

N° d'ordre : 05/2014 – M / INF

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'enseignement supérieur et de la recherche Scientifique
Université des sciences et de la Technologie Houari Boumediène

Faculté de l'Electronique et d'Informatique



MEMOIRE

Présenté pour l'obtention du **diplôme** de **MAGISTER**

En : INFORMATIQUE

Spécialité : Informatique Mobile

Par : Rédha BENAÏSSA

Sujet

Opérations de bases de données avec MapReduce.

Soutenu publiquement, le 10/12/2014, devant le jury composé de :

Mme.	Aïcha AISSANI-MOKHTARI	Professeur	à l'USTHB	Présidente
M.	Walid-Khaled HIDOUCI	Professeur	à l'ESI	Directeur de mémoire
M.	Kadda BEGHDAÏ BEY	Maître de conférences/A	à l'EMP	Examineur

Résumé

Aujourd'hui, avec l'émergence des applications intensives en manipulation de données, les approches d'évaluation de jointures réparties basées sur le modèle de programmation MapReduce de la technologie Hadoop ont gagné en popularité et ont pu venir à bout d'un ensemble de problèmes que les bases de données relationnelles, toujours plus volumineuses pour accélérer leurs analyses et raccourcir les temps de réponse des requêtes utilisateurs. Néanmoins, les modèles proposés dans ces approches généralement n'évitent pas le problème des répartitions inégales obtenues dans la première phase du modèle *MapReduce* (le déséquilibre probable des valeurs de l'attribut de jointure). Donc *Que faut-il faire pour faire face au déséquilibre de données manipulées dans la réalisation des opérations de jointure?*

Pour répondre à cette question et après une étude des techniques proposées qui traitent la jointure avec *MapReduce*, le présent travail propose une approche pour équilibrer les charges de traitements de jointure des différentes machines. Cette approche exploite une technique d'agrégation simple et sa généralisation (agrégation multiple). Pour ce faire, nous nous sommes basés sur l'utilisation d'un mécanisme d'agrégation des valeurs de l'attribut de jointure et les autres valeurs des attributs de la table répliquée sur ces machines pour la redistribution des données.

Contrairement aux algorithmes présentés dans la littérature pour l'évaluation des requêtes de jointure répartie, notre approche évite de matérialiser le résultat intermédiaire de la jointure tout en traitant de manière efficace le problème de déséquilibre des valeurs de l'attribut de jointure. Ceci permet, d'une part, de réduire considérablement la taille des données envoyées à chaque machine, et d'autre part, de minimiser les coûts de communication et par conséquent le temps global de traitement lors de la phase de redistribution des données de jointure même en présence d'un fort déséquilibre des données à partir de 40%.

Mots-clés : *Jointure répartie, Jointure par agrégation, MapReduce, Big Data, NoSQL, Hadoop, HDFS.*

Remerciements

J'exprime mes grandes reconnaissances et mes vifs remerciements à mon encadreur le Professeur Walid-Khaled HIDOUCI pour le temps et l'attention qu'il a bien voulu consacrer au bon déroulement de ce projet.

J'exprime ma profonde gratitude au Dr. Farid BENHEMMADI, pour sa disponibilité, ses conseils, ses orientations et ses encouragements tout au long de ce travail.

Je remercie vivement les membres de jury pour avoir accepté de juger ce modeste travail.

Enfin, je remercie tous ceux qui de près ou de loin ont bien voulu m'encourager pour que ce travail puisse être achevé.

Table des matières

Introduction Générale	1
Problématique	2
Organisation du Mémoire	4
Chapitre 1 : Big Data.....	6
1.1. Introduction.....	6
1.2. Les limites des SGBD Relationnels.....	7
1.2.1. Limitation vis-à-vis des architectures distribuées	8
1.2.2. Limitation théorique	10
1.2.3. Limitation face aux usages	11
1.3. Big Data	11
1.3.1. Historique	13
1.3.2. Caractéristiques du BigData.....	13
1.3.3. Les Système de Gestion de Bases de Données NoSQL	14
1.3.4. Le stockage des données du BigData (modèle NoSQL).....	15
1.3.4.1. <i>Paradigme clé / valeur</i>	16
1.3.4.2. <i>Bases documentaires</i>	17
1.3.4.3. <i>Bases orientées colonnes</i>	18
1.3.4.4. <i>Paradigme graphe</i>	19
1.4. Conclusion	21
Chapitre 2 : La Plate-forme Hadoop.....	22
2.1. Introduction.....	22
2.2. Historique du projet Hadoop.....	23
2.3. Écosystème Hadoop.....	24
2.4. Fonctionnement de Hadoop	28

2.4.1.	HDFS : Hadoop Distributed File System.....	28
2.4.1.1.	<i>Architecture du HDFS</i>	30
2.4.1.2.	<i>Lecture et écriture d'un fichier dans HDFS</i>	32
2.4.1.3.	<i>Tolérance aux pannes</i>	34
2.4.2.	Le paradigme MapReduce	35
2.4.2.1.	<i>Fonctionnement du MapReduce</i>	36
2.4.2.2.	<i>Exemple de fonctionnement de MapReduce</i>	39
2.4.2.3.	<i>Tolérance aux pannes du MapReduce</i>	41
2.4.2.4.	<i>YARN</i>	41
2.5.	Conclusion	44
Chapitre 3 :	Algorithmes parallèles de jointure	47
3.1.	Introduction.....	47
3.2.	Opération de jointure.....	48
3.2.1.	La jointure par boucles imbriquées	49
3.2.2.	La jointure par tri-fusion.....	52
3.2.3.	La jointure par fonction de hachage	53
3.3.	Jointure Parallèle	56
3.3.1.	Jointure parallèle par boucles imbriquées	57
3.3.2.	Jointure parallèle tri-fusion	59
3.3.3.	Jointure parallèle par fonction de hachage.....	61
3.4.	Répartition des données et affectation des tâches.....	62
3.4.1.	Découpage des relations	62
3.4.2.	Affectation des tâches.....	63
3.5.	Synthèse	64
Chapitre 4 :	La jointure avec MapReduce	66
4.1.	Introduction.....	66
4.2.	La jointure répartie.....	68

4.2.1.	La jointure standard répartie.....	68
4.2.2.	La jointure répartie standard améliorée:	70
4.2.3.	Prétraitement pour la jointure répartie:.....	72
4.3.	Jointure par diffusion.....	74
4.3.1.	Principe de fonctionnement	74
4.3.2.	Prétraitement pour la jointure par diffusion:.....	76
4.4.	La semi-jointure.....	76
4.4.1.	Principe de fonctionnement	76
4.4.2.	Prétraitement pour la semi-jointure:.....	79
4.5.	Synthèse.....	80
Chapitre 5 :	Etude des performances de la jointure répartie	82
5.1.	Introduction.....	82
5.2.	Environnement d'évaluation	82
5.2.1.	Configuration matérielle	82
5.2.2.	Configuration logicielle.....	83
5.2.2.1.	<i>VirtualBox</i>	83
5.2.2.2.	<i>Citrix XenServer</i>	84
5.2.2.3.	<i>Citrix XenCenter</i>	<i>Error! Bookmark not defined.</i>
5.2.2.4.	<i>Installation de la plateforme Hadoop</i>	84
5.3.	Exemple d'exécution d'un job MapReduce.....	85
5.4.	Evaluation de la jointure Répartie.....	88
5.4.1.	Distribution homogène.....	90
5.4.2.	Distribution hétérogène.....	91
5.5.	Conclusion	95
Chapitre 6 :	La jointure par agrégation avec MapReduce	97
6.1.	Introduction.....	97

6.2. Principe de base.....	98
6.2.1. Jointure répartie par agrégation simple	98
6.2.2. Jointure répartie par agrégation multiple	106
6.3. Conclusion	112
Conclusion Générale.....	114
Bibliographie	117

Liste des figures

Figure 1.1 : Théorème de CAP [1]	10
Figure 1.2 : Accroissement du stockage de données entre 2009 et 2020 Digital Universe [2].....	12
Figure 1.3 : Paradigme clé / valeur.	16
Figure 1.4 : Bases documentaires.	17
Figure 1.5 : Bases orientées colonnes.....	18
Figure 1.6 : Famille de colonnes.....	19
Figure 1.7 : Bases orientées graphe.	20
Figure 2.1 Ecosystème Hadoop [15]	25
Figure 2.2 : Schéma de stockage des blocs dans HDFS [15]	29
Figure 2.3 : Le fonctionnement du Secondary NameNode [23]	32
Figure 2.4 : Lecture d'un fichier dans HDFS. [15]	33
Figure 2.5 : Écriture d'un fichier dans HDFS . [15]	33
Figure 2.6 : Processus MapReduce de Google [21].....	36
Figure 2.7 : Framework MapReduce et HDFS [21]	37
Figure 2.8 : Communication entre Jobtracker et les Tasktrackers [24].....	39
Figure 2.9 : Schématisation du paradigme Map-Reduce [25].....	39
Figure 2.10 : Flux de données Map-Reduce.....	40
Figure 2.11 : Architecture de YARN [24].....	42
Figure 3.1: Algorithme Boucle Imbriquée Basique.	49
Figure 3.2 : Algorithme Basique de boucles imbriquées par blocs.....	50
Figure 3.3 : Algorithme de boucle imbriquée indexée.....	51
Figure 3.4: Algorithme de la jointure par tri-fusion.....	52
Figure 3.5 : Jointure de base par hachage.....	54

Figure 3.6 : Jointure simple par hachage.	55
Figure 3.7 : Jointure parallèle par boucles imbriquées.....	57
Figure 3.8 : Jointure parallèle par boucles imbriquées améliorée.....	59
Figure 3.9 : Performances des trois(03) algorithmes [40].	65
Figure 4.1 : Les types de jointures	67
Figure 4.2 : les phases de la jointure standard répartie [23].....	69
Figure 4.3 : Pseudo-Algorithme de la jointure standard répartie [41].	69
Figure 4.4 : Les phases de la jointure standard répartie améliorée [23].	71
Figure 4.5 : Pseudo-Algorithme de la jointure standard répartie améliorée [41].	72
Figure 4.6 : Pseudo-Algorithme de la jointure directe [45].	73
Figure 4.7 : Les phases de la jointure par diffusion[23].	75
Figure 4.8 : Pseudo-Algorithme de la jointure par diffusion[41].	76
Figure 4.9 : Première phase de la semi-jointure[23].	77
Figure 4.10 : Seconde phase de la semi-jointure [23].	78
Figure 4.11 : Troisième phase de la semi-jointure [23].	78
Figure 4.12 : Pseudo-Algorithme de la semi-jointure [41].	79
Figure 4.13 : Arbre de décision relatif aux stratégies de jointure.	80
Figure 5.1 : Fichiers des entrées de la fonction MAP	86
Figure 5.2 : Les lignes de la fonction MAP en forme de paires clé-valeur	86
Figure 5.3 : Les Sorties de la fonction MAP	86
Figure 5.4 : Les entrées de la fonction Reducer.....	87
Figure 5.5 : La température maximale par année.....	87
Figure 5.6: Temps d'exécution en fonction des configurations du cluster	88
Figure 5.7 : Les informations relatives aux stations climatologiques.....	89
Figure 5.8 : Mesures effectuées par les stations climatologiques.....	89

Figure 5.9 : Distribution homogène des données en entrée.	90
Figure 5.10 : Distribution hétérogène avec une domination de 30%	91
Figure 5.11 : Distribution hétérogène avec une domination de 40%	92
Figure 5.12 : Distribution hétérogène avec une domination de 70%	93
Figure 5.13 : Distribution hétérogène avec une domination de 90%	94
Figure 6.1 : Exemple de deux tables.....	101
Figure 6.2 : Exemple d'agrégation de deux valeurs d'attributs L_1 et L_2	102
Figure 6.3 : Génération des nouvelles clés dans la fraction $F^{(R)}$	102
Figure 6.4 : Regroupement des Mappers suivant la nouvelle clé de jointure $H_{L_1}(L_2)$	103
Figure 6.5 : Le regroupement du <i>Mapper</i> suivant la clé de jointure originale L_1	104
Figure 6.6 : Le regroupement du Mapper suivant la nouvelle clé de jointure $H_{L_1}(L_3)$	105
Figure 6.7 : Regroupement du Mapper suivant la nouvelle clé $H_{L_1}(L_2 \oplus L_3)$	107
Figure 6.8 : Un exemple de la jointure standard répartie.....	110
Figure 6.9 : Jointure répartie par agrégation multiple de trois fragments.....	111

Introduction Générale

Le World Wide Web a connu une révolution avec l'avènement du Web 2.0. Les applications Web sont devenues plus interactives, permettant aux utilisateurs la liberté d'interagir et de collaborer sur Internet d'une manière nullement possible auparavant. Les utilisateurs ont commencé à prendre le rôle de créateurs de contenu plutôt que spectateurs passifs de pages Web. Les sites web ont été submergés de contenus générés par les utilisateurs à partir de blogs, de vidéos, de sites de médias sociaux et d'autres technologies Web 2.0. Par conséquent, la taille des informations stockées au niveau des serveurs a explosé d'une manière surprenante, jamais vue auparavant.

Chaque jour, de plus en plus d'entreprises basculent au numérique, de plus en plus d'informations commence à être numérisées et le volume des données publiées et stockées augmente. Ceci a conduit à l'âge de pétaoctets des informations stockées. Une technologie d'émergence, qui gère des quantités de données de très grandes tailles : le « Big Data ».

La gestion et le stockage de données constituent pour les entreprises les principaux obstacles et défis en matière de développement de capacités d'analyse détaillée des données. La performance d'un système de gestion de bases de données est portée essentiellement par le temps dédié au processus d'évaluation des requêtes, surtout quand

il s'agit de grandes quantités de données manipulées. L'utilisation de la puissance de traitement, et en particulier, en exploitant le traitement parallèle qui est la solution intuitive.

Hadoop, composé de son système de gestion de fichiers distribués HDFS (Hadoop Distributed File System) et d'un modèle de traitements parallèles en occurrence le paradigme MapReduce, est une plate-forme qui prend en charge le traitement de grands ensembles de données dans un environnement de calcul distribué. Conçue pour offrir une évolutivité optimale, cette solution permet aux entreprises de bénéficier d'un niveau de performances prévisible, à mesure que le volume des ensembles de données (datasets) à traiter s'accroît.

L'une des opérations la plus courante et la plus complexe dans l'évaluation des requêtes est la jointure. La jointure combine les enregistrements de deux ensembles de données ou plus, généralement sur la base de certaines conditions.

Problématique

Hadoop permet aux organisations de stocker et d'analyser de grand volumes de données, qu'elles soient structurées ou non, relationnelles ou non. Il vient compléter les plates-formes traditionnelles d'analyse des données, telles que les bases de données relationnelles. Hadoop améliore les performances de ces mêmes plates-formes en matière d'analyse SQL interactive des sources de données. Dès lors, *Quel est l'apport de Hadoop dans le cadre de l'analyse Big Data ?*

Hadoop associe le Framework MapReduce au système de fichiers distribués(HDFS). Le Framework MapReduce effectue le calcul distribué sur de grandes

quantités de données, avec une répartition de la charge sur un grand nombre de machines et avec un niveau d'abstraction quasi-totale de la couche matérielle. Les entreprises ont commencé à utiliser MapReduce pour gérer de grandes quantités de données et quand il y a plusieurs ensembles de données concernées, il y aura toujours un besoin de réaliser la jointure entre ces ensembles de données. De ce fait, la question majeure qui se pose est : *comment peut-on réaliser la jointure dans un environnement distribué? Comment réalise-t-on la jointure avec le Framework MapReduce ?*

La jointure répartie est l'approche la plus répandue pour effectuer la jointure avec MapReduce, *Que faut-il donc faire pour faire face à ses limites en termes d'équilibrage de données manipulées ?*

L'objectif de ce mémoire est de tenter d'apporter des réponses (solutions) aux différentes questions posées ci-dessus. Nous allons ainsi étudier et comparer les performances de quelques propositions faites pour effectuer des jointures dans un environnement MapReduce, afin de voir l'effet que pourrait avoir la présence d'un déséquilibre de données sur les performances en générale. Ensuite nous proposerons des améliorations, basées sur l'agrégation des valeurs de clés de jointure, capables de supporter un tel déséquilibre pouvant être produit par la non-uniformité dans la distribution des valeurs de l'attribut de jointure.

Organisation du Mémoire

Ce mémoire est organisé en six chapitres :

Le chapitre 1, *Big Data*, présente les limites des systèmes de gestion de bases de données relationnelles (section 1.2.). Dans la section (1.3.), nous passons en revue le « Big Data », les caractéristiques de ses données, les systèmes de gestion de bases de données(SGBD) NoSQL et les catégories des SGBD NoSQL.

Dans le chapitre 2, *La Plate-forme Hadoop*, nous détaillerons la plateforme Hadoop dans ses deux principaux volets : le système HDFS (section 2.4.1) et le paradigme MapReduce (section 2.4.2).

Le chapitre 3, *Algorithmes parallèles de jointure*, nous y détaillons les différentes approches de jointure et leur implantation, indépendamment de l'environnement d'exécution. Nous distinguons trois (03) principales approches algorithmiques permettant de réaliser la jointure : La jointure par boucles imbriquées (nested-loop) (section 3.2.1), La jointure par tri-fusion (sort-merge) (section 3.2.2) et la jointure par fonction de hachage (hash-join) (section 3.2.3). Ensuite, nous exposerons leur adaptabilité aux architectures parallèles (section 3.3.). Par la suite, nous montrerons l'impact de la répartition des données à joindre et la manière dont les tâches de traitement sont affectées (section 3.4.).

Dans le chapitre 4, *La jointure avec MapReduce*, nous allons examiner trois stratégies de jointure avec MapReduce qui soutiennent les deux types de jointures

(intérieure et extérieure). Les trois stratégies exécutent la jointure soit dans la phase MAP ou dans la phase REDUCE en tirant profit de l'avantage tri-fusion de l'architecture MapReduce. La jointure répartie est détaillée dans la section 4.2. , la jointure par diffusion dans la section 4.3. , et la semi-jointure dans la section 4.4. Dans la section 4.5. , nous donnons un arbre de décision, afin que nous puissions choisir la meilleure stratégie de jointure pour une situation donnée.

Le chapitre 5, *Etude des performances de la jointure répartie*, nous y présentons les résultats des expérimentations effectuées pour évaluer la jointure répartie avec MapReduce. Nous décrivons l'environnement d'évaluation et les conditions expérimentales (section 5.2.), où nous donnons la configuration matérielle et logicielle de l'environnement d'évaluation. Après, Pour tirer profit du traitement parallèle que fournit Hadoop, un job de MapReduce est exécuté sur un cluster de machines (section 5.3.), où nous calculons la température globale maximale enregistrée, par des stations climatologiques, chaque année. Dans la section 5.4. , nous nous intéressons au déséquilibre des valeurs de l'attribut de jointure de la jointure répartie, auquel nous effectuerons des tests sur deux types de distributions : homogène et hétérogène.

Nous terminons avec le chapitre 6, *La jointure par agrégation avec MapReduce*, dans lequel nous proposons une approche pour équilibrer les charges des différents Reducers. Nous nous sommes basés sur l'utilisation d'un mécanisme d'agrégation des valeurs de l'attribut de jointure (clé originale) et les autres valeurs des attributs de la table de petit volume de données pour la redistribution des données. Nous présentons deux types d'agrégation : simple (section 6.2.1) et multiple (section 6.2.2).

Chapitre 1 : Big Data

1.1. Introduction

Les réseaux d'Internet fournissent une infrastructure pour le stockage des données et facilitent le partage d'un très grand volume de données via divers moyens de communication. Ce dernier (le partage) est une des principales clés de succès des applications Web.

Aujourd'hui, beaucoup d'applications Web voient leurs utilisateurs croître fortement pour atteindre des centaines de millions de personnes (en 2012, Facebook atteint plus de 845 millions de membres actifs [3]). Face à cette forte demande, les applications Web sont confrontées à un problème de passage à l'échelle. En particulier, les systèmes de gestion de données atteignent rapidement les limites de la charge qu'ils peuvent traiter. Cela peut provoquer une dégradation des temps de réponses perçus par l'utilisateur, situation qu'il n'accepte pas. Pour éviter cela, il est nécessaire d'augmenter les ressources de calcul, de stockage et de communication utilisées par les applications exploitées.

L'ajout de ressources est fait par étapes et progressivement. L'infrastructure obtenue est composée de ressources hétérogènes (diverses capacités de stockage, de calcul et de communication) réparties à l'échelle mondiale. Du fait de sa taille et son étendue, l'infrastructure est sujette à de nombreuses pannes dont il faut tenir compte en prévoyant la redondance des données.

Dans ce chapitre, nous allons présenter les limites des systèmes de gestion de bases de données relationnels. Par la suite, nous nous intéresserons à une technologie d'émergence, qui gère des quantités de données hétérogènes et de très grandes tailles, à savoir le « Big Data ».

Le présent chapitre est organisé comme suit :

- La section (1.2.) présente les limites des systèmes de gestion de bases de données relationnels;

- La section (1.3.) décrit le « Big Data », les caractéristiques de ses données, les systèmes de gestion de bases de données(SGBD) NoSQL, et les catégories des SGBD NoSQL;

1.2. Les limites des SGBD Relationnels

Depuis plusieurs années, les technologies de bases de données relationnelles et transactionnelles [4], qualifiées parfois “technologies SQL”, règnent en maîtres pour le stockage et la manipulation des données. Cette situation de leadership technologique peut facilement être expliquée par plusieurs facteurs :

Tout d’abord, SQL est un langage standardisé. Bien que chaque fournisseur de base de données ait implémenté quelques variantes, il n’en demeure pas moins que cette plate-forme commune est très utilisée par les développeurs. Pour une entreprise, investir dans une technologie SQL s’avère donc moins coûteux sur le plan de la formation que toute autre technologie. Cela explique sans doute l’échec des bases de données objets que l’on a vu fleurir durant les années 2000 et qui n’ont jamais véritablement percé, malgré une approche novatrice.

Par ailleurs, SQL embarque de nombreuses fonctionnalités importantes, adaptées aux besoins des entreprises, en particulier la gestion de l’intégrité des données et l’implémentation de transactions, indispensables pour bon nombre d’applications de gestion. Dans un monde où l’informatique de gestion représente une part très importante des besoins durant ces dernières décennies, ces fonctionnalités sont attendues et indispensables dans une base de données.

Enfin, les outils permettant l’exploitation de ce type de bases de données sont matures. Que l’on se place au niveau des outils de développements (IDE graphiques, intégration dans les langages et les frameworks, ...) ou d’exploitation (outils de sauvegarde, monitoring, ...), les solutions existent et sont prêtes depuis déjà un certain temps. La technologie SQL est considérée comme un standard pour toute problématique ayant trait au stockage et à la manipulation de données. Malgré cela, un certain nombre de limitations importantes sont apparues au fil des années. Les premiers acteurs à buter sur ces limites furent les fournisseurs de services en ligne les plus populaires, tels que Yahoo, Google et les acteurs du web social comme Facebook, Twitter ou LinkedIn. Le constat était simple [5] : *les SGBD relationnels ne sont pas adaptés aux environnements*

distribués requis par les volumes gigantesques de données et par les trafics tout aussi gigantesques générés par ces opérateurs.

Les besoins majeurs identifiés par ces acteurs sont les suivants [5]:

- Scaling des traitements : c'est la capacité à distribuer les traitements sur un nombre de machines important afin d'être en mesure d'absorber des charges très importantes.
- Scaling des données : c'est la capacité à répartir les données entre un nombre important de machines afin d'être en mesure de stocker de très grands volumes de données.
- La distribution de données sur plusieurs datacenters afin d'assurer une continuité de service en cas d'indisponibilité de service sur un datacenter. Cette approche doit par ailleurs permettre aux données d'être plus proche des personnes qui les utilisent.
- Une architecture qui fonctionne sur du matériel peu spécialisé et donc facilement remplaçable en cas de panne.

1.2.1. Limitation vis-à-vis des architectures distribuées

Les bases de données relationnelles mettent à la disposition des développeurs un certain nombre d'opérations de relations entre les tables :

- Un système de jointure entre les tables permettant de construire des requêtes complexes faisant intervenir plusieurs entités (les tables en l'occurrence).
- Un système d'intégrité référentielle permettant de s'assurer que les liens entre les entités sont valides.

Dans le contexte d'un système distribué, la mise en œuvre de tels mécanismes a un coût considérable. Sur la plupart des SGBD relationnels, il convient de s'assurer en permanence que les données liées entre elles sont placées sur le même nœud du serveur. Lorsque le nombre de

relations au sein d'une base augmente, il devient de plus en plus difficile de placer les données sur des nœuds différents du système.

En outre, la plupart des moteurs de SGBDs relationnels sont transactionnels ce qui leur impose le respect des contraintes Atomicity Consistency Isolation Durability, communément appelé par son acronyme ACID [6]:

- Atomicity (A) : signifie que les mises à jour de la base de données doivent être atomiques, c'est à dire qu'elles doivent être totalement réalisées ou pas du tout. Par exemple, sur 5000 lignes devant être modifiées au sein d'une même transaction, si la modification d'une seule échoue, alors la transaction entière doit être annulée. C'est primordial, car chaque ligne modifiée peut dépendre du contexte de modification d'une autre, et toute rupture de ce contexte pourrait engendrer une incohérence des données de la base.
- Consistency (C): signifie que les modifications apportées à la base doivent être valides, en accord avec l'ensemble de la base et de ses contraintes d'intégrité. Si un changement viole l'intégrité des données, alors soit le système doit modifier les données dépendantes, comme dans le cas classique d'une suppression en cascade, soit la transaction doit être interdite.
- Isolation (I): signifie que les transactions lancées au même moment ne doivent jamais interférer entre elles, ni même agir selon le fonctionnement de chacune. Par exemple, si une requête est lancée alors qu'une transaction est en cours, le résultat de celle-ci ne peut montrer que l'état original ou final d'une donnée, mais pas l'état intermédiaire. De fait, les transactions doivent s'enchaîner les unes à la suite des autres, et non de manière concurrentielle.
- Durability (D): signifie que toutes les transactions sont lancées de manière définitive. Une base ne doit pas afficher le succès d'une transaction, pour ensuite remettre les données modifiées dans leur état initial. Pour ce faire, toute transaction est sauvegardée dans un fichier journal, de sorte que si un problème survient empêchant sa validation complète, la transaction pourra être correctement terminée lors de la disponibilité du système.

Dans un contexte centralisé, les contraintes ACID sont plutôt aisées à garantir. Dans le cas de systèmes distribués, il est en revanche nécessaire de distribuer les traitements de données entre différents serveurs. Il devient alors difficile de maintenir les contraintes ACID à l'échelle du système distribué entier tout en maintenant des performances correctes.

1.2.2. Limitation théorique

Il existe par ailleurs, une limitation plus théorique à ce que peuvent réaliser les bases de données relationnelles distribuées qui est décrite par le théorème de CAP [1] (Consistency-Availability-Partition tolerance). Ce dernier énonce trois grandes propriétés pour les systèmes distribués :

- Coherence (C) ou Cohérence : tous les nœuds du système voient exactement les mêmes données au même moment.
- Availability (A) ou Disponibilité : la perte de nœuds n'empêche pas les survivants de continuer à fonctionner correctement.
- Partition tolerance (P) ou Résistance au partitionnement : aucune panne moins importante qu'une coupure totale du réseau ne doit empêcher le système de répondre correctement (ou encore : en cas de morcellement en sous-réseaux, chacun doit pouvoir fonctionner de manière autonome).

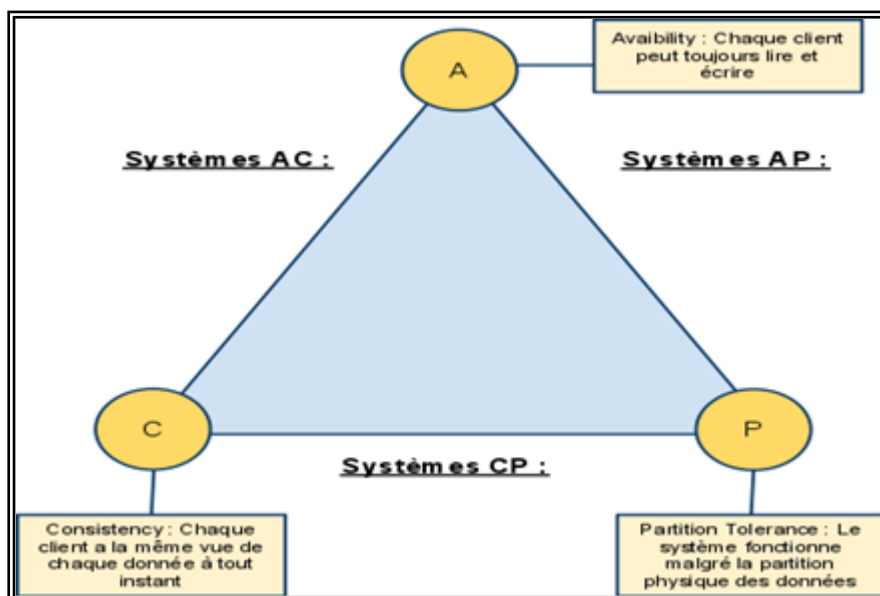


Figure 1.1 : Théorème de CAP [1]

Il est impossible d'obtenir ces trois propriétés du théorème de CAP en même temps dans un système distribué. Par conséquent, il faut donc en choisir deux parmi les trois. Les bases de données relationnelles implémentent les propriétés de Cohérence et de Disponibilité (système AC). Les bases de données NoSQL sont généralement des systèmes CP (Cohérent et Résistant au partitionnement) ou AP (Disponible et Résistant au partitionnement).

1.2.3. Limitation face aux usages

En outre du stockage distribué qui pèse à ce jour sur les systèmes relationnels, les structures de données manipulées par les systèmes sont devenues de plus en plus complexes avec en contrepartie des moteurs de stockage évoluant peu.

Le principal point faible des modèles relationnels est l'absence de gestion d'objets hétérogènes ainsi que l'obligation de déclarer au préalable l'ensemble des champs représentant un objet. Pour répondre au besoin des objets hétérogènes non pré-déclarés, le modèle Entity-Attribute-Value vu apparaître permettant ainsi la séparation de l'objet et de ses champs.

La majeure partie des technologies développées dans le cadre de la mouvance NoSQL permettent l'utilisation d'objets hétérogènes et de très grandes tailles, à savoir le *Big Data*, apportant comparativement une plus grande flexibilité dans les modèles de données ainsi qu'une simplification de la modélisation.

Dans la section suivante, nous détaillons le Big Data.

1.3. Big Data

Le Big Data est le phénomène d'émergence de quantités de données hétérogènes et de très grandes tailles. Ce phénomène est né du contexte actuel caractérisé par la généralisation de l'utilisation d'internet, des réseaux sociaux et d'appareils générant des données.

Les données créées, répliquées et consommées chaque année dans le monde devraient passer de 898 exabyte(10^{18}) à 6,6 zettabytes(10^{21}) entre 2012 et 2020, soit plus de 25% par an, ce qui signifie qu'ils doublent environ tous les trois ans [2]. La Figure 1.2 montre la croissance exponentielle des données.

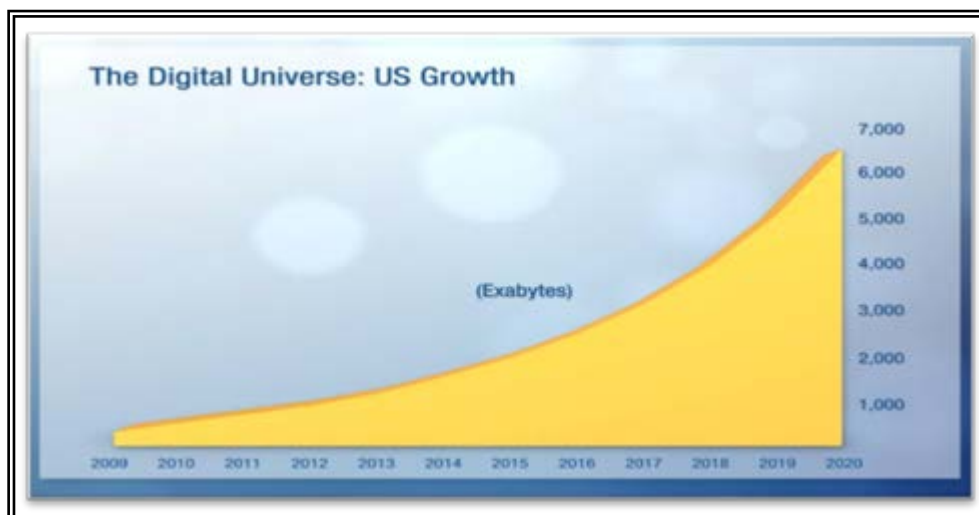


Figure 1.2 : Accroissement du stockage de données entre 2009 et 2020

Digital Universe [2]

Littéralement Big Data est défini comme : « Grosses données » ou « grande quantité de données » ou « données massives ». C'est une expression anglophone utilisée pour désigner des ensembles de données qui deviennent tellement volumineux qu'ils en deviennent difficiles à traiter avec des outils classiques de gestion des grands entrepôts de données [7].

Ce phénomène est considéré comme l'un des grands défis informatiques de la décennie 2010-2020 [8]. Il engendre une dynamique importante de la part des spécialistes sur le terrain des technologies informatiques.

Nous pouvons citer, à titre illustratif, quelques exemples de sources de données volumineuses [9]:

- Du Web : journaux d'accès, réseaux sociaux, e-commerce, indexation, stockage de documents, de photos, de vidéos ;
- De l'internet et des objets communicants : réseaux de capteurs, journaux des appels en téléphonie ;
- Des sciences : génomique, astronomie, physique subatomique, climatologie,
- De données commerciales (ex : historique des transactions dans une chaîne d'hypermarchés) ;
- De données personnelles (ex : dossiers médicaux) ;
- De données publiques (open data).

1.3.1. Historique

Le BigData n'est pas un phénomène nouveau. Cette discipline a vu le jour avec l'émergence de données trop volumineuses.

À ses débuts, le BigData était utilisé par les entreprises, telles que les banques (ex. Visa et Bank of America) en ce qui concerne les transactions par cartes de crédit. De même que les compagnies de téléphonie (ex. AT&T) qui l'utilisaient pour les registres d'appels téléphoniques. En outre, le BigData était utilisé par des sites de commerce électronique (ex. Amazon et eBay) pour améliorer le service en ligne [7].

1.3.2. Caractéristiques du BigData

Les caractéristiques des données du BigData peuvent être résumées dans les axes 3V (Volume, Vitesse, Variété) [8] [9].

➤ Volume

Le développement d'Internet conjugué à la production excessive de données par les utilisateurs, ont eu pour effet l'accélération de l'évolution des volumes de données disponibles. Donc une augmentation considérable des tailles de données à traiter. Si le volume est un critère majeur du BigData en revanche, son évaluation est délicate et évolutive.

Selon Eric Schmidt, PDG de Google de 2001 jusqu'à 2011, «il existe 5 millions de téraoctets d'informations dans le monde et nous n'en avons indexé que 170 téraoctets (c'est-à-dire beaucoup moins de 1%) ». De plus, il a estimé qu'il faudrait 300 ans pour indexer toutes les informations connues sur Terre.

➤ Vitesse

Aujourd'hui, les applications sur Internet tels que les réseaux sociaux, produisent des données en permanence (flux continu) qui sont difficilement maîtrisables. De tels volumes imposent un traitement instantané de l'information pour au moins sélectionner celle qui devra être analysée et écarter le reste. Et si l'activité impose une réaction en temps réel, il devient nécessaire de construire également des systèmes d'analyse quasiment en temps réel. Le BigData consiste donc

à analyser de gros volumes de données dans des délais très courts qui se rapprochent du temps réel.

➤ Variété

La variété représente la gamme de données structurées, semi structurées et non structurées. Le traitement de gros volumes de données structurées n'est ni récent, ni compliqué. En effet, les entrepôts de données des banques, des assureurs, des distributeurs et des compagnies de télécommunication sont gérés par ces derniers depuis plusieurs décennies. Le nouveau problème est de pouvoir analyser un grand volume de données semi-structurées et/ou non structurées, comme les données sociales (les photos, vidéos...) ou d'une autre provenance comme les sciences de la vie et de la terre ou la santé...

1.3.3. Les Système de Gestion de Bases de Données NoSQL

Aujourd'hui les sociétés doivent faire face à une très forte augmentation de la quantité de données qu'elles doivent stocker et traiter. Les systèmes de gestion bases de données relationnelles (SGBDR) sont la solution prédominante pour le stockage et le traitement de données, mais l'expérience des sociétés telles que Google et Facebook a montré que le modèle des SGBDR a atteint ses limites face à d'importantes quantités de données et ne sait répondre à des coûts exorbitants. Le mouvement NoSQL est devenu très populaire en 2009 et le terme NoSQL ne se réfère pas à une technologie mais à une appellation nébuleuse dans laquelle se retrouvent des bases de données et systèmes de stockage ; qui ont pour point commun d'avoir abandonné certaines fonctionnalités et caractéristiques fournies par les SGBDR et ce, pour être en mesure de répondre à la problématique de très gros volumes de stockage et de traitements. Ils visent à maintenir une augmentation linéaire des coûts par rapport aux volumes. Ces bases acceptent donc de faire un compromis notamment sur la cohérence afin de pouvoir gagner sur un des deux axes que sont la disponibilité et la tolérance au partitionnement, ce sont les trois axes du théorème de CAP (Consistency Availability Partition tolerance) [1].

Définition du NoSQL

NoSQL n'est pas une technologie mais plutôt une catégorie de produits et de technologies axés sur une nouvelle façon de stocker les données qui utilisent parfois des concepts communs sur le stockage et la manipulation des données. Il n'y a pas de caractéristiques ou critères précis qui permettent de classer un produit ou une technologie sous cette appellation. De plus, en parallèle, nous évoquons aussi l'appellation « Big Data » dans laquelle on retrouve la problématique du stockage et de l'accès à de large volume de données. NoSql peut être interprété comme le refus du langage SQL mais au lieu de NoSQL, il serait plus juste d'utiliser l'appellation NoRDBMS (Not Only Relational DataBase Management System). En effet, ce ne sont pas les langages SQL qui sont rejetés mais plutôt les systèmes de gestion de base de données classiques SGBDR/RDBMS (Relational DataBase Management System) basés sur le modèle ACID (Atomicité Cohérence Isolation Durabilité) et plus particulièrement le côté «Relationnel » qui se traduit le plus souvent par la présence d'un langage SQL. Aussi pour mettre en évidence cette distinction, aujourd'hui on retrouve souvent NoSQL défini plus justement comme étant l'acronyme de « Not Only SQL » [10].

1.3.4. Le stockage des données du BigData (modèle NoSQL)

Les besoins de stockage et de traitement des quantités importantes de données ont poussé beaucoup d'organisations à chercher de nouvelles techniques de stockage. En effet, la plupart des SGBD (Système de Gestion de Bases de Données) relationnels ne permettent plus de répondre aux besoins de stockage et de traitement de ces grandes quantités.

C'est dans ce contexte que nous avons vu apparaître de nouveaux SGBD qui sont regroupés sous l'appellation : les SGBD NoSQL (Not Only SQL). Ils représentent l'ensemble des technologies qui se distinguent par un relâchement des caractéristiques ACID (Atomicité, Consistance, Intégrité et Disponibilité) propres aux SGBDR (Système de Gestion de Bases de Données Relationnelles).

La famille des SGBD NoSQL se compose de plusieurs catégories [11] [12] [13] : orientés colonne, orientés document, orientés graphe et Clé/valeur.

1.3.4.1. Paradigme clé / valeur

Il s'agit de la catégorie de base de données la plus basique. Dans ce modèle chaque objet est identifié par une clé unique qui constitue la seule manière de le requêter. La structure de l'objet est libre et le plus souvent laissée à la charge du développeur de l'application (XML, JSON, ...), la base ne gérant généralement que des chaînes d'octets [13].

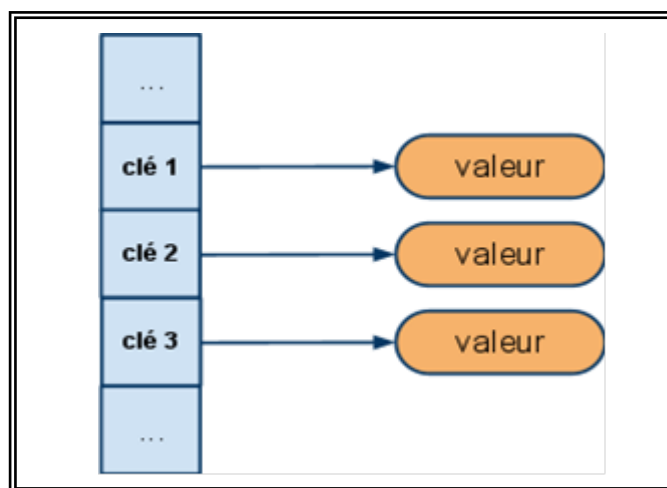


Figure 1.3 : Paradigme clé / valeur.

Dans ce modèle on ne dispose généralement que des quatre opérations Create Read Update Delete (CRUD):

- Create : créer un nouvel objet avec sa clé → `create(key, value)`
- Read : lit un objet à partir de sa clé → `read(key)`
- Update : met à jour la valeur d'un objet à partir de sa clé → `update(key, value)`
- Delete: supprime un objet à partir de sa clé → `delete(key)`

Les bases de ce type disposent pour la plupart d'une interface HTTP REST permettant de procéder très simplement à des requêtes, ceci depuis n'importe quel langage de développement.

L'approche volontairement très limitée de ces systèmes sur le plan fonctionnel est radicale et leur permet d'afficher des performances exceptionnellement élevées en lecture et en écriture ainsi qu'une scalabilité horizontale considérable. Le besoin de scalabilité verticale est fortement réduit au niveau des bases par le caractère très simple des opérations effectuées.

Du fait de leur modèle de requête très simple et réduit aux opérations CRUD, ces systèmes sont principalement utilisés comme dépôt de données dès lors que les besoins en termes de requêtes sont très simples. On les retrouve très souvent comme système de stockage de cache ou de sessions distribuées, notamment là où l'intégrité relationnelle des données est non significative.

1.3.4.2. Bases documentaires

Les bases de données documentaires sont constituées de collections de documents [13]. Un document est composé de champs et de valeurs associées, ces dernières pouvant être requêtées.

Par ailleurs, les valeurs peuvent être, soit d'un type simple (entier, chaîne de caractère, date, ...), soit elles-mêmes composées de plusieurs couples clé/valeur.

Bien que les documents soient structurés, ces bases sont dites "schemaless". A ce titre il n'est pas nécessaire de définir au préalable les champs utilisés dans un document. Les documents peuvent être très hétérogènes au sein de la base.

Le stockage structuré des documents leur confère des fonctionnalités dont ne disposent pas les bases clés/valeurs simples dont la plus évidente est la capacité d'effectuer des requêtes sur le contenu des objets.

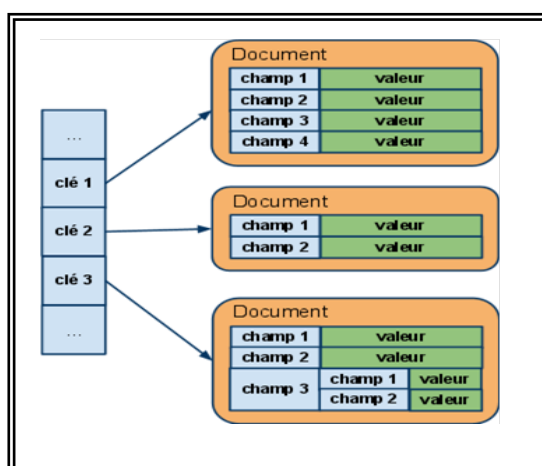


Figure 1.4 : Bases documentaires.

Nous conservons généralement une interface d'accès HTTP REST permettant d'effectuer simplement des requêtes sur la base mais celle-ci est plus complexe que l'interface CRUD des bases clés/valeurs. Les formats de données JSON et XML sont le plus souvent utilisés afin d'assurer le transfert des données sérialisées entre l'application et la base.

Du fait de leur conception plus simple que les bases de données relationnelles, ces bases conservent des performances élevées. Elles disposent par ailleurs de fonctionnalités très intéressantes en terme de flexibilité du modèle documentaire, ce que les bases relationnelles ne peuvent offrir. Cette dernière fonctionnalité leur confère une popularité importante, notamment dans le milieu du développement de CMS (Content Management System, outils de gestion de contenus) pour lesquels ce modèle est très adapté.

1.3.4.3. Bases orientées colonnes

Les bases orientées colonnes sont résolument les plus complexes à appréhender parmi la mouvance NoSQL. Bien que l'on obtienne au final un schéma relativement proche des bases documentaires, l'organisation sous-jacente des données permet à ces bases d'exceller dans les traitements d'analyse de données et dans les traitements massifs (notamment via des opérations de type Map-Reduce).

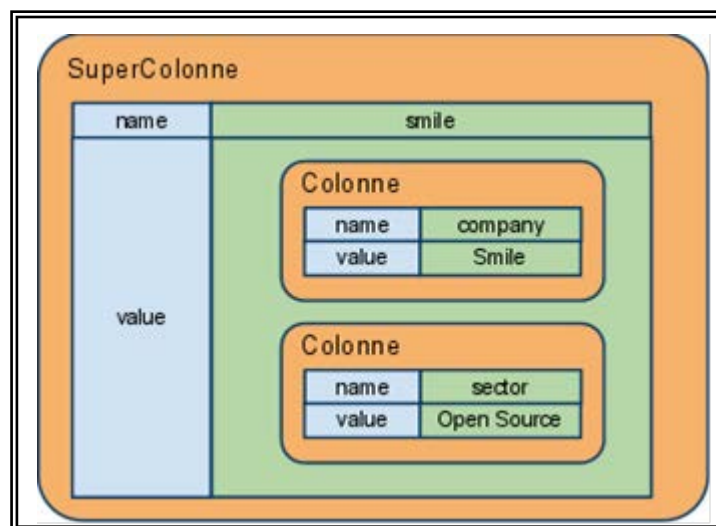


Figure 1.5 : Bases orientées colonnes.

Bien que l'on trouve des variations en fonction de la base considérée, les concepts essentiels sont les suivants [11] :

- ## Famille de colonnes : companies

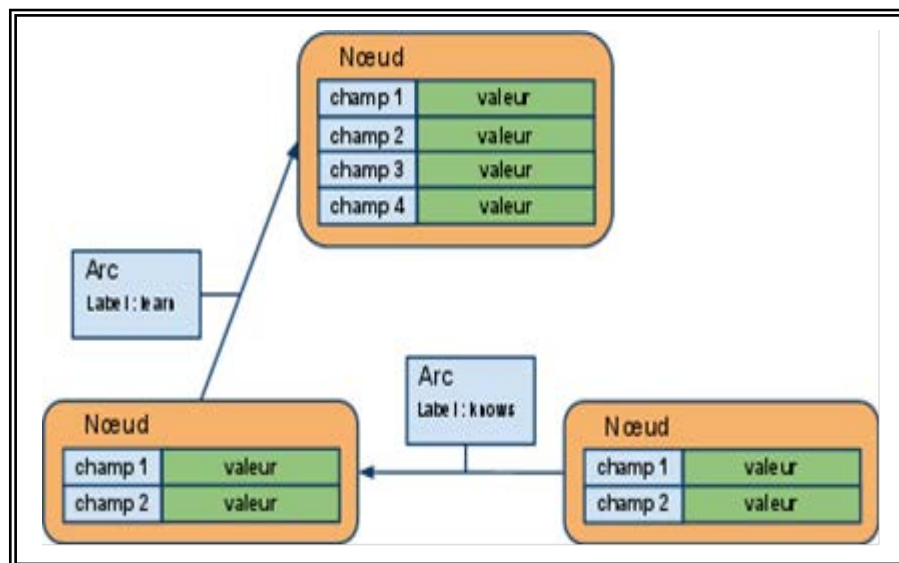


Figure 1.7 : Bases orientées graphe.

Par essence ces bases de données sont nettement plus efficaces que leur pendant relationnel pour traiter les problématiques liées aux réseaux (cartographie, relations entre personnes, ...). En effet, lorsqu'on utilise le modèle relationnel, cela nécessite un grand nombre d'opérations complexes pour obtenir des résultats.

Nous listons une douzaine de solutions existantes, qu'elles soient de type :

- base clé/valeur : comme Redis, Riak, Voldemort.
- base documentaire : comme MongoDB, CouchDB, Terrastore.
- base orientée colonne : comme Cassandra, Amazon SimpleDB, Google BigTable, HBase.
- base orientée graphe : comme Neo4j, OrientDB.

Chacune des catégories des SGBD NoSQL présente des caractéristiques différentes en termes de scalabilité et de complexité des données supportées.

Le modèle clé/valeur passe à l'échelle rapidement, mais il est déconseillé pour les données complexes en raison des difficultés de modélisation adéquate de ces données. Par contre, les SGBD orientés graphe ne passent pas à l'échelle rapidement à cause de la grande complexité du traitement à effectuer. Néanmoins, ils sont indispensables dans certains cas où on a une structure de données complexe.

1.4. Conclusion

Le phénomène du Big Data s'est façonné en 2011, mais a connu une croissance fulgurante en 2012, modifiant partiellement, mais d'une façon drastique, tous les aspects de la gestion des données. Les systèmes de Big Data ont ainsi entraîné des changements dans les méthodes de gestion et de traitements des données, dans les opérations d'extraction (ETL). L'apparition de ces nouveaux paradigmes a entraîné des changements majeurs dans la manière d'aborder la conception des applications et leur rapport à la base.

Pourtant, à l'aube de 2013, aucune des technologies Big Data ne semblent être plus actives que les bases de données NoSQL et le Framework Hadoop. Et, surtout, leur marge de progression semble aujourd'hui presque sans limites.

Ainsi, les technologies NoSQL et Hadoop semblent être les deux moyens clés pour prendre en charge les données non-structurées. Comme le montre Apache Hadoop, ces technologies ont souvent des racines liées à l'Open Source, et sont entrées très récemment dans le monde commercial.

Dans le chapitre suivant, nous détaillons une technique de programmation distribuée très utilisée dans le milieu NoSQL, en l'occurrence MapReduce du Framework Hadoop.

Chapitre 2 : La Plate-forme Hadoop

2.1. Introduction

Pour que les entreprises tirent pleinement profit de leurs informations, la gestion des données volumineuses, la gestion de la croissance des données, l'approvisionnement du stockage et la réalisation d'analyses « Big data » rapides et fiables constituent, des priorités.

La plate-forme émergente Hadoop est l'une des solutions récemment adoptée par les entreprises afin de répondre à leurs besoins d'analyse Big Data. Conçue pour offrir une évolutivité optimale, cette solution permet aux entreprises de bénéficier d'un niveau de performances prévisibles, à mesure que le volume des données à traiter/analyser s'accroît.

Hadoop est un Framework de programmation libre, basé sur Java qui prend en charge le traitement de grands ensembles de données dans un environnement de calcul distribué. Il fait partie du projet Apache parrainé par l'Apache Software Foundation¹.

Hadoop permet d'exécuter des applications sur les systèmes avec des milliers de nœuds impliquant des milliers de Téraoctets. Son système de fichiers distribué facilite le transfert de données rapide entre les nœuds et permet au système de continuer à fonctionner sans interruption en cas de défaillance d'un nœud. Cette approche réduit le risque de défaillance catastrophique du système, même si le nombre important de nœuds devient inopérant.

¹ Apache Software Foundation est une organisation à but non lucratif qui développe des logiciels open source sous la licence d'Apache.

Le Framework Hadoop créé par Doug Cutting en 2009 (projets de la fondation Apache) est utilisé par les principaux acteurs tels que Google, Amazon, Yahoo, Facebook, eBay et IBM, en grande partie pour des applications impliquant des moteurs de recherche et la publicité.

Dans ce chapitre, nous détaillerons la plateforme Hadoop dans ses deux principaux volets : le système HDFS et le paradigme MapReduce

2.2. Historique du projet Hadoop

Hadoop a ses origines dans Apache Nutch² [14] qui était un grand projet lancé en 2002. Il a rencontré des problèmes avec la création d'une architecture qui prend en charge des milliards de pages Web sur Internet [15].

Mike Cafarella et Doug Cutting qui ont participé dans le projet Nutch et créé HDFS ont estimé qu'un système d'indexation de milliards de pages coûterait environ un demi-million de dollars, comprenant le matériel couplé d'un coût de fonctionnement mensuel de 30.000 \$ US.

En 2004, Google a publié un document qui a présenté au monde le paradigme MapReduce et la structure de son système de fichier GFS (Google File System). L'idée a été reprise dans le projet de Nutch qui a développé le NDFS (Nutch Distributed File System).

Hadoop est né en février 2006. NDFS et Nutch sont devenus des sous-projets séparés sous Lucene³.

² Nutch est une initiative visant à construire un moteur de recherche open source. Il utilise Lucene comme bibliothèque de moteur de recherche et d'indexation.

³ Lucene est un moteur de recherche libre écrit en Java qui permet d'indexer et de rechercher du texte. C'est un projet open source de la fondation Apache.

En janvier 2008, Hadoop a réalisé son propre projet de niveau supérieur sous Apache. NDFS a été remplacé par HDFS (Hadoop Distributed File System). Voici une chronologie rapide qui illustre le progrès d'Hadoop [15] :

- En 2004 : création des versions initiales de ce qui est maintenant appelé système de fichiers distribué d'Hadoop « HDFS » et MapReduce mis en application par Doug Cutting et Mike Cafarella.
- Décembre 2005 - Hadoop fonctionne sûrement sur 20 nœuds.
- Janvier 2006 - Doug rejoint Yahoo!
- Février 2006 - le projet d'Apache Hadoop a officiellement commencé à soutenir le développement autonome de MapReduce et de HDFS.
- Février 2006 - adoption d'Hadoop par Yahoo!
- Avril 2006 - tri d'un benchmark (10 GB/ nœud) sur 188 nœuds en 47.9 heures.
- Mai 2006 - Yahoo! Mise en place d'un cluster de recherche de 300 nœuds Hadoop.
- Mai 2006 - tri d'un benchmark sur 500 nœuds en 42 heures
- Octobre 2006 - le cluster de recherche atteint 600 nœuds.
- Janvier 2007 - le cluster de recherche atteint 900 nœuds.
- Avril 2007 - les clusters de recherche atteignent 2 clusters de 1000 nœuds.
- Octobre 2008 - les clusters de recherche chargent 10 Téra-octet de données par jour.
- Avril 2009 - un record en triant 500 gigaoctet en 59 secondes (sur 1.400 nœuds) et 100 Téra-octet en 173 minutes (sur 3.400 nœuds).

2.3. Écosystème Hadoop

Hadoop contient de nombreux sous-projets qui relèvent de la même catégorie de l'informatique distribuée et du traitement de données à grande échelle. L'écosystème Hadoop est composé des principaux composants représentés dans la Figure 2.1 mais il

n'est pas limité à cette liste, l'écosystème est construit à une vitesse très rapide et de nouveaux projets lui sont ajoutés [16] .

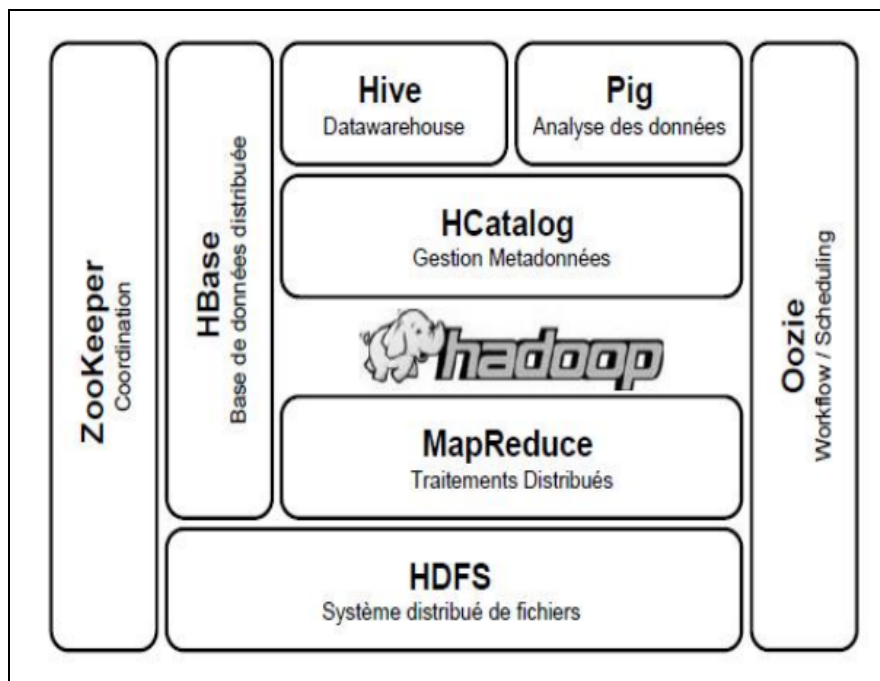


Figure 2.1 Ecosystème Hadoop [15]

- **HDFS** : est le système de fichiers distribués de l'écosystème Hadoop, ce système est le mécanisme par lequel une grande quantité de données peut être répartie sur un groupe d'ordinateurs. Les données sont écrites une seule fois, mais lues plusieurs fois pour analyses. Il fournit une base pour d'autres outils, tels que HBase [16].
- **Mapreduce** (traitement parallèle) : est un élément fondamental et principal pour l'exécution de Hadoop. Il est un modèle de programmation pour le traitement de données distribuées en parallèle. Pour son fonctionnement il sépare les jobs en phases de Mapping et des phases de Reduce (d'où son nom). Les développeurs écrivent les jobs MapReduce pour Hadoop, en utilisant les données stockées dans HDFS pour un accès rapide aux données. En raison de la nature du fonctionnement de MapReduce,

Hadoop apporte le traitement des données de façon parallèle, ce qui entraîne sa rapidité de mise en œuvre [16] .

- **Apache Zookeeper** : est un service de coordination distribuée de Hadoop. ZooKeeper est conçu pour fonctionner sur un cluster de machines, il fournit des verrous distribués qui peuvent être utilisés pour construire des applications distribuées. Il est utilisé pour la gestion des opérations Hadoop. De nombreux composants de Hadoop en dépendent [17].
- **Apache HBase** : est une base de données NoSQL orientée colonne située au-dessus de HDFS. HBase est utilisé pour sa rapidité d'accès en lecture/écriture aux grandes quantités de données. Elle utilise Zookeeper pour sa gestion afin de s'assurer que tous ses composants sont en cours d'exécution [16].
- **Apache Pig** : est un langage de flux de données et d'environnement d'exécution pour explorer de très grands ensembles de données. Pour son fonctionnement, Pig nécessite la présence du HDFS et du MapReduce sur cluster. Pig [18] est une abstraction sur la complexité de la programmation MapReduce. La plate-forme Pig comprend un environnement d'exécution et un langage de script nommé Pig Latin ; il est utilisé pour analyser des ensembles de données Hadoop. Son compilateur traduit le langage Pig Latin en séquences de programmes MapReduce.
- **Apache Hive** : Est un entrepôt de données distribuées. Hive gère les données stockées dans HDFS et fournit un langage de requêtes basées sur SQL pour exécuter des requêtes sur les données stockées dans Hadoop [19]. Hive permet aux développeurs qui ne connaissent pas MapReduce d'écrire des requêtes qui sont traduites par des jobs MapReduce sur la plateforme Hadoop. Comme Pig, Hive a été développé comme une couche d'abstraction, mais davantage orienté vers les analystes de base de données plus familiarisés avec SQL que la programmation Java [16].

- **Apache Oozie** : Développé par Yahoo!, c'est un gestionnaire de flux de travail pour coordonner l'exécution de plusieurs jobs MapReduce. Oozie est intégré à la pile Hadoop, il est capable de gérer un nombre important de complexité, en se basant sur l'exécution d'événements extérieurs qui comprennent la synchronisation et la présence de données nécessaires [16].
- **Apache Sqoop** : Est un outil de connectivité pour transférer des données entre des bases de données relationnelles et l'entrepôt de données HDFS de Hadoop. Sqoop s'appuie sur la base de données pour décrire le schéma des données importées/exportées et de MapReduce pour le fonctionnement de la parallélisation et la tolérance aux pannes [16].
- **Apache HCatalog** : Est une entrée relativement nouvelle, Apache HCatalog est un service qui fournit des schémas partagés et des services d'accès aux données d'abstraction pour les applications avec l'écosystème. L'objectif à long terme de HCatalog est de permettre l'interopérabilité entre les outils tels que Apache Hive et Pig afin qu'ils puissent partager des informations de métadonnées de Dataset [20].

Il existe trois modes de déploiement [21] de la plateforme Hadoop :

- **Mode autonome locale** (Local standalone mode) : c'est le mode par défaut de la plateforme où rien ne doit être configuré. Dans ce mode tous les composants de Hadoop s'exécutent sur un seul processus java ;
- **Mode pseudo-distribué** (Pseudo-distributed mode) : dans ce mode une JVM (Java Virtual Machine) est créé pour chaque composant de Hadoop, elles communiquent entre elles via les sockets réseau, ce qui fournit un mini-cluster, hébergé par un seul hôte ;
- **Mode entièrement distribué** (Fully-distributed mode) : dans ce mode, Hadoop est répartie sur une multitude de machines, où un nœud sera le

maître qui orchestre le travail des autres machines esclaves qui exécutent le travail.

Le mode fully distributed est le seul mode capable de déployer la plateforme Hadoop sur un cluster de machines, mais ce mode nécessite un plus grand travail de configuration.

2.4. Fonctionnement de Hadoop

Le Framework Hadoop est largement inspiré par deux produits Google [15]. Le premier est le Google File System (GFS) : un système de fichiers distribués conçu initialement par Google pour répondre aux besoins de stockage des données de leurs applications. Le second est le MapReduce, une architecture informatique popularisée par Google afin d'effectuer des calculs en parallèle. Hadoop ne requiert pas un matériel fiable et coûteux pour son fonctionnement.

La configuration technique de la plateforme Hadoop repose sur :

- HDFS (Hadoop Distributed File System);
- Le paradigme MapReduce.

2.4.1. HDFS : Hadoop Distributed File System

Quand les tailles des ensembles de données manipulées (Dataset) dépassent largement la capacité de stockage d'une seule machine physique, il serait nécessaire de les partager à travers un réseau de machines créant ainsi un système de fichiers distribués. Néanmoins, dans environnement réseau, toutes les complications de la programmation réseau apparaissent, ce qui le rend plus complexe qu'un système fichiers non distribués. Dans ce contexte, un des grands challenges est la réalisation d'un système de fichiers qui tolère la panne de nœuds sans souffrir d'une perte de donnée [22].

Hadoop est caractérisé par une architecture maître-esclave où le maître HDFS est le NameNode et les esclaves HDFS sont les DataNodes. Ces deux agents principaux composent ce système de fichiers [15]:

- Le NameNode gère l'arborescence et les métadonnées ;
- Les DataNodes fournissent l'espace de stockage pour des blocs de fichiers.

La Figure 2.2 illustre la méthode de stockage d'un fichier qui est décomposé en quatre (04) blocs où chaque bloc est répliqué trois (03) fois. Chaque répliqua d'un bloc est stocké dans un DataNode différent des autres répliquas du bloc.

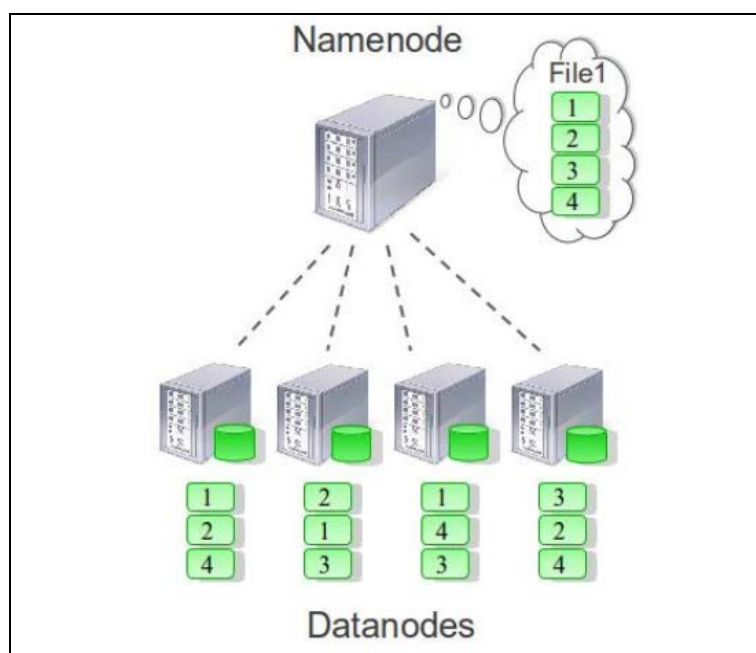


Figure 2.2 : Schéma de stockage des blocs dans HDFS [15]

Les données sont stockées par le système de fichiers de la machine par blocs de 64 Mo ou de 128 Mo par défaut. La lecture est séquentielle et l'écriture est en mode append-only⁴ [22].

⁴ Append-only files : c'est des fichiers dont les données peuvent être ajoutées, mais dans lequel les données existantes ne peuvent pas être modifiées. Ils conviennent parfaitement pour les fichiers de journalisation.

2.4.1.1. Architecture du HDFS

HDFS est un système de fichiers désigné pour stocker de très gros fichiers avec un modèle d'accès aux données en continu fonctionnant sur un cluster. La notion «gros fichiers» (Big data) désigne les fichiers dont la taille dépasse les centaines de méga-octets, giga-octets ou Téra-octet. Il existe des clusters avec une plateforme Hadoop qui stocke des péta-octets de données. HDFS fonctionne sans interruption visible pour l'utilisateur face à un nombre d'échecs considérables. HDFS est conçu pour écrire une seule fois et lire une multitude de fois c'est son modèle d'accès aux données [10].

Les fichiers hébergés dans HDFS doivent être écrit par un seul auteur. L'écriture est toujours faite à la fin du fichier. Il ne supporte pas l'écriture multiple ou la modification à un offset arbitraire dans un fichier. Cela pourrait être supporté dans les versions futures [19].

NameNode

Le NameNode fournit le stockage des métadonnées pour le système de fichiers partagés. Il assure également les opérations sur les métadonnées comme l'ouverture et la fermeture des fichiers. Il y a un seul serveur NameNode sur un cluster. Les métadonnées contiennent des informations critiques comme la structure des dossiers et l'emplacement des blocs constitutifs de chaque fichier sur le cluster. Le NameNode représente le maître dans le cluster HDFS.

Fonctionnalités du NameNode:

- Renseigne le client HDFS sur l'emplacement des données à compiler pour la requête et le prévient si un nœud de données est défaillant ;
- Responsable de la distribution et de la réplique des blocs ;
- Stocke et gère les métadonnées :
 - liste des fichiers ;
 - liste des blocs pour chaque fichier ;
 - liste des DataNodes pour chaque bloc ;

- attributs des fichiers (ex : nom, date de création, facteur de réplication).
- Journalise toute métadonnée et toute transaction sur un support persistant ;
 - lectures/écritures ;
 - créations/suppressions.
- Démarre à partir d'une image de HDFS (fsimage).

DataNodes

Les DataNodes sont l'emplacement où le fichier est hébergé. Ils stockent et reçoivent les blocs quand ils ont l'ordre de le faire par le NameNode ou par le client. Chaque DataNode envoie périodiquement un rapport au NameNode avec une liste des blocs qu'il stocke.

Fonctionnalité du DataNodes :

- Stocke des blocs de données dans le système de fichier local ;
- Maintien des métadonnées sur les blocs possédés (ex : CRC) ;
- Serveur de bloc de données et de métadonnées pour le client HDFS ;
- Envoie des Heartbeat⁵ avec le NameNode :
 - Message-aller vers le NameNode indiquant : son identité , sa capacité totale, son espace utilisé et son espace restant.
 - Message-retour depuis le NameNode : des commandes (copie de blocs vers d'autres DataNodes, invalidation de blocs, etc.)
- En plus du heartbeat, il informe régulièrement le NameNode des blocs qu'il contient.

5 Heartbeat : Signal qui permet au NameNode d'avoir, en continu, les états (actif/inactif) des DataNodes. Chaque DataNode envoie un « Heart Beat Signal » (un signal de battement de cœur) chaque 10 secondes (par default)

Secondary NameNode

Il s'agit d'une copie répliquée périodiquement du NameNode, dans le cas où celui-ci serait défaillant. Une énorme quantité de logs et de transactions (stockage conséquent) cause un redémarrage très lent du NameNode dû à sa prise en compte de beaucoup de changements. La solution est de réduire la taille des logs en utilisant un ou plusieurs processus distants. Cet élément n'avait pas été développé à l'origine mais il fait partie aujourd'hui de la plupart des architectures Hadoop en service [23].

Fonctionnalité du Secondary NameNode: Voir figure (Figure 2.3)

- Télécharge régulièrement les logs sur le NameNode.
- Crée une nouvelle image en fusionnant les logs avec l'image HDFS.
- Renvoie la nouvelle image au NameNode.

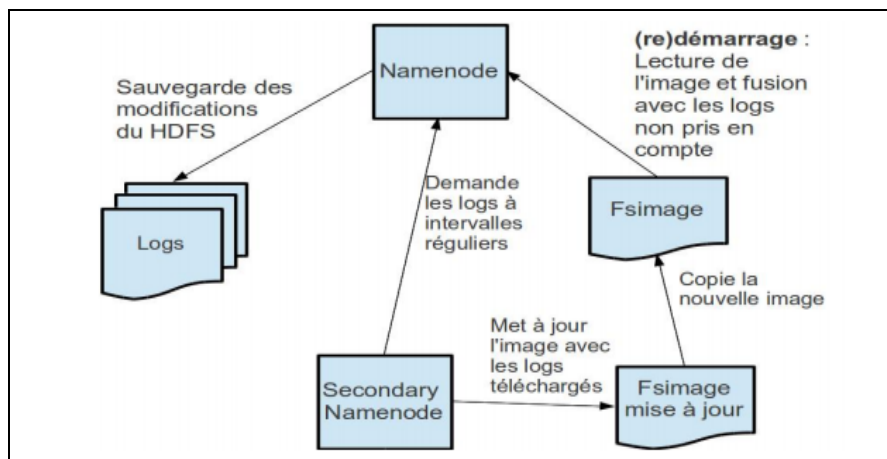


Figure 2.3 : Le fonctionnement du Secondary NameNode [23]

2.4.1.2. Lecture et écriture d'un fichier dans HDFS

Pour la lecture d'un bloc dans HDFS, le client HDFS interroge le NameNode pour récupérer la liste des DataNodes hébergeant ses différents réplicas, le client lit ensuite le bloc du DataNode, et s'il y a un problème avec le bloc, il lit le bloc d'un autre DataNode. Le client lit le plus proche réplica [15]. Voir la Figure 2.4 .

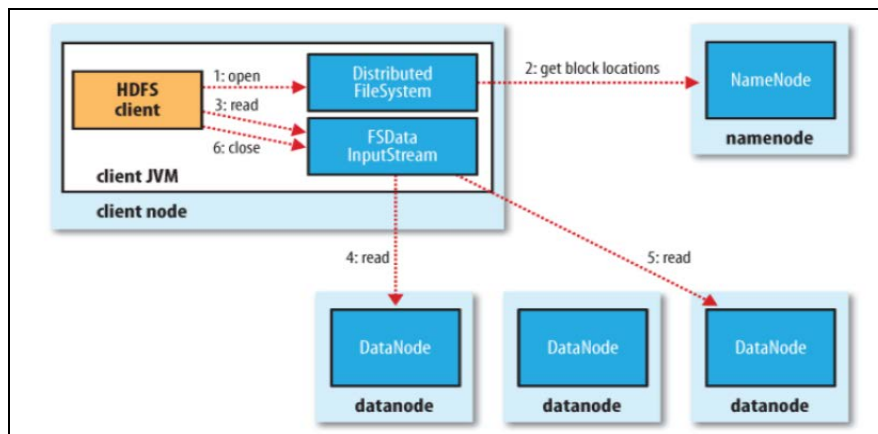


Figure 2.4 : Lecture d'un fichier dans HDFS. [15]

Le client récupère la liste des DataNodes sur lesquels sont placés ses différents réplicas et il écrit le bloc sur le premier de la liste. Ce dernier transfère les données au prochain DataNode dans le pipeline. Quand tous les réplicas sont écrits, le client recommence la procédure pour le prochain bloc du fichier (voir la Figure 2.5). La Stratégie de placement des blocs est comme suit:

- Un réplica sur le rack local.
- Un second réplica sur un autre rack.
- Un troisième réplica sur un rack d'un autre datacenter.
- Les réplicas supplémentaires sont placés de façon aléatoire.

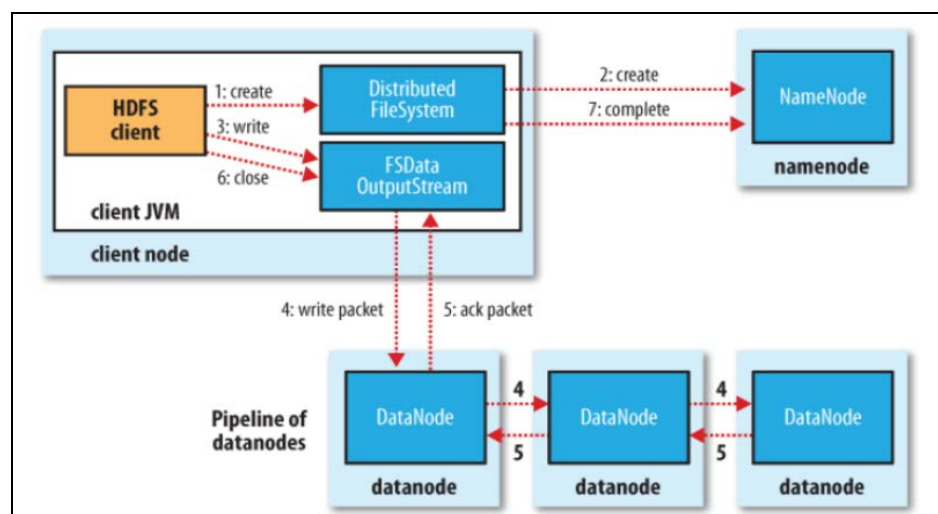


Figure 2.5 : Écriture d'un fichier dans HDFS . [15]

Remarque :

Au démarrage, le NameNode entre dans un mode spécial (safemode) dans lequel aucune réplication n'aura lieu, par contre, il recevra des rapports sur l'état actuel de la part des DataNodes (Blockreport), chaque rapport contiendra les informations concernant les blocs hébergés dans le DataNode émetteur. Un bloc de données est considéré dans un état « safe » s'il est répliqué autant de fois que la configuration du facteur de réplication, dans le cas contraire, un réglage de situation est effectué en ajoutant les blocs manquants. Par la suite, le NameNode quitte le mode « safe » pour reprendre son travail

2.4.1.3. Tolérance aux pannes

L'objectif principal de HDFS est de stocker des données sûrement même en présence des échecs. Les trois types d'échecs sont : les échecs de NameNode, les échecs de DataNode et les problèmes liés au stockage

- Échecs de NameNode

Le FsImage et l'EditLog sont les structures de données centrales de HDFS et l'échec du NameNode rend le cluster non fonctionnel. Dans le cas d'échec du DataNode, le dernier point de contrôle (checkpoint) peut être importé vers le NameNode. Une intervention est requise dans un tel cas. Afin de le faire, il faut :

- Créer un répertoire vide dans le répertoire spécifié dans la variable de configuration de dfs.name.dir.
- Spécifier l'endroit du répertoire de point de contrôle dans la variable de configuration fs.checkpoint.dir.
- Puis redémarrer le NameNode avec - l'option de importCheckpoint.
bin/hdfs namenode - importCheckpoint.

- Échecs de DataNode

Chaque DataNode envoie périodiquement un message au NameNode (Heartbeat message). Une séparation réseau peut faire perdre un sous-ensemble de DataNodes. Le

NameNode détecte cette situation par l'absence d'un message de « battement de cœur ». A cet effet, le NameNode marque les DataNodes sans battements de cœur comme morts. Aucune donnée qui a été enregistrée dans un tel DataNode mort n'est désormais disponible à HDFS. La mort de DataNode peut causer la diminution du facteur de répétition de quelques blocs, pour cela, le NameNode lance une ré-réplication pour les blocs perdus.

- Échecs de stockage

Un échec peut se produire en raison des défauts dans un dispositif de stockage ou des défauts de réseau. Donc il est possible qu'un bloc de données cherché d'un DataNode arrive au client corrompu. En fait, quand un client veut créer un fichier sur HDFS, il calcule une valeur de contrôle (checksum) pour chaque bloc et la sauvegarde dans un fichier caché dans le système HDFS. Puis, quand il récupère un bloc, il vérifie s'il s'agit de la même valeur de checksum, s'il ne s'agit pas de la même valeur il récupère le bloc auprès d'un autre DataNode.

2.4.2. Le paradigme MapReduce

MapReduce est un modèle de programmation proposé par Google. L'objectif est de traiter de manière parallèle de grandes quantités de données suivant un paradigme inspiré des langages de programmation fonctionnels. Le traitement de ces données s'effectue en deux étapes [21] :

- MAP : qui prend en entrée un ensemble de données et le transforme en un ensemble de couples de type (clé, valeur), triés en fonction des valeurs de la clé et attribués pour chaque donnée reçue par la fonction en entrée.
- REDUCE : qui a comme entrée les couples générés par la fonction de mappage, les transforme en une autre liste de couples (clé, valeur) plus réduite. La réduction de la liste initiale est faite en fusionnant les clés de chaque couple.

2.4.2.1. Fonctionnement du MapReduce

Le schéma du processus MapReduce tel que présenté par Google est représenté dans la Figure 2.6 suivante :

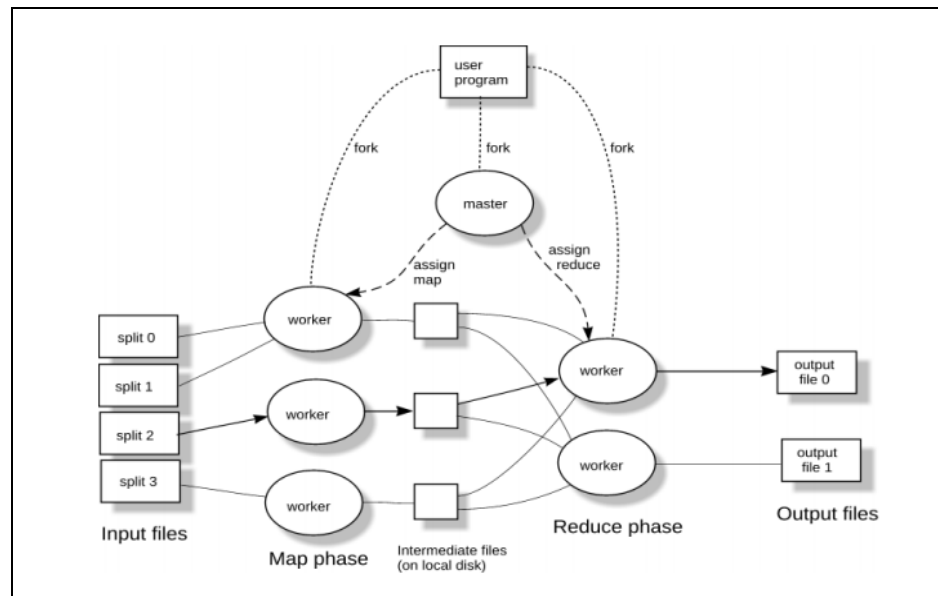


Figure 2.6 : Processus MapReduce de Google [21]

Le framework MapReduce est caractérisé par une architecture maître-esclave où :

- Le maître MapReduce (masters dans la Figure 2.6) : c'est le Jobtracker pour les versions de Hadoop inférieures à 2.0.0 et le Ressourcesmanager pour les versions supérieures à 2.0.0.
- Les esclaves MapReduce (workers dans la Figure 2.6) : sont les Tasktrackers pour les versions de Hadoop inférieures à 2.0.0 et les Nodemanagers pour les versions supérieures à 2.0.0.

La Figure 2.6 illustre le fonctionnement d'un programme MapReduce, au début, nous avons des données brutes (input data) et un programme d'utilisateur (user program). Quand l'utilisateur lance un programme, le master assigne des travaux de map à effectuer pour chaque TaskTracker « worker », chaque « worker » fait son traitement de map sur une partie «split» de données brutes. Des données intermédiaires sont générées par les

TaskTrackers, elles seront les entrées de la deuxième phase (reduce) qui sera effectuée elle-même aussi par les TaskTrackers. Chaque TaskTracker écrit ses fichiers de sortie dans un fichier séparé. Le détail de chaque composant est invoqué dans le point suivant.

La Figure 2.7 montre que l'exécution du Jobtracker et du NameNode est effectuée sur une même machine et que l'exécution du Tasktracker et du DataNode est réalisée sur une même machine.

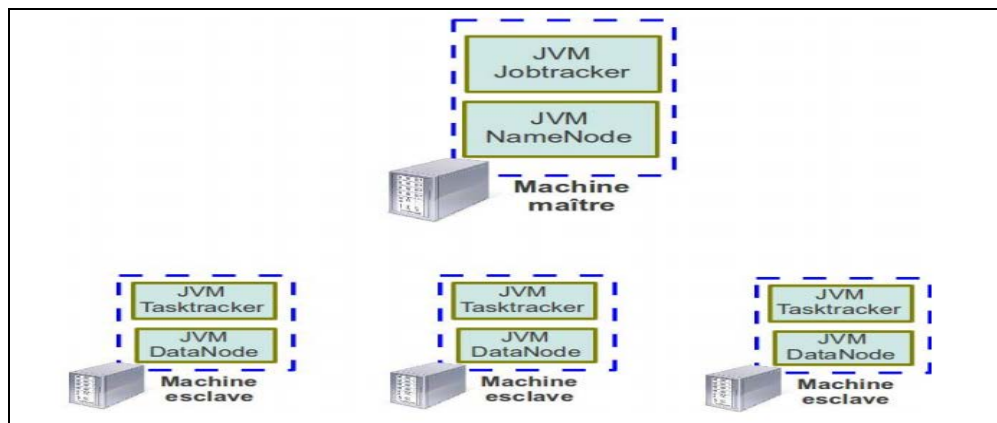


Figure 2.7 : Framework MapReduce et HDFS [21]

Jobtracker (gestionnaire de tâche)

C'est un processus qui fait la liaison entre l'application (le client) et Hadoop. Une fois que le code est soumis au cluster, le JobTracker détermine le plan d'exécution en déterminant quels répertoires de données à traiter, assigne à des nœuds (TaskTrackers) différentes tâches et surveille toutes les tâches durant leur fonctionnement. Si une tâche échoue, le JobTracker relancera automatiquement la tâche, probablement sur un nœud différent, jusqu'à une limite prédéfinie des tentatives. Les Fonctionnalités du Jobtracker sont les suivantes :

- Réception des jobs des clients ;
- Détermination des splits des données d'entrée d'un job ;
- Création des tâches de map en fonction du nombre de splits ;
- Création des tâches de reduce en fonction de la configuration du job ;

- Assignement des tâches aux Tasktrackers ;
- Réaffectation des tâches défaillantes ;
- Maintien des informations sur l'état d'avancement des jobs.

Tasktracker (moniteur de tâches)

Il représente l'exécutant dans ce modèle, chaque TaskTracker est responsable de l'exécution des différentes tâches que le JobTracker lui assigne. Bien qu'il y ait un TaskTracker simple par nœud esclave, chaque TaskTracker peut engendrer JVMs multiples pour manipuler beaucoup de tâches en parallèle « map » ou « reduce ».

Une responsabilité du TaskTracker est de communiquer constamment avec le JobTracker. Si ce dernier ne reçoit pas de rapports d'un TaskTracker dans une période spécifique, il supposera que le TaskTracker s'est brisé et resoumettra les tâches correspondantes à d'autres nœuds dans le cluster. Les fonctionnalités du Tasktracker sont les suivantes :

- Exécuter les tâches données par le Jobtracker.
- Exécuter des tâches dans une autre JVM (Java Virtual Machine nommée Child) en fonction du nombre de tâches qu'il peut exécuter simultanément (slot) :(paramétré par l'administrateur de la plate-forme et dépend du nombre de cœurs et de la capacité mémoire de la machine physique).
- Envoi des heartbeats avec le Jobtracker
 - message-aller : demande ou non d'une nouvelle tâche en fonction des slots disponibles
 - envoi de diverses informations (avancement des tâches en cours, espace mémoire, nombre de cœurs, charge)
 - message-retour : contenant une liste d'ordres : lancer une tâche, annuler une tâche, écriture définitive de la sortie sur le HDFS (commit) et réinitialisation du Tasktracker.

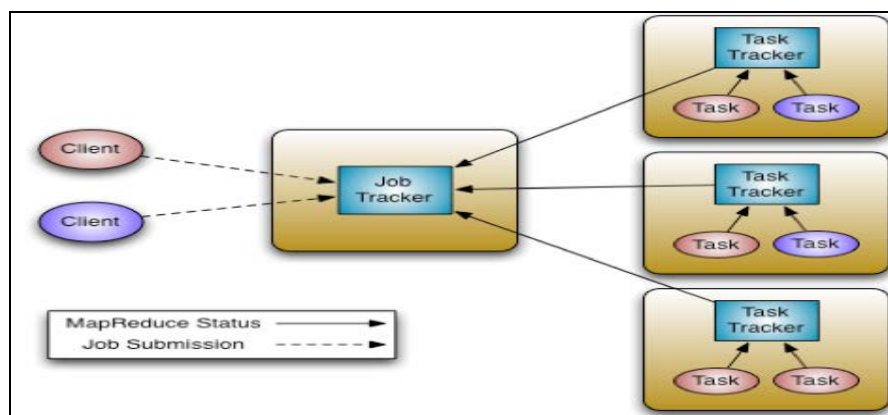


Figure 2.8 : Communication entre Jobtracker et les Tasktrackers [24]

2.4.2.2. Exemple de fonctionnement de MapReduce

La Figure 2.9 illustre les différentes étapes d'un job MapReduce

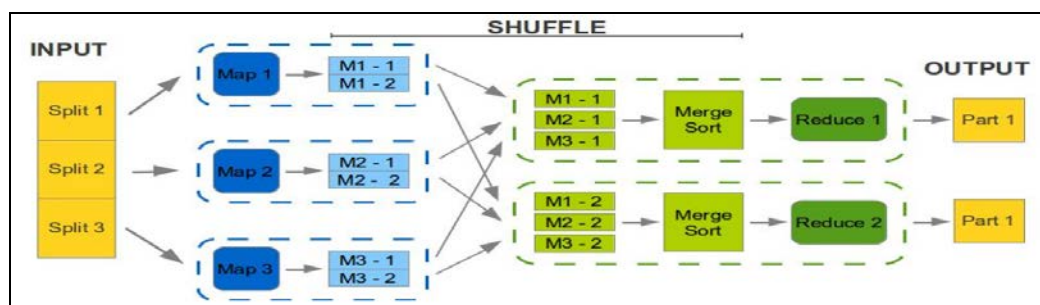


Figure 2.9 : Schématisation du paradigme Map-Reduce [25]

Un job de type MapReduce se déroule comme suit :

- Lecture d'une grande quantité de données(Input).
- **Phase MAP** : Chaque bloc de données (split) correspond à une tâche MAP. Un élément du split (ligne de fichier, tuple d'une base de données) est équivalent à une clé de type K1. A chaque clé de type K1 lue depuis le split, le processus MAP correspondant fait un appel à la fonction MAP(), cette fonction produit dans le flux d'information une liste de <clé, valeur> intermédiaire de type <K2,V2>.

MAP: (K1, V1) → list (K2, V2)

- **Phase Shuffle** : consiste en une transmission de données de la phase MAP vers la phase REDUCE. La responsabilité des Maps se résume au stockage local partitionné des couples clé/valeur de sortie et à l'assignement des clés parmi un nombre de partitions Reduce (nombre de partitionnement) préconfiguré par l'utilisateur. Chaque valeur de clé est associée à un Reduce unique. La responsabilité des Reduces se divise en 3 phases :
 - **Copy** : téléchargement sur chaque map de la partition qui lui est associée.
 - **Merge** : agrégation de l'ensemble des partitions téléchargées et agrégation des valeurs pour une clé donnée.
 - **Sort** : tri des différentes clés définissant l'ordre de lecture par le Reduce. Un ordre doit être défini pour chaque type de clé.
- **Reduce** : Le nombre de Reduce est défini à priori par l'utilisateur. Une fois la phase de map terminée, elle est suivie par une agrégation en liste de toutes les valeurs intermédiaires de type V2 associées à une clé de type K2. A chaque clé de type K2 le processus Reduce correspondant fait appel à la fonction `Reduce()`. Cette fonction produit dans le flux d'information une liste de <clé, valeur> de type <K3, V3>. Chaque paire <K3, V3> émise est enregistrée dans l'ensemble de données de sortie.

Reduce: (K2, list (V2)) → list (K3, V3)

- Écriture du résultat.

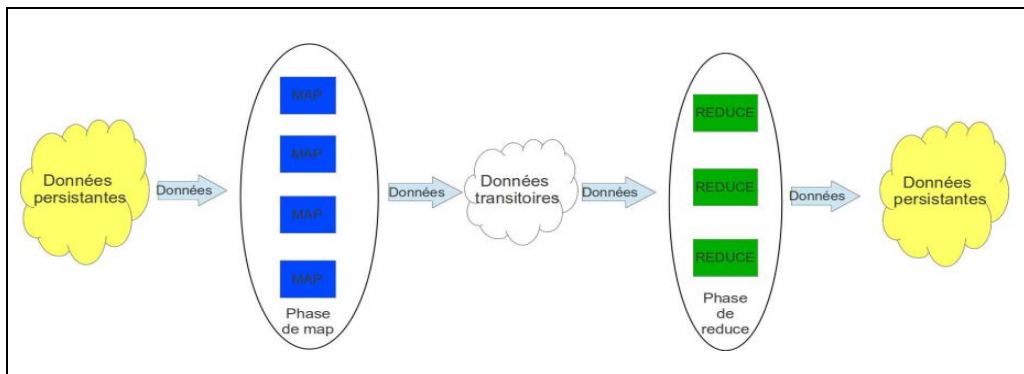


Figure 2.10 : Flux de données Map-Reduce.

La Figure 2.10 montre que la masse de données d'entrée est divisé en blocs appelés split, ils seront traités par différentes tâches, le calcul se fait en deux phases : Map et Reduce. Un flux de paires de <clé, valeur> se déplace entre : les données d'entrée et la phase de map, la phase de map et phase de Reduce et enfin la phase de Reduce et les données de sortie.

2.4.2.3. Tolérance aux pannes du MapReduce

Étant donné que la librairie MapReduce est destinée à traiter de très grandes quantités de données en utilisant des milliers de machines, ce procédé doit avoir une bonne tolérance aux pannes. Nous distinguons deux types de pannes qui sont :

Pannes de la machine esclave

Le master envoie un signal de test (ping) à toutes les machines esclaves périodiquement. S'il ne reçoit pas de réponse au bout d'un certain temps t , il les considère comme déconnectées. La tâche exécutée par cette machine sera redistribuée à une autre machine esclave. L'échec de cette dernière nécessite la reprise de la tâche depuis le début. Les tâches sont rapides à traiter donc cela n'influe pas sur le temps d'exécution total. On assure ainsi l'exécution de toutes les tâches indépendamment du travail des machines esclaves.

Pannes du master

Le master (machine maître) établit périodiquement des checkpoints de sa structure de données. Si le master tombe en panne, une nouvelle copie est générée et reprend l'exécution au dernier checkpoint. Ce fonctionnement, fait de la librairie MapReduce, une entité très robuste avec une forte tolérance aux pannes.

2.4.2.4. YARN

YARN est une évolution de l'architecture d'Hadoop permettant de décharger le Jobtracker qui avait tendance à cumuler trop de rôles et devenait donc complexe à

maintenir et à mettre à l'échelle (scaler). Cette remise à plat des rôles a permis aussi de découpler Hadoop de MapReduce et de permettre à des Framework alternatifs d'être portés directement sur Hadoop et HDFS et non au-dessus du MapReduce. Cependant, une couche de rétrocompatibilité avec les versions antérieures de Hadoop est maintenue afin de permettre une migration en douceur. La seule contrainte étant de recompiler ses jobs avec la nouvelle version d'Hadoop qui garantit une meilleure scalabilité

L'idée fondamentale de YARN est de diviser les deux principales responsabilités du JobTracker/TaskTracker en deux entités distinctes. Voir la Figure 2.11:

- Un ResourceManager global (RM).
- Un ApplicationMaster par application (AM).
- Un NodeManager esclave par nœud (NM).
- Un conteneur par application fonctionnant sur un NodeManager.

Le ResourceManager et le NodeManager forment le nouveau système générique de gestion des applications de manière distribuée [24].

L'ApplicationMaster par application a la responsabilité de négocier les conteneurs de ressources appropriée du planificateur, le suivi de leur état et le suivi de leurs progrès. Du point de vue système, l'ApplicationMaster fonctionne comme un conteneur normal.

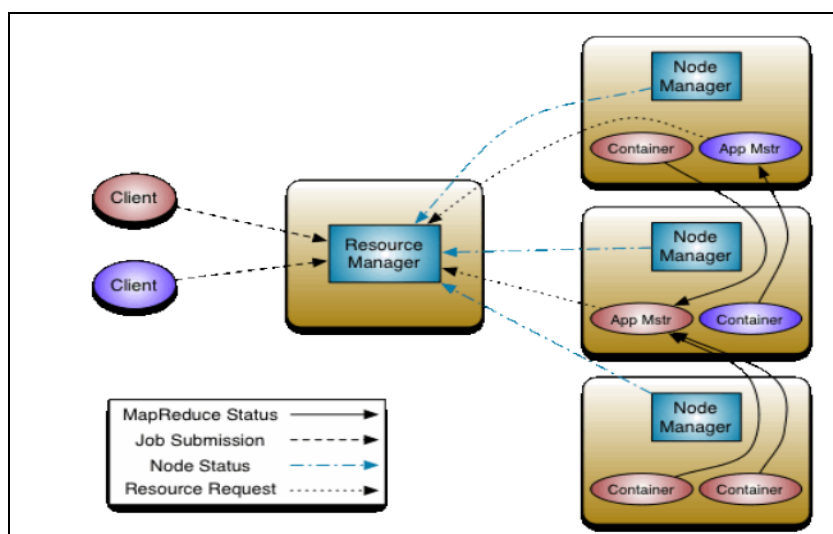


Figure 2.11 : Architecture de YARN [24].

ResourceManager

Le ResourceManager (RM) est le maître qui arbitre toutes les ressources disponibles du cluster et qui permet de gérer les applications distribuées en cours d'exécution sur le système YARN. Il travaille en collaboration avec les NodeManagers par nœud (NMs) et les ApplicationMasters par application (AMs).

- NodeManagers acceptent les instructions du ResourceManager et gèrent les ressources disponibles sur un seul nœud.
- ApplicationMasters sont des entités spécifiques responsables de la négociation des ressources avec le ResourceManager et du travail avec les NodeManagers pour démarrer les conteneurs [25].

Le ResourceManager a un ordonnanceur qui est responsable de l'allocation des ressources aux différentes applications en cours d'exécution, selon des contraintes telles que les capacités de la file d'attente et les limites d'utilisation. Le planificateur exécute sa fonction de planification basée sur les besoins des applications.

NodeManager

Le NodeManager (NM) est un agent qui prend en charge les nœuds de calcul individuels dans un cluster Hadoop. Il est responsable du lancement des conteneurs des applications. Il tient au courant le ResourceManager (RM), supervise la gestion du cycle de vie des conteneurs, surveille l'utilisation des ressources (mémoire, CPU, disque, réseau) des conteneurs individuels, suit la santé des nœuds, gère enfin le journal et les services auxiliaires qui peuvent être exploités par des applications différentes du YARN [25].

Une des distributions la plus complète, la plus testée et la plus largement déployée dans le monde d'Apache Hadoop est CDH (Cloudera's Distribution Including Apache Hadoop) est une distribution. CDH est 100% open source et c'est la seule solution Hadoop pour offrir le traitement par lots, le SQL interactif et la recherche interactive. Cet outil puissant d'automatisation de gestion offre la manière la plus rapide et la plus simple

pour obtenir un cluster Hadoop. La combinaison de Hadoop et Cloudera conduit à un projet Hadoop qui sera déployé plus rapidement et avec plus de succès [26].

2.5. Conclusion

Ce chapitre a été consacré à l'étude de la plateforme Hadoop, dans lequel nous avons présenté la plate-forme Hadoop, son mode de fonctionnement et quelques sous projets qui relèvent de la même catégorie de l'informatique distribuée et du traitement de données à grande échelle. Le fonctionnement de Hadoop se base sur le système de fichier distribué HDFS et un paradigme Map-Reduce de traitement de données très volumineuses.

HDFS est le système de fichiers distribués utilisé, pour le stockage des données Hadoop. HDFS a une architecture de type maître/esclave et est constitué d'un unique NameNode, d'un espace de nom et de plusieurs DataNodes. HDFS se caractérise par sa grande tolérance aux fautes et le fait qu'il soit conçu pour stocker de très gros volumes de données de manière fiable.

Le modèle MapReduce est aujourd'hui l'un des modèles de programmation parallèle les plus utilisés. Définissant une architecture Maître-Esclave, dans laquelle s'exécute une succession d'ensembles de tâches indépendantes, elle permet le traitement parallèle de grandes masses de données. Un des principaux avantages de ce modèle est sa simplicité. Chaque Job MapReduce soumis par un utilisateur est une unité de travail composée de données en entrée, des deux fonctions Map et Reduce et d'informations de configuration.

Hadoop se distingue par :

- Son accessibilité: Hadoop fonctionne sur les grands clusters, de machines différentes, tel que le Cloud « Amazon's Elastic Compute Cloud » (EC2).
- Sa robustesse: il fonctionne sur des clusters de différents types de matériel et permet la tolérance aux pannes lors de l'exécution des tâches par la

réplication. Il assure l'intégrité des données, l'équilibrage du cluster et la création des points de restauration.

- Sa simplicité: Le parallélisme est géré automatiquement sur Hadoop ce qui facilite l'écriture des codes parallèles.
- Sa scalabilité (passage à l'échelle): Le rendement ou temps d'exécution des clusters gérés par Hadoop est optimisé.

Comme pour toute technologie émergente, Hadoop présente encore quelques défauts intrinsèques qui doivent être corrigés. Nous citons quelques-uns :

- La distribution Hadoop offre un point unique de défaillance avec le serveur NameNode, qui gère les métadonnées de fichiers stockés dans le HDFS. Le serveur NameNode est l'élément central de tout système de fichiers HDFS : lorsqu'il tombe en panne, le système de fichiers passe hors ligne. Ce problème, la communauté Open Source de Hadoop essaie de le résoudre en cherchant un composant offrant une haute disponibilité pour les plateformes d'analyse Big Data.
- Hadoop manque de fonctionnalités avancées de protection de données. Ainsi, au lieu d'utiliser RAID, les développeurs doivent définir manuellement le paramètre de réplication des données pour HDFS(valeur par défaut :3),ceci limite fortement l'efficacité du stockage.
- Hadoop exige la construction de clusters de calcul dédiés. Dans une architecture de référence Hadoop conventionnelle, le cluster de calcul utilise le stockage en attachement direct(DAS), difficilement partageable avec d'autres applications.
- Hadoop exige un investissement supplémentaire dans des individus possédant des compétences Hadoop. En outre, les déploiements traditionnels de Hadoop sont parfois difficiles à intégrer avec les applications d'entreprise existantes. Les données sont accessibles via des protocoles Hadoop et HDFS ou via une interface SQL, ce qui implique le développement d'interfaces de programmation.

Pour manipuler les données, avec des interfaces de programmation dans un environnement distribué, réaliser la jointure entre les différents DataSets est une opération inéluctable pour effectuer des analyses Big Data.

Dans le chapitre suivant, nous détaillons les algorithmes parallèles de jointure.

Chapitre 3 :

Algorithmes parallèles de jointure

3.1. Introduction

La jointure est une opération indispensable dans le domaine des bases de données relationnelles. Elle requiert des ressources machines : mémoire, CPU et stockage. Selon l'algorithme mis en œuvre, ces trois ressources pouvant être différemment sollicitées.

Au-delà des difficultés intrinsèques à la jointure, l'évolution constante des environnements techniques a ouvert d'autres voies visant à utiliser de façon optimale les capacités des machines (mémoire, multiprocessing...).

En plus des évolutions des architectures des ordinateurs, la jointure est également impactée par des évolutions en matière d'utilisation des données et de leur répartition. De fait, l'extension des réseaux informatiques et l'inter-connectivité entre une multitude de sites, implique une dissémination des données au sein d'un vaste réseau, et donc une distribution des données exigeant toujours des traitements de type jointure.

Dans ce chapitre, nous détaillons les différentes approches de jointure et leur implantation, indépendamment de l'environnement d'exécution. Nous distinguons trois (03) principales approches algorithmiques permettant de réaliser la jointure : La jointure par boucles imbriquées (nested-loop), La jointure par tri-fusion (sort-merge) et la jointure par fonction de hachage (hash-join).

Ensuite, nous exposerons leur adaptabilité aux architectures parallèles, en essayant de dégager parmi les trois méthodes de base, celles qui sont les mieux appropriées à une situation donnée. Par la suite, nous montrerons l'impact de la répartition des données à joindre et la manière dont les tâches de traitement sont affectées.

3.2. Opération de jointure

La jointure entre deux relations « R » et « S » sur les attributs « R.A » et « S.B » consiste à coupler les tuples de « R » et « S » ayant les attributs « R.A » et « S.B » égaux. De façon générale, un tuple de R (respectivement de S) peut être couplé à un nombre quelconque de tuples de S (respectivement de R). Dans le cas particulier où les attributs R.A et S.B sont tous les deux des clés uniques de leurs relations respectives, un tuple de R (respectivement de S) ne peut être couplé qu'au plus avec un tuple de S (respectivement de R). Il existe une multitude de techniques de mise en œuvre de cette opération ainsi que de nombreuses études portant sur le sujet. Trois méthodes se sont dégagées regroupant l'ensemble des approches qui peuvent être proposées.

Les trois principales méthodes visant à réaliser ces opérations sont :

- la jointure par boucles imbriquées qui consiste pour deux relations R et S, d'une part à parcourir une de ces relations de façon séquentielle et d'autre part, à comparer chaque tuple de R lu avec l'ensemble des tuples de S afin d'extraire les tuples pour lesquels $A = B$. La première relation parcourue (ici R) est désignée sous le nom de «inner» et la seconde (ici S) est désignée sous le nom de «outer».
- La deuxième méthode se base sur un tri préalable des deux relations R et S, suivi d'une mise en œuvre équivalente aux boucles imbriquées avec toutefois une économie du nombre de lecture de la seconde relation.
- Enfin, la troisième méthode, quant à elle, repose sur une fonction de hachage des deux relations visant à permettre des accès directs aux tuples de S ayant une relation avec les tuples de R, sans nécessiter la recherche séquentielle par parcours de la relation S.

3.2.1. La jointure par boucles imbriquées

Cette jointure, présentée dans l'algorithme ci-dessous, est la plus naturelle des trois approches. Elle est composée de deux (02) boucles, la première boucle parcourt la première relation, et pour chaque tuple de la relation parcourue, le second parcours désigne la seconde relation pour permettre de trouver les tuples ayant une égalité des attributs de jointure des deux relations [27].

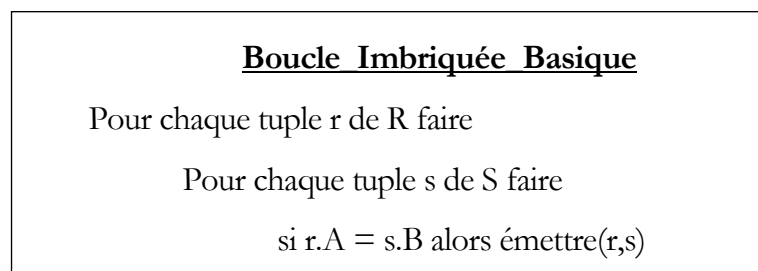


Figure 3.1: Algorithme Boucle Imbriquée Basique.

Cet algorithme est robuste et simple mais présente néanmoins un défaut majeur : la seconde relation est parcourue par un nombre important de fois; $||R||$ (nombre de tuples de R) fois. Par conséquent, il engendre des accès disques répétés aux mêmes données des deux relations provoquant un nombre important de défaut de pages.

Pour pallier à l'inconvénient de cette méthode, une modification consiste à bufferiser les données en mémoire pour limiter le nombre de défauts de pages. L'algorithme à accès par blocs explique cette modification où M représente le nombre de pages mémoire disponibles pour le traitement (Figure 3.2).

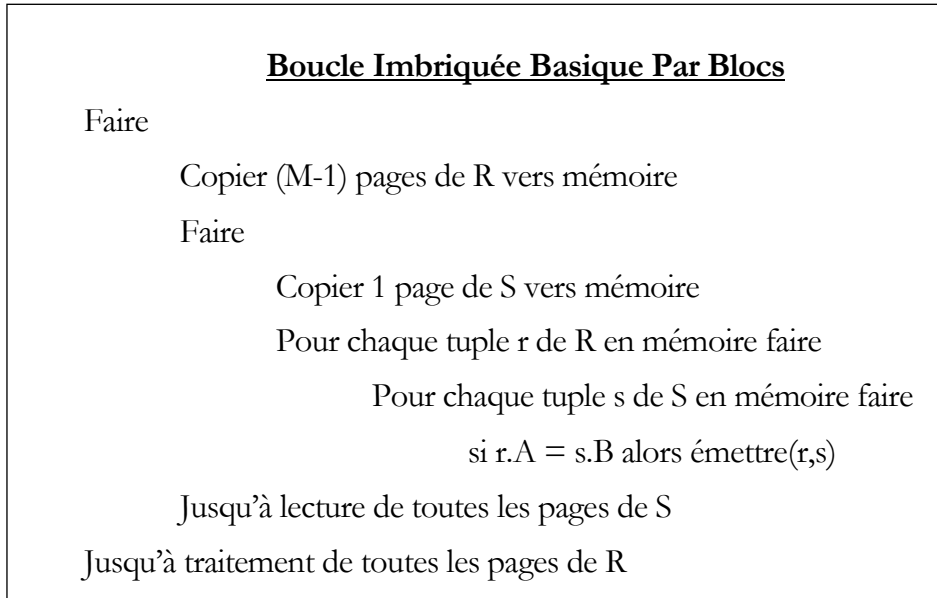


Figure 3.2 : Algorithme Basique de boucles imbriquées par blocs.

Cette modification rend le nombre de lecture en disque de l'ordre de [27]:

$$|R| + \left\lfloor \frac{|R|}{(M-1)} \right\rfloor * |S| \quad \text{Où } |R| \text{ et } |S| \text{ sont le nombre de pages de R et S.}$$

Cependant, nous parcourrons toujours $|R|$ fois la relation S, que cela soit en mémoire ou en disque. Ce qui implique une complexité des deux algorithmes de l'ordre de $O(|R| * |S|)$, avec $|R|$ et $|S|$ le nombre de tuples respectivement de R et S.

En effet, dans tous les cas, S est parcourue pour chaque tuple de R, et chaque tuple de R peut ne pas avoir ou très peu de tuples correspondants par rapport au prédicat de la jointure dans S.

L'utilisation d'index sur les attributs de jointure vise à repousser cette limite, plus particulièrement sur l'attribut de jointure de la relation S. Le principe d'un index d'une relation sur un attribut, consiste à associer à chaque valeur d'attribut une position « physique » (numéro de page...) sur disque associé permettant un accès direct au tuple. Ainsi, il n'est plus utile de parcourir à chaque itération entièrement la relation S puisque il suffit d'accéder sur disque au tuple associé à la relation R directement à travers son index (Figure 3.3).

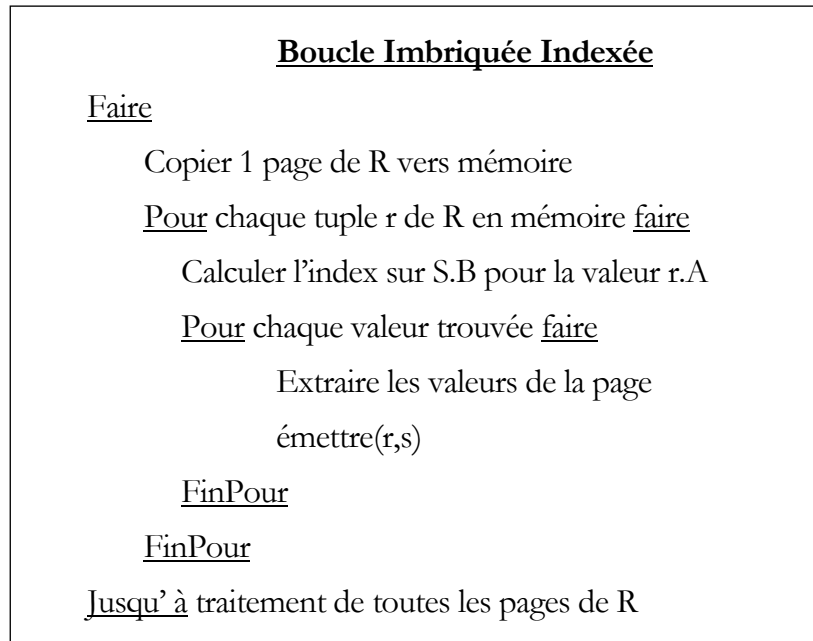


Figure 3.3 : Algorithme de boucle imbriquée indexée.

Le parcours de S ne dépend plus du nombre de pages de R en mémoire. En d'autres termes, il n'est plus utile de lire un nombre important de pages de R. Ainsi, la zone mémoire libérée peut alors contenir l'index et éviter qu'il ne devienne un goulet d'étranglement. La complexité théorique de cet algorithme est de l'ordre de $O(|R| + |S|)$. Le nombre d'accès disque est proportionnel à la sélectivité de la jointure, qui est définie par le rapport entre le nombre de tuples résultant de la jointure et le produit du nombre de tuple de R par le nombre de tuples de S :

$$\frac{||R \bowtie S||}{||R|| * ||S||}$$

Pour consolider cette méthode, le tri préalable des relations aura un apport significatif sur cette dernière (sans négliger le coût du tri). Ce qui nous amène à une autre famille d'algorithmes de jointure basés sur la tri-fusion.

3.2.2. La jointure par tri-fusion

La méthode de jointure par tri-fusion requiert un tri préalable suivant les attributs de jointure des deux relations. Un seul parcours des relations est nécessaire pour extraire les tuples ayant une égalité sur les attributs de jointure. Il est toutefois nécessaire de procéder par moment à des retours arrière sur l'une des relations dans le cas de non unicité de son attribut de jointure. L'impact de ce retour arrière est toutefois limité car il est la plupart du temps réalisé en mémoire, mais des relectures sur disque suite à un défaut de page restent possibles. L'algorithme résultant est le suivant :

Tri-fusion

Trier R sur R.A et S sur S.B selon l'ordre ascendant
 Initialiser r et rp avec le premier tuple de R
 Initialiser s et sp avec le premier tuple de S
 Tantque non.eof(R) ou eof(S) faire
 Tantque non.eof(R) ou eof(S) or r.A = s.B faire
 Si (r.A < s.B) initialiser r et rp avec le tuple suivant de R
 Sinon initialiser s avec le tuple suivant de S
 Fintantque
 Sinon (eof(R) ou eof(S))
 tantque r.A = rp.A et non eof(R) faire
 tantque s.B = r.A et non eof(S) faire
 émettre(rp, s)
 initialiser s avec le tuple suivant de S
 fintantque
 Initialiser r avec le tuple suivant de R
 Si r.A = sp.B alors initialiser s à sp // retour arrière
 Fintantque
 Fintantque

Figure 3.4: Algorithme de la jointure par tri-fusion.

Cet algorithme fonctionne indépendamment de la cardinalité des clés de jointure. Dans le cas de clés uniques, il n'est pas nécessaire de vérifier le test du retour arrière et l'algorithme devient alors simplifié. L'algorithme est cependant simple et robuste, le principal prérequis étant la

nécessité d'un tri; mais, dans le cas où les relations sont déjà maintenues triées, son intérêt devient plus grand.

Dans la section suivante nous détaillons une autre méthode fondée sur une fonction de hachage visant à segmenter les données d'une part, et en accélérer l'accès d'autre part.

3.2.3. La jointure par fonction de hachage

Les algorithmes de jointure préalablement décrits souffrent de problème de parcours d'un certain nombre de fois de l'une des deux relations, alors même que la plupart des valeurs parcourues ne participent pas au résultat. Un tri préalable pouvant être coûteux ou la disponibilité d'un index précisément sur les attributs de jointure n'étant pas toujours garantie. Une autre technique permet l'accès direct aux données pertinentes pour un tuple courant donné, avec un surplus de coût acceptable. La jointure par fonction de hachage découpe les relations R et S en partitions $R_1, R_2, \dots, R_n$ et $S_1, S_2, \dots, S_n$ tel que pour tout tuple r et s avec $r \in R_i$ et $s \in S_j$, si $i \neq j$ alors $r.A \neq s.B$. Dans ce cas, la jointure globale de R et S peut être ramenée à l'union des jointures R_i et S_i . Cette condition de découpage est purement théorique et il est difficile d'arriver immédiatement à cette situation. En outre, les différentes partitions d'une même relation peuvent être de tailles différentes, et il en est de même pour les partitions R_i et S_i .

L'algorithme de base peut être directement amélioré dès lors que l'on dispose de suffisamment de mémoire pour stocker plusieurs partitions et limiter ainsi les accès disque répétés. Cet algorithme simple a été implémenté pour la première fois au sein du système de base de données GRACE [28]. Il se décline de la façon suivante :

Jointure de base par hachage

```

Découper R en n partitions avec une fonction de hachage sur R.A
Découper S en n partitions avec la même fonction de hachage sur S.B
  Pour chaque partition i faire
    Pour chaque tuple r de Ri faire
      hachage sur r.A et insertion dans la table de hachage
    FinPour
  Pour chaque tuple s de Si faire
    hachage sur s.B et sonder la table de hachage
    si r.A = s.B alors émettre(r,s)
  FinPour

```

Figure 3.5 : Jointure de base par hachage.

Cette jointure de base identifie deux fonctions de hachage. La première est utilisée lors de la première phase du traitement pour la partition des relations. Cette phase est nommée phase de construction (built). Générer la table de hachage lors de la seconde phase est nommée phase de sondage (probe). Les deux fonctions peuvent être identiques, mais l'utilisation de deux fonctions distinctes évite les problèmes de collisions liées aux fonctions de hachage. Par ailleurs, l'objectif de l'une et l'autre des fonctions n'est pas le même. L'une a pour but de distribuer les tuples vers un ensemble de partition de la façon la plus équitable et éviter ainsi une forte disparité des tailles des partitions. L'autre a pour but l'accès efficace aux tuples de la première relation.

Dans cette version d'algorithme un axe de recherche à explorer se dégage, à savoir l'impact du nombre d'entrées/sorties sur les performances. En effet, dans le cas de « la boucle imbriquée », la mise en concurrence des deux relations (i.e. Boucles) induit un nombre de défaut de pages pouvant être conséquent et donc extrêmement pénalisant car il est en fonction de la taille des relations et la quantité mémoire disponible. Une évolution naturelle consiste donc à veiller à ce qu'une partition de la relation R puisse tenir dans une quantité mémoire fixée, donnant ainsi une seconde contrainte qui est :

$$|R_i| \leq M-2, \quad \text{pour } 1 \leq i \leq n \text{ où } M : \text{le nombre de pages mémoire disponibles.}$$

Les pages restantes (02) sont utilisées pour le stockage de la partition S_i pour la première page et, comme tampon du résultat pour la seconde page. Néanmoins, les partitions étant générées à l'aide d'une fonction de hachage, elles-mêmes prenant comme critère de calcul l'attribut de jointure, il est extrêmement difficile de prédire la taille de toutes les partitions. De ce fait, il n'est pas exclu qu'une partition soit supérieure en pages au nombre maximum de pages disponibles ($M-2$). Pour éviter ce cas de figure, une solution consiste à partitionner à nouveau la ou les partitions posant problème jusqu'à atteindre une taille inférieure à $M-2$.

Les algorithmes vus jusqu'à maintenant favorisent les accès aléatoires au disque, qui sont beaucoup moins efficace que les accès disque séquentiels. L'algorithme de la figure suivante, se propose donc de favoriser les accès séquentiels.

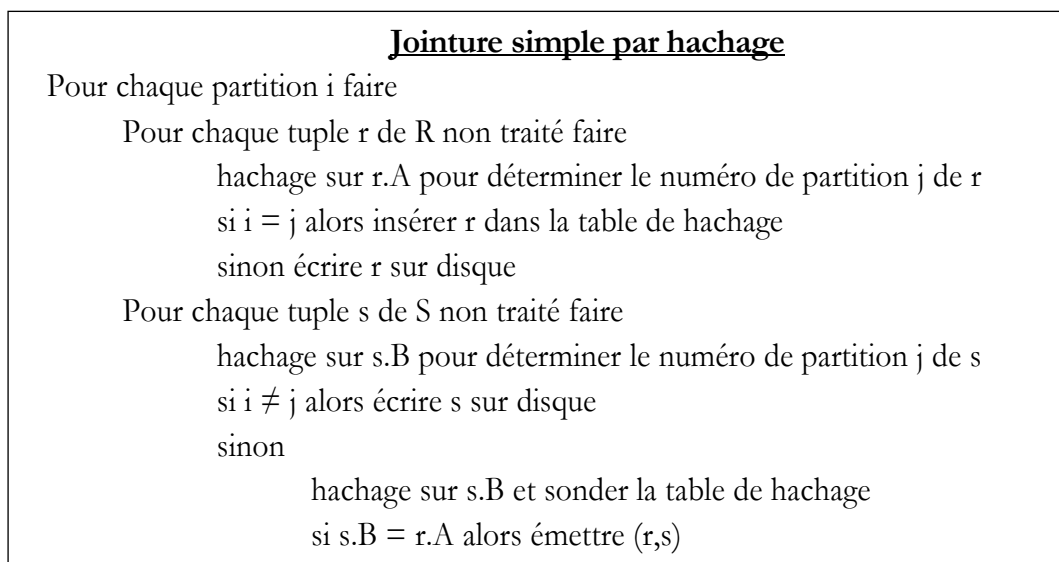


Figure 3.6 : Jointure simple par hachage.

Cet algorithme procède par itération sur le numéro de la partition. Ainsi, à chacune de ces itérations, soit un tuple fait partie de la partition courante et dans ce cas il est traité (mis en table de hachage ou jointure selon qu'il s'agit de R ou S), soit il n'en fait pas partie et sera enregistré sur disque. Si nous souhaitons éliminer les écritures aléatoires et donc favoriser les écritures séquentielles, il suffit, dans le cas où un tuple n'est pas concerné lors de l'itération courante, de le rejeter sans écriture sur disque. Certes, cela multipliera les lectures à partir du disque, mais ces

lectures restent séquentielles et donc beaucoup plus performantes que les lectures aléatoires les plus concurrentes[29].

Nous avons présenté trois familles d'algorithmes de jointure. D'autres algorithmes, visant à apporter des améliorations particulières ou à répondre à des jointures particulières, ont été également proposés. Nous nous sommes contentés de présenter les plus importants dont le pivot intrinsèque relatif à chacun est :

- a) Les boucles imbriquées
- b) Le tri des relations (suivi de boucles imbriquées)
- c) Le partitionnement et le hachage des relations

Ces algorithmes trouvent leur justification dans diverses configurations. L'un va être plus performant en cas de présence d'index sur l'une des relations ; L'autre sera favorisé dans le cas de relations triées. Mais d'autres critères de choix, plus techniques, peuvent également entrer en ligne de compte tel par exemple la taille mémoire disponible eu égard à la taille des relations, ou plus fonctionnelles, telle que la distributivité des valeurs des attributs de jointure.

Un autre axe d'évolution souhaité ou subit est la parallélisation des traitements de jointure. Dans cet autre environnement, tous les algorithmes cités ne sont pas équivalents face la parallélisation comme nous le présentons ci-après.

3.3. Jointure Parallèle

Dans le cas où les données des relations sont distribuées sur un vaste réseau ou si nous disposons de machines à architecture parallèle, l'opération de jointure est forcément parallèle. En outre, la jointure parallèle est utilisée pour diminuer les temps de traitement en exploitant au mieux les ressources machine.

Des extensions des algorithmes vus auparavant vers le parallélisme ont été proposées. Et, comme tout traitement parallèle, ceux-ci se déclinent en 3 étapes qui sont :

- a) Décomposition de la jointure en n tâches.

- b) Assignment des n tâches à p processeurs.
- c) Assemblage des résultats locaux.

L'étape « c » est fortement dépendante de la façon dont l'étape « a » est réalisée. Cette étape « c » peut également être optionnelle. Par ailleurs, nous pouvons distinguer deux familles d'architectures parallèles : les architectures partagées et les architectures distribuées.

- a) Architectures partagées : dans ce type d'architectures des ressources telles que la mémoire ou l'espace de stockage disque sont partagées et accessibles de façon concurrentielle par l'ensemble des processeurs utilisés.
- b) Architectures distribuées : dans ce cas, les ressources sont distribuées sur différentes machines interconnectées par un réseau de communication.

Notre problématique relative à la jointure parallèle avec MapReduce étant plutôt concernée par la seconde architecture, nous mettrons essentiellement l'accent sur les algorithmes adaptés à ce type d'architectures.

3.3.1. Jointure parallèle par boucles imbriquées

La jointure par boucles imbriquées est l'un des algorithmes qui se prête le mieux à la parallélisation. En effet, une parallélisation naturelle du traitement consiste à distribuer la relation R vers les p processeurs (i.e. machines) concernés. Chaque processeur réalisant localement la jointure de sa partition avec l'ensemble de la relation S . Par conséquent, avant d'effectuer la jointure, la réplication de la relation S vers l'ensemble des processeurs (i.e. nœuds) concernés est nécessaire. L'algorithme résultant est le suivant :

Jointure parallèle par boucles imbriquées

Chaque nœud qui dispose d'un fragment de S le dispatche aux autres nœuds
Distribution de R vers l'ensemble des nœuds concernés
Chaque nœud i joint la partition R_i avec l'ensemble de la relation S

Figure 3.7 : Jointure parallèle par boucles imbriquées.

La jointure locale des partitions R_i avec S peut être réalisée selon les algorithmes précédemment exposés. Si le nœud i en charge de la partition R_i dispose d'un index sur celle-ci, alors l'algorithme boucles imbriquées indexé (section 3.2.1) peut être utilisé. En effet, la relation S peut être déjà distribuée, sur un ensemble de machines pour des besoins techniques (exemple : stockage) ou fonctionnels, et donc disposer d'index permanents. Cette distribution préexistante permet également une autre optimisation : seuls les nœuds disposant d'une partition de la relation S sont utilisés pour la jointure parallèle. Cependant, cette optimisation peut être opposée à la nécessité de distribuer le traitement sur un large ensemble de nœuds afin de garantir le meilleur équilibre de charge.

En général, une relation est distribuée pour raison de contrainte technique ou fonctionnelle, suivant la valeur d'un attribut de cette même relation. Cet attribut peut être choisi suivant :

- Sa bonne distribution : lorsqu'il s'agit de contrainte technique visant notamment à faire supporter le stockage d'une relation par plusieurs nœuds. Dans ce cas, nous souhaitons que le partage soit équitable entre les différents nœuds.
- Des bases fonctionnelles : les traitements produisant la relation en question sont distribués sur un ensemble de nœuds. A titre d'exemple, la prise en charge d'un intervalle de numéros de clients par chaque nœud.

En effet, l'attribut de distribution de la relation n'est pas nécessairement l'attribut sur lequel porte la jointure. Les auteurs Lu et al.[30] proposent une solution d'abstraction à cette problématique. Elle s'appuie sur une table de translation MapS dont le schéma est (P,B) . Sur la base de cette table, chaque nœud disposant d'une partition de R peut identifier les nœuds disposant d'une partition de S vers lesquels il doit envoyer sa partition R . Ainsi, pour chaque tuple s de S , il existe un tuple m dans MapS tel que $m.B = s.B$ et $m.P$ est l'attribut de distribution de la relation S sur les différents nœuds.

Pour un tuple r , le tuple m de MapS tel que $m.B = r.A$ nous permet d'extraire l'attribut $m.P$ correspondant et donc identifier le site sur lequel la partition de S est localisée. Une extension de cette méthode consiste à distribuer également MapS selon l'attribut $m.B$, ce qui produit l'algorithme suivant prenant comme hypothèse structurante le fait que S est distribuée selon l'attribut $S.P$:

Jointure parallèle par boucles imbriquées

Pour chaque tuple s de S faire
 maintenir un tuple m de MapS avec $m.P = s.P$ et $m.B = s.B$
Partition de MapS selon MapS B et stockage de MapSi sur le nœud N_i
Pour chaque tuple r de R faire
 Trouver N_i contenant MapSi avec $m.B = r.A$
 Trouver N_j contenant S_i avec $s.P = m.P$
 Envoyer r à tous les nœuds N_j
Réaliser la jointure sur chaque nœud N_i

Figure 3.8 : Jointure parallèle par boucles imbriquées améliorée.

Dans le cas où l'attribut de jointure est également l'attribut de distribution de la relation S , il est inutile de maintenir la table MapSparce dont le nœud i contenant la partition S_i est directement obtenu avec l'attribut $S.B$.

La jointure par boucles imbriquées se prête naturellement à la parallélisation, mais le placement initial des données est essentiel. En effet, à défaut, de très nombreuses communications sont nécessaires ce qui peut être extrêmement pénalisant.

3.3.2. Jointure parallèle tri-fusion

La méthode de tri-fusion s'apprête moyennement à la parallélisation, sa phase de tri des relations peut être naturellement parallélisée sur chacun des nœuds disposant d'une partition R ou S , mais un traitement préalable et complexe est indispensable dans la phase de jointure pouvant

conduire à des performances équivalentes à un traitement séquentiel. Par conséquent, cette seconde phase a été souvent réalisée de façon séquentielle.

La parallélisation de cette seconde phase est envisageable selon deux modes opératoires distincts, qui sont :

- 1) Jointure des différentes partitions triées une à une (R_i et S_i) via une combinaison de l'ensemble des partitions S_i et R_i sur chaque nœud détenant l'une ou l'autre des partitions. De ce fait, $m \times n$ jointures sont nécessaires pour m partitions R_i et n partitions S_i .
- 2) Jointure de la petite relation en une seule partition R , réplication de cette partition vers les différents nœuds détenant une partition S_i , puis jointure de chaque partition S_i avec la totalité de la jointure R .

Les deux modes se ramènent à une jointure par boucles imbriquées sans aucun apport de la phase de tri préalable particulière à la jointure par tri-fusion.

Pour tirer profit de la propriété du tri, il est nécessaire de maintenir cette propriété au sein des différentes partitions i traitées par les différents nœuds. Ainsi, les relations S et R sont découpés en k partitions chacune.

Il existe de nombreuses solutions de tri parallèle pouvant être adaptées aux jointures, dès lors qu'elles prennent en compte la distributivité des attributs et assure une répartition équilibrée des données (entre les partitions) [31].

Une étude comparative entre un algorithme de tri parallèle avec découpage exact et un autre utilisant une méthode stochastique [32] a été effectuée, dont l'objectif consiste à atteindre des performances meilleures attendues par un traitement parallèle de la jointure, sur une quantité de données très importantes distribuées. Le second algorithme présente une efficacité beaucoup plus grande dès lors que la distributivité des données est maîtrisée et que l'échantillonnage choisi soit représentatif.

3.3.3. Jointure parallèle par fonction de hachage

La jointure par fonction de hachage s'articule autour du partitionnement des relations. Elle est totalement compatible avec la parallélisation. Dans le cas d'architecture parallèle à mémoire et disques partagés, les algorithmes monoprocesseur fonctionnent au sein d'une architecture parallèle, puisque chaque processeur peut accéder de façon concurrentielle aux différentes partitions [29].

Par contre, dans le cas d'architecture distribuée, la distribution des tuples associés à un nœud est aisément réalisable à l'aide de la fonction de hachage. Cette fonction est utilisée pour découper les relations en partitions et les associer à une table de routage prédéfinie. À l'issue de cette distribution, chaque nœud dispose alors de deux parties S_i et R_i sur lesquelles il peut appliquer la jointure par fonction de hachage.

La principale difficulté réside dans la bonne distribution des relations. En effet, il est difficile de prévoir la taille de chaque partition résultant de la fonction de hachage, puisque cela dépend de la fonction elle-même mais également de la distribution inhérente aux valeurs de l'attribut sur lequel porte la jointure. Une mauvaise distribution des données implique fatalement un dépassement de la capacité mémoire des nœuds de traitement dégradant ainsi fortement l'efficacité de l'algorithme parallèle.

D'un autre côté, un découpage trop fin des relations entraîne une sous-utilisation des capacités des nœuds de traitement.

L'approche générale consiste à découper les relations en un nombre de partitions suffisamment important afin d'éviter un dépassement mémoire et, durant la phase de construction, si des partitions sont jugées trop petites, alors on procède à leur assemblage en une partition assez grande afin d'optimiser l'usage de la mémoire.

Il faut signaler que la bonne ou mauvaise distribution des données ainsi que celle de l'attribut de jointure joue un rôle primordial dans le cas d'environnement monoprocesseur et encore plus dans la cadre d'un environnement parallèle.

3.4. Répartition des données et affectation des tâches

Une bonne distribution des données entre les différents nœuds assure un bon équilibre de charge des traitements. Or, le facteur majeur qui conditionne la possibilité de procéder à une telle répartition est la distribution des données à joindre, mais également de la distribution des valeurs de l'attribut de jointure [33].

En effet, malgré une bonne distribution des données sur un attribut quelconque des relations en jeu, il peut s'avérer qu'un nœud se voit affecté une paire de partitions S_i et R_i telle que la distribution des valeurs de l'attribut engendre un traitement lourd pour un ensemble de nœuds et un traitement quasi nul pour un autre ensemble. En conséquence, un déséquilibre se produit annulant l'efficacité attendue d'un traitement parallèle et en introduisant des goulets d'étranglement.

Des travaux de recherche ont été orientés à ce type de problématique. Une méthode efficace consiste à envisager plusieurs cas de figure qui sont ensuite mis en œuvre durant l'exécution selon une estimation de la distribution des données [34]. D'autres plus formelles s'appuient sur des histogrammes visant à équilibrer la charge selon la fréquence des valeurs de l'attribut de jointure [35, 36]. Cette étape est un élément clé du traitement parallèle d'une jointure entre deux relations.

3.4.1. Découpage des relations

C'est une phase clé du traitement parallèle d'une jointure. Elle a pour but de découper la jointure en un ensemble de jointures sur des sous-ensembles des relations. L'union de ces jointures devant impérativement produire le même résultat qu'une jointure directe sur la totalité des relations initiales sans découpage. Trois combinaisons de découpage des relations sont possibles[37, 38]:

- 1) Découpage total: dans lequel aucune réplique de tuples n'est réalisée. Il consiste à découper les deux relations en partitions dont l'objectif est que tout tuple de l'une ou l'autre relation participe à une et une seule tâche de jointure. Cela implique que les tuples en relation doivent être regroupés au sein des mêmes partitions. Ceci est

réalisable soit par découpage des relations en partitions selon des intervalles de valeurs de l'attribut de jointure, soit par utilisation d'une fonction de hachage. Ce type de découpage est le plus recherché et le plus utilisé quand cela est possible.

- 2) Découpage et réplique: dans ce cas une relation est découpée tandis que l'autre est répliquée. Généralement, c'est la relation la plus volumineuse qui est découpée tandis que la plus petite est répliquée. Ainsi, chaque tâche se voit affectée d'une part une partition de la plus grande relation et d'autre part, la totalité de la plus petite relation. Cette méthode est particulièrement pertinente quand la relation répliquée est suffisamment petite pour tenir en mémoire et dans ce cas, l'usage de la jointure par boucles imbriquée avec blocs est le plus approprié.
- 3) Réplique totale: les deux relations sont répliquées dans le découpage. Les m partitions de la première relation et les n partitions de la seconde sont jointes une à une, donnant lieu à $m*n$ tâches différentes. Cette méthode peu efficace n'est utilisée que dans des situations particulières au sein d'architectures à ressources partagées.

3.4.2. Affectation des tâches

Une fois les relations de jointure découpées, nous disposons d'un ensemble de tâches à affecter aux différents nœuds pour un traitement parallèle. Deux modes d'affectations peuvent être mis en œuvre, à savoir l'allocation statique ou dynamique[33, 39]. Chacune de ces méthodes est mise en œuvre dans des situations appropriées :

- 1) L'allocation statique : exige une gestion minimale de l'affectation puisque les tâches sont affectées une fois pour toute aux nœuds élus pour le traitement. Cette approche peut s'avérer efficace si tous les nœuds ont la même capacité de traitement et les données sont équitablement distribuées. Dans le cas contraire, les temps de calcul globaux peuvent s'avérer incertains voire désastreux [33].
- 2) L'allocation dynamique: Elle est basée sur un découpage adapté des relations de jointure et en conséquence un nombre suffisamment important de tâches à

affecter. Il est possible de piloter l'allocation en veillant à ce que les temps d'exécution soient approximativement les mêmes à travers tous les nœuds qui participent au traitement. L'une des stratégies les plus utilisées afin de maintenir cet équilibre de charge est celle dite *best-fit-like* qui consiste à : trier les tâches de manière décroissante selon un critère donné, puis selon un algorithme glouton, affecter la tâche suivante au nœud qui est sur le point de terminer sa tâche courante. Cette méthode se décline selon deux axes qui sont les mesures utilisées et la stratégie d'allocation choisie. Une étude dans un contexte d'architecture de type *Shared-nothing* a fait usage d'une telle méthode, mais en ramenant cette architecture à une architecture de type mémoire virtuelle partagée [39].

3.5. Synthèse

Il est difficile de désigner, parmi l'ensemble des algorithmes exposés, un comme étant le plus performant que tous les autres. L'ensemble des études partent d'hypothèse restrictive sur la distribution des données. Les algorithmes exposés trouvent leur intérêt dans des configurations particulières, telles que la bonne distribution des attributs de jointure, la différence de la taille des deux relations ou encore la présence d'index.

La jointure basée sur une fonction de hachage (H dans la figure ci-dessous) semble selon les études réalisées assez généraliste et globalement efficace. Toutefois, celle-ci est fortement sensible à la distribution de l'attribut de jointure ainsi qu'à la taille des relations versus la quantité de mémoire disponible. La jointure par boucles imbriquées (NL) est quant à elle complètement insensible à la distribution de l'attribut de jointure et gagne en efficacité dans un environnement parallèle. Enfin, la jointure par tri-fusion (SM) n'est réellement efficace que lorsque les relations de jointure sont déjà triées, et, dans ce cas cette méthode est de loin la plus performante, même dans une configuration monoprocesseur [40].

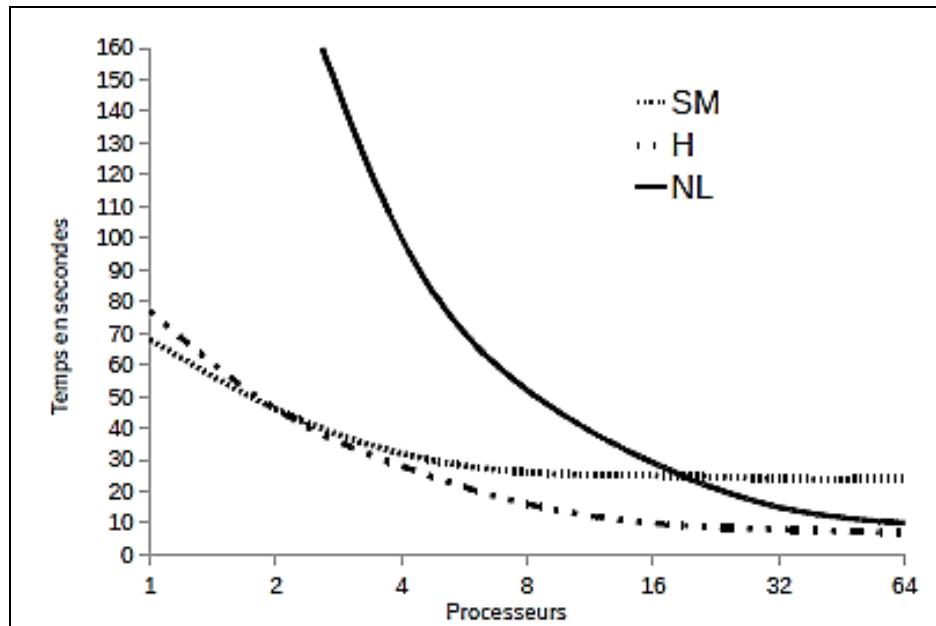


Figure 3.9 : Performances des trois(03) algorithmes [40].

En outre, une bonne répartition des données à joindre garantis une efficacité optimale des algorithmes de jointure parallèle.

Nous exposons dans le chapitre suivant les approches de jointure avec le Framework MapReduce.

Chapitre 4 : La jointure avec MapReduce

4.1. Introduction

La jointure est une construction relationnelle que nous utilisons pour combiner un ensemble de relations (concept essentiel dans le cas des bases de données). En MapReduce, les jointures sont applicables dans des situations où nous avons deux ou plusieurs ensembles de données que l'on souhaite combiner. Un exemple serait la combinaison des informations sur les utilisateurs (extraites d'une la base de données OLTP) avec les fichiers journaux qui contiennent des informations relatives aux activités des utilisateurs. Voici quelques scénarios où il serait utile de combiner ces ensembles de données:

- Agrégations de données basées sur des données démographiques des utilisateurs (telles que les différences dans les habitudes de l'utilisateur entre les adolescents et les utilisateurs dans les trentaines)
- Pour envoyer un email aux utilisateurs qui n'ont pas utilisé le site pour un nombre prescrit de jours
- Une boucle d'évaluation qui examine les habitudes de navigation de l'utilisateur, permettant au système de recommander le site inexploré jusqu'à une date donnée afin de les offrir à l'utilisateur

Tous ces scénarios ont besoin de joindre des ensembles de données, et les deux types les plus communs de jointures sont la jointure interne et la jointure externe.

- La jointure interne « inner join » compare toutes les lignes dans les relations L et R, pour produire un résultat si un prédicat de jointure est satisfait.

- En revanche, Les jointures externes « outer join » sont plus "complètes" que les jointures internes. En plus de sélectionner l'ensemble des lignes des 2 relations respectant le prédicat de jointure comme le font les jointures internes, elles sélectionnent les données d'une des tables qui ne respectent pas le prédicat de jointure.

La Figure 4.1 montre les différents types de jointures.

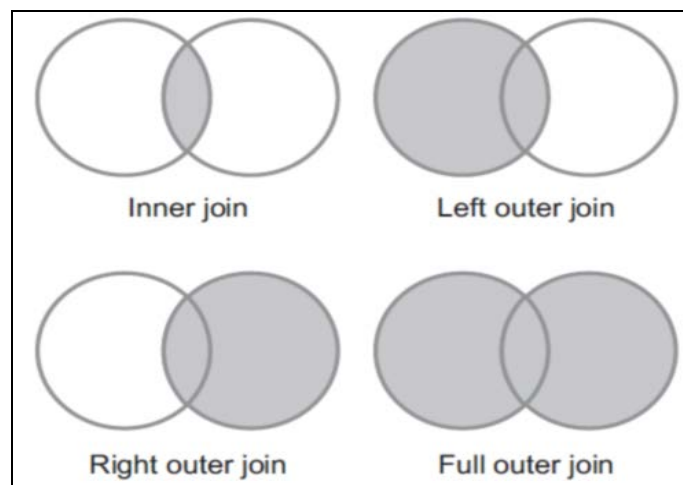


Figure 4.1 : Les types de jointures

Dans ce chapitre, nous allons examiner trois stratégies de jointure avec MapReduce qui soutiennent les deux types les plus communs de jointures (intérieure et extérieure). Les trois stratégies exécutent la jointure soit dans la phase MAP ou dans la phase REDUCE en tirant profit de l'avantage tri-fusion de l'architecture MapReduce.

- Jointure répartie : pour les situations où nous joignons ensemble deux ou plusieurs ensembles de données volumineux ;
- Jointure par diffusion : applicable dans les cas où l'un des ensembles de données est assez petit (pour le sauvegarder en mémoire cache) ;

- Semi-jointure : où un ensemble de données est initialement trop grand pour le tenir en mémoire, mais qui peut, après un certain filtrage, être réduit à une taille pouvant être sauvegardé en mémoire.

Après la présentation de ces stratégies de jointure, nous donnons un arbre de décision, afin que nous puissions choisir la meilleure stratégie de jointure pour une situation donnée.

4.2. La jointure répartie

Généralement, La jointure répartie [23] est la stratégie de jointure la plus utilisée avec le Framework MapReduce. Dans cette stratégie de jointure, L et R sont dynamiquement divisés sur la clé de jointure et les paires correspondantes de partitions sont jointes.

4.2.1. La jointure standard répartie

La jointure répartie standard peut être mise en œuvre dans un seul job MapReduce.

Chaque Mapper fonctionne sur une fraction de R ou L. Pour identifier la table de l'enregistrement en entrée, chaque Mapper tag l'enregistrement(le record) avec sa table d'origine et extrait la valeur de la clé de jointure de l'enregistrement traité. Ensuite, il le délivre comme étant une paire <clé, valeur>.

Les sorties quant à elles sont partitionnées, triées et regroupées par le Framework MapReduce. Pour chaque valeur de la clé de jointure, tous les enregistrements correspondants sont regroupés et finalement envoyés à un Reducer.

Enfin, le *Reducer* ; pour chaque clé de jointure ; sépare et mémorise les enregistrements en deux ensembles de données en fonction des tags de la table d'entrée L ou R, puis effectue un produit cartésien entre les enregistrements de ces ensembles pour réaliser l'opération de la jointure.

La Figure 4.2 montre la jointure standard répartie à un niveau élevé en utilisant le paradigme MapReduce.

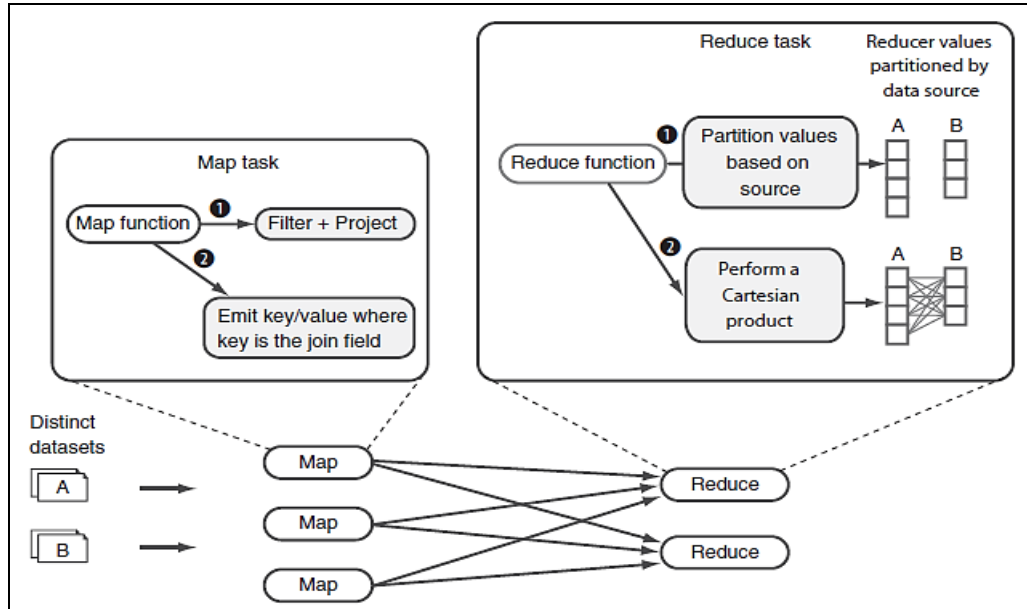


Figure 4.2 : les phases de la jointure standard répartie [23]

Le Pseudo-Algorithm (Figure 4.3) ci-dessous représente cette stratégie de jointure.

```

Map ( $K$ : null,  $V$ : a record from a split of either  $R$  or  $L$ )
   $join\_key \leftarrow$  extract the join column from  $V$ 
   $tagged\_record \leftarrow$  add a tag of either  $R$  or  $L$  to  $V$ 
   $emit(join\_key, tagged\_record)$ 

Reduce ( $K'$ : a join key,
   $LIST\_V'$ : records from  $R$  and  $L$  with join key  $K'$ )
  create buffers  $B_R$  and  $B_L$  for  $R$  and  $L$ , respectively
  for each record  $t$  in  $LIST\_V'$  do
    append  $t$  to one of the buffers according to its tag
  for each pair of records  $(r, l)$  in  $B_R \times B_L$  do
     $emit(null, new\_record(r, l))$ 

```

Figure 4.3 : Pseudo-Algorithm de la jointure standard répartie [41].

Un problème potentiel avec la répartition standard de la jointure réside dans le fait que tous les enregistrements pour les valeurs de la clé de jointure de L et R doivent être mis en mémoire tampon.

Quand la cardinalité (le nombre de valeurs uniques que contient un attribut) principale est petite ou quand les données sont à forte asymétrie, tous les enregistrements d'une valeur correspondante à une clé de jointure ne peuvent être mémorisés.

Cet algorithme souffre, comme la jointure repartie dans Pig [42] Hive [43] et Jaql [44], du même problème car tous les enregistrements de la plus grande table L doivent être mémorisés.

En outre, avec la jointure standard répartie et avec MapReduce en général, il est indispensable de réduire la quantité de données échangées entre les phases MAP et REDUCE, car le tri et le transfert de données entre les deux phases sur un réseau est très coûteux.

4.2.2. La jointure répartie standard améliorée:

Pour résoudre le problème de mise en mémoire tampon de la répartition standard de la jointure, une amélioration de cette stratégie a été proposée par Blanas [41] avec quelques changements cruciaux.

Premièrement, dans la fonction MAP, la clé de sortie est modifiée à une composition de la valeur de la clé de jointure et la table cible. Les tables cibles sont générées d'une manière qui assure que les enregistrements de R seront triés avant ceux de L par rapport à la clé jointure.

En second lieu, une fonction de partition est adaptée de telle sorte que le code de hachage soit calculé seulement à partir de la partie clé de jointure de la clé composite (sans tenir compte de la table cible). De cette façon les enregistrements ayant la même valeur de la clé de jointure sont toujours affectés au même Reducer.

Dans le Reducer, La fonction de regroupement est également personnalisée afin que les enregistrements soient regroupés par rapport à la clé de jointure.

Finalement, comme les enregistrements de la plus petite table R sont garantis d'être en avance sur ceux de L pour la même valeur de la clé de jointure, seuls les enregistrements de R sont mémorisés et les enregistrements de L sont diffusés pour générer le résultat de jointure.

La jointure répartie standard arrange le problème de mémoire tampon de la version standard. Toutefois, les deux versions défavorisent les performances (temps de réponse pour réaliser la jointure). En plus, les ensembles de données L et R doivent être triés et envoyés sur le réseau pendant la phase shuffle de l'architecture MapReduce.

La Figure 4.4 montre la jointure standard répartie à un niveau élevé en utilisant le paradigme MapReduce.

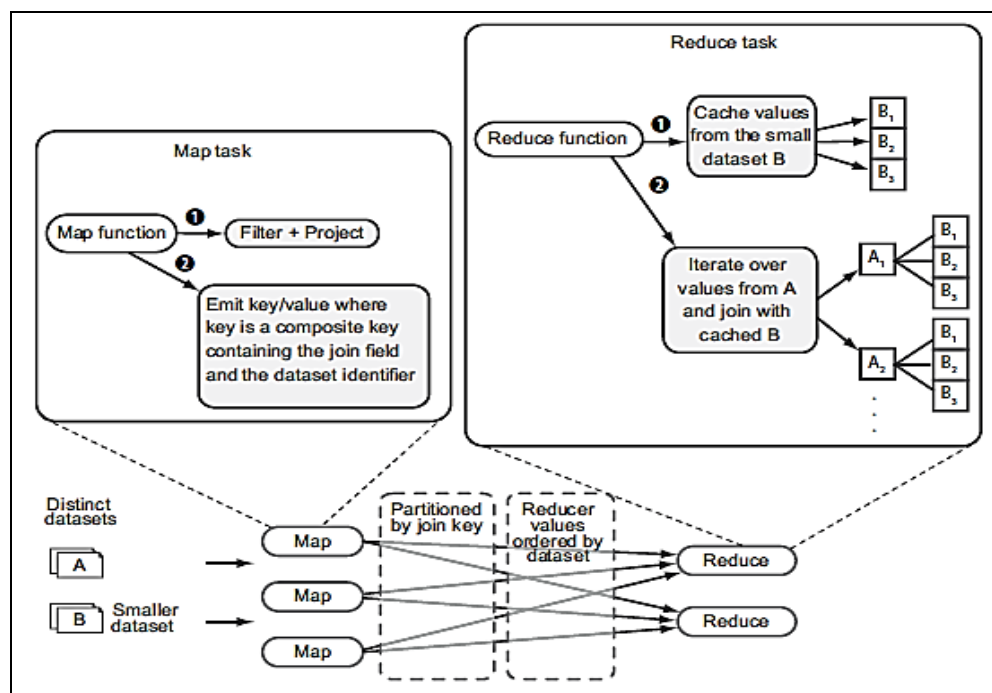


Figure 4.4 : Les phases de la jointure standard répartie améliorée [23].

Voici la conception selon le paradigme MapReduce de la jointure standard répartie améliorée (Figure 4.5).

```

Map (K: null, V: a record from a split of either R or L)
  join_key ← extract the join column from V
  tagged_record ← add a tag of either R or L to V
  composite_key ← (join_key, tag)
  emit (composite_key, tagged_record)

Partition (K: input key)
  hashcode ← hash_func(K.join_key)
  return hashcode mod #reducers

Reduce (K': a composite key with the join key and the tag
  LIST_V': records for K', first from R, then L)
  create a buffer BR for R
  for each R record r in LIST_V' do
    store r in BR
  for each L record l in LIST_V' do
    for each record r in BR do
      emit (null, new_record(r, l))

```

Figure 4.5 : Pseudo-Algorithme de la jointure standard répartie améliorée [41].

4.2.3. Prétraitement pour la jointure répartie:

La surcharge dans la jointure répartie peut être diminuée si les deux tables *L* et *R* ont été déjà partitionnées suivant la clé de jointure, avant d'entamer l'opération de jointure. Ceci peut être accompli par un pré-partitionnement des enregistrements de *L* et des enregistrements de *R* lorsqu'ils sont chargés dans le DFS (Distributed File System) ; qui fournit un moyen simple pour accéder à des données réparties et distribuées sur un réseau. Ensuite, pour exécuter la requête de jointure, les partitions correspondantes de *L* et *R* peuvent être directement jointes.

Contrairement à un SGBDR parallèle, il n'est pas possible de garantir que les partitions correspondantes de *L* et *R* soient situés sur le même nœud de traitement. C'est parce que le DFS prend des décisions indépendantes relatives à l'endroit de stockage d'un

bloc de données particulier. Par conséquent, pour accomplir l'opération de jointure, la stratégie de jointure directe pourra l'effectuer.

L'algorithme de jointure directe [45] est implémenté uniquement sur des Mappers du Framework MapReduce (Pas de traitement pour le Reducer). Chaque Mapper traite une fraction de L (L_i).

Durant la phase d'initialisation, une fraction de R (R_i) est extraite du DFS, si elle n'existe pas déjà dans le système de stockage local, et une table de hachage de la mémoire principale est construite sur cette dernière. Par la suite, la fonction MAP inspecte les enregistrements d'une fraction L_i et fouille la table de hachage pour réaliser la jointure.

A noter que le nombre de partitions peut être choisi de telle sorte que chaque fraction R_i corresponde à la taille de la mémoire du nœud de traitement.

Le Pseudo-Algorithme (Figure 4.6) suivant représente la jointure directe :

```
Init ()
  if  $R_i$  not exist in local storage then
    remotely retrieve  $R_i$  and store locally
   $H_{R_i} \leftarrow$  build a hash table from  $R_i$ 

Map ( $K$ : null,  $V$ : a record from a split of  $L_i$ )
  probe  $H_{R_i}$  with the join column extracted from  $V$ 
  for each match  $r$  from  $H_{R_i}$  do
    emit (null, new_record( $r, V$ ))
```

Figure 4.6 : Pseudo-Algorithme de la jointure directe [45].

4.3. Jointure par diffusion

4.3.1. Principe de fonctionnement

Dans la plupart des applications, la table de référence R est beaucoup plus petite que la table du journal L , ($|R| \ll |L|$). Au lieu de déplacer les deux tables R et L sur le réseau comme dans le cas de la jointure répartie, nous ne diffusons que la petite table R . Car ceci évite le tri des deux tables et surtout la surcharge du réseau lors du déplacement de la table de la plus grande taille, en occurrence de L [41] .

La jointure par diffusion est effectuée dans une seule tâche MAP. Sur chaque nœud, tous les enregistrements R sont récupérés du DFS pour éviter l'envoi de L sur le réseau. Chaque tâche de MAP utilise une table de hachage stockée en mémoire principale pour joindre une fraction de L (L_i) avec R .

La jointure par diffusion vérifie dans sa fonction d'initialisation `init()` de chaque tâche de MAP, si R est déjà enregistrée dans le système de fichiers local. Dans le cas contraire, elle récupère R de la DFS et la partitionne(R) suivant la clé de jointure, et stocke ces partitions dans le système de fichiers local afin que toutes les partitions de R soient chargées en mémoire lors de la jointure.

La jointure par diffusion construit dynamiquement de la table de hachage sur L ou R . Le plus petit des R et la partition des L est choisi pour construire cette table de hachage, qui est supposée être toujours sauvegardée en mémoire. Si R est inférieur à la fraction de L (L_i), la fonction `init()` est utilisée pour charger l'ensemble des partitions de R dans la mémoire pour construire la table de hachage. Ensuite, la fonction de MAP extrait la clé de jointure de chaque enregistrement de L et l'utilise pour le comparer avec la table de hachage pour enfin générer la jointure en sortie.

D'autre part, si la partition de L est plus petite que R , la jointure ne se fait pas dans la fonction MAP. Au lieu de cela, elle partitionne L de la même manière que la partition de la R . Puis dans la fonction `Close()`, les partitions correspondantes de R et L sont

jointes. Le chargement des partitions de R est évité si la partition correspondante de L n'a pas d'enregistrements.

La jointure par diffusion est utile lorsque le domaine des valeurs de la clé de jointure est large.

La Figure 4.7 montre les phases de la jointure par diffusion avec le paradigme MapReduce à un niveau élevé.

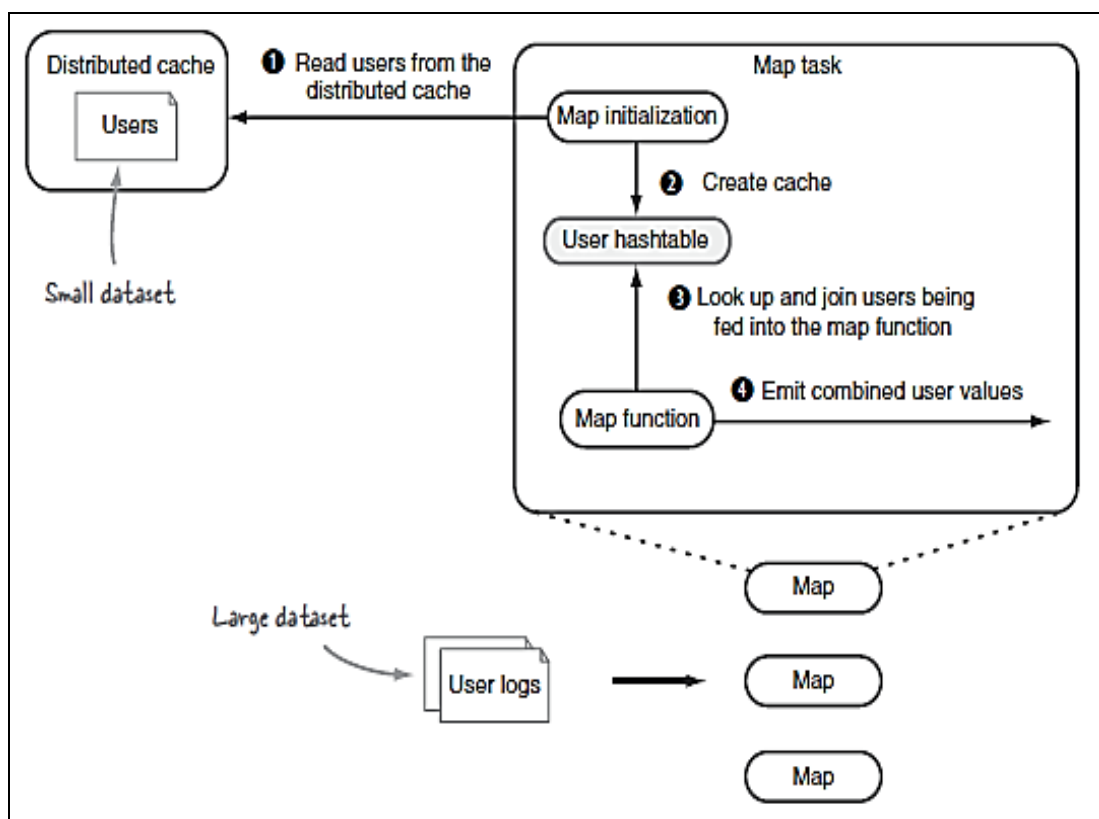


Figure 4.7 : Les phases de la jointure par diffusion[23].

Le Pseudo-Algorithmme suivant (Figure 4.8) présente les étapes de la jointure par diffusion.

<pre> Init () if R not exist in local storage then remotely retrieve R partition R into p chunks $R_1..R_p$ save $R_1..R_p$ to local storage if $R < \text{a split of } L$ then $H_R \leftarrow \text{build a hash table from } R_1..R_p$ </pre>
<pre> Map (K: null, V: a record from an L split) if H_R exist then probe H_R with the join column extracted from V for each match r from H_R do emit (null, new_record(r, V)) </pre>

Figure 4.8 : Pseudo-Algorithme de la jointure par diffusion[41].

Pig [42] dispose également d'une mise en œuvre de la jointure par diffusion. Dans les tâches MAP de la jointure par diffusion, les partitions de R peuvent être rechargées plusieurs fois, car chaque tâche du MAPPER fonctionne comme un processus séparé. Par conséquent, ce rechargement répété dégrade les performances de cette approche.

4.3.2. Prétraitement pour la jointure par diffusion:

Bien qu'il n'y ait aucun moyen de contrôler précisément l'emplacement physique des réplicas dans le DFS, en augmentant le facteur de réplication pour R , nous pouvons veiller à ce que la plupart des nœuds du cluster aient une copie locale de R permettant ainsi l'amélioration de la jointure par diffusion tout en évitant la récupération de R du DFS dans sa fonction `Init()`.

4.4. La semi-jointure

4.4.1. Principe de fonctionnement

Souvent, lorsque R est grand, plusieurs enregistrements de R ne peuvent pas être effectivement référencés par les enregistrements dans la table L . Prenons l'exemple de Facebook, sa table d'utilisateurs contient des centaines de millions d'enregistrements.

Cependant, une heure d'historique contenant un fichiers de journalisation, en terme de données, devrait contenir probablement les activités de seulement quelques millions d'utilisateurs et la majorité des utilisateurs ne sont pas présents dans ce journal [41] . La semi-jointure réalise la jointure dans ce contexte.

La semi-jointure est mise en œuvre en trois phases, chacune correspondant à une tâche de MapReduce séparée.

Phase 1:

La première phase de semi-jointure est gérée dans un job MapReduce complet. Dans la fonction MAP, une table de hachage, sauvegardée en mémoire principale, est utilisée pour déterminer l'ensemble des clés de jointure uniques dans une partition de L. Cette table de hachage est utilisée dans la mesure où MapReduce ne supporte pas directement l'agrégation basée sur le hachage.

La fonction MAP, en sortie, envoie seulement les clés uniques permettant ainsi la réduction de la quantité de données qui doivent être triées. La tâche REDUCE est utilisée pour consolider toutes les clés uniques dans un fichier unique « L.uk », caractérisé par sa petite taille afin de le tenir en mémoire centrale.

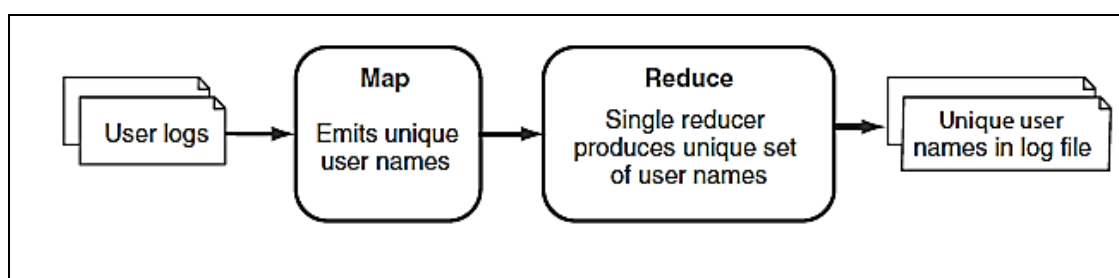


Figure 4.9 : Première phase de la semi-jointure[23].

Phase 2:

La deuxième phase de la semi-jointure est similaire à la jointure par diffusion et est réalisée par un job MapReduce contenant seulement la tâche MAP. La fonction `init()`

charge le fichier « L.uk » dans la table de hachage en mémoire principale. Puis la fonction MAP parcourt chaque enregistrement de R et l'envoie si sa clé de jointure se trouve dans le fichier « L.uk ». La sortie de cette phase est une liste de fichiers R_i pour chaque partition de R.

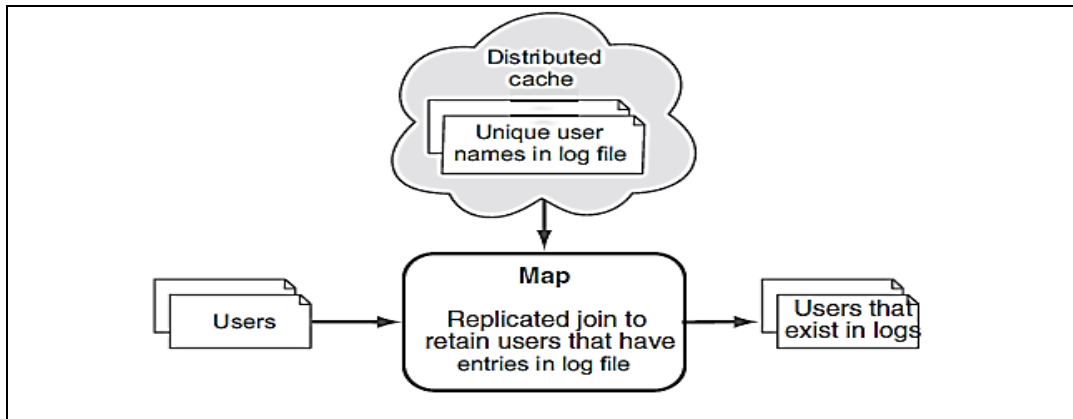


Figure 4.10 : Seconde phase de la semi-jointure [23].

Phase 3:

Enfin, dans la troisième phase de la semi-jointure, tous les fichiers R_i sont joints avec L, en utilisant la jointure par diffusion décrite précédemment.

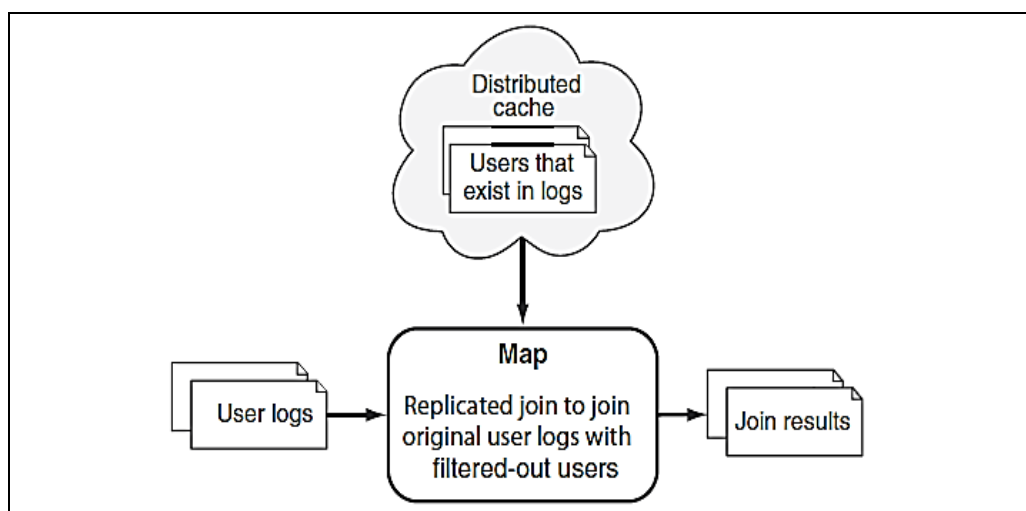


Figure 4.11 : Troisième phase de la semi-jointure [23].

Ce Pseudo-Algorithme résume les phases de la semi-jointure(Figure 4.12).

```

Phase 1: Extract unique join keys in  $L$  to a single file  $L.uk$ 
  Map ( $K$ : null,  $V$ : a record from an  $L$  split)
     $join\_key \leftarrow$  extract the join column from  $V$ 
    if  $join\_key$  not in  $unique\_key\_table$  then
      add  $join\_key$  to  $unique\_key\_table$ 
      emit ( $join\_key$ , null)

  Reduce ( $K'$ : a unique join key from table  $L$ ,
     $LIST\_V'$ : a list of null)
    emit ( $K'$ , null)

Phase 2: Use  $L.uk$  to filter referenced  $R$  records;
  generate a file  $R_i$  for each  $R$  split
  Init ()
     $ref\_keys \leftarrow$  load  $L.uk$  from phase 1 to a hash table

  Map ( $K$ : null,  $V$ : a record from an  $R$  split)
     $join\_col \leftarrow$  extract join column from  $V$ 
    if  $join\_col$  in  $ref\_keys$  then
      emit (null,  $V$ )

Phase 3: Broadcast all  $R_i$  to each  $L$  split for the final join

```

Figure 4.12 : Pseudo-Algorithme de la semi-jointure [41].

4.4.2. Prétraitement pour la semi-jointure:

Il est possible de déplacer les deux premières phases de la semi-jointure à une étape de prétraitement et seulement exécuter la dernière phase au moment du lancement de l'opération de jointure. Ce prétraitement peut même être fait progressivement. A titre d'exemple, pour les nouveaux enregistrements de L générés, les clés uniques de jointure de ces enregistrements peuvent être accumulés et joints avec R pour déterminer le sous-ensemble de R qui se joint à L .

Dans cette technique, nous avons examiné la façon d'utiliser la semi-jointure pour combiner deux ensembles de données. La construction de semi-jointure implique plusieurs étapes par rapport aux autres approches de jointure, mais c'est un moyen puissant lorsque nous travaillons avec de grands ensembles de données (où l'un des ensembles de données doit être réduit à une taille qui correspond à la mémoire).

Bien que la semi-jointure évite d'envoyer les enregistrements de R sur le réseau qui ne se joindront pas à L, l'analyse supplémentaire de L engendre un coût considérable de traitement qui influence les performances de cette approche.

4.5. Synthèse

Chacune des stratégies de jointure que nous avons détaillées possède des forces et des faiblesses, alors comment peut-on déterminer qui conviendra mieux pour les données que nous traitons ?

La Figure 4.13 résume les compromis de l'étude des stratégies de jointure dans un arbre de décision.

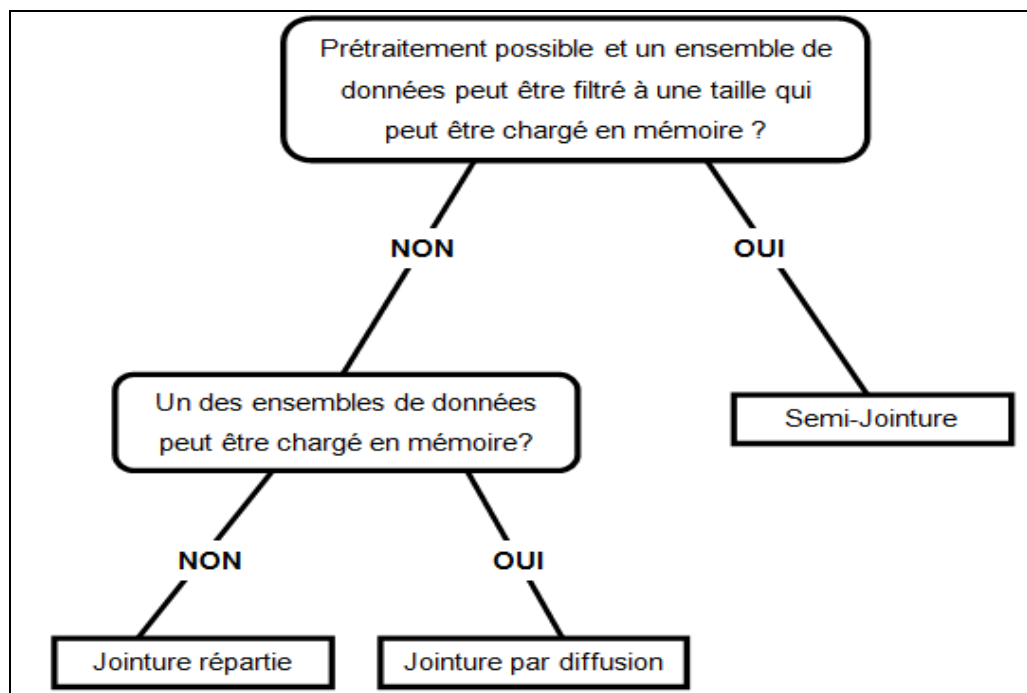


Figure 4.13 : Arbre de décision relatif aux stratégies de jointure.

Basé sur les statistiques, telles que la taille des données relatives et la fraction de la clé de jointure référencée (ces statistiques pourraient être recueillies sur un échantillon des entrées). Cet arbre de décision tente de déterminer quelle est la bonne stratégie de jointure pour une circonstance donnée. Si les données ne sont pas prétraitées, la meilleure stratégie

de jointure dépend de la taille des données transférées via le réseau. Si le coût de la diffusion dans le réseau de la table R à chaque nœud est moins cher que le transfert de R et L, alors la jointure par diffusion doit être utilisée.

Lorsque le prétraitement est permis, la semi-jointure, avec suffisamment de partitions est le meilleur choix. Elle offre une plus grande flexibilité puisque ses étapes de prétraitement sont insensibles à la façon dont la table de journal est structurée, et donc adaptée à un certain nombre de tables de référence [46].

En outre, les étapes de prétraitement des deux algorithmes : jointure répartie et jointure par diffusion sont moins coûteuses car il n'y a pas de réarrangement des données du journal.

- Si l'un des ensembles de données est assez petit pour tenir dans la mémoire d'un MAPPER, la jointure par diffusion est suffisante.
- Si le volume des deux ensembles de données est important, et un ensemble de données peut être considérablement réduit par des éléments de filtration, la mise en œuvre de semi-jointure est adéquate.
- Si nous ne pouvons pas prétraiter les deux ensembles de données et les tailles de ces ensembles sont trop grandes par rapport à la mémoire cache d'un nœud de traitement, ceci signifie l'obligation d'effectuer la jointure dans le REDUCER, alors la jointure répartie doit être utilisée.

Chapitre 5 :

Etude des performances de la jointure répartie

5.1. Introduction

Dans ce chapitre, nous présentons les résultats de l'évaluation expérimentale de la jointure répartie avec MapReduce, détaillée dans la section 4.2.

En premier lieu nous décrivons l'environnement d'évaluation et les conditions expérimentales (section 5.2.), où nous donnons la configuration matérielle et logicielle de l'environnement d'évaluation. Par la suite, Pour tirer profit du traitement parallèle que fournit Hadoop, un job de MapReduce est exécuté sur un cluster de machines (section 5.3.), où nous calculons la température globale maximale enregistrée, par des stations climatologiques, chaque année. Dans la section 5.4. , nous nous intéressons au déséquilibre des valeurs de l'attribut de jointure de la jointure répartie, auquel nous effectuerons des tests sur deux types de distributions : homogène et hétérogène.

5.2. Environnement d'évaluation

Dans cette section nous détaillons la configuration matérielle et logicielle de l'environnement d'évaluation.

5.2.1. Configuration matérielle

Notre environnement d'évaluation est composé d'un cluster de calcul et d'une machine d'administration.

a.Cluster de calcul

Le cluster Fujitsu Primergy utilisé est composé de deux nœuds et d'une baie de stockage. Les nœuds Fujitsu RX300S6 ont les caractéristiques suivantes:

- Processeur Intel Xeon X5675, 12 processeurs, 24 Threads, 3.06 Ghz ,12 Mo mémoire cache.
- 64 Go DDR3.
- Disque dur SAS 2x146Go.

La baie de stockage Fujitsu Ethernus DX80 est caractérisée par:

- Deux (02) ports fibre Channel Channel 8 Go.
- Disque dur SAS 6 x 600 Go.

Pour effectuer les différents tests d'évaluation, le cluster de calcul présenté précédemment sera utilisé pour le déploiement d'une architecture virtualisée.

b. Machine d'administration

La machine d'administration est utilisée pour manipuler à distance, via le réseau, le cluster de calcul. Elle se caractérise par :

- Une unité centrale de traitement (CPU) Pentium (R) D 3,20 Ghz ;
- Une mémoire vive (RAM) de deux (02) Go ;
- Un Système Microsoft Windows XP Professionnel.

5.2.2. Configuration logicielle

Un ensemble d'outils a été utilisé notamment :

5.2.2.1. VirtualBox

VirtualBox est un outil libre de virtualisation publié par Oracle, disponible sur les systèmes d'exploitation Linux (32 et 64 bits), Mac OS X, Solaris et Windows. Il être lancé n'importe où,

depuis de petits systèmes embarqués jusqu'aux machines de bureaux en passant par des déploiements en centre de traitement de données (Datacenter) ou même des environnements en nuages (Cloud).

5.2.2.2. Citrix XenServer

C'est l'une des premières plateformes de virtualisation open source du secteur pour la gestion des infrastructures virtuelles Cloud. XenServers s'installe pour virtualiser les charges les plus contraignantes et automatiser les processus de gestion ; améliorant ainsi la souplesse et la mobilité de l'informatique et réduire les coûts. XenServer est conçue pour optimiser les centres de données (Datacenter). Nous avons installé XenServer sur le cluster pour le déploiement d'une architecture virtualisée.

5.2.2.3. Putty

C'est une interface permettant aux utilisateurs de se connecter à distance au XenServer depuis n'importe quel PC Windows. Elle permet l'installation complète et la configuration des machines virtuelles, Elle permet aussi la configuration du réseau. Nous avons installé Citrix XenCenter pour pouvoir gérer Le cluster à distance.

5.2.2.4. Installation de la plateforme Hadoop

Un cluster Hadoop est typiquement organisé en configuration maître/esclaves, il fait cohabiter les agents (ou services) suivants :

- Un NameNode (exécuté sur la machine maître) et plusieurs DataNode (exécutés sur les machines esclaves). Dans le cluster on a utilisé une machine maître et 20 machines esclaves.
- Un Ressourcemanager (exécuté sur la machine maître) et plusieurs Nodemanagers (exécutés sur les machines esclaves).

Dans le cluster utilisé nous allons d'abord préparer les nœuds en installant et en configurant les composants du CDH (Cloudera's Distribution Including Apache Hadoop) en

mode distribué (Fully-distributed mode). L'architecture du cluster est composée d'un Nœud maître nommé Master qui héberge le NameNode de la plateforme et de plusieurs nœuds esclaves nommés DataNode qui hébergent les DataNode de la plateforme Hadoop. La configuration des composants Hadoop se déroule comme suit :

1) Installation des composants Hadoop.

- Pour la machine Maître : Installation du NameNode et du ResourceManager ;
- Pour les machines esclaves : Installation du DataNode et du NodeManager ;
- L'agent JobHistoryServer : qui stocke l'historique des Jobs MapReduce exécutés sur le cluster ;
- L'agent SecondaryNameNode sur une autre machine : qui est une copie du NameNode, répliquée périodiquement.

2) Configuration de la plateforme et des agents (Configuration des fichiers `hadoop-env.sh`, `core-site.xml`, `hdfs-site.xml`, `mapred-site.xml` et `yarn-site.xml`) ;

3) Ajout des esclaves dans le fichier « slave ».

5.3. Exemple d'exécution d'un job MapReduce

Pour tirer profit du traitement parallèle que fournit Hadoop, nous exprimons notre requête comme un job de MapReduce, qui sera exécuté sur un cluster de machines. Dans cet exemple, nous voulons calculer la température globale maximale enregistrée, par des stations climatologiques, chaque année.

Les données que nous allons utiliser sont issues du centre national américain de données climatologiques (NCDC, <http://www.ncdc.noaa.gov/>). Elles sont stockées dans un format ASCII orienté ligne, dans lequel chaque ligne est un enregistrement. Le format prend en charge un ensemble complet d'éléments météorologiques, dont beaucoup sont en option ou avec des longueurs de données variables. Par souci de simplicité, nous allons mettre l'accent sur les éléments de base, tels que la température, qui sont toujours présents et sont de largeur fixe.

La fonction MAP est simple. Nous retirons l'année et la température de l'air, car ce sont les seules informations qui nous intéressent. La fonction MAP est juste une phase de préparation des données. Elle prépare les données de manière à ce que la fonction Reducer puisse faire sa tâche: trouver la température maximale pour chaque année.

Pour visualiser la façon dont fonctionne MAP, la Figure 5.1 montre les lignes d'échantillons de données d'entrée :

```
0067011990999991950051507004...9999999N9+00001+9999999999...
0043011990999991950051512004...9999999N9+00221+9999999999...
0043011990999991950051518004...9999999N9-00111+9999999999...
0043012650999991949032412004...0500001N9+01111+9999999999...
0043012650999991949032418004...0500001N9+00781+9999999999...
```

Figure 5.1 : Fichiers des entrées de la fonction MAP

Ces lignes sont présentées à la fonction MAP sous la forme de paires clé-valeur:

```
(0, 0067011990999991950051507004...9999999N9+00001+9999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+9999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+9999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+9999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+9999999999...)
```

Figure 5.2 : Les lignes de la fonction MAP en forme de paires clé-valeur

La fonction MAP extrait l'année et la température de l'air et les émet à sa sortie.

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

Figure 5.3 : Les Sorties de la fonction MAP

Les sorties de la fonction MAP sont traitées par le Framework MapReduce avant d'être envoyée à la fonction Reducer. Ce genre de traitement consiste à grouper les paires clé-valeur de clé. Ainsi, en reprenant l'exemple, la fonction Reducer voit les entrées suivantes:

(1949, [111, 78])
(1950, [0, 22, -11])

Figure 5.4 : Les entrées de la fonction Reducer

Chaque année apparaît avec une liste de tous les relevés de température de l'air. Toutes les fonctions Reducer parcourent la liste et cherchent la température maximale.

(1949, 111)
(1950, 22)

Figure 5.5 : La température maximale par année

Un ensemble de tests a été effectué pour montrer le passage à l'échelle, sur le cluster de calcul. Plusieurs configurations ont été testées, où nous varions le nombre de nœuds esclaves. Le Tableau ci-dessous résume les résultats obtenus

Configuration du Cluster	C1	C2	C3	C4	C5
Nombre de Nœuds Esclaves	3	5	10	15	20
Temps d'exécution (ms)	19137	18197	17393	17047	16087
Ecart entre 2 configurations consécutives (ms)	0	940	804	346	960
Ecart entre la configuration Ci la configuration C5 (ms)	3050	2110	1306	960	0

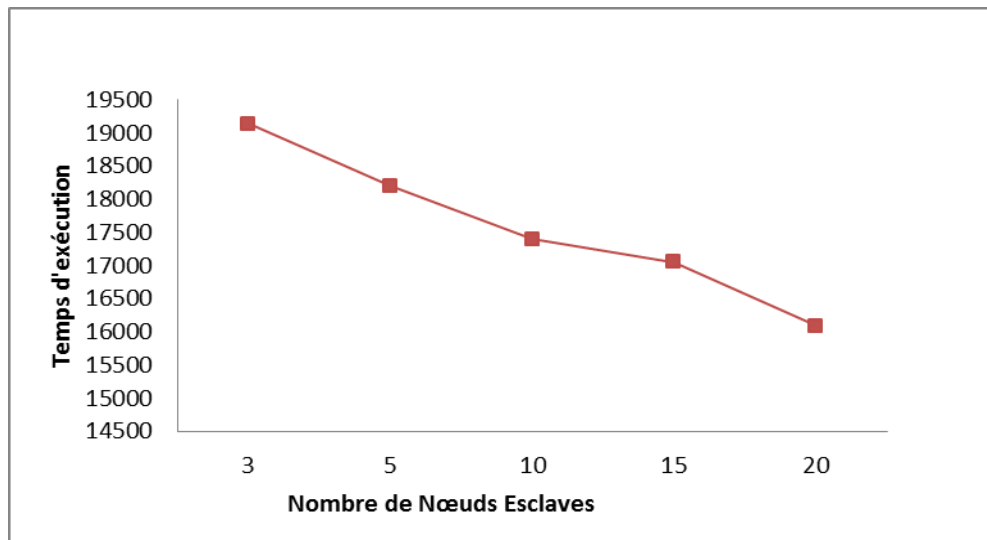


Figure 5.6: Temps d'exécution en fonction des configurations du cluster

Le graphe ci-dessous montre la diminution du temps d'exécution quand nous augmentons le nombre de nœuds dans le cluster de calcul. Un gain de plus de 3 secondes entre le régime plein du cluster (20 nœuds) par rapport au cas où nous disposons de 05 nœuds.

5.4. Evaluation de la jointure Répartie

Dans cette section nous allons évaluer la jointure répartie, détaillée dans la section 4.2. , où nous nous intéressons au déséquilibre des valeurs de l'attribut de jointure auquel nous voulons étudier la limite. Pour nos tests, nous les effectuerons sur deux types de distributions : homogène et hétérogène.

Les données que nous allons utiliser sont issues du centre national américain de données climatologiques (NCDC, <http://www.ncdc.noaa.gov/>), associées à un travail de préparation de données pour avoir ces deux types de distributions.

Nous réalisons la jointure entre deux ensembles de données(Dataset) dont la clé de jointure est l'identifiant de la station climatologique :

- 1) le premier dataset : contient les informations relatives aux stations climatologiques.
Ci-dessous un extrait de ce dataset :

013000 99999 GULLFAKS C	NO NO	+61200 +002267 +00780
013005 99999 STATFJORD OIL RIG &	NO NO	+61217 +001833 +00150
013040 99999 YTTEROYANE FYR	NO NO	+61567 +004683 +00250
013060 99999 HELLISOY FYR	NO NO	+60750 +004717 +00230
013070 99999 ALGER	NO NO	+60750 +004717 +00200

Figure 5.7 : Les informations relatives aux stations climatologiques

- 2) le second dataset : contient les différentes mesures effectuées, à des dates différentes, par les stations climatologiques.

0029725043999991901122506004+62167+030650FM12+010299999V0203201N002119999999N0000001N9-02391+99999101101ADDGF1089919999999999999999
0029010000999991901122513004+62167+030650FM12+010299999V0200501N001019999999N0000001N9-02391+99999100901ADDGF1029919999999999999999
0029010010999991901122520004+62167+030650FM12+010299999V0200501N002119999999N0000001N9-02671+99999100821ADDGF1039919999999999999999
0029029500999991901122606004+62167+030650FM12+010299999V0200501N002119999999N0000001N9-02899999100881ADDGF1029919999999999999999
0029029600999991901122613004+62167+030650FM-12+010299999V0200501N002119999999N0000001N9-02781+99999100831ADDGF1029919999999999999999
0029029580999991901122620004+62167+030650FM-12+010299999V0200501N001019999999N0000001N9-03001+99999100871ADDGF1029919999999999999999

Figure 5.8 : Mesures effectuées par les stations climatologiques

Les deux ensembles de données(Dataset) sont sauvegardées au niveau du système de fichiers distribués de la plate-forme Hadoop, en l'occurrence HDFS.

5.4.1. Distribution homogène

Dans cette première distribution le nombre d'occurrences d'une valeur de l'attribut de jointure est reparti d'une manière homogène, par rapport au nombre d'enregistrements du fichier en entrée du job MapReduce qui réalise la jointure. Le nombre d'enregistrements représentant les mesures effectuées par chaque station climatologique est identique.

Le tableau ci-dessus résume les résultats obtenus de ce test.

Configuration du Cluster	C1	C2	C3	C4
Nombre de Nœuds Esclaves	4	10	15	20
Temps d'exécution (ms)	325350	241400	199353	157353
Ecart entre 2 configurations consécutives (ms)	0	83950	42047,5	42000
Ecart entre la configuration Ci la configuration C4 (ms)	167997,5	84047,5	42000	0

Une représentation graphique des résultats obtenus du test, où le nombre d'occurrences d'une valeur de l'attribut de jointure dans le cas d'une distribution homogène.

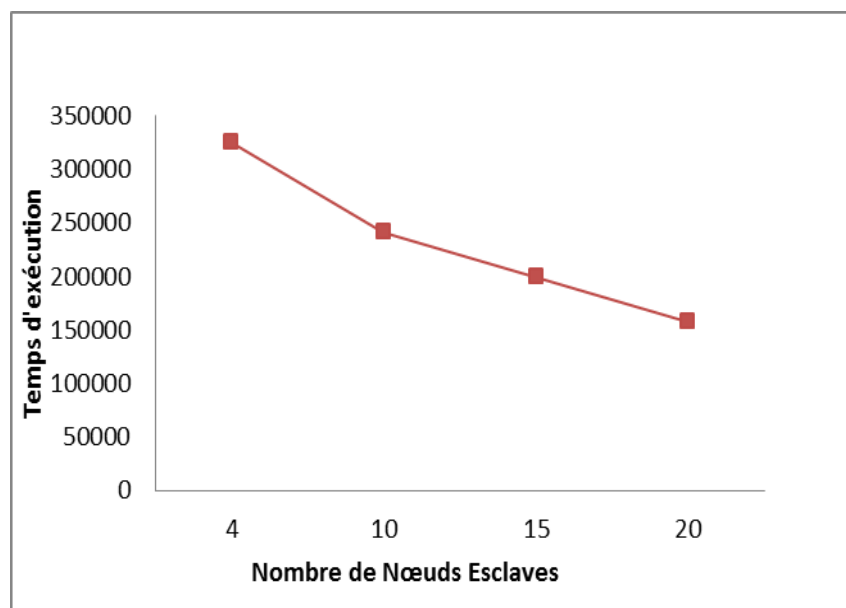


Figure 5.9 : Distribution homogène des données en entrée.

5.4.2. Distribution hétérogène

Dans cette deuxième distribution nous varions le nombre d'enregistrements représentant les mesures effectuées par les stations climatologiques. Le nombre d'occurrences d'une valeur de l'attribut de jointure (identifiant de la station climatologique) est reparti d'une manière hétérogène, par rapport au nombre d'enregistrements du fichier en entrée du job MapReduce qui réalise la jointure.

Test 1 : Distribution hétérogène avec une domination de 30%

Le tableau ci-dessous résume les résultats obtenus de ce test, où le nombre d'occurrences d'une valeur de l'attribut de jointure représente 30% du fichier en entrée.

Configuration du Cluster	C1	C2	C3	C4
Nombre de Nœuds Esclaves	4	10	15	20
Temps d'exécution (ms)	430005	398500	377520	367020
Ecart entre 2 configurations consécutives (ms)	0	31505	20980	10500
Ecart entre la configuration Ci la configuration C4 (ms)	62985	31480	10500	0

Une représentation graphique des résultats obtenus du test, où le nombre d'occurrences d'une valeur de l'attribut de jointure représente 30% du fichier en entrée du job MapReduce qui réalise la jointure.

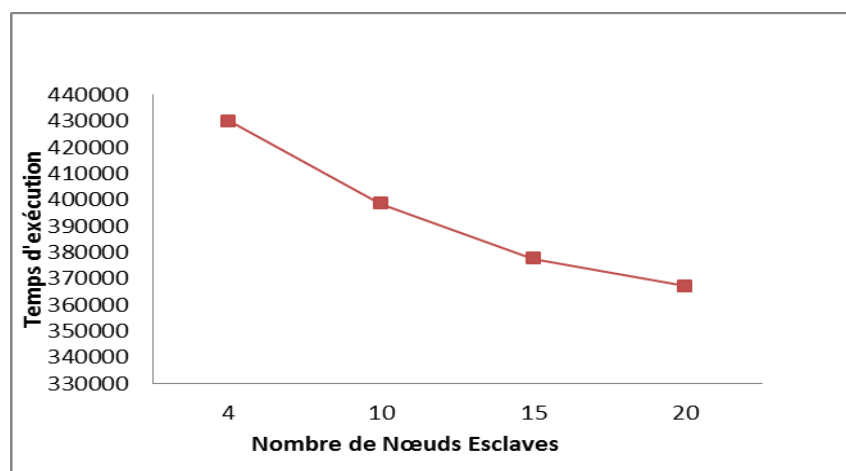


Figure 5.10 : Distribution hétérogène avec une domination de 30%

Test 2 : Distribution hétérogène avec une domination de 40%

Le tableau ci-dessous résume les résultats obtenus de ce test, où le nombre d'occurrences d'une valeur de l'attribut de jointure représente 40% du fichier en entrée.

Configuration du Cluster	C1	C2	C3	C4
Nombre de Nœuds Esclaves	4	10	15	20
Temps d'exécution (ms)	604257,5	530758	521558	521558
Ecart entre 2 configurations consécutives (ms)	0	73500	9200	0
Ecart entre la configuration Ci la configuration C4 (ms)	82700	9200	0	0

Une représentation graphique des résultats obtenus du test, où le nombre d'occurrences d'une valeur de l'attribut de jointure représente 40% du fichier en entrée du job MapReduce qui réalise la jointure.

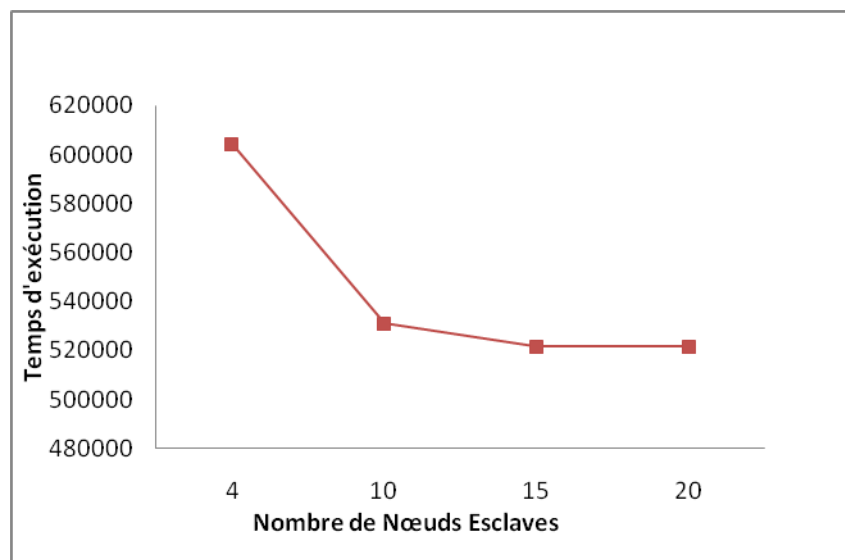


Figure 5.11 : Distribution hétérogène avec une domination de 40%

Test 3 : Distribution hétérogène avec une domination de 70%

Le tableau ci-dessous résume les résultats obtenus de ce test, où le nombre d'occurrences d'une valeur de l'attribut de jointure représente 70% du fichier en entrée.

Configuration du Cluster	C1	C2	C3	C4
Nombre de Nœuds Esclaves	4	10	15	20
Temps d'exécution (ms)	768162,5	631663	627463	627463
Ecart entre 2 configurations consécutives (ms)	0	136500	4200	0
Ecart entre la configuration Ci la configuration C4 (ms)	143850	7350	0	0

Une représentation graphique des résultats obtenus du test, où le nombre d'occurrences d'une valeur de l'attribut de jointure représente 40% du fichier en entrée du job MapReduce qui réalise la jointure.

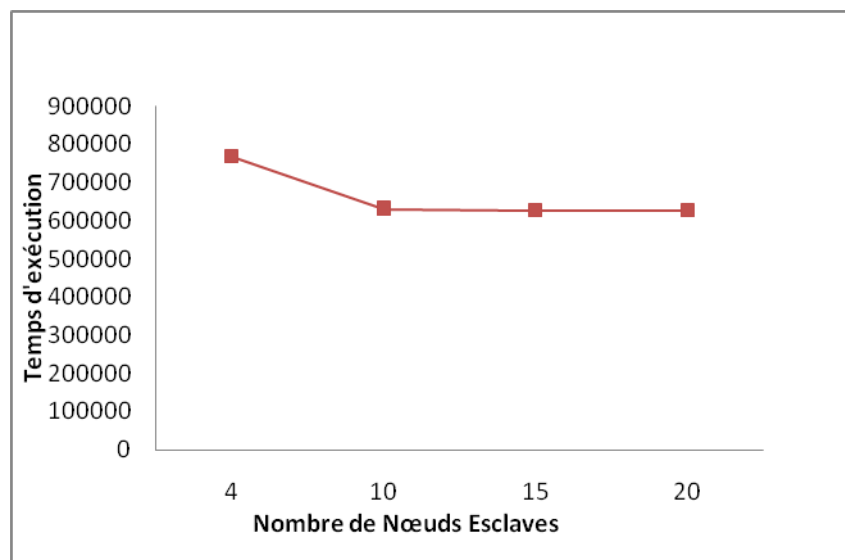


Figure 5.12 : Distribution hétérogène avec une domination de 70%

Test 4 : Distribution hétérogène avec une domination de 90%

Le tableau ci-dessous résume les résultats obtenus de ce test, où le nombre d'occurrences d'une valeur de l'attribut de jointure représente 90% du fichier en entrée.

Configuration du Cluster	C1	C2	C3	C4
Nombre de Nœuds Esclaves	4	10	15	20
Temps d'exécution (ms)	940087,5	793088	780488	780488
Ecart entre 2 configurations consécutives (ms)	0	147000	12600	0
Ecart entre la configuration Ci la configuration C4 (ms)	159600	12600	0	0

Une représentation graphique des résultats obtenus du test, où le nombre d'occurrences d'une valeur de l'attribut de jointure représente 90% du fichier en entrée du job MapReduce qui réalise la jointure.

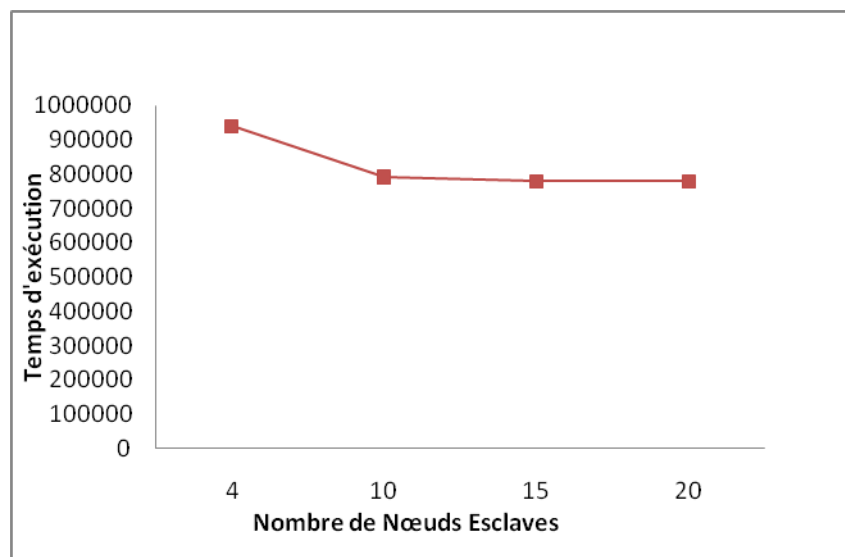


Figure 5.13 : Distribution hétérogène avec une domination de 90%

5.5. Conclusion

Tout au long de ce chapitre nous avons présenté les résultats des expérimentations effectuées pour évaluer la jointure répartie par MapReduce présentée au chapitre 4. Les évaluations se focalisent essentiellement sur l'impact du nombre d'occurrences d'une valeur de l'attribut de jointure sur le temps d'exécution de cette approche de jointure. Nous avons décrit en premier lieu l'environnement d'évaluation et les conditions expérimentales qui le caractérisent. Par la suite, nous avons montré la fonctionnalité de la plate-forme déployée sur un cluster de calcul via un exemple d'exécution d'un job MapReduce. Les résultats obtenus montrent la scalabilité de cette plate-forme de tests.

Nous avons organisé cette expérimentation en deux parties:

- Dans la première partie, la distribution homogène, nous avons évalué la jointure répartie suivant le nombre d'occurrences d'une valeur de l'attribut de jointure, par rapport au nombre d'enregistrements du fichier en entrée du job MapReduce qui réalise la jointure. Dans ce cas, le nombre d'enregistrements représentant les mesures effectuées par chaque station climatologique est identique. Cette évaluation nous a permis de constater que tous les résultats obtenus montrent que le temps d'exécution est proportionnel au nombre de nœuds qui composent le cluster de calcul, ce qui garantit le passage à l'échelle.
- Dans la deuxième partie, la distribution hétérogène, nous avons évalué la jointure répartie en variant le nombre d'occurrences d'une valeur de l'attribut de jointure, par rapport au nombre d'enregistrements du fichier en entrée. Les résultats obtenus nous ont permis de constater qu'avec un taux de distribution inférieur à 40%, les résultats sont identiques à la distribution homogène. Au-delà de 40%, nous constatons que le temps d'exécution demeure inchangé à partir de la configuration du cluster où nous avons 15 nœuds qui participent au traitement et c'est la même constatation si nous augmentons le nombre de nœuds du cluster de calcul. En outre, plus le taux de domination ou distribution d'une valeur de l'attribut de jointure est important, plus le temps d'exécution est aussi important.

Par conséquent, le déséquilibre des valeurs de l'attribut de jointure dégrade les performances de la jointure répartie.

Pour pallier à cette limitation, nous proposons dans le chapitre suivant une approche pour équilibrer les valeurs de l'attribut de jointure.

Chapitre 6 :

La jointure par agrégation avec MapReduce

6.1. Introduction

Etant donné que, ces dernières années, les paradigmes d'évaluation des jointures basés sur le modèle de programmation *MapReduce* ont gagné en popularité avec l'émergence des bases de données de tailles énormes et qu'ils ont pu venir à bout d'un ensemble de problèmes que les bases de données relationnelles étaient incapables de résoudre, ils restent néanmoins limités dans d'autres problèmes tel que le déséquilibre des valeurs de l'attribut de jointure. Pour remédier à cette limitation, nous proposons dans ce chapitre une approche pour équilibrer les charges des différents *Reducers*. Pour ce faire, nous nous sommes basés sur l'utilisation d'un mécanisme d'agrégation des valeurs de l'attribut de jointure (clé originale) et les autres valeurs des attributs de la table de petit volume de données pour la redistribution des données. Nous reviendrons plus en détail sur le paradigme de notre approche dans la suite de ce chapitre.

Contrairement aux algorithmes présentés dans le chapitre précédent pour l'évaluation des requêtes de jointure, notre approche évite de matérialiser le résultat intermédiaire de la jointure tout en traitant de manière très efficace le problème de déséquilibre des valeurs de l'attribut de jointure. Ceci permet, d'une part, de réduire considérablement la taille des données envoyées à chaque *Reducer*, et d'autre part, de minimiser les coûts de communication et par conséquent le temps global de traitement lors de la phase de redistribution des données de jointure.

6.2. Principe de base

Cette section présente une première tentative vers l'implémentation d'un mécanisme de jointure pour interroger des bases de données de tailles énormes basée sur le modèle *Map-Merge-Reduce* pour répondre à la problématique du déséquilibre des valeurs de l'attribut de jointure. Dans une première partie, nous présenterons les principes de base de notre approche pour l'agrégation des valeurs de l'attribut de jointure et un seul attribut de la table de petit volume de données (agrégation simple). Nous détaillerons ensuite dans une deuxième partie la généralisation aux autres valeurs des autres attributs (agrégation multiple) en précisant spécificités, forces et faiblesses de chacune.

Dans la suite de ce chapitre, notre approche prend en compte la réplication possible de données de la table de petit volume de données sur l'ensemble des nœuds pour réduire le coût de traitement de la jointure. En effet, la table de petit volume de données peut être déjà répliquée, sur un ensemble de nœuds pour des besoins techniques (ex. stockage) ou fonctionnels, et donc disposer d'index permanents. Ainsi chaque nœud traite chacun un sous-ensemble de la table de gros volume de données (fragments) afin de diminuer le moins possible le recours de l'immigration des données des deux tables entre les nœuds. Cette réplication préexistante nous permet également d'envisager une autre optimisation : seuls les nœuds disposant de la table de petit volume de données sont utilisés pour la jointure parallèle en recevant les fragments de la table de gros volume de données. Nonobstant, cette optimisation peut être antinomique avec la nécessité de distribuer le traitement sur un large ensemble de nœuds afin de garantir le meilleur équilibre de charge.

6.2.1. Jointure répartie par agrégation simple

On peut décomposer la méthodologie de jointure répartie par agrégation simple en trois phases qui sont :

- Génération de nouvelles clés de jointure.

- Jointure des données locales à chaque nœud.
- Répartition des données sur nœuds *Reducers*.

Dans la jointure répartie par agrégation simple, nous utilisons uniquement un seul attribut pour construire les nouvelles clés de jointure. Cette construction est basée sur la concaténation de deux valeurs à savoir celle de l'attribut de jointure initiale (la valeur de la clé originale) et celle d'un attribut quelconque de la table de petit volume de données.

Soient $L(L_1, L_2, \dots, L_m)$ et $R(R_1, R_2, \dots, R_m)$ deux tables de petit et de gros volumes de données respectivement et soit $F^{(R)}$ une fraction de la table R (fragment).

Comme mentionné dans le chapitre précédent, la jointure répartie naïve $F^{(R)} \propto L$ peut être mise en œuvre dans un seul job MapReduce où chaque *Mapper* délivre le regroupement des paires $\langle \text{Clé Originale}, \text{Valeur} \rangle$ sous la forme :

$\langle L_l, \text{Tuples correspondants du fragment } F^{(R)} \rangle$
 $\langle L_l, \text{Tuples correspondants de la table } L \rangle$

Avec L_l correspond aux valeurs de la clé de jointure.

A contrario, dans la jointure répartie par agrégation simple, le *Mapper* délivre des paires $\langle \text{nouvelle clé}, \text{valeur} \rangle$. Cette nouvelle clé est générée en concaténant chaque valeur de l'attribut de jointure de l'enregistrement traité (clé originale : L_l) avec la valeur correspondante d'un attribut de la même occurrence L_j pour $j \neq l$.

Avec la fonction Map, chaque Mapper va calculer un résultat partiel de l'agrégation en associant toutes les valeurs correspondant à la même clé en un unique couple $\langle \text{Nouvelle Clé}, \text{Valeur} \rangle$. Dans la suite de cette section, ce résultat partiel d'agrégation va être décrit avec un ensemble d'illustrations.

Ainsi, notre démarche consiste dans un premier temps à préparer, dans le cadre d'une jointure de données massives et à l'aide des fonctions Map et Reduce, décrites dans le chapitre 3 de ce document, une stratégie de détermination des nouvelles clés de jointure

afin d'utiliser par la suite le mécanisme initial Map-Reduce. Toutefois, notons que les étapes de la fonction Reduce dans notre approche restent identiques à celles du mécanisme de base (la jointure naïve), à la différence près que la sortie produite par la fonction Map inclut la nouvelle clé pour chaque agrégation d'une valeur d'un attribut associée à cette même clé originale de jointure.

Cependant, la fonction Map est déclinée en trois phases qui sont :

- La génération des nouvelles clés de jointure
- La restitution sous forme d'une liste de tuples du type <Nouvelle Clé, Valeur>
- Emettre la nouvelle paire :

$\langle H_{L_t}(L_j), \text{tuples correspondants du fragment } F^{(R)} \rangle$

$\langle H_{L_t}(L_j), \text{Tuples correspondants de la table } L \rangle$

La première phase de cette fonction consiste à préparer les données avant l'application du mécanisme initial Map-Reduce. Ainsi, les enregistrements des deux tables seront lus puis modifiés en une structure de type **(nouvelle clé,v)** avant d'être émise au Reducer. Cette restructuration est basée sur un mécanisme d'agrégation des valeurs des attributs en correspondance de la clé originale de jointure de la table de petit volume de données.

La seconde phase est une pré-jointure et qui consiste à formater les données suivant la nouvelle clé. Ces données sont pré-formatées pour être ensuite orientées vers le Reducer. Soulignons que cette agrégation des valeurs d'attributs nous aide à avoir une information très détaillée sur la répartition des valeurs de la nouvelle clé de jointure qui participent effectivement dans le résultat envoyé au Reducer.

Nous décrivons dans un premier temps le fonctionnement de l'algorithme pour la génération de cette nouvelle clé par le Mapper.

Algorithme de génération de nouvelle clé

Pour chaque valeur de la clé de jointure originale L_j de la table de petit volume L :

1. Générer la nouvelle clé de jointure $H_{L_j}(L_j)$ dans L en concaténant les valeurs : $H_{L_j}(L_j) = L_j L_j$.
2. Parcourir $F(R)$ et remplacer les valeurs de la colonne de jointure $F(R)$ par la nouvelle clé de jointure $H_{L_j}(L_j)$.

Dans la première étape de cet algorithme, nous générons la nouvelle clé de jointure $H_{L_j}(L_j)$ en choisissons un attribut de la petite table L . Dans cette étape, nous ajoutons $H_{L_j}(L_j)$ comme étant un contenu du champ « Nouvelle clé » dans la table L . Puis dans la seconde étape, nous remplaçons les valeurs de la colonne de jointure de la fraction de la table de gros volume de données $F(R)$ en répliquant la nouvelle clé $H_{L_j}(L_j)$ pour l'ensemble des tuples correspondants.

A titre d'illustration, considérons les deux tables suivantes et comme clé de jointure originale L_2 :

Fraction $F^{(R)}$				Table : L		
R_1	R_2	R_3	R_4	L_1	L_2	L_3
a ₀	b ₀	c ₀	d ₀	b ₀	e ₀	f ₀
a ₀	b ₀	c ₁	d ₀	b ₀	e ₀	f ₁
a ₁	b ₀	c ₀	d ₀	b ₀	e ₁	f ₀
a ₂	b ₁	c ₁	d ₁	b ₁	e ₀	f ₁
a ₃	b ₁	c ₁	d ₂			

Figure 6.1 : Exemple de deux tables

Comme il a été mentionné dans l'algorithme précédent, chaque fonction *Mapper* a deux phases :

Dans la première phase, nous générons la nouvelle clé de jointure $H_{L_j}(L_j)$ en choisissons un attribut de la petite table L . Supposons que le choix s'est porté sur

l'attribut L_2 ($j=2$). Le résultat de la génération de la nouvelle clé et l'ajout du champ de cette dernière est reporté dans la table suivante :

Table : L			
$H_{L_1}(L_2)$	L_1	L_2	L_3
b_0e_0	b_0	e_0	f_0
b_0e_0	b_0	e_0	f_1
b_0e_1	b_0	e_1	f_0
b_1e_0	b_1	e_0	f_1

Figure 6.2 : Exemple d'agrégation de deux valeurs d'attributs L_1 et L_2

Tout en parcourant le fragment $F^{(R)}$, dans la seconde étape de la génération de la nouvelle clé, nous ajoutons cette dernière nouvelle clé générée dans tous les enregistrements en dupliquant cette dernière clé par rapport à la clé de jointure originale dans ce fragment. Ainsi, le résultat obtenu par la duplication est comme suit:

Fraction $F^{(R)}$			
R_1	R_3	R_4	$H_{L_1}(L_2)$
a_0	c_0	d_0	b_0e_0
a_0	c_0	d_0	b_0e_1
a_0	c_1	d_0	b_0e_0
a_0	c_1	d_0	b_0e_1
a_1	c_0	d_0	b_0e_0
a_1	c_0	d_0	b_0e_1
a_2	c_1	d_1	b_1e_0
a_3	c_1	d_2	b_1e_0

Figure 6.3 : Génération des nouvelles clés dans la fraction $F^{(R)}$.

Finalement, la deuxième phase de la fonction Map consiste à l'émission des nouvelles paires <Nouvelle Clé, Valeur> qui est basée sur les valeurs de la clé de jointure $H_{L1}(L_2)$. Ainsi, pour notre exemple, le Mapper délivre trois paires d'enregistrements sous la forme de trois regroupements différents. Comme le montre la Figure 6.3, nous avons obtenu trois groupes dont la fonction du Mapper a donné trois regroupements d'enregistrements. Dans le premier regroupement, nous avons obtenu trois (03) et deux (02) enregistrements (tuples) à partir des tables $F^{(R)}$ et L respectivement. Dans le second regroupement, le Mapper produit trois (03) et un (01) enregistrement respectivement. Or le dernier regroupement obtenu a deux (02) enregistrements du fragment et un seul de la table L .

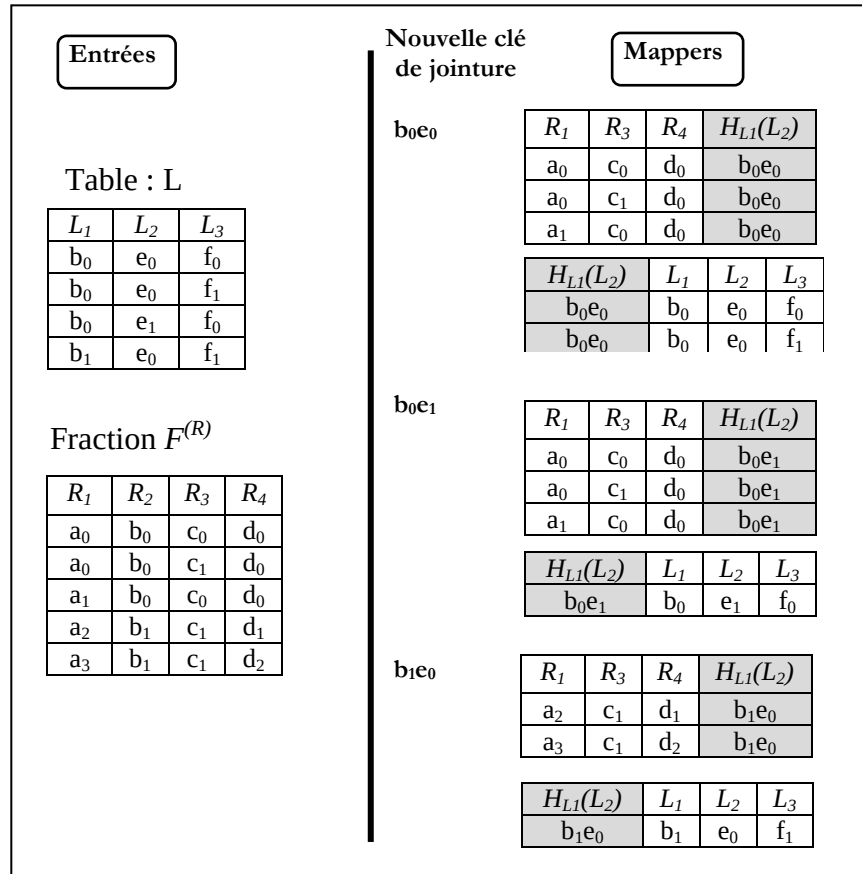


Figure 6.4 : Regroupement des Mappers suivant la nouvelle clé de jointure $H_{L1}(L_2)$.

A titre de comparaison, dans l'approche de jointure repartie (section 4.2. page68), nous obtenons deux enregistrements suivant la clé de jointure originale en regroupant ces derniers enregistrements comme suit :

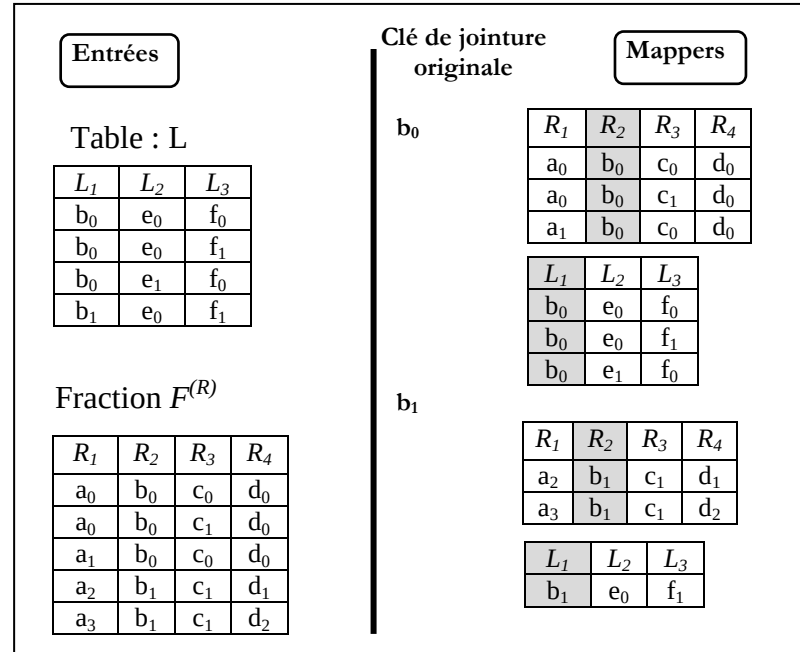
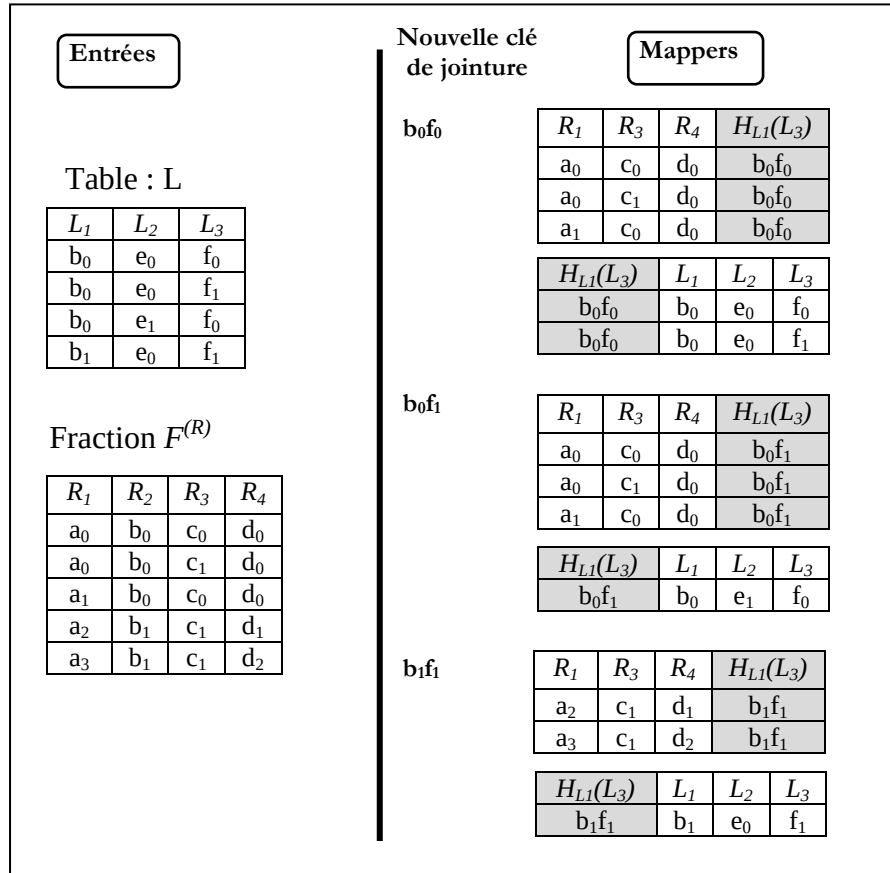


Figure 6.5 : Le regroupement du Mapper suivant la clé de jointure originale L_1

Suivant la configuration de la Figure 6.5, le Mapper envoie chaque regroupement d'enregistrements à un Reducer au lieu d'envoyer trois regroupements. Ainsi, dans notre approche en se basant sur la nouvelle clé, la jointure est évaluée sur trois Reducers au lieu de deux.

Notons que notre approche d'évaluation des jointures a plusieurs configurations de redistribution des regroupements d'enregistrements suivant les choix des attributs d'agrégation utilisés.

Reconsidérons à nouveau l'exemple précédent. En utilisant les valeurs de l'attribut L_3 , la Figure 6.6 illustre les configurations de regroupements possibles. Dans cette tâche de jointure, le Mapper délivre aussi trois groupes d'enregistrements aux trois Reducers différents au même titre que le choix des valeurs de l'attribut L_2 dans la redistribution.

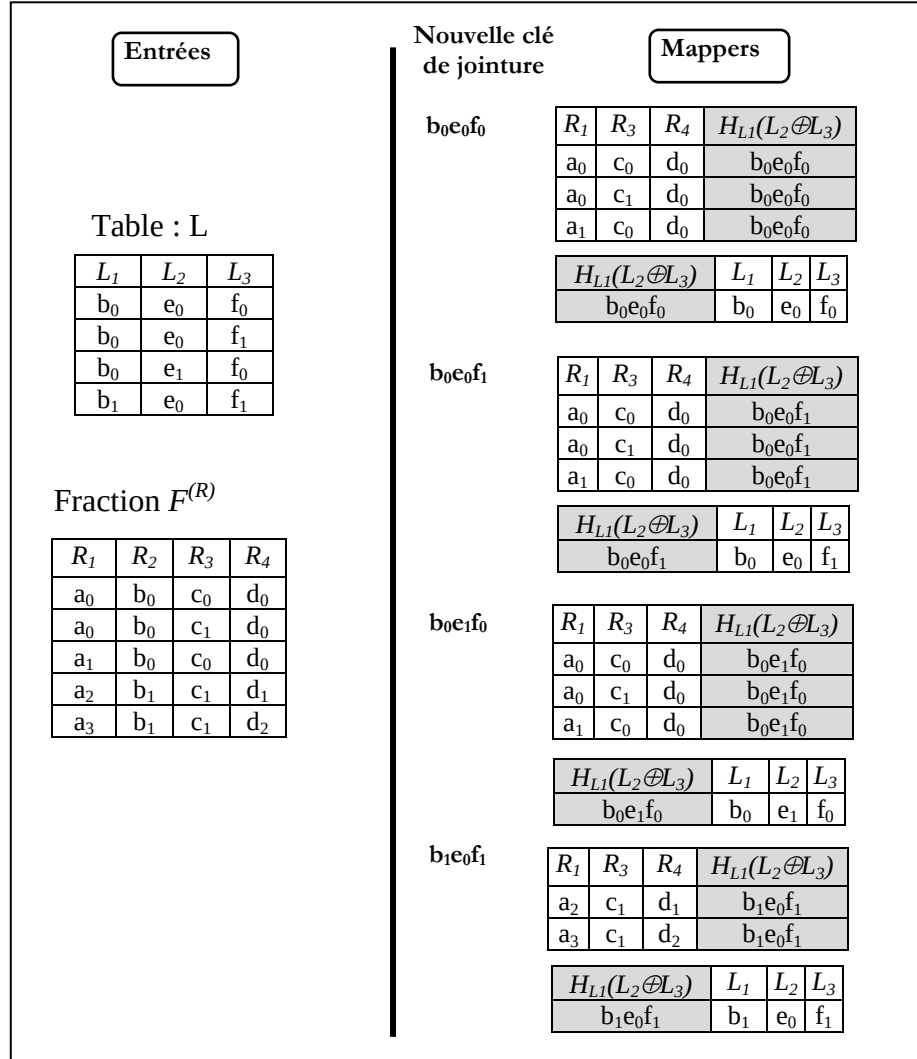

 Figure 6.6 : Le regroupement du Mapper suivant la nouvelle clé de jointure $H_{L1}(L_3)$.

Notons que le surcoût lié à l'utilisation de cette notion d'agrégation des valeurs d'attributs reste négligeable du fait que les regroupements à distribuer sont créés durant la phase Mapping. La raison est l'utilisation d'un simple algorithme de parcours pour les deux tables logeant un nœud *Mapper*.

6.2.2. Jointure répartie par agrégation multiple

Dans cette stratégie, nous le *Mapper* exploite plusieurs attributs pour l'agrégation des valeurs de ces attributs afin de construire la nouvelle clé de jointure. Ainsi, le regroupement des enregistrements à envoyer aux *Reducers* est assuré par la concaténation des valeurs de l'attribut de jointure L_l et les valeurs des attributs $L_1 \oplus L_2 \oplus \dots \oplus L_j$ pour $j \neq l$. La génération de la nouvelle clé de jointure $H_{L_l}(L_1 \oplus L_2 \oplus \dots \oplus L_j)$ est faite à la base du dispatching de l'ensemble des concaténations de chaque tuple de la table de petit volume de données sur les tuples appartenant aux fragments de la table de gros volume de données $F_i^{(R)}$. Notons que les tuples des fragments doivent être en correspondance avec la clé de jointure originale L_l .

Dans le cadre des jointure des données massives, notre fonction Map assure un calcul partiel d'agrégation en associant toutes les concaténations des valeurs correspondant à la même nouvelle clé en un unique couple ***$\langle$ Nouvelle Clé, Valeur $\rangle$*** . Ensuite, un regroupement des enregistrements global permet la détermination le nombre des opérations de jointure suivant la nouvelle clé et de délivrer les résultats de regroupements aux *Reducers*.

Figure 6.7 : Regroupement du Mapper suivant la nouvelle clé $H_{L_1}(L_2 \oplus L_3)$.

Dans le but d'illustrer le fonctionnement de cette opération de jointure par agrégation multiple et de faire une comparaison avec celle dite simple, nous reconsidérons à nouveau l'exemple d'illustration précédent. Le résultat obtenu par la fonction Map est donné par la figure 6.7. Ainsi nous avons quatre (04) regroupements d'enregistrements au lieu de trois (03) regroupements dans la jointure par agrégation simple et de deux (02) regroupements dans la jointure répartie naïve (section 4.2. page 68). Notons que si l'ensemble des agrégations des valeurs des attributs donne différentes nouvelles clés de jointure, notre approche de jointure par agrégation multiple, chaque *Mapper* délivre un sous ensemble d'enregistrements de la table de gros volume de données et un unique enregistrement de la table de petit volume de données. Autrement dit, chaque

enregistrement de la table de petit volume de données est dispatché sur un regroupement par la nouvelle clé de jointure. Dans cette situation, le nombre des *Reducers* nécessaire pour la redistribution des regroupements est égale au nombre des tuples de cette petite table.

Dans la suite de ce chapitre, nous nous proposerons d'envisager l'exploration de la jointure de données massives sous le modèle *MapReduce* est toujours sous l'hypothèse de la réplication de la table de petit volume de données et notre génération de nouvelles clés de jointure. Ainsi, une jointure massive se déroule en cinq étapes, donnant lieu chacune à un workflow *MapReduce*, qui sont :

- La réplication de la table de petit volume de données avec le marquage d'un ou plusieurs attributs d'agrégation ;
- La répartition de la table de gros volume de données en fragments (splits) entre différents Mappers;
- L'exécution de la jointure par agrégation pour chaque fragment.
- Le regroupement de l'ensemble des valeurs ayant la même valeur de nouvelle clé ;
- La répartition des regroupements entre différents Reducers.

Ces étapes permettent l'élaboration du plan d'exécution d'une jointure répartie dont la première étape consiste à répliquer la table de petit volume de donnée dans différents *Mappers* tout en fixant le type d'agrégation (simple ou multiple suivant le choix du nombre des attributs) pour la valeur de clé de jointure. Dans la seconde étape, nous répartissons les fragments de la table de gros volume de données à l'ensemble des *Mappers*. Puis la troisième étape exécute la jointure répartie décrite précédemment. Après avoir effectué les jointures de tous les fragments, la quatrième étape effectuera des regroupements visant à regrouper l'ensemble des valeurs ayant la même valeur de la nouvelle clé en une seule structure (deux fragments des deux tables) à des fins de dispatching des données regroupées aux *Reducers*. Après l'émission de ces fragments, la dernière étape se contente de passer d'une forme groupée des fragments en entrée vers la forme initiale avant de les émettre vers le fichier de sortie DFS.

A titre d'exemple, nous donnons ici l'exécution d'une jointure utilisant un cluster de trois nœuds (*Mappers* ou *Reducers*). Pour une raison de présentation, supposons que nous avons trois fragments de la table de gros volume de données $R : F_1^{(R)}, F_2^{(R)} \text{ et } F_3^{(R)}$.

Dans une première illustration, nous revenons sur les algorithmes standards (ou naïfs) de la jointure basés sur MapReduce (section 4.2. page 68) en apportant une analyse dans le contexte de la fragmentation de la table de petit volume de données. Comme le montre la figure 6.8, la table de petit volume L a été fragmentée suivant la cardinalité de chaque valeur de la clé de jointure originale (l'attribut L_I). Ainsi, nous avons obtenu deux fragments dont le premier contient six (06) enregistrements de la table de gros volume de données et deux (02) enregistrements de la table L qui correspondent à la valeur b_0 de l'attribut L_I . Le second fragment contient un fragment de neuf (09) enregistrements et d'un (01) enregistrement des deux tables R et L respectivement. En conclusion, tablant sur le fait que nous voulons optimiser le temps de traitement dans l'opération finale de jointure (produit cartésien des deux fragments), cet algorithme naïf n'assure généralement pas un équilibre des charges dans le calcul du résultat final pour chaque Reducer (Cette optimisation vise à limiter la charge d'un Reducer). Ce constat est dû au fait que le premier Reducer effectue Douze (6X2) fois l'opération finale de jointure alors que le second Reducer contiendra d'effectuer neuf (9X1) fois cette opération.

A contrario, dans notre méthode de jointure par agrégation multiple, nous avons obtenu trois fragments des deux tables. Le premier regroupe six (06) enregistrements de la table R et un seul enregistrement de la table L . Le second fragment a la même quantité d'enregistrements que le premier fragment. En somme, ces deux fragments émis aux *Reducers* 1 et 2 représentent en réalité la distribution de la charge du premier *Reducer* dans l'approche de jointure naïve décrite précédemment. Ainsi, nous constatons que notre méthode élabore un plan d'exécution de l'opération finale de la jointure naïve limitant l'effet de l'attribut à deux fréquences b_0 et du déséquilibre des données. Par contre, le troisième *Reducer* a la même charge que celui dans la jointure naïve car le fragment de la

table de petit volume de données L contient un seul enregistrement. Nous déroulons un exemple de la jointure standard répartie (section 4.2. page 68).

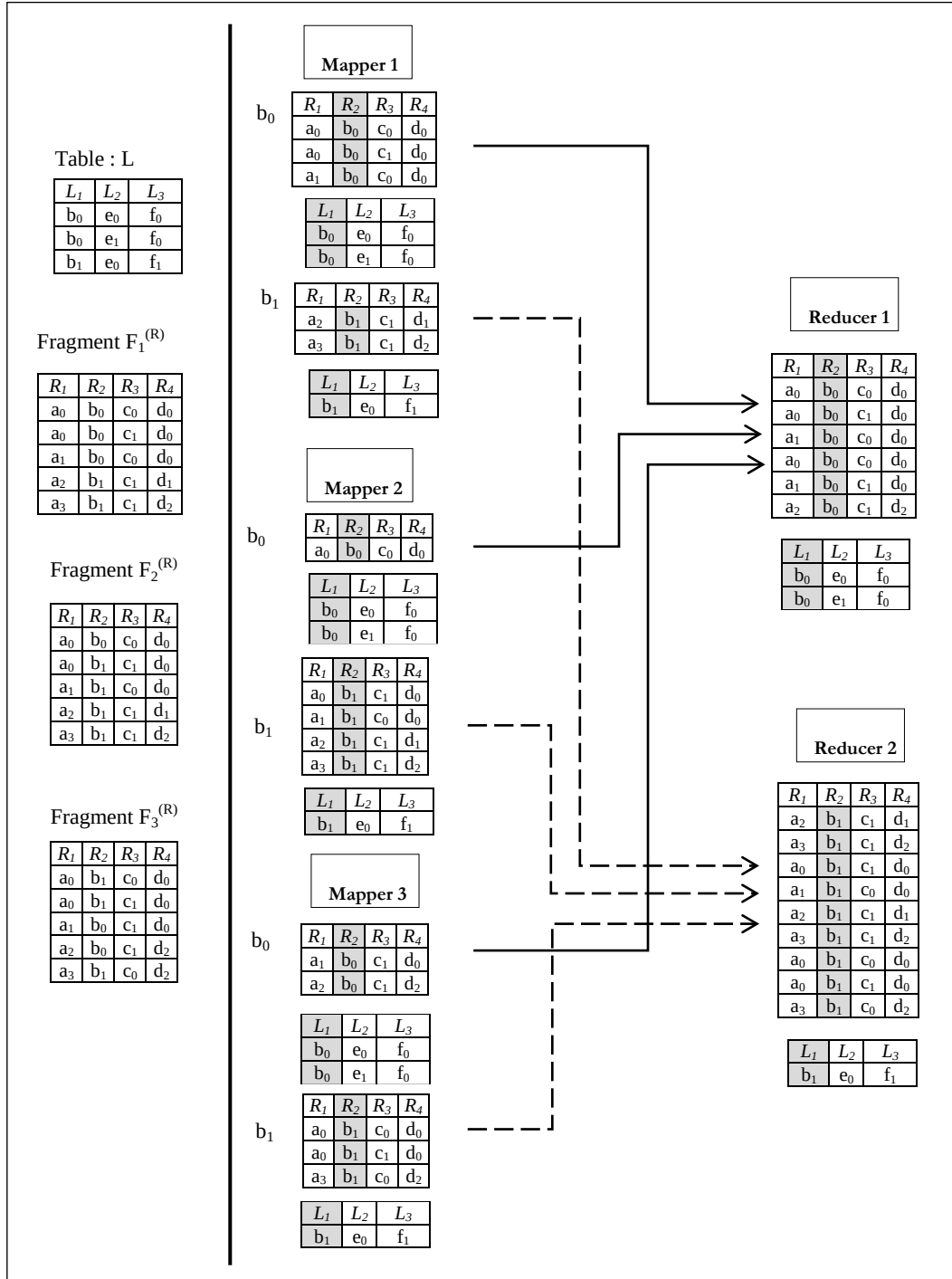


Figure 6.8 : Un exemple de la jointure standard répartie

Ci-dessous un exemple de jointure répartie par agrégation multiple de trois fragments.

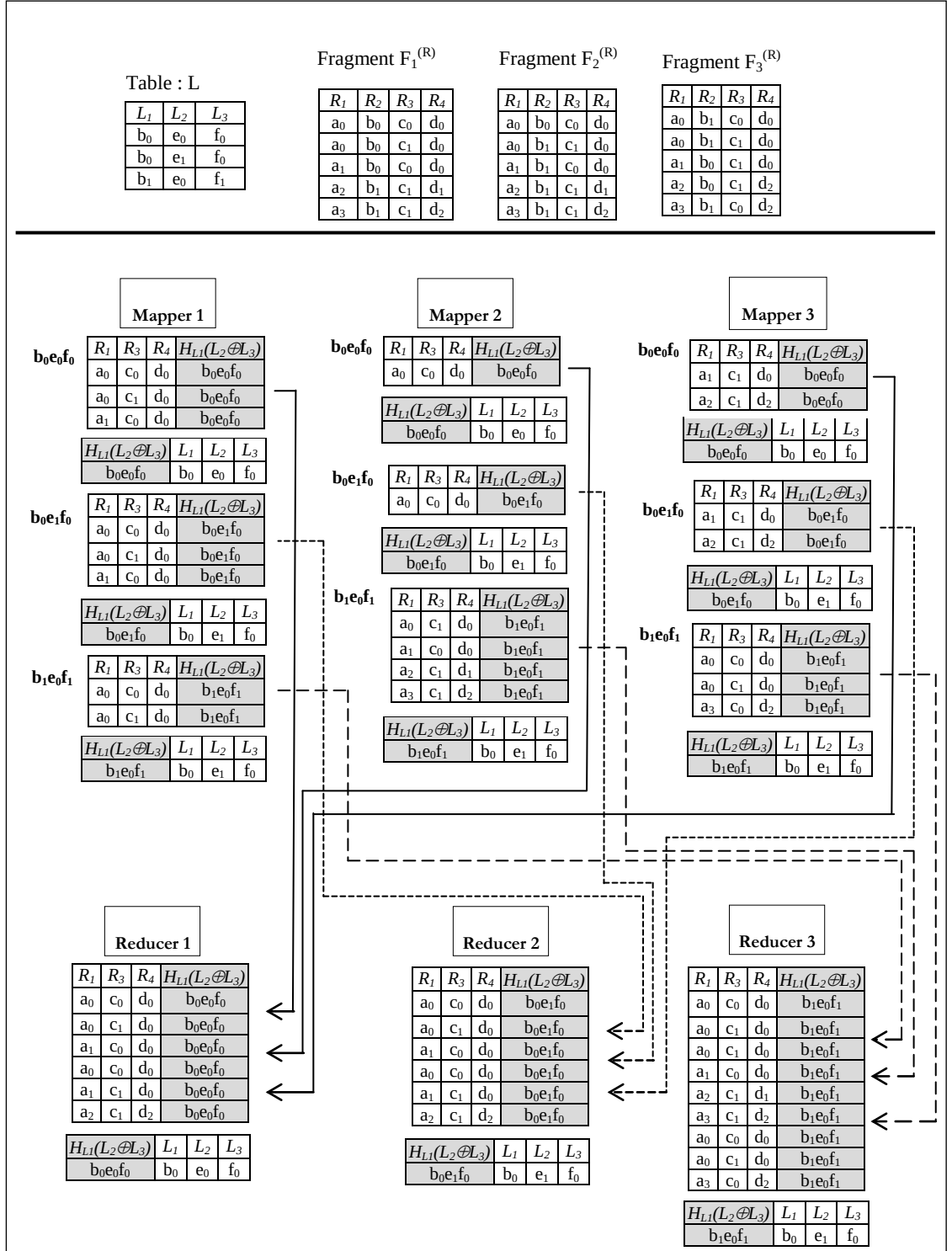


Figure 6.9 : Jointure répartie par agrégation multiple de trois fragments.

En conclusion, désirant sur le fait que nous voulons optimiser le temps de traitement dans l'opération finale de jointure, la jointure naïve n'assure généralement pas un équilibre des charges dans le calcul du résultat final pour chaque *Reducer*. (Cette optimisation vise à limiter la charge d'un *Reducer*). Cependant, notre méthode de jointure par agrégation multiple assure une répartition des fragments qui vise à limiter la charge d'un *Reducer*.

6.3. Conclusion

L'objectif principal de ce chapitre est la proposition d'une approche pour le traitement des requêtes de jointure simple dans un modèle MapReduce. Cette approche est basée sur un mécanisme de jointure répartie par agrégation des valeurs d'attributs d'une table de petit volume avec la clé de jointure initiale.

Comme le montre le mécanisme décrit dans ce chapitre, la jointure répartie par agrégation se prête naturellement à la répartition. Il ressort du précédent mécanisme de jointure par agrégation simple ou multiple qu'une distribution des données obtenues assure une efficacité du traitement de jointure répartie. Cette agrégation d'un seul attribut ou de plusieurs attributs peut être choisie pour raison de sa distribution où le regroupement des enregistrements selon la nouvelle clé de jointure soit équitable entre les différents nœuds *Reducers*. A la lumière du mécanisme proposé, nous pouvons conclure que notre approche assure une répartition des fragments qui vise à limiter la charge d'un nœud *Reducer*. Cependant, cette répartition n'est pas toujours atteignable à cause du déséquilibre probable des données.

En effet, malgré une bonne distribution des données sur un attribut quelconque des deux tables à joindre, il peut s'avérer qu'un nœud se voit affecté une paire de regroupements admettant un seul enregistrement de la table de petit et gros volumes de données chacune telle que la distributivité des valeurs de l'attribut implique un traitement lourd pour un ensemble de nœud et un traitement quasi-nul pour un autre ensemble. Ceci

produit un déséquilibre qui annule l'efficacité attendue d'un traitement de jointure parallèle en introduisant des goulets d'étranglement. Cependant il existe une multitude de méthodes pour appréhender ce type de problématique. Une méthode visant à équilibrer les regroupements selon la fréquence des valeurs de l'agrégation des attributs de jointure peut être envisagée selon la distributivité des données de la nouvelle clé de jointure dans les deux relations. Cette étape est un élément clé du traitement réparti d'une jointure entre deux relations.

Conclusion Générale

Aujourd'hui, avec l'émergence des applications intensives en manipulation de données, les approches d'évaluation de jointures basées sur le modèle de programmation MapReduce ont gagné en popularité et ont pu venir à bout d'un ensemble de problèmes que les bases de données relationnelles, toujours plus volumineuses pour accélérer leurs analyses et raccourcir les temps de réponse aux requêtes utilisateurs. Néanmoins, les modèles proposés dans ces approches généralement n'évitent pas le problème des répartitions inégales obtenues dans la première phase du modèle MapReduce (le déséquilibre probable des charges des différents Reducers).

Afin de réduire ce déséquilibre tout en redistribuant les fortes répartitions des données de jointure aux différents Reducers, nous avons proposé une première tentative d'équilibrage qui est basée sur un mécanisme d'agrégation simple ou multiple des valeurs de la clé de jointure initiale, avec une ou plusieurs autres valeurs des attributs.

Ainsi, le travail présenté dans ce mémoire a porté sur la réduction probable des charges des Reducers dans le modèle MapReduce lors de l'évaluation des requêtes de jointure sur des architectures distribuées. Dans ce cadre, nous avons proposé deux mécanismes d'agrégation des valeurs de la clé de jointure :

- Pour répondre à la problématique du déséquilibre des valeurs de l'attribut de jointure, la première variante de jointure répartie est basée sur un mécanisme d'agrégation simple des valeurs de l'attribut de jointure et une seule valeur d'un seul attribut de l'ensemble de données (DataSet), ayant le petit volume de données répliquées dans l'architecture distribuée. Notre démarche de jointure consiste dans un premier temps à préparer, dans le cadre d'une jointure de données massives et à l'aide des fonctions Map et Reduce, une stratégie de détermination des nouvelles clés de

jointure afin d'utiliser par la suite le mécanisme initial MapReduce. Toutefois, notons que les étapes de la fonction Reduce dans notre approche restent identiques à celles du mécanisme de base (la jointure répartie), à la différence près que la sortie produite par la fonction Map inclut la nouvelle clé pour chaque agrégation d'une valeur d'un attribut associée à cette même clé originale de jointure. Cependant, notre fonction Map est déclinée en deux phases que sont : La première phase prépare les données en une nouvelle structure de type (Nouvelle clé, valeurs) avant l'application du mécanisme initial MapReduce. Ainsi, les enregistrements des deux tables seront lus puis modifiés avant d'être émise au Reducer. La seconde phase est une pré-jointure qui consiste à formater les données suivant la nouvelle clé pour être ensuite orientées vers un Reducer.

➤ La seconde variante de jointure répartie est basée sur un mécanisme d'agrégation multiple. Pour équilibrer les charges des différents Reducers, nous nous sommes basés sur l'utilisation de plusieurs valeurs d'agrégation afin de construire la nouvelle clé de jointure. Ces nouvelles clés permettent la redistribution des données tout en réduisant la charge possible de chaque Reducer. Dans ce cas, chaque Reducer s'occupe d'un sous-ensemble d'agrégation des valeurs de l'attribut de jointure original et non de ses propres données initiales.

Ces deux variantes de mécanismes d'agrégation ont montré leur efficacité sur une possibilité de rééquilibrage des charges des Reducers en réduisant le temps global de traitement de la jointure répartie. Généralement, ce temps global est relatif au traitement envoyé au Reducer ayant la plus grande nouvelle répartition en matière de taille de données car les données appartenant à un Reducer surchargé sont transférées vers les Reducers les moins chargés durant l'évaluation de la jointure. Durant l'expérimentation, le second mécanisme garantit généralement que la taille du résultat de la jointure dans chaque nœud soit proportionnelle à la capacité de calcul, même en présence d'un fort déséquilibre des données à partir de 40%.

Par ailleurs, contrairement aux algorithmes présentés dans la littérature, notre approche évite de matérialiser le résultat intermédiaire de la jointure tout en traitant de manière efficace le problème de déséquilibre des valeurs de l'attribut de jointure par ces

mécanismes d'agrégation proposés. Ceci permet, d'une part, de réduire considérablement la taille des données envoyées à chaque Reducer, et d'autre part, de minimiser les coûts de communication et par conséquent le temps global de traitement lors de la phase de redistribution des données de jointure. De plus, notre approche exploite efficacement la puissance des architectures distribuées lorsque le nombre de Reducers de la répartition des données initiales est largement moindre par rapport au nombre des nœuds effectifs.

Au niveau de l'approche de jointure répartie par agrégation proposée, plusieurs améliorations pourraient être effectuées :

- ❖ Les transformations et les traitements de jointure répartie proposés permettent de pallier facilement à un ensemble de carences de l'évaluation de jointure simple. Toutefois, il serait également intéressant de travailler sur le traitement de requêtes complexes dans les systèmes de fichiers distribués (HDFS).
- ❖ Envisager l'utilisation des histogrammes des valeurs d'agrégation pour rééquilibrer les charges des nœuds. Nous pensons que ces histogrammes apportent la possibilité de réduire efficacement le temps d'exécution d'une requête dans les systèmes de fichiers distribués.
- ❖ Il serait possible d'investiguer la combinaison de valeurs des attributs de l'ensemble des tables pour la génération de la nouvelle clé de jointure. De surcroît, nous pensons que l'adoption d'une technique de combinaison des valeurs d'attributs de plusieurs tables avec le recours aux histogrammes serait particulièrement prometteuse dans l'optimisation du temps d'exécution des requêtes de jointure et d'équilibrage de charge dans une architecture distribuée.

Bibliographie

1. Brewer, E., *CAP twelve years later: How the "rules" have changed*. Computer, 2012. **45**(2): p. 23-29.
2. Gantz, J. and D. Reinsel, *The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east*. IDC iView: IDC Analyze the Future, 2012.
3. Wilson, R.E., S.D. Gosling, and L.T. Graham, *A review of Facebook research in the social sciences*. Perspectives on Psychological Science, 2012. **7**(3): p. 203-220.
4. Palmerini, P., S. Orlando, and R. Perego. *Statistical properties of transactional databases*. in *Proceedings of the 2004 ACM symposium on Applied computing*. 2004. ACM.
5. Feuerlicht, G. and J. Pokorný, *Can Relational DBMS Scale Up to the Cloud?*, in *Information Systems Development*. 2013, Springer. p. 317-328.
6. Lewis, P.M., A.J. Bernstein, and M. Kifer, *Databases and transaction processing: an application-oriented approach*. 2002: Addison-Wesley.
7. Russom, P., *Big data analytics*. TDWI Best Practices Report, Fourth Quarter, 2011.
8. Agrawal, D., S. Das, and A. El Abbadi, *Big data and cloud computing: new wine or just new bottles?* Proceedings of the VLDB Endowment, 2010. **3**(1-2): p. 1647-1648.
9. Herodotou, H., et al. *Starfish: A Self-tuning System for Big Data Analytics*. in *CIDR*. 2011.
10. Tiwari, S., *Professional NoSQL*. 2011: John Wiley & Sons.
11. Hecht, R., *NoSQL Evaluation*. 2011.
12. Vicknair, C., et al. *A comparison of a graph database and a relational database: a data provenance perspective*. in *Proceedings of the 48th annual Southeast regional conference*. 2010. ACM.
13. Cattell, R., *Scalable SQL and NoSQL data stores*. ACM SIGMOD Record, 2011. **39**(4): p. 12-27.
14. Moreira, J.E., et al. *Scalability of the Nutch search engine*. in *Proceedings of the 21st annual international conference on Supercomputing*. 2007. ACM.
15. White, T., *Hadoop: The Definitive Guide: The Definitive Guide*. 2009: O'Reilly Media.

16. Lublinsky, B., K.T. Smith, and A. Yakubovich, *Professional Hadoop Solutions*. 2013: John Wiley & Sons.
17. Borthakur, D., et al. *Apache Hadoop goes realtime at Facebook*. in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 2011. ACM.
18. Taylor, R.C., *An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics*. BMC bioinformatics, 2010. 11(Suppl 12): p. S1.
19. Venner, J. and S. Cyrus, *Pro Hadoop*. Vol. 1. 2009: Springer.
20. Georgiou, A., *Storing Data Flow Monitoring in Hadoop*. 2013.
21. Turkington, G., *Hadoop Beginner's Guide*. 2013: Packt Publishing Ltd.
22. Borthakur, D., *HDFS architecture guide*. HADOOP APACHE PROJECT <http://hadoop.apache.org/common/docs/current/hdfs design. pdf>, 2008.
23. Holmes, A., *Hadoop in Practice*. 2012: Manning Publications Co.
24. Perera, S., *Hadoop MapReduce Cookbook*. 2013: Packt Publishing Ltd.
25. Arantes, L., et al., *Vers une plate-forme MapReduce tolérant les fautes byzantines*. Technique et Science Informatiques, 2012. 31(8-10): p. 1079-1099.
26. Cloudera, I., *CDH Proven, enterprise-ready Hadoop distribution—100% open source*. 2012, Oct.
27. Li, J., Z. Cai, and S. Chen. *Multi-weighted tree based query optimization method for parallel relational database systems*. in *Cooperative Database Systems for Advanced Applications, 2001. CODAS 2001. The Proceedings of the Third International Symposium on*. 2001. IEEE.
28. Kitsuregawa, M., H. Tanaka, and T. Moto-Oka, *Application of hash to data base machine and its architecture*. New Generation Computing, 1983. 1(1): p. 63-74.
29. DeWitt, D.J. and R.H. Gerber, *Multiprocessor hash-based join algorithms*. 1985: University of Wisconsin-Madison, Computer Sciences Department.
30. Lu, H., K. Tan, and B.-C. Ooi, *Query processing in parallel relational database systems*. 1994: IEEE Computer Society Press.
31. Iyer, B.R., G.R. Ricard, and P.J. Varman. *Percentile finding algorithm for multiple sorted runs*. in *Proceedings of the 15th international conference on Very large data bases*. 1989. Morgan Kaufmann Publishers Inc.
32. DeWitt, D.J., J.F. Naughton, and D.A. Schneider. *Parallel sorting on a shared-nothing architecture using probabilistic splitting*. in *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*. 1991. IEEE.

-
33. Lakshmi, M.S. and P.S. Yu, *Effectiveness of parallel joins*. Knowledge and Data Engineering, IEEE Transactions on, 1990. 2(4): p. 410-424.
 34. DeWitt, D.J., et al., *Practical skew handling in parallel joins*. 1992: University of Wisconsin-Madison, Computer Sciences Department.
 35. Bamha, M. and G. Hains, *An efficient equi-semi-join algorithm for distributed architectures*, in *Computational Science-ICCS 2005*. 2005, Springer. p. 755-763.
 36. Bamha, M. and G. Hains. *A skew-insensitive algorithm for join and multi-join operations on shared nothing machines*. in *Database and Expert Systems Applications*. 2000. Springer.
 37. Copeland, G., et al., *Data placement in Bubba*. Vol. 17. 1988: ACM.
 38. Lowenthal, D.K. and G.R. Andrews. *An adaptive approach to data placement*. in *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*. 1996. IEEE.
 39. Shatdal, A. and J.F. Naughton, *Using shared virtual memory for parallel join processing*. Vol. 22. 1993: ACM.
 40. Schneider, D.A. and D.J. DeWitt, *A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment*. Vol. 18. 1989: ACM.
 41. Blanas, S., et al. *A comparison of join algorithms for log processing in mapreduce*. in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010. ACM.
 42. Olston, C., et al. *Pig latin: a not-so-foreign language for data processing*. in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008. ACM.
 43. Thusoo, A., et al., *Hive: a warehousing solution over a map-reduce framework*. Proceedings of the VLDB Endowment, 2009. 2(2): p. 1626-1629.
 44. Beyer, K.S., et al. *Jaql: A scripting language for large scale semistructured data analysis*. in *Proceedings of VLDB Conference*. 2011.
 45. Mi, C., Q. Chen, and T. Liu. *An Efficient Cross-Match Implementation Based on Directed Join Algorithm in MapReduce*. in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*. 2011. IEEE.
 46. Lin, J. and C. Dyer, *Data-intensive text processing with MapReduce*. Synthesis Lectures on Human Language Technologies, 2010. 3(1): p. 1-177.