T[k]
16             H_E ← H_E ∪ {T[k]}
```

### 4.3.2 Repartitioning

Once we have modeled the workload of each input file through a hypergraph, we apply a min-cut $k$-way graph partitioning algorithm. The algorithm takes as input a value $k$ and a hypergraph, and produces $k$ disjoint subsets of vertices minimizing the sum of the weights of the edges between vertices of different subsets. Weights can be associated to vertices, for instance to represent different sizes. We set $k$ as the number of chunks in the input file. Using the min-cut

algorithm, the tuples that are used for generating the same intermediate key are usually assigned to the same partition.

The output of the algorithm indicates the set of tuples that have to be assigned to each of the input file chunks. Then, the input file should be repartitioned using the produced assignments. However, the file repartitioning cannot be done in a straightforward manner, particularly because the chunks are created by HDFS automatically as new data is appended to a file. We create a set of temporary files, one for each partition. Then, we read the original file, and for each read tuple, the graph algorithm output indicates to which of the temporary files the tuple should be copied. Then, two strategies are possible: 1) create a set of files in one directory, one per partition, as it is done in the reduce phase of MapReduce executions and 2) write the generated files sequentially in the same file. In both cases, at the end of the process, we remove the old file and rename the new file/directory to its name. The first strategy is straightforward and instead of writing data in temporary files, it can be written directly into HDFS. The second one has the advantage of not having to deal with more files but has to deal with the following issues:

- *Unfitted partitions*: The size of partitions created by the partitioning algorithm may be different than the predefined chunk size, even if we set strict imbalance constraints in the algorithm. To approximate the chunk limits to the end of the temporary files when written one after the other, we can modify the order in which temporary files are written. We used a greedy approach in which we select at each time the temporary file whose size, added to the total size written, approximates the most to the next chunk limit.
- *Inappropriate last chunk*: The last chunk of a file is a special case, as its size is less than the predefined chunk size. However, the graph partitioning algorithm tries to make all partitions balanced and does not support such a constraint. In order to force one of the partitions to be of the size of the last chunk, we insert a virtual tuple, $t_{virtual}$, with the weight equivalent to the empty space in the last chunk. After discarding this tuple, one of the partitions would have a size proportional to the size of the last chunk.

The repartitioning algorithm's pseudo-code is shown in Algorithm 5. In the algorithm we represent $RR$ as the `RecordReader` used to parse the input data. We need to specify the associated `RecordWriter`, here represented as $RW$, that performs the inverse function as $RR$. The reordering of temporary files is represented by the function *reorder*().

The complexity of the algorithm is dominated by the min-cut algorithm execution. Min-cut graph partitioning is NP-Complete, however, several polynomial approximation algorithms have been developed for it. In this work we use PaToH [121] to partition the hypergraph. In the rest of the algorithm, an inner loop is executed $n$ times, where $n$ is the number of tuples. *generateTupleID*() can be

**Algorithm 5:** Repartitioning

**Input**:
$F$: Input file
$H = (H_V, H_E)$: Hypergraph modeling the workload
$k$: Number of partitions

**Result**:
$F'$: The repartitioned file

```
1  begin
2      H_V ← H_V ∪ t_virtual
3      {P_1, ..., P_k} ← minCut(H, k)
4      for i ∈ (1, ..., k) do
5          create tempf_i
6      foreach c_i ∈ F do
7          initialize(RR, c_i)
8          rc ← 0
9          while t.data ← RR.next() do
10             t.id ← generateTupleID(c_i, rc)
11             p ← getPartition(t.id, {P_1, ..., P_k})
12             RW.write(tempf_p, t.data)
13             rc ← rc + 1
14     (j_1, ..., j_k) ← reorder(tempf_1, ..., tempf_k)
15     for j ∈ (j_1, ..., j_k) do
16         write tempf_i in F'
```

executed in $O(1)$ if we keep a table with $n_i$, the number of input tuples, for all input chunks. *getPartition*() can also be executed in $O(1)$ if we keep an array storing for each tuple the assigned partition. Thus, the rest of the algorithm is done in $O(n)$.

### 4.3.3 Reduce Tasks Locality-Aware Scheduling

#### Overview

In order to take advantage of the repartitioning, we need to maximize data locality when scheduling reduce tasks. We have adapted LEEN, the algorithm proposed in [74], in which individual keys are assigned dynamically to reduce tasks once all the map tasks have completed. Each node executes a single reduce task. Each (key,node) pair is given a fairness-locality score representing the ratio between the imbalance in reducers input and data locality when a key is assigned to a given reducer. Each key is processed independently in a greedy algorithm. For each key, candidate nodes are sorted by their key frequency in descending order (nodes with higher key frequencies have better data locality). But instead of selecting the node with the maximum frequency, further nodes are considered if they have a better fairness-locality score. The aim of this strategy is to balance reduce input sizes as much as possible. On the whole, we made the following modifications in the MapReduce framework:

- The partitioning function is changed to assign a unique partition for each intermediate key, i.e., $part(k) = k$.
- Map tasks, when finished, send to the master a list with the generated intermediate keys and their frequencies. This information is included in the Heartbeat message that is sent at task completion.
- The master assigns intermediate keys to the reduce tasks relying on this information in order to maximize data locality and to achieve load balancing. As a result, a given reduce task will process as many partitions as intermediate keys are assigned to it.

#### LEEN Scheduling Algorithm

In the LEEN algorithm proposed in [74] two elements are taken into account when scheduling keys: the locality and the fairness. The fairness is defined as the variation on the size of the reduce tasks' input, i.e., a scheduling which assigns keys to reduce tasks in a way in which the input sizes are similar is better (fairer) than a scheduling which produces skew on the reduce input sizes. The goal of the algorithm is to schedule keys so that locality is maximized while maintaining fairness, i.e., not producing skewed distributions on the reducer's input size.

The algorithm's pseudocode is shown in Algorithm 6. It follows a greedy approach in which fairness is computed based on the distribution of the keys that

only consider the partitioning assignments decided so far. That is, in the iteration $(i+1)$, keys from $k_1$ to $k_i$ have been already assigned to a reduce task and this assignment is fixed and only the placement of key $k_{i+1}$ is decided. We assume that node

Locality is defined as follows. Let $k$ be an intermediate key. We define its frequency at node $n$ as the number of intermediate pairs produced in the map tasks executed in $n$ with that key:

$$freq(k,n) = \sum_{\substack{m_j \in M_\alpha: \\ node(m_j)=n}} |\{ip \in output(m_j) : key(ip) = k\}| \tag{4.7}$$

From the frequency we can then define the locality of $k$ when assigned to reduce task $r_i$ as follows:

$$locality(k, r_i) = \frac{freq(k, n_i)}{\sum_{n \in N_\alpha} freq(k, n)} \tag{4.8}$$

Intuitively, the locality of key $k$ at node $n_i$ is the number of intermediate tuples with key $k_i$ that have been produced in map tasks executed at $n$ divided by the total number of tuples with that key.

Keys are first sorted in descending order of their fairness-locality value, $FLK(k)$ (line 3), defined as the standard deviation of the distribution of frequencies of $k$ divided by the maximum frequency:

$$FLK(k) = \frac{\sigma(freq(k))}{\max(freq(k))} \tag{4.9}$$

For each key, nodes are considered in order of their frequency (see Line 5 in Algorithm 6): first the nodes where the frequency is higher (as it would maximize locality). If the next node in the list would produce a more balanced assignment, i.e., with better fairness, it is explored; if not, the current node is retained, as the next one has worse values both on locality and fairness. In order to calculate the fairness we first define $hosted(n)$ as the total number of keys produced in $n$, minus the number of pairs already assigned to other nodes, plus the number of pairs already assigned to $n$. That is, $hosted(n)$ is initialized to the sum of the frequencies of all the keys produced at $n$ in the map phase (see Line 2). For each iteration $i$, if $k_i$ is assigned to $n$, we add the frequency that this key has in the rest of nodes (see Line 13), as those tuples will be transferred to $n$. If $k_i$ is assigned to other node, we subtract the frequency of $k_i$ (see Line 14), as the tuples will be processed in other node. In the algorithm, $fairness(k_i, n)$ represents the standard deviation on $hosted(n)$ if $k_i$ is assigned to $n$.

**Algorithm 6:** LEEN scheduling algorithm

**Input**:
$K$: Set of intermediate keys
$N$: Set of nodes in the cluster
**Result**:
*part*: Partitioning function (assignments): $part(k)$ returns the reducer responsible of key $k$

```
1  begin
2      hosted(n) = Σ_{k∈K} freq(k,n) // number of keys produced at node n
3      K_sorted ← sort(K, FLK(k), desc) // sort keys in descending order of
       fairness-locality
4      for k_i ∈ K_sorted do
5          N_sorted ← sort(N, freq(k_i,n), desc) // sort nodes in descending order of
           k_i frequency
6          for n_j ∈ N_sorted do
7              if fairness(k_i,n_j) ≤ fairness(k_i,n_{j+1}) then
8                  break
9          part(k_i) ← r_j // assign key k_i to reduce task r_j
10         for n ∈ N do
11             // update the number of keys at each node
12             if n ≠ n_j then
13                 hosted(n_j) ← hosted(n_j) + freq(k,n)
14                 hosted(n) ← hosted(n) − freq(k,n)
15     return part
```

### 4.3.4 Improving Scalability

Two strategies can be taken into account to improve the scalability of our approach:

**Grouping of intermediate keys:** In order to deal with a high number of intermediate keys, we use the concept of virtual reducers, *VR*. Instead of using intermediate keys both in the metadata and the modified partitioning function we use $part(key) = (key \mod |VR|)$. Actually, this is similar to the way in which keys are assigned to reduce tasks in the original MapReduce, but in this case we set $|VR|$ to a much greater number than the actual number of reduce tasks. This decreases the amount of metadata that should be transferred to the master and the time to process the key frequencies and also the amount of edges that are generated in the hypergraph.

**Reducing the number of vertices in the graph:** To reduce the number of vertices that should be processed in the graph partitioning algorithm, we perform a preparing step in which we coalesce tuples that always appear together in the edges, as they should be co-located together. The weights of the coalesced tuples would be the sum of the weights of the tuples that have been merged. Let $t_1, t_2 \in H_V$ be two tuples in the graph. We would create a new tuple $t'$ with weight $w(t') = w(t_1) + w(t_2)$ if the following condition is satisfied

$$\forall e \in H_E : t_1 \in e \iff t_2 \in e \tag{4.10}$$

If the condition holds, the appearances of $t_1$ and $t_2$ are replaced by $t'$ in all the edges of the graph. This step can be performed as part of the combination algorithm described in Section 4.3.1.

## 4.4 Experimental Evaluation

In this section, we report the results of our experiments done for evaluating the performance of *MR-Part*. We first describe the experimental setup, and then present the results.

### 4.4.1 Set-Up

We implemented a prototype of *MR-Part* on top of Hadoop-1.0.4 and evaluated it on Grid5000 [64], a large scale infrastructure composed of different sites with several clusters of computers (see Section 2.1.3). In our experiments we employed PowerEdge 1950 servers with 8 cores and 16 GB of memory. We deployed Debian GNU/Linux 6.0 (squeeze) 64-bit in all nodes, and used the default parameters for Hadoop configuration, unless otherwise is indicated.

We tested the proposed algorithm with queries from TPC-H [119], a decision support benchmark. Queries have been written in Pig [94] [1], a dataflow system on top of Hadoop that translates queries into MapReduce jobs. Scale factor (which accounts for the total size of the dataset in GBs) and employed queries are specified on each specific test. After data population and data repartitioning the cluster is rebalanced with a balancing threshold of 5% in order to minimize the effects of remote transfers in the map phase.

As input data, we used `lineitem`, which is the biggest table in TPC-H dataset. In our tests, we used the queries for which the shuffle phase has a significant impact in the response time. In particular, we used the following queries: Q5 and Q9 that are examples of hash joins on different columns, Q7 that executes a replicated join and Q17 that executes a co-group. Note that, for any query data locality will be at least that of native Hadoop.

We compared the performance of *MR-Part* with that of native Hadoop (NAT) and reduce locality-aware scheduling (RLS) [74], which corresponds to changes explained in Section 4.3.3 but over the non-repartitioned dataset. We measured the percentage of transferred data in the shuffle phase for different queries and cluster sizes. We also measured the response time and shuffle time of MapReduce jobs under varying network bandwidth configurations.

### 4.4.2 Results

#### Transferred Data for Different Query Types

We repartitioned the dataset using the metadata information collected from monitoring query executions. Then, we measured the amount of transferred data in the shuffled phase for our queries in the repartitioned dataset. Figure 4.3(a) depicts the percentage of data transferred for each of the queries on a 5 nodes cluster and scale factor of 5. As we can see, transferred data is around 80% in non repartitioned data sets (actually the data locality is always around 1 divided by the number of nodes for the original datasets), while *MR-Part* obtains values for transferred data below 10% for all the queries. Notice that, even with reduce locality-aware scheduling, no gain is obtained in data locality as intermediate keys are produced with similar frequencies in all input chunks.

#### Transferred Data for Different Cluster Sizes

In the next scenario, we have chosen query Q5, and measured the transferred data in the shuffle phase by varying the cluster size and input data size. Input data size has been scaled depending on the cluster size, so that each node is assigned 2GB of data. Fig 4.3(b) shows the percentage of transferred data for the

1. We have used the implementation provided in `http://www.cs.duke.edu/starfish/mr-apps.html`

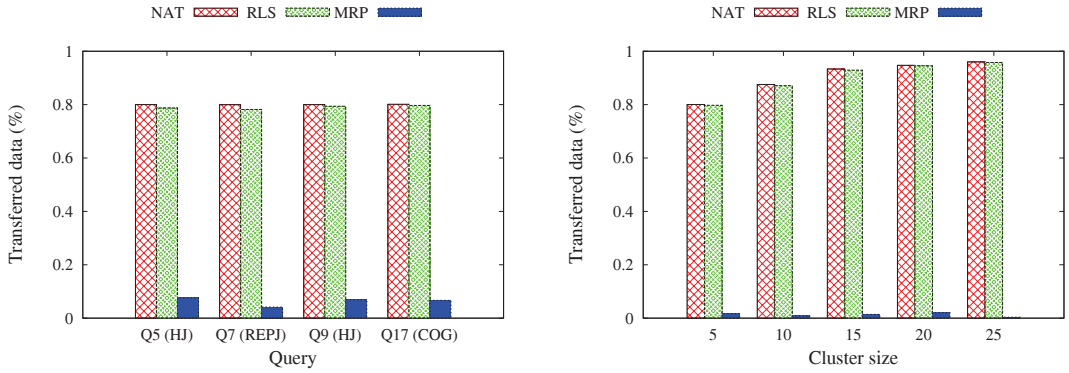


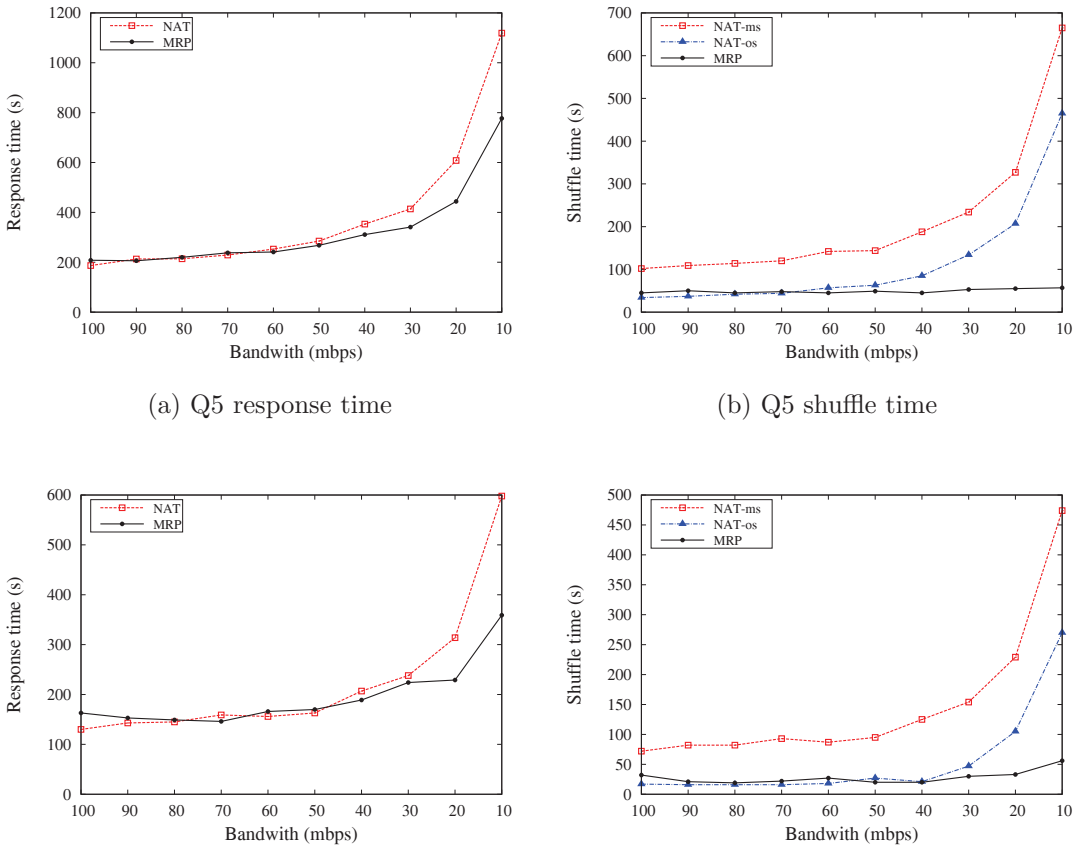

**Figure 4.3:** Percentage of transferred data for a) different type of queries b) varying cluster and data size

three approaches, while increasing the number of cluster nodes. As shown, with increasing the number of nodes, our approach maintains a steady data locality, but it decreases for the other approaches. Since there is no skew in key frequencies, both native Hadoop and RLS obtain data localities near 1 divided by the number of nodes. Our experiments with different data sizes for the same cluster size show no modification in the percentage of transferred data for *MR-Part*.

### Response Time

As shown in previous subsection, *MR-Part* can significantly reduce the amount of transferred data in the shuffle phase. However, its impact on response time strongly depends on the network bandwidth. In this section, we measure the effect of *MR-Part* on MapReduce response time by varying network bandwidth. We control point-to-point bandwidth between nodes with Linux `tc` command line utility. We execute all the queries on a cluster of 20 nodes with scale factor of 40 (40GB of dataset total size).

The results are shown in figure 4.4 and 4.5. As we can see in the left side of the figures, the slower is the network, the bigger is the impact of data locality on response time. To show where the improvement is produced we also report the time spent in data shuffling in the right side of the figures. Measuring shuffle time is not straightforward since in native Hadoop it starts once 5% of map tasks have finished and proceeds in parallel while they are completed. Because of that, we depict two lines: *NAT-ms*, that represents the time spent since the first shuffle byte is sent until this phase is completed, and *NAT-os*, that represents the period of time where the system is only dedicated to shuffling, i.e., from the completion of the last map task to the end of the copy phase. For *MR-Part* only the second line has to be represented as the system has to wait for all map tasks to complete




(a) Q5 response time

(b) Q5 shuffle time

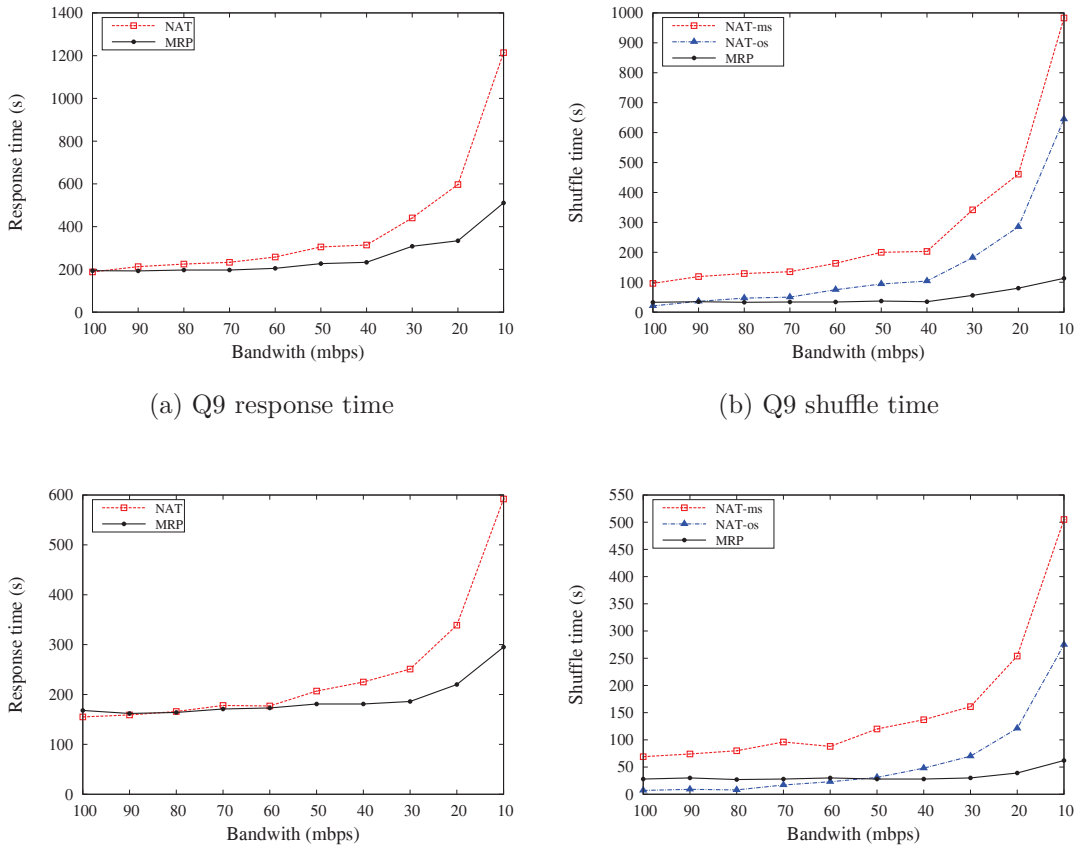
(c) Q7 response time

(d) Q7 shuffle time

**Figure 4.4:** Results for varying network bandwidth in queries Q5 and Q7

in order to schedule reduce tasks (both lines would coincide).

We can observe that, while shuffle time is almost constant for *MR-Part*, regardless of the network conditions, it increases significantly as the network bandwidth decreases for the other alternatives. For instance, in Q5 and with a bandwidth of 10 mbps, the shuffle time in *MR-Part* is just 12% of *NAT-os* (only shuffling) and 8,5% of *NAT-ms*. This results are similar in other queries: 20% (11,8%) in Q7, 17,5% (11,5%) in Q9 and 22% (12%) in Q17. Depending on the query characteristics, the shuffle phase would have more or less importance in the response time. The consequence is that the response time of *MR-Part* ranges from from 42% of that of native Hadoop in Q9 to 69% in Q5. In any case, , with *MR-Part* significant gains can be obtained.




(a) Q9 response time

(b) Q9 shuffle time

(c) Q17 response time

(d) Q17 shuffle time

**Figure 4.5:** Results for varying network bandwidth in queries Q9 and Q17

## 4.5 Related Work

Reducing network transfers in the shuffle phase is important because they may produce a significant overhead in job execution, as already explained in Section 2.3.3. Simulation and experiments on real production clusters under different scenarios have shown how this problem can affect MapReduce's performance [127, 98]. As a consequence, several works have tried to overcome this limitations.

Seo et al. [111] propose a pre-shuffling scheme to reduce data transfers in the shuffle phase. Their approach looks over the input splits before the map phase begins and predicts the reduce task the intermediate key-value pairs will be assigned to. Then, the data is assigned to a map task near the expected future reducer. Similarly, in [67], reduce tasks are assigned to the nodes that maximize data locality, but in this case the decision is taken at reduce scheduling time, once a given percentage of map tasks have finished.

Ibrahim et al. [74] propose a more fine-grained approach, as intermediate keys

are assigned to reduce tasks at scheduling time, instead of using a predefined partitioning function. However, data locality is limited by how intermediate keys are spread over all the map outputs, i.e., if a given key is produced with the same frequency at all nodes, the possible gain is limited, as only the pairs produced at one node can be local. *MR-part* employs this technique as part of the reduce scheduling, but improves its efficiency by partitioning intelligently input data, thus modifying the distribution of intermediate keys among nodes.

In the literature, there have been many other improvements to the MapReduce framework. Some of them are also related to *MR-part* but in a lesser extent. For example, CoHadoop [52] aims to improve the performance of joins by partitioning input datasets over the join column and co-locating the corresponding chunks in the same nodes, thus avoiding the need of a shuffle phase. This approach is only applicable to a very specific type of queries, as opposed to ours which aims at a greater type of jobs. An alternative to repartitioning when executing a set of queries over the same dataset is to store intermediate results as a form of caching, as is proposed in [51]. However, this may pose a high overhead in storage requirements. Our approach, on the other hand, improves queries performance while requiring the same storage size as the original dataset.

Graph and hypergraph partitioning have been used to guide data partitioning in databases and in general in parallel computing, as it was explained in Section 2.2.4. They allow to capture data relationships when no other information, e.g., the schema, is given. The work in [41, 90] uses this approach to generate a database partitioning. The approach in Curino et al. [41] is similar to our approach in the sense that it tries to co-locate frequently accessed data items, although it is used to avoid distributed transactions in an OLTP system.

## 4.6 Discussion

In this section, we show why the related approaches, e.g. [67] and [74], that try to reduce data transfer in the shuffle phase can not be efficient without using proper partitioning of the input. We consider two approaches: 1) locality-aware scheduling of reduce tasks [67]; 2) assignment of intermediate keys at scheduling time [74].

### 4.6.1 Locality-aware Scheduling of Reduce Tasks

The approach taken in [67] consists in scheduling reduce tasks to the nodes that decrease the amount of network transfers. Formally, given a reduce task $r_i$ in job $job_\alpha$, the node chosen for its execution would be:

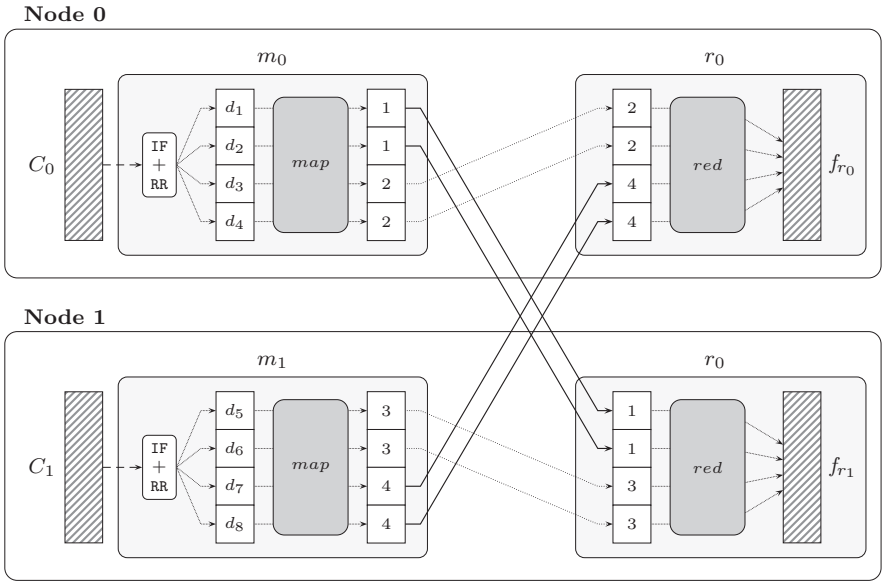



**Figure 4.6:** Limitations of locality-aware scheduling of reduce tasks

$$node(r_i) = \arg\max_{n \in N} \sum_{\substack{m_j \in M_\alpha: \\ node(m_j) \neq n}} |output(m_j) \cap input(r_i)| \tag{4.11}$$

However, this approach does not make any assumption on the reduce partitioning function (by default $part(key) = key \mod |R_\alpha|$). As a consequence, even if all the intermediate pairs could be scheduled locally, the partitioning function may assign them to different reduce tasks. This is the case of the example shown in Figure 4.6. Default partitioning assigns keys 2 and 4 to reduce $r_0$ and keys 1 and 3 to reduce $r_1$. As a consequence, no matter where those tasks are scheduled, always half of the keys have to be transferred through the network.

### 4.6.2 Assignment of Intermediate Keys at Scheduling Time

In [74] one reduce task is assigned to each of the nodes participating in the computation of the MapReduce job, i.e., $node(r_i) = n_i, \forall i \in \{1, ..., |N|\}$. Then, at the end of the map phase, intermediate keys are assigned individually to one of the reduce tasks (bypassing the partitioning function) in a way such that the amount of transferred data is reduced.

Apart from locality, in their algorithm, they also consider load balancing when assigning intermediate tuples. If we ignore that part, which can only worsen data locality, a given intermediate key $ip_j$ is assigned to the reduce task given by this expression:

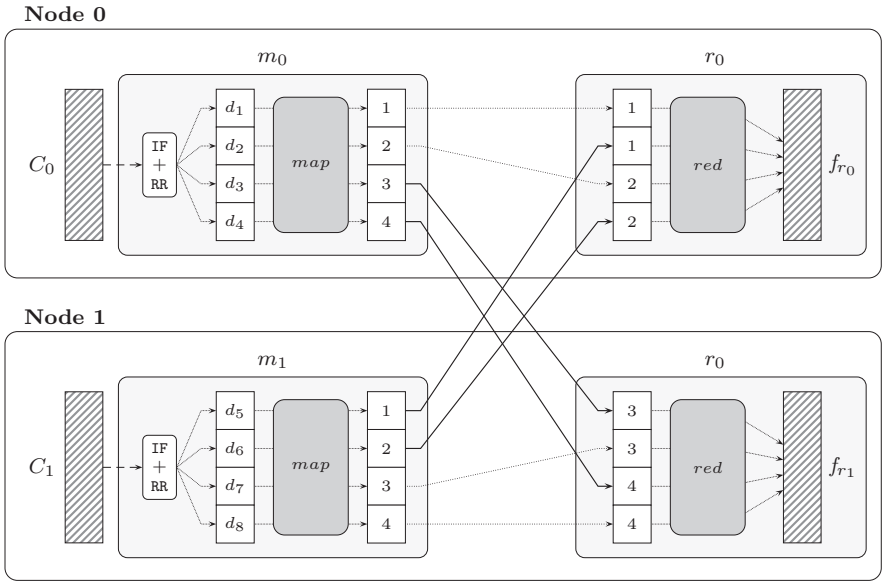



**Figure 4.7:** Limitations of scheduling time key assignment

$$part(k) = \arg\max_{r \in R_\alpha} locality(k, r) \tag{4.12}$$

With this approach, in the example shown in Figure 4.6 no network transfers would be required between map and reduce tasks, as keys 1 and 2 would be assigned to reduce $r_0$ and 3 and 4 to $r_1$ at scheduling time. Nevertheless, if the keys are produced by all map tasks with similar frequencies, no gain is obtained. The extreme case is the scenario shown in Figure 4.7 where all the intermediate keys are generated by all map tasks with equal frequency. In that case, no matter the assignments of the keys, $\frac{|N|-1}{|N|} \times 100\%$ of the keys have to be transferred through the network.

If we employ *MR-Part*, input tuples will be repartitioned, for instance in the following way:

$$loc(d_1) = loc(d_5) = loc(d_2) = loc(d_6) = C_0$$
$$loc(d_3) = loc(d_7) = loc(d_4) = loc(d_8) = C_1$$

That would guarantee the schedule strategy used in [74] would obtain perfect data locality in reduce task execution.

## 4.7 Conclusions

In this chapter, we focused on the shuffle phase of MapReduce computations. In particular, we have studied the problem of remote data transfers, as they can cause a big overhead on the response time of the jobs. The main approach used to tackle this problem is to consider data locality when scheduling reduce tasks. The idea is to allocate those tasks as close as possible to the nodes that executed the map tasks producing their input. However, even if we take the most fine-grained approach, in which individual keys are assigned to reduce tasks, we are limited by how those keys are generated in the map tasks. The idea of our approach is to break that limitation, and in order to do so, we had to modify the input of the map tasks.

We proposed *MR-Part*, an approach that repartitions the input files of MapReduce jobs in order to co-locate the input pairs that frequently generate the same intermediate keys in the same file chunks, as they will be processed by the same map task. *MR-Part* monitors a set of MapReduce jobs and captures the relationships between input tuples and intermediate keys. It makes no assumptions about the structure of the input data or the semantics of the map tasks, a feature very appropriate to MapReduce computations, which aim to be general and schema independent. Of course, some implicit structure is needed, as we work with input tuples that should be equivalent among all the jobs. But this is a very weak assumption.

*MR-Part* uses a hypergraph to model the relationships between input tuples and intermediate keys and then relies on a min-cut graph partitioning algorithm to generate the partitions. The hypegraph model is well suited to our problem since it is independent of the schema and the complexity of the workload. A possible limitation would be its scalability with respect to the number of tuples. However, in MapReduce, tuples may represent large pieces of data, so big datasets do not necessary imply huge amounts of tuples. Moreover, as MapReduce is already deployed in shared-nothing clusters for parallel computing, we can make use of the infrastructure and execute the graph partitioning in parallel, since there already exist some tools that do that.

We built a prototype of *MR-Part* and tested it in the Grid5000 experimental platform. The results show a significant reduction in transferred data in the shuffle phase. This reduction has a significant impact on the response time when the network bandwidth is limited, as the shuffle time is considerably reduced. Reduction of network traffic is important in data centers, where many jobs can be executed in parallel and share some network links which can become a bottleneck. This feature is even more important in some deployment models such as hybrid or community clouds, where some links are especially slower when compared to others. In order to better adapt to those scenarios, alternative scheduling algorithms that account for network links with varying characteristics could also

be developed and evaluated.

# Chapter 5

# MR-Part Prototype

In this chapter, we describe the implementation of *MR-Part* prototype in more details. First, in Section 5.1, we provide a general vision of the prototype and introduce the main components and their role in *MR-Part*. In Section 5.2, we describe the interface for job monitoring and its integration to the MapReduce framework. Section 5.3 describes how this information is combined and employed for repartitioning the input files. Then, in Section 5.4, we give the modifications which we have made in the MapReduce framework in order to modify the reduce scheduling behavior and implement the desired scheduling algorithm. Finally, we conclude by pointing out the key features of our prototype and discuss the main challenges found in the implementation.

## 5.1 Overview

As explained in Chapter 4, MR-Part captures the relationship between input key-value pairs and intermediate keys from a set of MapReduce jobs representing the workload and uses this information in order to repartition the input data and co-locate the pairs that frequently produce the same intermediate keys. At the end of the process, related tuples will be stored either in the same chunk (if a single input file is generated) or in the same file (if one file per partition is generated). In both cases, future MapReduce jobs will assign the tuples that were co-located together by the repartitioning algorithm to the same input splits and, consequently, process them in the same node with the same map task. This will allow future executions of MapReduce jobs to take full advantage of reduce locality-aware scheduling in order to reduce the transfers in the shuffle phase.

*MR-Part* proceeds in three main phases (see Figure 5.1):

- **Monitoring:** the monitoring phase is responsible for capturing the relationships between the input tuples and the intermediate keys from the jobs in the workload model. It is implemented through a component called *collector*, which is integrated into the MapReduce framework and can be

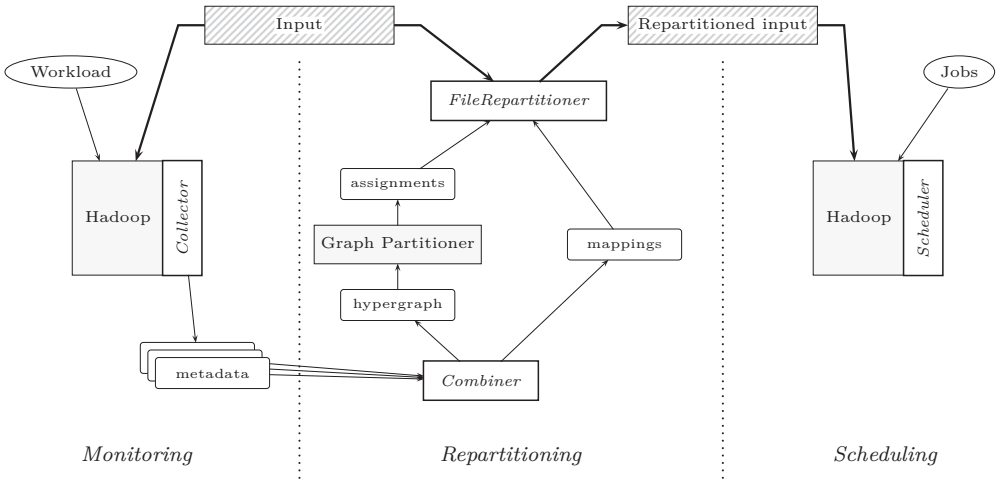



**Figure 5.1:** *MR-Part* implementation overview

activated or deactivated by the user. It generates a set of metadata files, one per map task, that are later used in the repartitioning.

- **Repartitioning:** the repartitioning phase is responsible for reorganizing the tuples of the input dataset according to the metadata collected from the workload in the monitoring phase. It works in three steps: 1) it combines all the metadata files produced by the collector and generates an hypergraph and a mapping between tuple identifiers and their location at the input dataset (file, offset and record counter); 2) it uses a graph-partitioning algorithm, e.g., PaToH, to obtain a set of partition assignments from the hypergraph; and 3) it repartitions the input file based on the obtained partition assignments.
- **Scheduling:** in this phase, future jobs are executed by using the repartitioned input and a scheduling algorithm that is added to the Hadoop framework.

We have implemented MR-Part on top of Hadoop-1.0.4 framework. Two APIs coexist in MapReduce at the moment, the older `org.apache.hadoop.mapred`, maintained for backwards compatibility but most commonly used in the examples of books and manuals, and the newer `org.apache.hadoop.mapreduce`. Some parts of the framework are common to both APIs, but the behavior changes slightly in other parts depending on the choice. In this chapter, whenever both versions are available, we always refer to the newer API.

## 5.2 Monitoring

The objective of the monitoring component of *MR-Part* is to detect the relationships between input key-value pairs and intermediate keys.

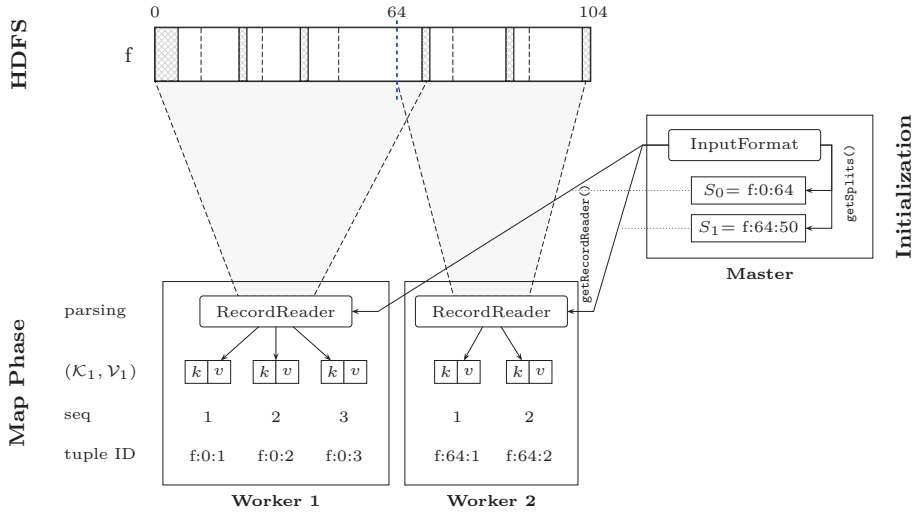



**Figure 5.2:** Tuple identification procedure

The first task of the monitoring component is to uniquely identify input tuples in a transparent way to the user. We have to take into account the following issues:

- There is no constraint on the uniqueness of the key or key-value pairs that are passed as input to the map function. As a consequence, the only way to uniquely identify a given tuple is to consider its position in the sequence of pairs that is obtained when parsing the input.
- The input is not parsed sequentially at a single node, but divided into several splits and parsed in parallel by different map tasks. Input splits are defined by a file path, an offset (starting position) and a length. The `InputFormat` is in charge of creating the splits from the input data in a MapReduce execution. Typically, the input of MapReduce is stored in HDFS and there is a one-to-one correspondence between the file chunks and the input splits. Therefore, we can consider together the tuples generated for chunks of the same file, and order them by the offset of the splits from which they are generated.
- Input tuples are generated from the input splits with the `RecordReader`. This class reads the data from the file system, parses it and generates the corresponding input key-value pairs one after another. Consequently, we can uniquely identify tuples in a split by keeping a record counter which is incremented each time a new tuple is parsed.

We show the procedure in Figure 5.2. The input file is stored in HDFS and contains the input key-value pairs in a specific format. In the figure we depict both the marks that separate different pairs and the marks that separate keys from values within a pair. In the job initialization, the master uses the `InputFormat` class to generate the input splits, one per file chunk. A split is defined by the file path, the starting offset in the file and its length and is assigned to a map task.

When a map task is executed in a worker it creates the appropriate instance of `RecordReader` both from the `InputFormat` and the split that has been assigned to the task and parses the input. As a result it generates a sequence of key-value pairs. Notice that if a pair is cut at the middle of the chunk boundary, the `RecordReader` also reads the beginning of the next chunk in order to retrieve the whole tuple. We assign a sequence number to each pair. The resulting tuple identifier will contain the information necessary to uniquely identify the split (file path and offset) and the sequence number. For instance, f:64:2 represents in the figure the second key-value pair produced when processing the split generated in file f at starting position 64.

Once we have a mechanism to uniquely identify input tuples, we just have to associate the pairs to the values produced by the map function when processing them. We do not need to be aware of the body of the map function since for us, it just consumes one input tuple and generates zero or more intermediate tuples. We link the input tuple identifier and the set of intermediate keys generated by the map function.

The class that performs the mentioned task is called collector. We first define an interface that allows the developer to implement different versions of the collector and then explain how it is integrated into the MapReduce framework.

### 5.2.1 Collector Interface

To collect the information about the relationship between input pairs and intermediate keys, we have defined an interface called `WLCollector`. This interface defines the following methods:

- `startTask(MapTask task, TaskAttemptContext context)`: This method indicates the start of the execution of the map task associated with the collector. Information about the task and the context is passed so that the metadata of the task attempt can be accessed.
- `startSplit(Path f, long offset, long length)`: This method indicates the information about the split from which input tuples are produced. All the intermediate keys (notified through `addRecord()`) produced between the execution of this method and that of `endSplit()` are associated to the split.
- `addRecord(int rc, K1 kIn, V1 vIn, K2 kOut, V2 vOut)`: Each time a new intermediate key is produced, the information about the current record counter (position of the key-value pair in the sequence of generated tuples) along with the input and intermediate key-value pairs are communicated to the collector.
- `endSplit(int rc)`: This method is necessary to store the number of input records produced for a given split. It is necessary since `addRecord()` is only called when an intermediate key is produced and this is not always the case.
- `endTask()`: Communicates the collector that the task has finished its exe-

cution.

Different implementations of this interface can be provided. We have defined several versions, which vary in generality and efficiency. In order to optimize the performance of the collector and reduce the size of the metadata, we make use of the concept of virtual reducer, that we defined in Section 4.3.4. Recall that instead of keeping the individual relationships between intermediate keys and reduce tasks, we consider several keys as a group (virtual reducer) and assign all of them to the same reduce task.

We implement the following classes for the `WLCollector` interface:

- `TupleWLCollector`: It is the most general implementation, which associates the input tuples (using the split information and the record counter) with intermediate keys.
- `HashWLCollector`: Instead of storing the intermediate keys, it stores their `hashCode()`. This is only appropriate when, in the partitioning function, only the information of the `hashCode()` is employed to determine the reduce task. This is the case of the default partitioner in MapReduce.
- `VRWLCollector`: If we are using virtual reducers and the number is fixed among the different jobs, instead of storing the `hashCode()`, we can directly use the assigned virtual reducer. Since they are going to be assigned to the same reduce task by the scheduler, there is no need to treat them differently. As a more optimized data structure, we can use a fixed size array of record counters where the $i$-th position stores the values of the record counters when producing an intermediate key assigned to virtual reducer $i$.

The implementation of the collector is coupled with the implementation of the combiner, as the metadata that is stored in the former is parsed by the latter. From the above mentioned classes, `VRWLCollector` is the most optimal implementation, and it is the one that has been employed in the experimental evaluation. It generates two metadata files: `header`, which contains the information about the split (file, offset and length) and about the intermediate keys; and `rcounters`, which contains the list of record counters ordered by the virtual reducer assigned to the intermediate keys that they generate. The information about intermediate keys is composed of: 1) the number of input tuples; 2) the number of input tuples that produce intermediate keys; 3) the number of generated intermediate keys; and 4) a list of pairs $\langle vr, freq \rangle$ with the number of keys produced for each virtual reducer.

In Figure 5.3, the operation of `VRWLCollector` is depicted through an example. Map $m_i$ is assigned a split created from file `foo` at position 128, with a length of 64. In the left-hand side of the figure, the map task execution is represented. Each input tuple may generate an intermediate key that is then assigned a virtual reducer by the partitioning function. On the top right hand-side of the picture, the main memory structure kept by the collector is represented. This data structure consists of an array storing for each virtual reducer, the tuples producing

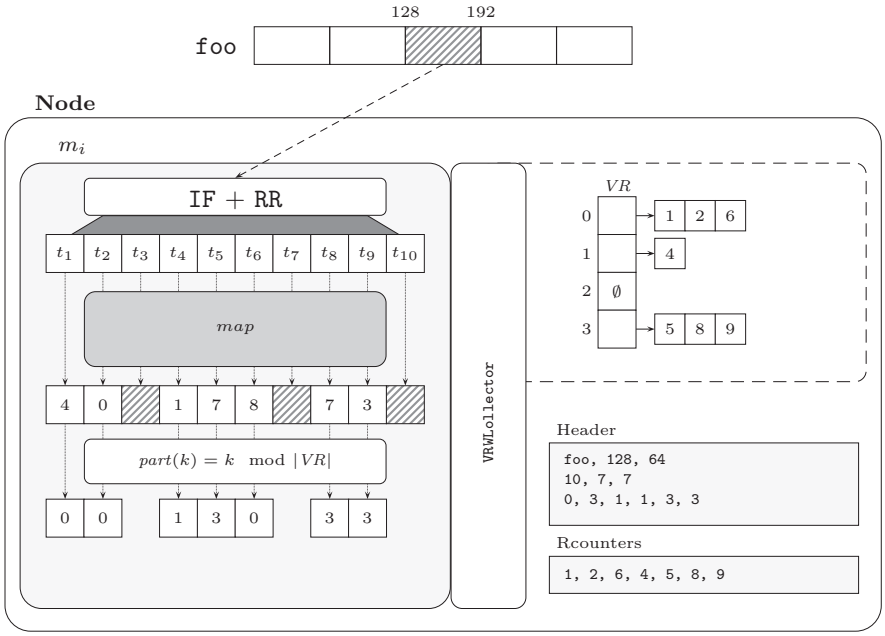



**Figure 5.3:** Example of `VRWLCollector` execution

intermediate keys that were later assigned to the virtual reducer. Finally, on the bottom right side of the figure, the two files generated from this information are depicted. Notice that the information for virtual reducer 2 is not written, as no intermediate keys have been assigned to it.

### 5.2.2 Collector Integration in MapReduce

There are several ways in which the collector can be integrated into the MapReduce framework so that it can be notified whenever a new intermediate key is produced. In our implementation, it is executed transparently when `FileSplit` is employed. `FileSplit` is the subclass of `InputSplit` defined by Hadoop to be used when the input data of a MapReduce job comes from files. All provided `InputFormat` implementations that parse files make use of this class. If other types of input splits are used, the `RecordReader` is responsible of indicating the split information to the collector.

At the beginning of the map execution (`MapTask.runNewMapper()` method) the collector is created and initialized. The specific implementation to be used is indicated by the parameter `mapred.wl.collector.class`. To disable the monitoring, a special implementation called `NoOpWLCollector` should be used.

The management of the split information and record counter can get complicated when using multiple threads or several splits in a map task. In our prototype, we only consider jobs that read one file per split and do not use

`MultithreadedMapper`. We have modified `MapContext` to keep track of the current record count. Whenever a new input pair is read, its value is incremented. Whenever an intermediate key is written, the `addRecord()` is called in the collector.

## 5.3 Repartitioning

Repartitioning of input files is done in three steps: 1) the metadata of different map tasks in different jobs is combined and an hypergraph is generated for each of the input files; 2) the graph partitioning algorithm is executed; and 3) the input files are repartitioned according to the results of the graph partitioning.

### 5.3.1 Metadata Combination

As a result of the job monitoring process, we obtain a set of files with all the workload information, two files per map task: the *header*, with summary information for the split, and the *rcounters*, which contains for each virtual reducer the set of input tuples that produce it. The goal of the metadata combination phase is to create a graph representing the workload of all those files. As mentioned in the previous section, the combiner is coupled with the chosen implementation, as it has to parse the produced metadata files. We have an interface, called `WLCombiner`, and several implementations.

The execution of the combiner is performed in two main steps, mainly because of efficiency reasons. In the first step, general information from the metadata is retrieved. This information is used to execute more efficiently the second step, where the information about the relationships between input tuples and intermediate keys is actually processed. The steps for executing the combiner are as following:

1. In this step, only the *header* file of each map task is read. The following information is retrieved:
   - The number of input tuples generated at each split ($n_i$): this information is used to assign consecutive integers as tuple identifiers, e.g., split 1 would assign ids from 1 to $n_1$, split 2 from $n_1 + 1$ to $n_1 + n_2$, etc.
   - The frequency of intermediate keys for each virtual reducer: this allows to represent the hyperdges as fixed-size arrays, which is a more compact data structure.
2. In this step, the *rcounters* files are processed and the hypergraph is generated. If there is more than one input file (this information is obtained when reading the headers), each file is processed independently and produces a different hypergraph. For a given file, one job is handled after another. Within a job, the sets of record counters producing the same key at different

splits are merged by just concatenating the arrays, as there are no intersections. Each of these sets generates an hyperedge in the graph. Notice that it could be possible to generate exactly the same hyperedge several times for different jobs. However, modifying the corresponding weight has the same effect as writing the edge several times in the hypergraph. Computing the weights would require to keep all the information in memory between the processing of different jobs and perform plenty of set comparisons. That is why, repeating them in the file is preferred and after processing each job all the hyperedges are written to the file and the memory is freed.

In the experimental evaluation we have used `VRWLCombiner`, which is the combiner associated to `VRWLCollector`. This combiner incorporates the tuple coalescing strategy into the combining process, requiring an additional pass over the *rcounters* files and producing an additional file with the mapping between the original tuple identifiers and the new coalesced tuples identifiers. Coalescing is the strategy proposed in Section 4.3.4 to reduce the size of the hypergraph. It merges the vertices that are equivalent in the graph, i.e., belong exactly to the same set of edges, into a unique virtual vertex.

**Algorithm 7:** Coalescing Procedure

**Input**:
*CoalMap*: Array with tuple mappings
*VRtoID*: Producing tuple ids for virtual reducer
*nextId*: identifier to assign to next virtual tuple
**Result**:
*CoalMap*: Modified tuple mappings
*nextId*: updated identifier for next virtual tuples

```
1  begin
2      for i = 0 to |VR| do
3          aux ← emptyMap()
4          foreach id ∈ VRtoID(i) do
5              oldId ← CoalMap[id]
6              if ∃ aux(oldId) then
7                  newId ← aux(oldId)
8              else
9                  newId ← nextId
10                 aux(oldId) ← newId
11                 nextId ← nextId + 1
12             CoalMap[id] ← newId
```

For tuple coalescing an array needs to be kept where position $i$ represents the mapping of tuple $i$ to a given virtual tuple $j$. At the end of the process, all the

tuples with the same mapping value are considered as a single virtual tuple. The pseudo-code of the coalescing procedure is shown in Algorithm 7. It is executed each time the metadata of a given job is read. It receives the coalescing mapping array (*CoalMap*), the data structure containing the relationships between tuples and virtual reducers for the job (*VRtoID*), and the next identifier to be assigned when a new virtual tuple is created. Let $|I_\alpha|$ be the number of intermediate tuples in job $\alpha$, then the complexity of the algorithm is $O(|I_\alpha|)$. If at most one intermediate pair is created by input key (which is the most frequent situation), then the execution of the coalescing process for all the jobs is in the order of $O(n \times |W|)$ in the worst case, where $n$ is the total number of input tuples and $|W|$ the number of jobs in the workload. The space complexity of the algorithm is $O(n)$.

As mentioned before, when using tuple coalescing, the metadata files are read twice, first to generate the coalescing and then to generate the hypergraph file from the monitoring information and the tuple mappings. This avoids to keep in memory more than $O(|I_\alpha|)$ elements at a time.

Once the mapping between input tuples and intermediate keys is obtained, it can be used to generate the graph partitioning algorithm's input. Weights of coalesced tuples include the total count of original tuples that have been merged. If a single input file is created (see Section 5.3.3), a fake tuple is generated with a weight proportional to the unused space of the last chunk.

If coalescing is used, the combiner generates a file with the mapping between tuple locations and the corresponding virtual tuple. Otherwise the mapping would not need to be stored individually as simply applying the formula in Equation 4.6 is enough. Only the number of input tuples per split and their offsets are required.

### 5.3.2 Graph Partitioning

For partitioning the hypergraphs, we have used PaToH [121], a library for hypergraph partitioning based on multilevel hypergraph bisection. This library defines a set of functions to be used in C, where the input is passed as variables in memory, but can also be used as a standalone program. In this case, the input is formatted in text files, which are parsed and loaded into memory by the library itself. The output is also given in a text file.

In our implementation we have used the second standalone version of PaToH. Consequently, the combiner generates a file with the hypergraph representation and this file is passed as input to PaToH. The file is divided into three parts: 1) the header, which specifies the index base (0 or 1), the number of vertices, edges and pins (one vertex in an edge accounts for a pin), and the used weighting scheme (no weights, weights in tuples, edges or both); 2) the edges, one line per hyperedge with the connected tuples; and 3) the tuple weights, which are specified

in the final line. If the same hyperedge is generated in several jobs, it is repeated in the file instead of modifying its weight.

The number of partitions into which the graph is divided is equal to the number of chunks in which the original input file was stored. The rest of the parameters used in the partitioning are set to default values. The output of the execution is a new file which contains a list of integers, one per input tuple with the assigned partition id.

### 5.3.3 File Repartitioning

The last step of *MR-Part* consists of repartitioning the input files according to the results obtained in the graph partitioning. Two implementations of this phase are provided: `SingleFileRepartitioner`, which repartitions each input file into a new single file and `MultiFileRepartitioner`, which generates one new input file for each fragment of the partitioning.

Both implementations extend from `FileRepartitioner`, an abstract class which implements the common parts of the process. The repartitioning of each input file consists of the following steps:

1. Read the file properties, including splits' information, `RecordReader`, etc. All this information has been generated by the combiner from the metadata gathered by the collectors.
2. Load the partitioning assignments into memory. In the case of PaToH, the assignments are stored in a text file, as explained in the previous section. Note that with coalescing, the assignments associate virtual tuple ids with fragments.
3. Read the input file, apply the partitioning and generate $k + 1$ temporary files, where $k$ is the number of chunks into which the input file is divided. For each chunk, the following procedure is performed:
   - **3.1.** Read the mappings of the tuples corresponding to the current chunk and store them in memory.
   - **3.2.** Read the chunk tuples one by one, keeping the count of the current record number within the chunk.
   - **3.3.** Obtain the assigned partition for the given tuple using the mapping between locations and tuple identifiers (produced in the combiner) and the partition assignment (produced by the graph partitioner).
   - **3.4.** Write the tuple into the temporary file corresponding to the assigned partition. If no partition has been assigned (because the tuple never produces intermediate keys), the tuple is written into a special temporary file, which we call leftovers file.
4. Based on the partitioner implementation, do one of the following options:

- `MultiFileRepartitioner`: This implementation is straightforward as it just writes each temporary file into a new HDFS file. Tuples in the leftovers file are used in the smaller files to make their sizes balanced, as they can be placed anywhere without harming the partitioning.
- `SingleFileRepartitioner`: In this case, the temporary files are written one after another in the same HDFS file. As explained in Section 4.3.2, the size of the temporary files may not correspond to that of the chunks and this may cause tuples assigned to one partition to be written in the chunk associated to another partition. In order to minimize the effects of that scenario, the temporary files are first reordered in order to approximate the chunk and the file boundaries.

## 5.4 Scheduling

In order to include the scheduling strategy in the MapReduce framework, we modified several parts:

- Information about the frequency of the intermediate keys is captured and sent to the master (jobtracker) when the map task completes, as it is required by the scheduling algorithm.
- The reduce scheduling algorithm is modified so as to implement the desired scheduling strategy. The user specifies the strategy to be used through a MapReduce property. LEEN algorithm, presented in Section 4.3.3, is one of the possible strategies to use.
- The map tasks partition the intermediate keys by assigning them to virtual reducers, as proposed in Section 4.3.4. As a consequence the spills generated from the buffer are also partitioned in as many parts as virtual reducers.
- The shuffle mechanism has been extended in order to include the possibility of specifying which virtual reducers a given reduce task needs to fetch. The data corresponding to those virtual reducers is merged before being sent to the reduce task.

### 5.4.1 Frequency Information

The scheduling algorithm (see Algorithm 6 in Section 4.3.3) needs to know the number of tuples (frequency) for each key produced at each node. We have modified the MapReduce framework in order to collect and transfer this information to the algorithm.

Map tasks capture information about the number of intermediate pairs that have been produced for each virtual reducer. The implementation of this mechanism is simple, as it only requires to keep an array which is updated each time `RecordWriter.write()` method is called.

When a map task completes its execution, a heartbeat message is sent to the master (jobtracker) with information about the task, which is stored in a class called `MapTaskStatus`. We have modified this class in order to transfer the array including the virtual reducers frequencies.

### 5.4.2 Reduce Scheduler

The information about the frequencies produced by the map tasks should be collected by the master and then used in order to assign virtual reducers to reduce tasks. We have defined a new interface called `ReduceScheduler` that is injected into the reduce task scheduling mechanism of MapReduce in order to implement the desired strategy, e.g., LEEN algorithm. The user is able to provide an implementation of this class and instruct the framework to use it by modifying the parameter `mapred.jobtracker.taskScheduler.reduceScheduler.class`. The main methods of this interface are the following:

- `mapFinished(MapTaskStatus status)`: This method is called each time a map task reports its completion. The reduce scheduler may then query the status in order to get the needed information, e.g., the virtual reducer frequencies. Recall that this information have been included in the modified status.
- `calculateReduceAssignments(int clusterSize)`: This method is called once all the map tasks have finished, when all the frequencies information is available. It executes the reduce scheduling algorithm and produces the assignments of virtual reducers to reduce tasks. For instance, if the LEEN algorithm is used, it will execute the code corresponding to Algorithm 6.
- `getNextReduceID(String host)`: Gets the id of the next reduce task to be executed in the specified host. In the LEEN algorithm, each host is assigned a single reduce task. Hence, the first call to this method would return the id of the reduce task to be executed at that node and future calls will return invalid ids.
- `getVRs(int reduceID)`: Returns an array with the virtual reducers that have to be processed by the reduce task with the specified id.

We provide several implementations of `ReduceScheduler` interface, including `DefaultScheduler`, which implements the default MapReduce scheduling behavior, `LEENScheduler`, which implements Algorithm 6, and `DataLocalityScheduler`, which assigns intermediate keys by just considering data locality and not fairness.

The `ReduceScheduler` is loaded by the `JobInProgress` when created. This class manages the execution of a single job and is called by the jobtracker whenever a relevant event occurs, e.g., a map task finishes. When a task is reported complete, method `completedTask()` at `JobInProgress` is called and the information about the task status is passed. If the task is a map, this information is forwarded to the reduce scheduler through the `mapFinished()` method. If the

completed task is the last map task, then `calculateReduceAssignments()` is executed on the scheduler. This is because the scheduling algorithm needs the information about all key frequencies and this is only available at the end of the map phase.

When a tasktracker reports that a reduce slot (processor) is idle, the method `obtainNewReduceTask()` is called in one `JobInProgress`. The selection of the task to execute is delegated to the reduce scheduler. Recall from Section 2.3.2 that scheduling in MapReduce is done in two steps: first a job is selected and then a task within the job. The first step is done by the MapReduce framework as usual. The second step is carried out by the reduce scheduler. If a valid id is returned, the reduce scheduler is queried in order to obtain the list of virtual reducers assigned to that task (method `getVRs(int reduceID)`). This list is written into the reduce task configuration and then the task sent to the tasktracker selected for its execution. Invalid ids can be returned if no more schedule tasks are planned to execute at that node.

### 5.4.3 Shuffle Mechanism

The shuffle mechanism has been modified in order to manage the virtual reducers. The changes affect both the way in which intermediate pairs are partitioned by the map task and the communication protocol that is used between the reduce tasks and the tasktracker in order to retrieve the reduce input.

#### Number of Virtual Reducers

A new MapReduce configuration parameter has been included in order to specify the number of virtual reducers in the scheduling mechanism. The partitioning function will use this number instead of the actual number of reduce tasks when called. The consequence is that at the end of the map execution the output will be divided into $|VR|$ files, one per virtual reducer. As in default MapReduce, the pairs within these files are ordered by the `Comparator` specified by `mapred.output.key.comparator.class` parameter.

#### Reduce Input Retrieval

When a new reduce task is scheduled to a node, the corresponding instance of `ReduceTask` gets the information about the virtual reducers that has been assigned to it (which has been stored in the configuration). The protocol for the communication of this task with the tasktrackers that store the map outputs has been slightly modified. The HTTP request has been converted to use the `POST` method, since the number of required virtual reducers may be big and it may not be possible to include them in the HTTP query string. This list is written into the message body. The HTTP server at the tasktracker reads this list from

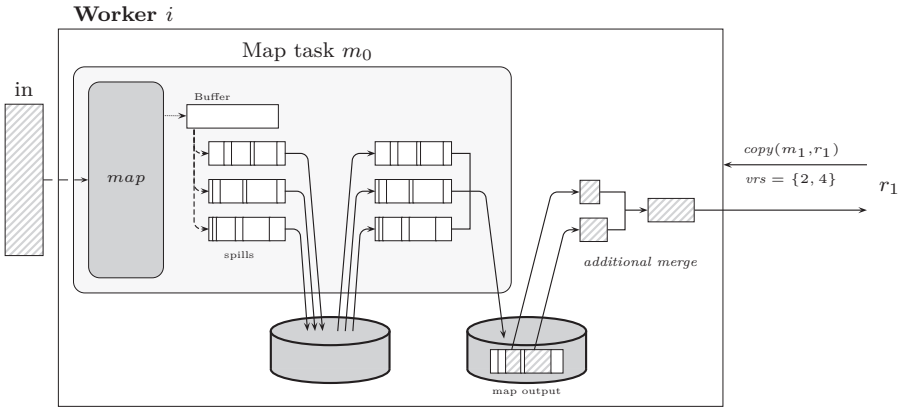



**Figure 5.4:** Modified shuffle

the message and merges the files corresponding to the requested virtual reducers before sending them to the reduce task. This step is needed as the reduce task expects the map output to be sorted to be able to merge the outputs coming from different map tasks.

An example of the modified shuffle phase is shown in Figure 5.4. In this case, we use 6 virtual reducers and 2 reduce tasks. When a reduce task, in this case $r_1$, sends a request, it also indicates the set of virtual reducers that it wants to retrieve, in this case, 2 and 4. The worker then reads the data corresponding to those virtual reducers and perform an additional merge before sending the response.

## 5.5 Conclusions

In this chapter, we described the implementation details of the *MR-Part* prototype, which we have integrated into Hadoop-1.0.4. Both the monitoring and the scheduling components are incorporated into Hadoop, while the repartitioning components are executed offline and outside the Hadoop framework.

Whenever possible, we have defined a general interface and provided one or more implementations. This allows to test several strategies with minimum effort and makes the maintenance and extension of the prototype simpler. In this way, alternative strategies can be envisioned for workload monitoring, but even more useful is the possibility of defining specific scheduling algorithms. New monitoring or scheduling implementations can be packaged in a `jar` file and loaded dynamically without having to modify the Hadoop framework code.

The Hadoop implementation of MapReduce is complex and difficult to modify. Its complexity comes, not only from the number of issues that are managed by the MapReduce framework, but also from the objective of making the framework as extensible as possible in order to provide the developers with a lot of freedom in

terms of implementation choices. Moreover, the coexistence of two APIs requires to duplicate many parts of the code and makes the Hadoop code difficult to understand and modify.

In our prototype, we have tried to make its applicability as wide as possible. As a consequence, we have not constrained our implementation to specific formats or job types but used the general interfaces provided in Hadoop. This covers the big majority of MapReduce jobs that are usually implemented. However, some of the most advanced features require a lot of engineering effort and have been left for future work, including a more transparent integration with multi-layered input format strategies (complex versions of `InputSplit` or the management of multi-threaded execution of map tasks (`MultithreadedMapper`).

# Chapter 6

# Conclusions

In this chapter, we summarize and discuss the main contributions made in the context of this thesis. Then we give some research directions for future work.

## 6.1 Contributions

This thesis is in the context of big data applications, using large-scale parallelism for the efficient processing and management of large datasets. In particular, we focused on the problem of data partitioning, which is fundamental to yield parallel processing and thus improve the performance of applications that deal with big datasets. We have addressed two problems in particular:

### 6.1.1 Dynamic Partitioning in Continuously Growing Large Databases

We tackled the problem of automatic partitioning in large scientific databases where new data items are inserted as new measurements are carried out. We identified the main limitations of existing approaches, basically: 1) the inefficiency of automatic approaches based on the basic techniques to handle the complexity of scientific applications; and 2) the overhead of executing partitioning algorithms over the whole dataset in the case of graph-based approaches. As a solution, we proposed two algorithms, *DynPart* and *DynPartGroup* that dynamically allocate the new data items based on the affinity they have with the current fragments in the partitioning. The first algorithm processes new data items one by one, finds the fragment with the highest affinity that satisfies the imbalance constraints and makes the partition assignment. *DynPartGroup*, on the other hand, performs an additional step in which equivalent data items are grouped together. Then, the assignments are carried out group by group from the biggest to the smallest group. This strategy requires less affinity calculations and deals better with imbalance constraints.

We evaluated our dynamic approach through implementation and compared its execution time with that of a static graph-based partitioning approach. The results show that the running time of the static approach increases as the database grows in size, while that of our algorithms remains stable. They also reveal that with our algorithms the partitioning efficiency is preserved when the database grows in size.

The experimental results also show that *DynPartGroup* algorithm presents a very good behavior when there is a high correlation between arriving elements, and this is not affected by the imbalance constraints on the fragments.

We have tested our algorithms in the context of astronomical catalogs, which store data elements with a high number of attributes which are queried through complex workloads. Overall, our experimental evaluation illustrates that our partitioning strategy is very good at efficiently handling the data partitioning in our astronomy application, but the results are not constrained to this application and can be employed in many other applications where data is continuously appended into the database.

### 6.1.2 Data Partitioning for Reducing Network Traffic in MapReduce

MapReduce has become one of the most popular frameworks for large scale data analysis and has been subject of many works aiming to improve its efficiency. We identified the overhead that the MapReduce's shuffle causes when large intermediate data transfers are involved. Then, we proposed *MR-Part*, a strategy that monitors a MapReduce workload by capturing relationships between input tuples and intermediate keys and repartitions the input files so that locality-aware reduce scheduling strategies can obtain the maximum benefit.

To validate our proposal, we built a prototype of *MR-Part* by modifying and extending the Hadoop framework, the most popular framework in open source for MapReduce. The prototype has been designed with the aim of maximum applicability and it is not limited to any particular type of MapReduce jobs. Indeed it is highly customizable as it provides several interfaces that allow the developers to implement personalized strategies with minimum effort.

We evaluated *MR-Part* in the Grid5000 experimental platform. The results reveal a significant reduction on the amount of data that is transferred through the network in the shuffle phase. This reduction has a significant impact on the total execution time when the network bandwidth is limited, as the shuffle time is considerably reduced.

## 6.2 Directions for Future Work

The results presented in this thesis leave room to further improvement. Some future directions of research are:

- **Adaptation to workload evolution:** We can consider changes in the workload of the partitioning when using the *DynPart* or *DynPartGroup* algorithms, which may lead to a degradation of partitioning efficiency. In order to deal with those scenarios, efficient reorganization of data items is required. This process should take into account the evolution of the partitioning efficiency and the cost of the data transfers needed to attain the new data partitioning.
- **Clustering of similar queries:** Queries with similar characteristics, e.g., accessing similar sets of data items, may be grouped together when constructing groups in the *DynPartGroup* algorithm (CreateGroups() function). This would allow *DynPartGroup* to execute even faster as less affinities should have to be calculated and reduce the amount of metadata, as only clusters and not individual queries would have to be taken into account.
- **Fully parallelization of the *MR-Part* approach:** The main goal is to make the system more scalable. This would require the usage of parallel partitioning libraries, e.g., Zoltan [26], and its integration into parallel versions of the combination and repartitioning algorithms, which could profit of the MapReduce framework for that task.
- **Improved scheduling of reduce tasks:** In particular, we plan to develop new intermediate key scheduling algorithms that take into account not only the data locality but also the characteristics of the network. A possible approach would be to design a cost model including the estimated cost of rack and off-rack data transfers and their impact on response time when used in combination with load balancing.

# Bibliography


[1] D. J. Abadi, S. R. Madden, and N. Hachem, « Column-stores vs. row-stores: how different are they really? », in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 967–980.

[2] A. Abouzeid, K. B. Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, « HadoopDB: an architectural hybrid of mapreduce and dbms technologies for analytical workloads », *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 922–933, Aug. 2009.

[3] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, « Database tuning advisor for Microsoft SQL Server 2005: demo », in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 930–932.

[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, « Automated selection of materialized views and indexes in sql databases », in *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000, pp. 496–505.

[5] S. Agrawal, V. R. Narasayya, and B. Yang, « Integrating vertical and horizontal partitioning into automated physical database design », in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 359–370.

[6] R. K. Ahuja, Özlem Ergun, J. B. Orlin, and A. P. Punnen, « A survey of very large-scale neighborhood search techniques », *Discrete Applied Mathematics*, vol. 123, no. 1–3, pp. 75 –102, 2002.

[7] A. Ailamaki, V. Kantere, and D. Dash, « Managing scientific data », *Communications of the ACM*, vol. 53, no. 6, pp. 68–78, Jun. 2009.

[8] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, « Weaving relations for cache performance », in *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001, pp. 169–180.

[9] *Amazon Elastic Block Store (EBS)*, `http://aws.amazon.com/ebs`, 2013.

[10] *Amazon Elastic Compute Cloud (Amazon EC2)*, `http://aws.amazon.com/ec2`, 2013.

[11] *Amazon Elastic MapReduce*, `http://aws.amazon.com/elasticmapreduce/`, 2013.

[12] *Amazon Relational Database Service (Amazon RDS)*, `http://aws.amazon.com/es/rds/`, 2013.

[13] *Amazon Simple Storage Service (Amazon S3)*, `http://aws.amazon.com/s3`, 2013.

[14] *Amazon Virtual Private Cloud (Amazon VPC)*, `http://aws.amazon.com/vpc/`, 2013.

[15] *Amazon Web Servoces*, `http://aws.amazon.com/`, 2013.

[16] G. Amdahl, « Validity of the single processor approach to achieving large scale computing capabilities », in *Proceedings of the AFIPS spring joint computer conference*, 1967, pp. 483–485.

[17] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, « Reining in the outliers in map-reduce clusters using Mantri », in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010, pp. 1–16.

[18] E. Anderson and J. Tucek, « Efficiency matters! », *SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 40–45, Mar. 2010.

[19] *Apache Hadoop*, `http://hadoop.apache.org`, 2013.

[20] *Apache Hadoop NextGen MapReduce (YARN)*, `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`, 2013.

[21] *Apache HBase*, `http://hbase.apache.org`, 2013.

[22] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, « Above the clouds: a Berkeley view of cloud computing », EECS Department, University of California, Berkeley, Technical report, Feb. 2009.

[23] *Aster MapReduce Appliance*, `http://www.asterdata.com/product/deployment/appliance.php`, 2013.

[24] B Bergsten, M Couprie, and P Valduriez, « Prototyping DBS3, a shared-memory parallel database system », in *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, Dec. 1991, pp. 226–234.

[25] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[26] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco, *Zoltan 3.0: parallel partitioning, load-balancing, and data management services; user's guide*, Tech. Report SAND2007-4748W, `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`, Sandia National Laboratories, 2007.

[27] P. Boncz, M. Zukowski, and N. Nes, « MonetDB/X100: hyper-pipelining query execution », in *Second Biennial Conference on Innovative Data Systems Research*, 2005, pp. 225–237.

[28] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, « The HaLoop approach to large-scale iterative data analysis », *The VLDB Journal*, vol. 21, no. 2, pp. 169–190, Apr. 2012.

[29] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, « HaLoop: efficient iterative data processing on large clusters », *Proceedings of the VLDB Endowment*, vol. 3, pp. 285–296, Sep. 2010.

[30] M. Burrows, « The Chubby lock service for loosely-coupled distributed systems », in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 335–350.

[31] N. Capit and J. Emeras, *OAR documentation - user guide*, `http://oar.imag.fr/dokuwiki/doku.php`, Laboratoire d'Informatique de Grenoble, 2012.

[32] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, « PVFS: a parallel file system for Linux clusters », in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 317–327.

[33] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, « Bigtable: a distributed storage system for structured data », *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, 2008.

[34] S. Chaudhuri and V. Narasayya, « Self-tuning database systems: a decade of progress », in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 3–14.

[35] S. Chaudhuri and V. R. Narasayya, « Autoadmin "what-if" index analysis utility », in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998, pp. 367–378.

[36] G. Chockler, I. Keidar, and R. Vitenberg, « Group communication specifications: a comprehensive study », *ACM Computiong Surveys*, vol. 33, no. 4, pp. 427–469, Dec. 2001.

[37] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears, « MapReduce online », in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010, p. 21.

[38] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, « PNUTS: Yahoo!'s hosted data serving platform », *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.

[39] G. Copeland, W. Alexander, E. Boughter, and T. Keller, « Data placement in Bubba », in *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1988, pp. 99–108.

[40] G. P. Copeland and S. N. Khoshafian, « A decomposition storage model », in *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, 1985, pp. 268–279.

[41] C. Curino, E. Jones, Y. Zhang, and S. Madden, « Schism: a workload-driven approach to database replication and partitioning », *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 48–57, Sep. 2010.

[42] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, « Automatic SQL tuning in Oracle 10g », in *Proceedings of the Thirtieth international conference on Very large data bases*, 2004, pp. 1098–1109.

[43] J. Dean and S. Ghemawat, « MapReduce: simplified data processing on large clusters », in *6th Symposium on Operating System Design and Implementation*, 2004, pp. 137–150.

[44] ——, « MapReduce: a flexible data processing tool », *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010.

[45] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, « Dynamo: Amazon's highly available key-value store », in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, vol. 41, 2007, pp. 205–220.

[46] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, « New challenges in dynamic load balancing », *Applied Numerical Mathematics*, vol. 52, pp. 133–152, Feb. 2005.

[47] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsaio, and R. Rasmussen, « The Gamma database machine project », *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 44–62, Mar. 1990.

[48] D. J. DeWitt and J. Gray, « Parallel database systems: the future of high performance database systems », *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, Jun. 1992.

[49] D. J. DeWitt, M. Smith, and H. Boral, « A single-user performance evaluation of the teradata database machine », in *High Performance Transaction Systems*, 1989, pp. 243–276.

[50] J. Dittrich, J. A. Q. Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, « Hadoop++: making a yellow elephant run like a cheetah (without it even noticing) », *Proceedings of the VLDB Endowment*, vol. 3, pp. 515–529, 1-2 Sep. 2010.

[51] I. Elghandour and A. Aboulnaga, « ReStore: reusing results of MapReduce jobs in Pig », vol. 5, pp. 586–597, 6 Feb. 2012.

[52] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, « CoHadoop: flexible data placement and its exploitation in Hadoop », *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 575–585, Jun. 2011.

[53] C. M. Fiduccia and R. M. Mattheyses, « A linear-time heuristic for improving network partitions », in *Proceedings of the 19th Design Automation Conference*, 1982, pp. 175–181.

[54] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata, « Column-oriented storage techniques for MapReduce », *Proceedings of the VLDB Endowment*, vol. 4, pp. 419–429, Apr. 2011.

[55] I. Foster, « What is the grid? - a three point checklist », *GRIDtoday*, vol. 1, no. 6, Jul. 2002.

[56] I. Foster, C. Kesselman, and S. Tuecke, « The anatomy of the Grid - enabling scalable virtual organizations », *International Journal of Supercomputer Applications*, vol. 15, 2001.

[57] François Pellegrini, *PT-Scotch and libPTScotch 6.0. User's Guide*, `http://www.labri.u-bordeaux.fr/perso/pelegrin/scotch/`, Université Bordeaux 1 & LaBRI, Dec. 2012.

[58] E. Friedman, P. Pawlowski, and J. Cieslewicz, « SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions », *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1402–1413, Aug. 2009.

[59] S. Ghandeharizadeh and D. J. DeWitt, « Hybrid-range partitioning strategy: a new declustering strategy for multiprocessor databases machines », in *Proceedings of the sixteenth international conference on Very large databases*, 1990, pp. 481–492.

[60] S. Ghemawat, H. Gobioff, and S.-T. Leung, « The Google file system », in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, vol. 37, 2003, pp. 29–43.

[61] *Google app engine*, `https://cloud.google.com/products/app-engine`, 2013.

[62] *Google Apps for business*, `http://www.google.com/intx/en/enterprise/apps/business/`, 2013.

[63] *Google compute engine*, `https://cloud.google.com/products/compute-engine`, 2013.

[64] *Grid 5000 project*, `https://www.grid5000.fr/mediawiki/index.php`, 2013.

[65] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, « Load balancing in MapReduce based on scalable cardinality estimates », in *2012 IEEE 28th International Conference on Data Engineering*, Apr. 2012, pp. 522–533.

[66] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, « Handling data skew in MapReduce », in *Proceedings of the 1st International Conference on Cloud Computing and Services*, May 2011, pp. 574–583.

[67] M. Hammoud, M. S. Rehman, and M. F. Sakr, « Center-of-gravity reduce task scheduling to lower MapReduce network traffic », in *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, 2012, pp. 49–58.

[68] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, « RCFile: a fast and space-efficient data placement structure in MapReduce-based warehouse systems », in *2011 IEEE 27th International Conference on Data Engineering*, Apr. 2011, pp. 1199–1208.

[69] B. Hendrickson and T. G. Kolda, « Graph partitioning models for parallel computing », *Parallel Computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[70] B. Hendrickson and R. Leland, « A multilevel algorithm for partitioning graphs », in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, 1995.

[71] M. Herlihy and J. Wing, « Linearizability: a correctness condition for concurrent objects », *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[72] *Hp cloud services*, `https://www.hpcloud.com/`, 2013.

[73] P. Hunt, M. Konar, F. Junqueira, and B. Reed, « ZooKeeper: wait-free coordination for internet-scale systems », in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010, p. 11.

[74] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, « LEEN: locality/fairness-aware key partitioning for MapReduce in the cloud », in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, 2010, pp. 17–24.

[75] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum, « Kadeploy3: efficient and scalable operating system provisioning », *USENIX ;login:*, vol. 38, no. 1, pp. 38–44, Feb. 2013.

[76] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, « The performance of MapReduce: an in-depth study », *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 472–483, Sep. 2010.

[77] A. Jindal, J. A. Q. Ruiz, and J. Dittrich, « Trojan data layouts: right shoes for a running elephant », in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[78] T. Kaldewey, E. Shekita, and S. Tata, « Clydesdale: Structured Data Processing on MapReduce », in *Proceedings of the 15th International Conference on Extending Database Technology*, 2012, pp. 15–25.

[79] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, « H-store: a high-performance, distributed main memory transaction processing system », *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008.

[80] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, « Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web », in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 654–663.

[81] G. Karypis, *METIS : a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Version 5.1.0*, `http://glaros.dtc.umn.edu/gkhome/views/metis`, Department of Computer Science & Engineering, University of Minnesota, Mar. 2013.

[82] B. W. Kernighan and S. Lin, « An efficient heuristic procedure for partitioning graphs », *The Bell System Technical Journal*, vol. 49, no. 1, pp. 291–307, 1970.

[83] W. Kohler and K. Steiglitz, « Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems », *Journal of the ACM*, vol. 21, no. 1, pp. 140–156, Jan. 1974.

[84] M. Koyutürk and C. Aykanat, « Iterative-improvement-based declustering heuristics for multi-disk databases », *Information Systems*, vol. 30, pp. 47–70, Mar. 2005.

[85] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, « SkewTune: mitigating skew in MapReduce applications », in *Proceedings of the 2012 international conference on Management of Data*, 2012, pp. 25–36.

[86] A. Lakshman and P. Malik, « Cassandra: a decentralized structured storage system », *SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[87] C. A. Lee, « A perspective on scientific cloud computing », in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 451–459.

[88] Y. Li and M. Mascagni, « Improving performance via computational replication on a large-scale computational grid », in *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, 2003.

[89] J. Lin, « The curse of zipf and limits to parallelization: a look at the stragglers problem in MapReduce », in *Proceedings of the 7th Workshop on LargeScale Distributed Systems for Information Retrieval*, 2009.

[90] D. R. Liu and S. Shekhar, « Partitioning similarity graphs: a framework for declustering problems », *Information Systems*, vol. 21, no. 6, pp. 475–496, Sep. 1996.

[91] R. V. Nehme and N. Bruno, « Automated partitioning design in parallel database systems », in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 1137–1148.

[92] *Nimbus project*, `http://www.nimbusproject.org/`, 2013.

[93] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, « The Eucalyptus open-source cloud-computing system », in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009, pp. 124–131.

[94] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, « Pig latin: a not-so-foreign language for data processing », in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1099–1110.

[95] *Open Nebula – open source data center virtualization*, `http://opennebula.org/`, 2013.

[96] *Oracle loader for Hadoop*, `http://www.oracle.com/technetwork/bdc/hadoop-loader/overview/index.html`, 2013.

[97] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd. Springer, 2011.

[98] B. Palanisamy, A. Singh, L. Liu, and B. Jain, « Purlieus: locality-aware resource allocation for MapReduce in a cloud », in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[99] S. Papadomanolakis and A. Ailamaki, « AutoPart: automating schema design for large scientific databases using data partitioning », in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, 2004, pp. 383–392.

[100] A. Pavlo, C. Curino, and S. B. Zdonik, « Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems », in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 61–72.

[101] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, « A comparison of approaches to large-scale data analysis », in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 165–178.

[102] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, « Interpreting the data: parallel analysis with Sawzall », *Scientific Programming*, vol. 13, no. 4, pp. 277–298, Oct. 2005.

[103] *Pivotal Greenplum Database*, `http://gopivotal.com/pivotal-products/pivotal-data-fabric/pivotal-analytic-database`, 2013.

[104] S. Ramakrishnan, G. Swart, and A. Urmanov, « Balancing reducer skew in MapReduce workloads using progressive sampling », in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[105] K Ranganathan, A Iamnitchi, and I Foster, « Improving data availability through dynamic model-driven replication in large peer-to-peer communities », in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2002, p. 376.

[106] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman, « Automating physical database design in a parallel database », in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002, pp. 558–569.

[107] *Salesforce - CRM and cloud computing to grow your business*, `http://www.salesforce.com/`, 2013.

[108] *SAP Sybase IQ Columnar Database, Column-Based & Oriented DBMS*, `http://www.sybase.com/products/datawarehousing/sybaseiq`, 2013.

[109] L. F. G. Sarmenta, « Sabotage-tolerance mechanisms for volunteer computing systems », in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, pp. 337–346.

[110] P. Schwan, « Lustre: building a file system for 1,000-node clusters », in *Linux Symposium*, 2003, p. 380.

[111] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, « HPMR: prefetching and pre-shuffling in shared MapReduce computation environment », in *Proceedings of the 2009 IEEE International Conference on Cluster Computing*, 2009, pp. 1–8.

[112] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, « The Hadoop distributed file system », in *IEEE 26th Symposium on Mass Storage Systems and Technologies*, May 2010, pp. 1–10.

[113] D. Silva, W. Cirne, and F. Brasileiro, « Trading cycles for information: using replication to schedule bag-of-tasks applications on computational grids », in *Euro-Par 2003 Parallel Processing*, vol. 2790, 2003, pp. 169–180.

[114] *Sloan Digital Sky Survey*, `http://www.sdss3.org`, 2013.

[115] M. Stonebraker, D. Abadi, D. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, « MapReduce and parallel DBMSs: friends or foes? », *Communications of the ACM*, vol. 53, no. 1, pp. 64–71, Jan. 2010.

[116] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, « C-store: a column-oriented DBMS », in *Proceedings of the 31st international conference on Very Large Data Bases*, 2005, pp. 553–564.

[117] X. Su and G. Swart, « Oracle in-database Hadoop: when MapReduce meets RDBMS », in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data SE - SIGMOD '12*, 2012, pp. 779–790.

[118] *The Dark Energy Survey*, `http://www.darkenergysurvey.org/`, 2013.

[119] *The TPCBenchmark$^{TM}$H (TPC-H)*, `http://www.tpc.org/tpch/`, 2013.

[120] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, « Hive - a petabyte scale data warehouse using Hadoop », in *Proceedings of the 26th International Conference on Data Engineering*, Mar. 2010, pp. 996–1005.

[121] Ümit V. Çatalyürek and Cevdet Aykanat, *PaToH: partitioning tool for hypergraphs*, `http://bmi.osu.edu/~umit/software.html`, Mar. 2011.

[122] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley, « DB2 advisor: an optimizer smart enough to recommend its own indexes », in *Proceedings of the 16th International Conference on Data Engineering*, 2000.

[123] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac, « Adaptive MapReduce using situation-aware mappers », in *Proceedings of the 15th International Conference on Extending Database Technology*, 2012, pp. 420–431.

[124] *Vertica*, `http://www.vertica.com/`, 2013.

[125] W. Vogels, « Eventually consistent », *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[126] C. Walton, A. Dale, and R. Jenevein, « A taxonomy and performance model of data skew effects in parallel joins », in *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991, pp. 537–548.

[127] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, « A simulation approach to evaluating design decisions in MapReduce setups », in *17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2009, pp. 1–11.

[128] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, « Ceph: a scalable, high-performance distributed file system », in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[129] *Windows Azure: Microsoft's cloud platform*, `http://www.windowsazure.com/`, 2013.

[130] Y. Xu, P. Zou, W. Qu, Z. Li, K. Li, and X. Cui, « Sampling-based partitioning in MapReduce for skewed data », in *2012 Seventh ChinaGrid Annual Conference*, 2012, pp. 1–8.

[131] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, « Improving MapReduce performance in heterogeneous environments », in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 29–42.

[132] J. Zhou, N. Bruno, M. Wu, P. Larson, R. Chaiken, and D. Shakib, « SCOPE: Parallel Databases Meet MapReduce », *The VLDB Journal*, vol. 21, no. 5, pp. 611–636, Oct. 2012.

[133] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden, « DB2 design advisor: integrated automatic physical database design », in *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, 2004, pp. 1087–1097.