Parallélisme et équilibrage de charges dans le traitement de la jointure sur des architectures distribuées.

Mohamad Al Hajj Hassan

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES

LABORATOIRE : LIFO

THÈSE présentée par :

Mohamad AL HAJJ HASSAN

soutenue le : **16 décembre 2009**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

Parallélisme et équilibrage de charges dans le traitement de la jointure sur des architectures distribuées

THÈSE DIRIGÉE PAR :

M. Frédéric LOULERGUE	Professeur, Université d'Orléans

RAPPORTEURS :

M^{me}. Anne BENOIT	MCF-HDR, LIP, ENS Lyon
M. Lionel BRUNIE	Professeur, INSA - Lyon

JURY :

M. Gaétan HAINS	Professeur, Université Paris 12, Président du jury
M^{me}. Anne BENOIT	MCF-HDR, LIP, ENS Lyon
M. Lionel BRUNIE	Professeur, INSA - Lyon
M^{me}. Zineb HABBAS	MCF-HDR, Université de Metz
M. Frédéric LOULERGUE	Professeur, Université d'Orléans
M. Mostafa BAMHA	MCF, Université d'Orléans, Co-encadrant

À mon père,
à ma mère,
à la mémoire de ma tante Hana,
à ma soeur et mes frères.

Remerciements

J'aimerais, en premier lieu, remercier vivement Mme Anne BENOIT, Maître de Conférences - HDR à l'ENS de Lyon, et M. Lionel BRUNIE, Professeur à l'INSA de Lyon, pour l'honneur qu'ils m'ont fait en acceptant d'être les rapporteurs de ma thèse. Je vous exprime toute ma reconnaissance pour les remarques pertinentes que vous avez formulées et pour la patience à la lecture du manuscrit.

Mes remerciements s'adressent également à M. Gaétan HAINS, Professeur à l'université Paris 12, et à Mme Zineb HABBAS, Maître de Conférences - HDR à l'université de Metz, pour l'honneur qu'ils m'ont fait en acceptant de bien vouloir participer à ce jury de soutenance.

J'aimerais exprimer toute ma profonde gratitude à M. Frédéric LOULERGUE, Professeur à l'université d'Orléans, pour avoir accepté de diriger cette thèse. Un grand merci pour sa disponibilité et son amabilité. Je tiens également à le remercier pour son soutien et les nombreux conseils qu'il m'a prodigués tout au long de cette thèse.

J'adresse également des vifs remerciements à M. Mostafa BAMHA, Maître de Conférences à l'université d'Orleans, qui a assuré le co-encadrement de cette thèse avec beaucoup de patience et d'enthousiasme. Je tiens tout particulièrement à lui exprimer ma plus profonde gratitude pour son soutien de tous les jours qui m'a permis de mener à terme ces travaux. Son expérience et ses conseils ont été décisifs pour le déroulement de ce travail. Je le remercie pour son extrême gentillesse et sa disponibilité de tous les instants.

Je souhaite aussi remercier Mme Christel VRAIN, la directrice du LIFO, et tous les membres de mon laboratoire et du département d'informatique à l'université d'Orléans pour l'accueil chaleureux.

J'adresse mes remerciements à l'équipe de GRID'5000 qui m'a permis de tester la performance de mes travaux sur la grille.

Un grand merci à mon cousin Nawfal. Merci à tous mes amis de Calais et d'Orléans avec lesquels j'ai passé de très bons moments pendant toutes ces années en France. Je remercie particulièrement Aziz, Hanaa, Oussama, Mirvatte, Katia, Jradeh, Bassem, Ibrahim et Eliane.

J'adresse un grand merci à toute ma famille qui a toujours été présente lorsque j'en ai eu besoin, en particulier, à mon père et ma mère. Je suis là grâce à vos sacrifices, votre soutien et vos encouragements pendant toutes

ces années. Mes remerciements vont aussi à tous mes frères et à ma chère soeur.

Enfin, j'adresse mes remerciements à l'âme de ma tante Hana. Tu es une personne qui a marqué ma vie et tu resteras pour toujours vivante dans mes pensées et dans mon coeur.

Orléans, le 10 décembre 2009.

Table of contents

List of figures

Introduction

1

La performance des processeurs et la capacité de stockage ne cessent de s'améliorer. Cependant, le ralentissement et l'embouteillage au niveau des entrées/sorties sur les disques ne permettent pas d'augmenter indéfiniment la performance des machines séquentielles. Pour cela, depuis le début des années 80, les chercheurs dans le domaine de bases de données ont commencé à s'intéresser au parallélisme. *Bubba* (Boral et al. 1990) et *Gamma* (DeWitt et al. 1990) sont des exemples des systèmes homogènes de bases de données parallèles sur des architectures à disques répartis (*l'architecture Shared Nothing* [1]). L'objectif principal de ces systèmes est l'accélération linéaire des algorithmes parallèles. Cependant, cette accélération n'est pas toujours atteignable à cause des coûts de communication et du déséquilibre probable des charges des différents processeurs en présence du déséquilibre des données (Mourad et al. 1994, Seetha and Yu December 1990, Kitsuregawa and Ogawa 1990, Hua and Lee 1991, Wolf et al. 1994, Hua et al. 1995, DeWitt et al. 1992, Harada and Kitsuregawa 1995, Bamha and Hains 2000; 1999).

L'opération d'*équi-jointure* (ou simplement la *jointure*) est l'une des opérations la plus utilisée dans les *Systèmes de Gestion de Base de Données* (SGBD). La jointure de deux relations R et S sur l'attribut A de R et l'attribut B de S est la relation, $R \bowtie S$, formée de l'ensemble des tuples du produit cartésien $R \times S$ satisfaisant $R.A = S.B$.
La jointure est une opération très coûteuse. Pour cela, sa parallélisation est bien étudiée par le comité de chercheurs dans le domaine de base de données afin d'atteindre des performances acceptables. Des algorithmes ont été proposés dans (Bamha and Hains 2000, Bamha 2005) pour l'évaluation de la jointure sur des architectures Shared Nothing homogènes. Ces algorithmes garantissent une accélération presque linéaire même en présence de déséquilibre de données tout en réduisant fortement les coûts de communication.

Cependant, d'autres types de requêtes telles que le "Group-By Join", souvent utilisées dans les systèmes interactifs d'aide à la décision, le traitement analytique en ligne (OLAP) et les entrepôts de données, n'ont pas été bien étudiées par les chercheurs. Ces systèmes sont largement utilisés dans les services bancaires et financiers, les grands distributeurs de produits alimentaires, la fabrication de produits de grande consommation,

[1] l'architecture Shared Nothing : une architecture où chaque processeur dispose de sa propre mémoire et de ses propres disques.

les télécommunications et le transport. Ils aident les analystes et les gestionnaires à analyser des données et à prendre des décisions. Ils stockent des bases de données contenant des informations historisées de tailles importantes. En plus, ces systèmes permettent d'effectuer des analyses complexes sur ces données et d'obtenir des données agrégées (Han and Kamber 2000). A cette fin, les fonctions d'agrégat sont souvent utilisées. Elles peuvent être appliquées au résultat de la jointure de plusieurs relations constituées de milliards de tuples. Par ailleurs, le temps d'exécution de ces fonctions doit rester raisonnable. Il est donc nécessaire d'exploiter le parallélisme (Liang and Orlowska 2000) en utilisant des algorithmes parallèles qui soient efficaces même en présence de déséquilibre de données tout en réduisant au minimum les coûts de communication entre les différents processeurs et les coûts de lecture/écriture sur les disques des résultats des jointures intermédiaires (Gupta et al. 1995, Li et al. 2005, Taniar and Rahayu 2001).

Aujourd'hui, avec l'émergence des applications telles que la physique des hautes énergies, la modélisation climatique et la bioinformatique qui sont des applications intensives en manipulation de données, des chercheurs géographiquement répartis sur plusieurs sites ont besoin d'accéder et d'interroger des bases de données de tailles énormes. Ces bases de données sont distribuées sur les sites de plusieurs organisations. Les requêtes exécutées par ces chercheurs exigent également le transfert d'un volume énorme de données sur le réseau dans un délai raisonnable. Pour des telles applications, les systèmes parallèles ne sont pas suffisants. Pour cela, Foster et Kesselman ont introduit, en 1998, dans leur livre *The Grid : Blueprint for a Future Computing Infrastructure* (Foster and Kesselman 1999) la notion de grille. Ils ont défini la grille comme étant *une infrastructure matérielle et logicielle qui fournit un accès fiable, cohérent, omniprésent, peu coûteux et avec une très grande capacités de calcul*. Cependant, la volatilité et l'hétérogénéité de nœuds, peuvent dégrader la performance des algorithmes sur la grille bien qu'ils sont très efficaces pour l'évaluation de la jointure sur des architectures *Shared Nothing* (SN) homogènes.

Les applications web telles que *Google Analytics* [2], *Google Earth* [3], *Personalized Search* [4] et *Facebook* [5] ont besoin de stocker de volumes de données importants et d'appliquer des opérations de recherche et d'extraction sur ces données. Pour répondre à ces besoins de stockage et de manipulation de données, des systèmes de stockage distribués de données structurées comme BigTable (Chang et al. 2006) et HBase[6] sont utilisés. Ces systèmes s'appuient sur les systèmes de fichiers distribués comme *Google File System* (GFS) (Ghemawat et al. 2003) et *Hadoop Distributed File System* (HDFS)[7]. Le modèle de programmation parallèle *Map-Reduce* permet d'effectuer des calculs parallèles et distribués sur les données stockées dans le DFS

[2]Google Analytics : `http://www.google.com/analytics/` .
[3]Google Earth : `http://earth.google.com/` .
[4]Personalized Search : `www.google.com/psearch` .
[5]Facebook : `http://www.facebook.com/` .
[6]HBase : `http://hadoop.apache.org/hbase/` .
[7]HDFS : Hadoop Distributed File System, `http://hadoop.apache.org/hdfs/` .

(Distributed File System) (Dean and Ghemawat 2004).

Dans cette thèse, nous nous intéressons particulièrement :

- *au traitement de requête "Group-By Join" sur des architectures Shared Nothing*. Les algorithmes que nous proposons garantissent une accélération presque linéaire même en présence d'un fort déséquilibre de données. Dans ces algorithmes, le résultat intermédiaire de jointure n'est pas matérialisé ainsi le coût des opérations de lecture/écriture sur les disques est réduit. En plus, le coût de communication est fortement réduit car seuls les tuples qui participent effectivement à la jointure sont redistribués.
- *au traitement de la jointure et multi-jointures sur des architectures distribuées hétérogènes*. Nous proposons une approche dynamique de distribution des tâches sur les processeurs qui prend en compte les caractéristiques et la charge de chaque processeur afin d'avoir un temps d'exécution presque équitable sur tous les processeurs durant toutes les étapes du traitement. Dans cette approche, les paquets de données non traités sur un processeur chargé sont transférés vers d'autres processeurs moins chargés dans le but de traiter d'une manière très efficace l'effet du déséquilibre des valeurs de l'attribut de jointure et du déséquilibre du résultat de la jointure. Les coûts de communication et de lecture/écriture sur le disque sont également fortement réduits car seuls les tuples qui participent effectivement au résultat de la jointure sont redistribués.
- *au traitement de jointure sur des architectures grille*. L'algorithme qu'on propose optimise l'allocation des ressources et minimise le coût de communication sur la grille. Cet algorithme est aussi insensible au déséquilibre de données et à la volatilité des machines.
- *au traitement de l'opération de semi-jointure sur les Distributed File Systems* en utilisant le modèle de programmation *Map-Reduce-Merge* (Yang et al. 2007) et les histogrammes distribués.

L'analyse théorique de complexité et les tests réalisés confirment une accélération presque linéaire pour nos algorithmes.

Organisation du mémoire

Dans le chapitre 2, nous présentons les notions de base dans les domaines de base de données et des architectures parallèles et distribuées. Nous commençons par un rappel sur les opérations et les requêtes dans les systèmes de gestion de base de données. Après avoir présenté les différentes architectures des systèmes de base de données parallèles, nous faisons une présentation des modèles de programmation parallèle avec une justification de notre choix du modèle du coût BSP (Bulk Synchronous Parallel) pour l'analyse de la complexité de nos algorithmes sur les systèmes parallèles et distribués. Nous présentons aussi la grille de calcul et le *cloud computing* avec une comparaison entre ces deux architectures.

Dans le chapitre 3, nous présentons l'état de l'art de travaux sur le traitement de la jointure et des requêtes de "Group-By Join" sur les architectures parallèles et distribuées. Nous commençons par une pré-

sentation des algorithmes parallèles basiques pour traiter la jointure sur des architectures parallèles homogènes ainsi que le problème du déséquilibre des données qui peut mener à la dégradation des performances de ces algorithmes. Nous faisons aussi un tour d'horizon des algorithmes d'équilibrage de charges, présentés dans la littérature, pour le traitement de la jointure sur des architectures *Shared Nothing* homogènes. Cependant, la performance de ces algorithmes peut se dégrader sur les architectures distribuées hétérogènes et sur la grille. Cela est dû aux caractéristiques hétérogènes des nœuds dans ces deux architectures et leur volatilité dans la grille. Pour cela, des algorithmes adaptatifs ont été proposés dans la littérature pour rééquilibrer les charges des nœuds tout au long de l'évaluation de la jointure. Nous présentons aussi le modèle *Map-Reduce-Merge* utilisé dans le *cloud computing* pour le traitement de la semi-jointure.

Dans le chapitre 4, nous présentons une nouvelle approche pour le traitement des requêtes de jointure avec regroupement dans deux algorithmes : *GBJFA-Join* utilisé pour le traitement de requêtes de "Group-By Join" sur des architectures SN homogènes dans le cas où les attributs de jointure figurent également parmi les attributs du Group-By et *GAJFA-Join* algorithme utilisé quand les attributs de jointures et du group-by sont distincts. Ces algorithmes sont basés sur l'utilisation des histogrammes distribués. Dans nos algorithmes, l'histogramme d'une relation R sur l'attribut de la jointure x est la liste des couples (v, n_v) où n_v est le nombre de tuples de R ayant la valeur v pour x. Ces histogrammes sont organisés sous la forme d'arbre balancé (B^+-tree) pour accélérer les opérations de recherche. En plus, ils sont totalement distribués, car dans toutes les phases de nos algorithmes, ils sont répartis sur tous les nœuds, et jamais centralisés sur un seul nœud. Ils nous donnent une connaissance détaillée de la distribution des valeurs de l'attribut de jointure ainsi que de la distribution du résultat de jointure. Ils sont, donc, très efficaces pour le traitement de requête de "Group-By Join" même en présence de déséquilibre des données. Nous avons réussi à réduire à la fois le coût de communication et de lecture/écriture sur les disques car dans nos algorithmes :

- l'utilisation des histogrammes nous permet de trouver les tuples qui participent effectivement au résultat de la jointure. Ainsi, seuls ces tuples sont redistribués, ce qui réduit les coûts de communication. L'utilisation des histogrammes permet de réduire également le nombre de paquets de données et par conséquent le coût total du traitement de la jointure.
- la fonction d'agrégat est partiellement évaluée avant la redistribution des données. Cela aide à réduire la taille des opérandes de la jointure et par conséquent le coût de la phase de redistribution des données ainsi que le temps global de traitement.
- les résultats intermédiaires de jointure ne sont pas matérialisés. Ceci aide également à réduire les coûts inutiles de lecture/écriture des données sur les disques.

Les performances de ces algorithmes ont été étudiées en utilisant le modèle de coût BSP qui prévoit une accélération presque linéaire pour nos algorithmes. Les tests réalisés ont confirmé la validité de cette prévision.

Dans le chapitre 5, nous proposons l'algorithme *DFA-Join* pour l'évaluation de la jointure sur les systèmes distribués hétérogènes et multi-utilisateurs. Cet algorithme est basé sur une technique parallèle de distribution de charges en deux-étapes : La première étape est statique où chaque processeur reçoit une charge proportionnelle à sa capacité. La deuxième est dynamique ainsi, les charges des processeurs surchargés sont transférées vers les processeurs les moins chargés. Nous présentons aussi, l'algorithme *PDFA-Join*, une version pipelinée de *DFA-Join* pour l'évaluation des requêtes complexes avec multi-jointures. Nous montrons aussi que *PDFA-Join* peut être appliqué de manière efficace dans différentes stratégies d'exécution parallèles permettant d'exploiter non seulement le parallélisme intra-opérateur mais aussi le parallélisme inter-opérateur pipeliné. Le modèle de coût et l'étude expérimentale effectuée sur la plate-forme Grid'5000 ont montré que nos algorithmes ont une accélération presque linéaire sur des architectures hétérogènes même en présence d'un fort déséquilibre des données.

Dans le chapitre 6, nous présentons l'algorithme *GDFA-Join* pour l'évaluation de la jointure sur les grilles. Cet algorithme est une variante de l'algorithme *DFA-Join*. Il prend en compte la réplication possible de données sur plusieurs nœuds pour réduire le coût de traitement de la jointure. Il est aussi conçu pour tolérer les pannes d'un ou de plusieurs nœuds. L'étude de coût de l'algorithme *GDFA-Join* et l'étude expérimentale ont montré qu'il a une accélération presque linéaire même en présence de déséquilibre des données.

Dans le chapitre 7, nous étendons les travaux réalisés sur les jointures au traitement des semi-jointures sur une architecture *Cloud*. Dans ce cadre, nous présentons l'algorithme *CFA-semi-Join* pour l'évaluation de la semi-jointure sur les systèmes de fichiers distribués. La *semi-jointure* de S par R est la relation $S \ltimes R$ composée des tuples de la relation S qui apparaissent dans la jointure de R et S ($R \bowtie S$). L'algorithme *CFA-Semi-Join* est basé sur le modèle de programmation *Map-Reduce-Merge* et l'utilisation des histogrammes distribués. Comme dans nos algorithmes précédents, l'utilisation des histogrammes nous permet de déterminer les tuples qui apparaissent dans le résultat final de la jointure, et seulement ces tuples sont redistribués. Donc, les coût de communication et de lecture/écriture sur disques sont fortement réduit, tout en traitant de manière efficace le problème du déséquilibre des résultats de jointure. L'étude de coût de chaque phase de l'algorithme *CFA-Semi-Join* montre qu'il a une accélération presque linéaire même en présence de déséquilibre des données.

La conclusion ainsi que nos perspectives de travail sont présentées dans le chapitre 8.

General Notions

2

Contents

In this chapter, we review the basic operations in the Database Management Systems (DBMS) such as the join, semi-join and group-by in addition to the aggregate functions. Then, we describe the basic parallel database architectures and parallel programming models. We also present the grid infrastructure with some existing middlewares and systems. Finally, we review cloud computing and compare it with grid computing.

2.1 Relational database model

The Relational database was born in 1970 with Codd's paper *"A Relational Model of Data for Large Shared Data Banks"* (Codd 1970). A relational database is formed of one or multiple tables or relations. Each table is formed of rows known as tuples describing one or more data categories in columns. Data stored in these tables is manipulated using a Structured Query Language (SQL). These queries allow us to apply several operations on the relations such as join, projection, restriction, group by and aggregate functions.

2.1.1 Basic operations in database management systems

In this section, we will review the *join* and *semi-join* operations. Then, we will give the different kinds of queries involving aggregate functions, group by and/or join operations mainly *simple aggregate* and *"GroupBy-Join" queries*.

We will use the following relations that represent respectively: the suppliers, the products and quantity of a product shipped by a supplier in a specific date.

```
SUPPLIER (Sid, Sname, City)
PRODUCT (Pid, Pname, Category)
SHIPMENT (Sid, Pid, Date, Quantity)
```

Join operation

The θ-join of two relations R and S on attribute A of R and attribute B of S (A and B of the same domain) is the relation written as $R \bowtie_{A\ \theta\ B} S$, containing the pairs of tuples from R and S for which $R.A\ \theta\ S.B$ where $\theta \in \{=, \neq, <, >, \leq, \geq\}$. If the operator θ is the equality operator, then this operation is called an *equi-join*. In general, join refers to equi-join.

Figure 2.1 is an example of the join of *SUPPLIER* relation and *SHIPMENT* relation on the common attribute *Sid*. This query is written in *SQL* (Structured Query Language) as:

```
SELECT SUPPLIER.Sid, Sname, City, Pid, Date, Quantity
FROM SUPPLIER, SHIPMENT
WHERE SUPPLIER.Sid = SHIPMENT.Sid;
```

To find the final result, each two tuples of *SUPPLIER* and *SHIPMENT* having respectively the same values of *Sid* attribute are concatenated to form one tuple in $SUPPLIER \bowtie SHIPMENT$.

Semi-join operation

The semi-join of S by R is the relation $S \ltimes R$ composed of the tuples of S which occur in the join of R and S.

The semi-join reduces the size of the join operands and satisfies:

$$R \bowtie S = R \bowtie (S \ltimes R) = (R \ltimes S) \bowtie (S \ltimes R).$$

Figure 2.2 shows the result of $SUPPLIER \ltimes SHIPMENT$.

SUPPLIER

Sid	Sname	City
1000	Dupont	Paris
1001	Durand	Orléans
1002	Mitchell	Lille
1003	Picard	Orléans
1004	Daniel	Marseille
1005	Mitchell	Calais
1006	Picard	Lyon

SHIPMENT

Sid	Pid	Date	Quantity
1000	20045	13/10/2008	700
1000	20135	10/01/2009	300
1004	40984	14/02/2009	550
1004	35468	20/02/2009	430
1004	98345	20/02/2009	800
1005	87935	15/04/2009	900
1005	24356	20/05/2009	250

SUPPLIER $\bowtie$ *SHIPMENT*

Sid	Sname	City	Pid	Date	Quantity
1000	Dupont	Paris	20045	13/10/2008	700
1000	Dupont	Paris	20135	10/01/2009	300
1004	Daniel	Marseille	40984	14/02/2009	550
1004	Daniel	Marseille	35468	20/02/2009	430
1004	Daniel	Marseille	98345	20/02/2009	800
1005	Mitchell	Calais	87935	15/04/2009	900
1005	Mitchell	Calais	24356	20/05/2009	250

FIG. 2.1 – *Join operation example.*

SUPPLIER $\ltimes$ *SHIPMENT*

Sid	Sname	City
1000	Dupont	Paris
1004	Daniel	Marseille
1005	Mitchell	Calais

FIG. 2.2 – *Semi-join operation*: $SUPPLIER \ltimes SHIPMENT$.

Simple aggregate queries

In this category, we have queries involving *scalar aggregate* or *Group-By queries*. In scalar aggregate queries, the result is a single value related to all the tuples of the relation. Whereas, Group-By queries produce a single value for each group of tuples of a relation.

The following query is an example of scalar aggregate queries allowing to find the total amount of products shipped by all the suppliers using the aggregate function *SUM* [1].

```
SELECT SUM(Quantity) FROM SHIPMENT;
```

So, in this query, we simply find the sum of the values related to attribute *Quantity* of all the tuples of relation *SHIPMENT*.

[1] The main aggregate functions in DBMS are: SUM, COUNT, AVG, MIN and MAX.

An example of Group-By query is:

```
SELECT Sid, SUM(Quantity)
FROM SHIPMENT
GROUP BY Sid;
```

In this query, we compute the sum of values related to the attribute *Quantity* of each group of tuples having the same value of *Sid*. This grouping is permitted by using the *Group By* clause. So, the above query is used to find the total quantity of the products shipped by each supplier.

In all the previous queries, the aggregate functions were used in the projection part of the relational queries, but they may also be used in the *HAVING* part, for example:

```
SELECT Sid, SUM(Quantity)
FROM SHIPMENT
GROUP BY Sid
HAVING SUM(Quantity) >200;
```

In such queries, the *having* part is treated after the *group by* part (Shatdal and Naughton 1994).

"GroupBy-Join" queries

As we have seen in *simple aggregate* queries, aggregate functions can be applied on the tuples of a single table. But, in most SQL queries they are applied on the output of the join of multiple relations. Such queries are known as *"GroupBy-Join" queries*. We can distinguish two types of "GroupBy-Join" queries illustrated using the following example.

Query 1:

```
SELECT p.Pid, p.Pname, SUM (Quantity)
FROM PRODUCT as p, SHIPMENT as s
WHERE p.Pid = s.Pid
GROUP BY p.Pid, p.Pname ;
```

Query 2:

```
SELECT p.Category, SUM (Quantity)
FROM PRODUCT as p, SHIPMENT as s
WHERE p.Pid = s.Pid
GROUP BY p.Category ;
```

The purpose of *Query*1 is to find the total quantity of each product shipped by all the suppliers, while that of *Query*2 is to find the total amount of each category of product shipped by all the suppliers. The difference between *Query*1 and *Query*2 lies in the group-by and join attributes. In *Query*1, the join attribute (*Pid*) is part of the group-by attributes. This is not the case in *Query*2 where group-by and join attributes are totally different. This difference plays an important role in "GroupBy-Join" query processing especially in PDBMS. Where for *Query*1, it is preferable to

carry out the group-by and aggregate functions first and then the join operation (Taniar et al. 2000, Taniar and Rahayu 2001). This helps in reducing the size of the relations to be joined. As a consequence, applying the group-by and aggregate functions before the join operation in PDBMS results in a huge gain in the communication cost and thus in the execution time of such queries. In the contrary, group-by cannot be applied before the join on *Query*2, because the join attribute (*Pid*) is different from the group-by attribute (*category*).

2.1.2 Join processing on mono-processor systems

Three main sequential algorithms, for evaluating the join operation on mono-processor systems, are known:

- Nested loop join,
- Sort-Merge join,
- Hash based join.

We will review, in the following subsection, these algorithms with their execution costs. We consider that we want to find the join of two relations R and S and that the common join attribute is x. The term $||R||$ (resp. $||S||$) represents the number of tuples of R (resp. S) and $|R|$ (resp. $||S||$) is the size (expressed in bytes or number of pages) of R (resp. S). The term $c_{r/w}$ represents the cost to read/write a page of data from/to disk. In all these algorithms, R and S are read from disk with a cost: $O\big(c_{r/w} * (|R| + |S|)\big)$ and the join result $R \bowtie S$ is written to disk with a cost $O(c_{r/w} * |R \bowtie S|)$. Thus, the total disk access cost is: $O\big(c_{r/w} * (|R| + |S| + |R \bowtie S|)\big)$.

Nested loop join algorithm

This is the simplest join algorithm. In this algorithm (Algorithm. 1), for each tuple of R all the tuples of S are scanned. Each two tuples of input relations that satisfy the join condition are appended to form the join result.

Algorithm 1: Nested loop join algorithm.

```
▷ for each tuple r in relation R do
    ▷ for each tuple s in relation S do
        ▷ if r.x = s.x then
            ▷ write the tuple < r, s > to the join result;
        ▷ endif
    ▷ endfor
▷ endfor
```

This algorithm induces $||R|| * ||S||$ join attribute values comparisons. Thus, the total result of computing the join using the Nested loop join algorithm is:

$$O(c_{r/w} * (|R| + |S| + |R \bowtie S|) + t_{comp} * ||R|| * ||S||),$$

where t_{comp} is the needed time to compare the join attribute values related to two tuples of relations R and S.

Sort-Merge join algorithm

This is a two-step algorithm. In the first step, the two relations are sorted based on the join attribute. Then, in the second step, the two sorted tables are merged in order to find the final join result as shown in Algorithm. 2. The costs of sorting R and S are respectively: $O(||R|| * \log(||R||))$ and $O(||S|| * \log(||S||))$ (Taniar et al. 2008b). The cost of merging the two relations is: $O(||R|| + ||S||)$. So, the total cost of computing the join using the Sort-Merge join algorithm is:

$$O(c_{r/w} * (|R| + |S| + |R \bowtie S|) + ||R|| * \log(||R||) + ||S|| * \log(||S||) + ||R|| + ||S||).$$

Algorithm 2: Sort-Merge join algorithm.

▷ Sort tuples of relation R;
▷ Sort tuples of relation S;
▷ Read a tuple r of R;
▷ Read a tuple s of S;
▷ **while** $r \neq EOF$ or $s \neq EOF$ **do**
  ▷ **if** $r.x = s.x$ **then**
    ▷ write the tuple $< r, s >$ to the join result;
    ▷ read the next tuple r of R;
    ▷ read the next tuple s of S;
  ▷ **else if** $r.x > s.x$ **then**
    ▷ read the next tuple s of S;
  ▷ **else**
    ▷ read the next tuple r of R;
  ▷ **endif**
▷**endwhile**

Hash based join algorithm

Several hash based algorithms are presented in the literature such as *Grace hash* and *Hybrid hash join*. The main idea behind these algorithms is to (*a.*) hash the smallest relation, known as *build relation*, in order to create a hash table using a hash function applied on the values of the join attribute, (*b.*) then probe the largest relation, known as *probe relation*, as shown in Algorithm 3.
The cost of this algorithm is of the order:

$$O((c_{r/w} * (|R| + |S| + |R \bowtie S|) + t_h * ||R|| + t_s * ||S||),$$

where t_s is the time needed to search for an entry in the hash table and t_h is the time to add an entry to the hash table. In this algorithm, probing can be done at a constant cost, since the inner relation (R) is in main memory after the build phase and has a hash access path on the join attribute (Zeller and Gray 1990). Whenever the hash tables cannot fit in memory, input relations are partitioned into buckets and written back to disks. So, the above algorithm can be applied without memory shortage.

Algorithm 3: Hash based join algorithm.

▷ Hash function $h()$;
▷ /* Build the hash table of the smallest relation R. */
▷ **for each** tuple r of relation R **do**
  ▷ write r into the hash table with index entry $h(r.x)$;
▷ **endfor**
▷ /* Probe S */
▷ **for each** tuple s of relation S **do**
  ▷ probe the hash table for index $h(s.x)$;
  ▷ **if** index $h(s.x)$ exists in the hash table **then**
    ▷ **for each** record r' with index entry $h(s.x)$ **do**
      ▷ **if** $r'.x = s.x$ **then**
        ▷ write the tuple $< r', s >$ to the join result ;
      ▷ **endif**
    ▷ **endfor**
  ▷ **endif**
▷ **endfor**

2.2 Parallel Architectures and Programming models

Flynn (Flynn 1966) has proposed in 1966 a taxonomy of computer architectures. He based his taxonomy on the notion of instructions and data streams that can be simultaneously treated by the machine. The term *stream* refers to a sequence of either instructions or data. The four categories of Flynn's taxonomy are :

- *SISD machine* : Single Instruction Single Data machine. This is a conventional serial computer (uniprocessor) that processes only one stream of instructions and one stream of data. SISD machine is also known as *von Neumann computer*.
- *SIMD machine* : Single Instruction Multiple data machine. In this category, machines have multiple processing units under the supervision of a single control unit. At any given time, all the processing units synchronously execute the same instruction but on different data streams. ILLIAC-IV is an example of SIMD machine.
- *MIMD machine* : Multiple Instructions Multiple Data machine. A MIMD machine is formed of multiple processors each executing its own instruction stream to process its allocated data stream. MIMD machine exploits asynchronous parallelism. Cray X-MP, Tandem/16 and Meiko Computing Surface (CS-1) are examples of MIMD machines.
- *MISD machine* : Multiple Instructions Single Data machine. A MISD machine is formed of multiple processors executing independent streams of instructions on the same data stream. The C.mmp built at Carnegie-Mellon University by William Wulf on 1971 is an example of MISD machine.

In the rest of this section, we will review three main parallel database architectures that fall under the MIMD machines. Then, we present several parallel programming models that can be followed to implement parallel

algorithms on parallel systems. These models also allow us to study the execution cost of the algorithms using their cost models.

2.2.1 Parallel database Architectures

Parallel database architecture can be divided into three principal categories: *shared memory*, *shared disk*, and *shared nothing* architectures.

Shared-Memory architectures

In the *Shared Memory* (SM) architectures, all the processors have a direct access to a common shared main memory and to all shared disks using a bus interconnection network as shown in figure 2.3.

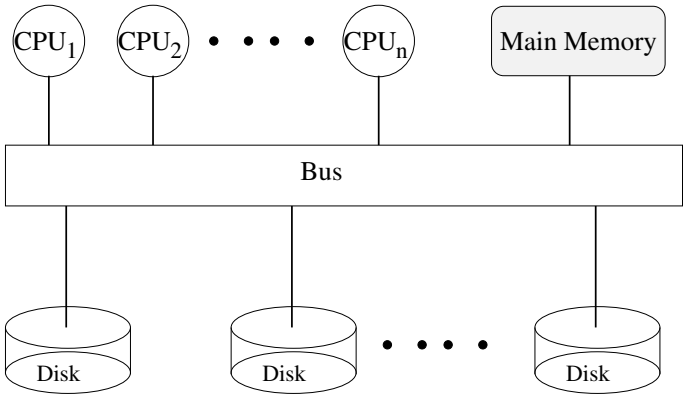

FIG. 2.3 – *A Shared-Memory architecture.*

In this architecture, the choice of the degree of parallelism is flexible and load balancing is easy to achieve because all needed data are located in a central shared memory. In addition, the communication cost is not high since all the processors have a direct access to the common data located in the central main memory.
This architecture has a main drawback: it is limited to a small number of processors (Taniar et al. 2008a). This is due to the following facts:

- many processors may need to access shared data which results in memory and bus contention,
- the physical bus interconnection network has a limited capacity of data transfer. So processor-to-memory connection becomes a bottleneck since the bus is shared by all the processors.

To conclude, load balancing in shared memory architectures is easy to achieve. However, these systems suffer from availability and scalability limitation problems (Rahm 1996).

Shared-Disk architectures

In the *Shared Disk architectures* (SD), each processor has its own local main memory while all the processors share the secondary memory as shown in figure 2.4. So, each processor can access any disk in the system. However, a processor can only read/write data in its local main memory but not in the local memory of other processors. In this architecture, communications

are performed throughout a high speed interconnection bus.

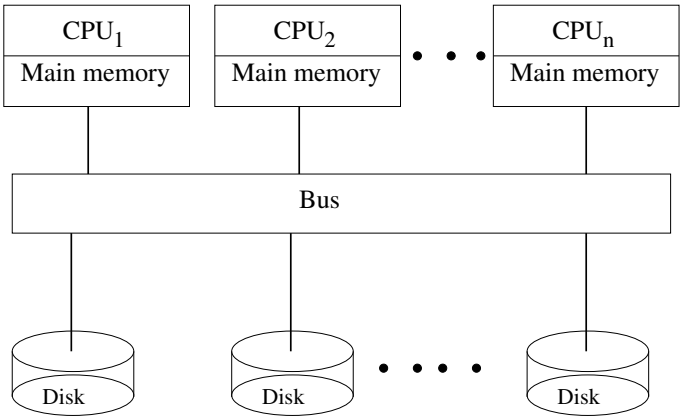

FIG. 2.4 – *A Shared-Disk architecture.*

The main advantage of shared disk architectures lies in the flexibility of choosing the degree of parallelism due to sharing the data stored in disks by all the processors. Load balancing between processors is easily achieved because intermediate data are available to all nodes through the shared disks. In addition, data sharing problems, in this architecture, is minimized because each processor stores its active data in its local memory. On the other hand, this architecture suffers from communication and disk bottleneck if many processors need to read/write data on the shared disks. Since the main memory is not shared by the processors, each processor has its own data cache. Hence, a cost overhead is needed to maintain the cache consistency.

Shared-Nothing architectures

In a *Shared-Nothing* (SN) architecture, each processor has its own local main memory and disks as shown in figure 2.5. So a processor cannot directly access neither the main memory nor the disks of other processors. Communication between processors is established by message passing through an interconnection network.

In this architecture, the database is partitioned over the processors and

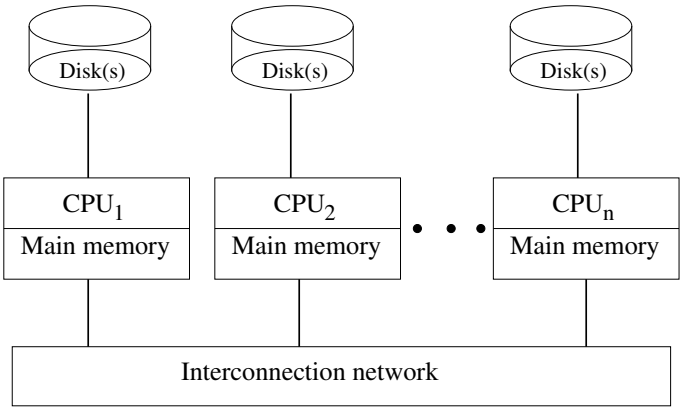

FIG. 2.5 – *A Shared-Nothing architecture.*

each processor can have a direct access to data stored in its local disk.

To access data of other processors, a distributed query and transaction execution is needed (Rahm 1996).

The main advantage of SN architecture is its scalability due to the fact that each processor is independent of the others. However, in this architecture, it is not easy to achieve load balancing between processors. It also suffers from high communication costs.

2.2.2 Parallel programming models

Several parallel programming models were presented in the literature. PRAM model is one of the first introduced ones. This model considers shared memory and uniform memory access. This is not realistic in parallel programming where communication cost is one of the most important factors. For this reason, other portable models such as BSP, LogP, CGM and EM-BSP were invented. These models take into account this factor. We review these models in the rest of this section.

The PRAM model

Forturne et al. (Fortune and Wyllie 1978) have introduced the Parallel Random Access Machine (PRAM) model for parallel computing. The PRAM model consists of p processors. Each one has its own private local memory, and they all share a global memory. In PRAM model, processors execute the computation instructions synchronously. At each step of a PRAM algorithm, some processors are active and execute: read, write or compute instructions. The other processors are inactive. In a read step, each active processor reads one global memory location into its local memory. In a compute step, each active processor executes a single operation and writes the result into its local memory. In a write step, each active processor writes one local memory location into the global memory.

It was necessary to define some memory access restrictions to resolve read and write conflicts to the same shared memory location. Depending on the restrictions on memory access, we have 4 different PRAM models:

- **Exclusive Read, Exclusive Write (EREW) PRAM**: At each time step, one and only one processor can read or write the same shared memory location.
- **Concurrent Read, Exclusive Write (CREW) PRAM**: At each time step, simultaneous reads of the same memory location are allowed, but only one processor can write to a shared memory location.
- **Concurrent Read, Concurrent Write (CRCW) PRAM**: At each time step, both simultaneous reads and writes of the same memory location are allowed. In CRCW PRAM model, we also need to specify what happens when several processors write to the same memory locations.
- **Queue Read, Queue Write (QRQW) PRAM** (Gibbons et al. 1994; 1999): At each time step, each memory location can be read or written by any number of processors. Concurrent read or write to a location are serviced one-at-a-time. The access time to read or write

a location is proportional to the number of concurrent readers or writers to the same location.

The BSP model

Bulk-Synchronous Parallel (BSP) model is a programming model introduced by L. Valiant (Valiant August 1990). It offers a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of multi-processor architectures (Skillicorn et al. 1997). A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed:

- the number of processor-memory pairs p,
- the time l required for a global synchronization,
- the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word).

The network is assumed to deliver an h-relation in time $g \times h$ for any arity h.

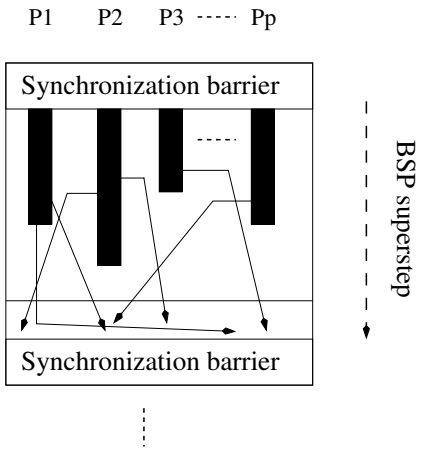

FIG. 2.6 – *A BSP superstep.*

A BSP program is executed as a sequence of *supersteps*, each one divided into (at most) three successive and logically disjoint phases. In the first phase, each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase, the network delivers the requested data transfers. And in the third phase, a global synchronization barrier occurs, making the transferred data available for the next superstep. The execution time of a superstep s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$\text{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l$$

where $w_i^{(s)}$ is the local processing time on processor i during superstep s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words

transmitted (resp. received) by processor i during superstep s. The execution time, $\sum_s \text{Time}(s)$, of a BSP program composed of S supersteps is therefore a sum of 3 terms: $W + H * g + S * l$ where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$. In general W, H and S are functions of p and of the size of data n, or (as in the present application) of more complex parameters like data skew and histogram sizes. To minimize execution time of a BSP algorithm, its design must jointly minimize the number S of supersteps and the total volume H (resp. W). In addition, for each superstep s, the volume $h_i^{(s)}$ (resp.$w_i^{(s)}$) must be balanced on all processors.

The LogP model

LogP, described in (Culler et al. 1993), is a distributed memory multiprocessor model where processors communicate by point-to-point messages. The model parameters are:

- **L**: an upper bound on the *latency* incurred in communicating a message containing a word (or small number of words) from its source module to its target module.
- **o**: the processor time *overhead* required to transmit or receive a message, during which the processor cannot perform other operations.
- **g**: the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. $\frac{1}{g}$ corresponds to the available per-processor communication bandwidth.
- **P**: the number of processor/memory couples.

The terms L, o and g parameters are measured as multiples of the processor cycle. In LogP model, processors work asynchronously and at most $\lceil \frac{L}{g} \rceil$ messages can be in transit, on the network, at any time. In LogP model, sending a small message (a datum) from one processor to another requires a time of $L + 2 * o$. Sending a long message formed of k bytes, by point-to-point messages, requires sending $\lceil \frac{k}{w} \rceil$ in $2 * o + (\lceil \frac{k}{w} \rceil - 1) * \max\{g, o\} + L$ cycles, where w is the underlying message size of the machine.

LogP model only deals with short messages. So, an extension of this model, named LogGP was described in (Alexandrov et al. 1995) to model small and long messages communication. LogGP extends LogP model where a G parameter is added. This parameter captures the bandwidth obtained for long messages and $\frac{1}{G}$ represents the available per processor communication bandwidth for long messages. Thus, sending a k byte message, in LogGP model, requires $2 * o + (k - 1) * G + L$. Other LogP extensions, such as (Löwe et al. 1997), were also presented in the literature for the same aim.

BSP model can be efficiently simulated by LogP and vice-versa. However, Bilardi et al. (Bilardi et al. 1996) claim that BSP is somewhat preferable to LogP due to its greater simplicity and portability. However, it was shown in (Bilardi et al. 1996, Eisenbiegler et al. 1998), that from an

asymptotic point of view, the two models are equivalent.

The CGM model

Dehne et al. have introduced in (Dehne et al. 1993), the *Coarse Grained Multicomputer* (CGM) model. A $CGM(n, p)$ machine is formed of p processors with $O(\frac{n}{p})$ local memory for each processor. The size of the local memory of each processor is larger than $O(1)$. The processors are connected using an interconnection network where each processor can exchange messages of size $O(\log n)$ with any one of its immediate neighbors in constant time. A CGM algorithm is a sequence of phases, alternating between local computing and global communication. The CGM model represents a special case of the BSP model where all the communication operations of one superstep are done in the h-relation ($h = O(\frac{n}{p})$) (Alves et al. 2002), i.e. each processor sends $O(\frac{n}{p})$ data and receives $O(\frac{n}{p})$ data.

The External Memory BSP (EM-BSP) model

In applications such as astronomical simulation, weather prediction, geographic information systems, computational biology, virtual reality, 3D simulation and modeling and genetic sequencing, the size of internal memory of the computer is only a small fraction of the problem size. External Memory (EM) algorithms are needed for such applications because the size of the main memory is always insufficient (Dehne et al. 1997; 2002). Parallel processing represents an important issue for EM algorithms for the same reasons that parallel processing is of practical interest in non-EM algorithm design (Dehne et al. 1997; 2002, Gava 2005). However, to obtain an acceptable performance, blockwise access to data and fully parallel disk I/O, when more than one disk is present, are critical issues in EM-algorithms (Dehne et al. 2002).

EM-BSP model, an extension of BSP model, was presented in (Dehne et al. 1997) to include secondary local memories. In this model, each processor has, in addition to its main memory, an external memory formed of a set of disks. EM-model has, in addition to the BSP parameters, the following ones:

- M is the local memory size of each processor,
- D is the number of drives for each processor,
- B is the transfer block size of a local disk drive,
- G is the ratio of local computational capacity (number of local computation operations) divided by the local I/O capacity (number of blocks of size B that can be transfered between the local disks and memory per unit time).

The model is restricted to the case where all processors have the same number of disks because it is mostly the case in practice. Each disk drive consists of a sequence of tracks. The tracks can be accessed by direct random access using their unique track number. Each processor can use all of its D disk drives concurrently, and transfer $D * B$ items from/to the local disks to/from its local memory in a single I/O operation and at cost

G. A processor is assumed to store in its main memory at least one block from each local disk at the same time, i.e. $M \geq D * B$.

As in BSP model, the computation on the EM-BSP model proceeds in a succession of supersteps. EM-BSP model adapts the communication and computation supersteps from the BSP model and allows multiple disk I/O operations during the computation phase of the superstep. For an h-relation, $g * h + L$ time units are charged per communication superstep. The I/O cost of a computation superstep is $t_{I/O} = \max_{j=1}^{v} w_{I/O}^{j}$ where $w_{I/O}^{j}$ is the I/O cost incurred by processor j. Each I/O operation costs G time steps. For a computation superstep with at most t_{comp} local operations, on each processor, $t_{comp} + t_{I/O} + L$ time units are charged. Thus, the total cost of each superstep is: $t_{comp} + t_{comm} + t_{I/O} + L$.

Chosen Model

We have chosen to use the BSP cost model to study the execution time of each step of our parallel algorithms on parallel architectures. This choice was due to the simplicity and portability of this model on different parallel architectures. In addition, its parameters g, p and l allow to describe the parallel machine characteristics and give a realistic performance prediction of the execution cost of the algorithms. For our algorithms, the exchanged messages have a large size, so using LogP model makes our cost analysis more complex without guaranteeing better performance or prevision. The CGM model is not adequate with our algorithms because data redistribution and load rebalance of processors are done in the h-relation with unpredictable value of h. The EM-BSP model, would have been a good choice for us. However, in our algorithms, we consider that each processor reads/writes data from/to one disk only. So, using the EM-BSP model will complicate the analysis cost without providing better performance prevision for our algorithms. Thus, to simplify the cost analysis, we used, in addition to the BSP parameters, a parameter $c_{r/w}^{i}$ that represents the disk Input/Output cost of each processor and allows to simulate the case of multiple disks attached to a processor.

2.3 The GRID

Foster gave in his paper, *"What is the Grid ? A Three Point Checklist"* (Foster 2002), three points to specify if a system is a grid or not. According to this checklist, a grid system:

- coordinates resources that are not subject to centralized control,
- uses standard, open, general-purpose protocols and interfaces,
- delivers nontrivial qualities of services.

So, the grid infrastructure enables users to share autonomous geographically distributed computing and data storage resources belonging to different administrative domains in a transparent way.

In order to efficiently access and employ the heterogeneous and volatile resources of the grid, we need software tools that allow us to (Wankar 2008, Taniar et al. 2008a):
- identify the resources needed to run the application,
- schedule the execution of the tasks on the grid resources,
- monitor the execution state of the jobs and system components,
- locate, transport and replicate data,
- manage security issues.

These software tools are known as *Grid Middleware*.

In the following subsections, we review *Globus* and *gLite* grid middleware systems. Then, we present several examples of existing grid systems.

2.3.1 Grid Middleware

Several middleware systems are implemented to build grid infrastructures such as: *Gridbus* [2], *NetSolve/GridSolve* [3], *UNICORE* [4], *Legion* [5] and *Advanced Resource Connector (ARC)* [6]. We will briefly present *Globus* and *gLite* which are the mostly used middleware systems.

Globus toolkit

Globus [7] is an open source project that offers software tools for building grid systems and applications. Globus toolkit (Foster 2005) provides a set of software services and libraries for: security, resource management and access, information infrastructure, data management, communication, fault detection and portability.
The primary components of Grid toolkit are:
- *Grid Resource Allocation and Management (GRAM)*: provides a web service interface for remote job submission and control;
- *Grid Security Infrastructure (GSI)*: addresses massage protection, authentication, delegation, and authorization;
- *GridFTP*: a high-performance, reliable and secure data transfer protocol;
- *Monitoring and Discovery Service (MDS)*: provides tools and APIs for discovering, publishing and accessing information about the grid resources.

Users can employ some or all the toolkit components to build their own applications. Globus toolkit is the most used grid middleware which is rapidly evolving (Taniar et al. 2008a).

[2] Gridbus, `http://www.gridbus.org/`.
[3] NetSolve/GridSolve, `http://icl.cs.utk.edu/netsolve/`.
[4] UNICORE, `http://www.unicore.eu/`.
[5] Legion, `http://www.legion.virginia.edu/`.
[6] ARC, `http://www.nordugrid.org/middleware/`.
[7] Globus, `http://www.globus.org/`.

gLite

gLite [8] is part of the *EGEE* project [9] funded by the European Union. It was born from the collaboration of more than 80 people in 12 different academic and industrial research centers. The gLite middleware is formed of a set of components to enable both computing and storage resource sharing. gLite integrates components from other middleware projects, such as *Condor* [10] and the *Globus toolkit*, in addition to components developed for the *LCG* project [11].
The development of the middleware is organized into three different activities:

1. Data management, workload management, monitoring, accounting, computing element, logging and bookkeeping services;
2. Security services;
3. Network monitoring and provisioning services.

The gLite Grid services follow a *Service Oriented Architecture* (SOA). So, it will be easy to connect the software to other Grid services. This will also facilitate compliance with upcoming Grid standards, for instance the *Web Service Resource Framework* (WSRF) [12] from OASIS [13] and the *Open Grid Service Architecture* (OGSA) [14] from the *Open Grid Forum* (OGF) [15].

2.3.2 Grid infrastructure examples

There are many projects for building grid infrastructures such as: the *Austrian Grid* [16], D-Grid [17], BEgrid [18], EGEE [19], Grid'5000 [20] and EGI [21]. We will briefly review some of these infrastructures.

EGEE

The *Enabling Grids for E-sciencE* (EGEE) is an European project, launched in 2004, for building grid infrastructure. The project gives access to researchers from diverse scientific domains to a grid infrastructure available 24 hours a day, 7 days a week. As stated in the above section, EGEE employs

[8] gLite: Lightweight Middleware for Grid Computing, `http://glite.web.cern.ch/glite/`.
[9] EGEE: Enabling Grids for E-sciencE, `http://www.eu-egee.org/`.
[10] Condor, `http://www.cs.wisc.edu/condor/`.
[11] LCG project: A data storage and analysis infrastructure for the high energy physics community that will use the Large Hadron Collider at CERN, `http://lcg.web.cern.ch/LCG/`.
[12] WSRF, `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf`.
[13] OASIS, `http://www.oasis-open.org`.
[14] OGSA, `http://forge.ogf.org/sf/projects/ogsa-wg`.
[15] OGF, `http://ogf.org/`.
[16] The Austrian Grid, `http://www.austriangrid.at/`.
[17] D-Grid: The German Grid Initiative, `http://www.d-grid.de/index.php?id=1&L=1`.
[18] BEgrid: The Belgian Grid for Research, `http://www.begrid.be/index.php`.
[19] EGEE: Enabling Grids for E-sciencE, `http://www.eu-egee.org/`.
[20] Grid'5000, `https://www.grid5000.fr/`.
[21] EGI: The European Grid Initiative, `http://web.eu-egi.eu/`.

gLite as the grid middleware. The grid was developed under the following three objectives :

- build a consistent, robust and secure Grid network that will attract additional computing resources ;
- continuously improve and maintain the middleware in order to deliver a reliable service to users ;
- attract new users from industry as well as science and ensure that they receive the high standard of training and support they need.

On January 2009, EGEE connected 267 sites from 54 countries, with approximately 114,000 CPUs available to users 24 hours a day and a 20 PB of storage capacity.

Grid'5000

Grid'5000 (Cappello et al. 2005) is a French research effort, started in 2003, for developing a large scale nation wide infrastructure for Grid research. 17 laboratories, in France, are involved in Grid'5000 in order to offer to the community of grid researchers a testbed allowing experiments in : network protocols, operating systems, Grid or P2P middleware, application runtime, programming environments and applications.
Grid'5000 currently connects resources distributed on 9 sites in France. Each local site platform is formed of at least one cluster. The total number of processors of all the sites is approximately 3202 (5714 cores). The sites are connected by the RENATER [22] Education and Research Network. All clusters are connected to Renater with at least 1Gb/s link. The processors of each cluster are connected via Myrinet or Infiniband.

Grid'5000 doesn't employ a specific Grid middleware. However, several services are offered to allow : the reservation of processors by the users, deploy the reserved machines with their personal computing environment and stock data. The most important ones are :

- *OAR* [23] used as a resource manager for large clusters. OAR doesn't execute jobs on the resources but manages them (reservation, access granting) in order to allow users to have access to reserved resources and use them. *OAR-GRID* is used to reserve nodes from different sites of Grid'5000.
- Kadeploy [24] which is a deployment system for cluster and grid computing. It allows the users to deploy their own computing environment.
- A local NFS on each site.

[22] RENATER : Le Réseau National de télécommunications pour la Technologie l'Enseignement et la Recherche, `http://www.renater.fr/` .

[23] OAR : Resource Management System for High Performance Computing, `http://oar.imag.fr/index.html` .

[24] Kadeploy, `http://kadeploy.imag.fr/` .

EGI

The *European Grid Initiative* (EGI) [25] Design Study, launched in September 2007 and ended on November 2009, is an effort to establish an European sustainable grid infrastructure. Many European countries have launched *National Grid Initiatives* (NGI) for building grid infrastructures at national level. The goal of EGI is to link existing NGIs and support the creation of new ones. To this day, 42 NGIs in Europe participated in EGI. Some of the main objectives of EGI, as stated in their website, are:

- Coordinate the integration and interaction between NGIs;
- Provide global services and support that complement and/or coordinate national services (Authentication, VO-support, security, etc);
- Coordinate middleware development and standardization to enhance the infrastructure by soliciting targeted developments from leading EU and National Grid middleware development projects;
- Advise National and European Funding Agencies in establishing their programs for future software developments based on agreed user needs and development standards;
- Integrate, test, validate and package software from leading grid middleware development projects and make it widely available;
- Link the European infrastructure with similar infrastructures elsewhere.

The EGI entity will start operation in 2010.

2.3.3 Cost notations for the grid architecture

The BSP cost model is not adequate with grid architecture because it assumes that all system components have equal computation and communication characteristics. So, we will use the following notations, which represent the characteristics of the used machines, to study the cost of each step of parallel algorithms for grid.

Network parameters

- m_p: Communication message protocol cost per page of data,
- m_l: Communication message latency for one page of data.

We consider that the values of the following parameters are determined based on the slowest processor and network characteristics in the system.

Nodes parameters

- $c^i_{r/w}$: Read/write cost of a page of data on a local disk of node i,
- t^i_r: Time to read a record from the main memory of node i,
- t^i_w: Time to write a record to the main memory of node i,
- t^i_d: Time to compute destination join evaluating node of a tuple on node i,
- t^i_h: Time to add an entry to a B^+-tree on processor i.

[25] EGI: The European Grid Initiative, `http://web.eu-egi.eu/`.

2.4 Cloud computing

Foster et al. (Foster et al. 2009) defined *Cloud Computing* as:

> *"A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services over the Internet."*

So, Cloud computing gives users access to dynamically scalable and often virtualized computing and storage resources, provided as services, over the internet. Users do not need to know where the resources are or how they are configured. They pay for the service provider only for the services that they use. Cloud computing can be seen as several layers of services (Armbrust et al. 2009), mainly:

- *Infrastructure-as-a-Service* (IaaS): delivers computer infrastructure (computing resources and storage) as services. Amazon EC2 [26], IBM Blue Cloud [27] and Sun Grid [28] are examples of IaaS.
- *Platform-as-a-Service* (PaaS): delivers operating systems and some specific applications such as Apache and MySQL for Web-based applications as services. IBM IT Factory, Google AppEngine [29] and Force.com [30] are examples of PaaS.
- *Software-as-a-Service* (SaaS): delivers applications as services over the internet. Google Apps is an example of SaaS.

2.4.1 Cloud Computing versus Grid Computing

In their paper *"Cloud Computing and Grid Computing 360-Degree Compared"* (Foster et al. 2009), Foster et al. have posed the following question: *Is "Cloud Computing" just a new name for Grid?"*. Then they claimed that there is no straightforward answer to such question. However, they argued that cloud computing does not only overlap with grid computing, it has indeed evolved out of grid computing and relies on grid computing as its backbone and infrastructure. However, the main differences may be classified by the following aspects:

- *Business Model*: in the cloud business model, the user pays to the service provider based on his consumption. On the other hand, the grid business model is project-oriented where users have certain number of CPU hours that they can employ.
- *Compute Model*: usually, grid systems are based on a batch-scheduled compute model for reserving the computing resources requested by users. In such a batch-scheduled model, jobs may wait in the *Local Resource Manager (LRM) wait queue* until all the requested computing resources are available for the duration of the job. At this time, resources are allocated and dedicated to the job. In contrast, resources in cloud computing are shared by all the users at the same time.

[26] Amazon Elastic Compute Cloud (Amazon EC2): `http://aws.amazon.com/ec2/`.
[27] IBM Blue Cloud: `http://www.ibm.com/ibm/cloud/`.
[28] Sun Grid: `http://www.sun.com/software/sge/`.
[29] Google AppEngine: `http://code.google.com/intl/fr/appengine/`.
[30] Force.com: `http://www.salesforce.com/platform/`.

– *Security Model*: security is one of the main issues of grid systems. This is mainly due to the fact that, grids connect heterogeneous and dynamic resources of different organizations where each grid site may have its own administration domain and operation autonomy. On the other hand, Cloud computing security model is simpler and less secure because clouds usually give access to dedicated data centers belonging to the same organization. In cloud computing , Web forms are used to create and manage users account informations. These web forms allow users to reset their passwords and receive new passwords via Emails in an unsafe and unencrypted communication.

Grid'5000 introduced in section 2.3.2 and Amazon EC2 allow users to upload their own operating systems. However, in Grid'5000 a user has an exclusive employments rights and access to his reserved machines. This is not the case with Amazon EC2 where machines are shared by several users at the same time.

2.4.2 Distributed File Systems

Internet search engines such as Google and Yahoo need to manage and query a huge amount of data every day. Parallel processing of such queries on hundreds or thousands of nodes is obligatory to obtain a reasonable processing time (Dean and Ghemawat 2004). However, building parallel programs on parallel and distributed systems is complicated. This is because programmers must treat several issues such as load balancing between processing nodes, fault tolerance, network performance, etc. Search engine companies have developed Distributed File Systems (DFS) and parallel programming infrastructures that treat these parallel processing related issues without the explicit participation of the programmers (Lämmel 2007). Hadoop (hadoop), Google's File System (GFS) (Ghemawat et al. 2003) and BigTable (Chang et al. 2006) are examples of such DFS. These systems form the infrastructure of Yahoo's and Google's cloud computing systems. They are build from thousands of commodity machines and assure scalability, reliability and availability issues (hadoop, Ghemawat et al. 2003, Chang et al. 2006). To reduce disk Input/Output, each file in such storage systems is divided into chunks or blocks of data and each block is replicated on several nodes for fault tolerance.

Parallel programs are easily written on such systems following the Map/Reduce paradigm where a program is composed of a workflow of user defined *map* and *reduce* functions (Dean and Ghemawat 2004).

2.4.3 Map-Reduce Programming Model

Google's Map-Reduce programming model presented in (Dean and Ghemawat 2004) is based on two functions: *Map* and *Reduce*. Dean and Ghemawat stated that they have inspired their Map-Reduce model from Lisp and other functional languages. Users must implement two function *Map* and *Reduce* having the following signatures:

map: $(k_1, v_1) \longrightarrow list(k_2, v_2)$,
reduce: $(k_2, list(v_2)) \longrightarrow list(v_3)$.

The user must write the *map* function that has two input variables, a key k_1 and an associated value v_1. Its output is a list of intermediate key/value pairs (k_2, v_2). This list is partitioned by the Map-Reduce library depending on the values of k_2, where all pairs having the same value of $k2$ belong to the same group.
The *reduce* function, that must also be written by the user, has two input parameters: an intermediate key k_2 and a list of intermediate values $list(v_2)$ associated with k_2. It applies the user defined merge logic on $list(v_2)$ and outputs a list of values $list(v_3)$.

2.5 Summary

In this chapter, we reviewed three main architectures in parallel programming: *parallel systems*, *grid systems* and *cloud computing*. Each one of these architectures is adequate with some kinds of applications as we have seen in chapter 1. We also reviewed several programming models adequate with these architectures. In chapter 3, we will review algorithms presented in the literature for treating database queries, mainly those involving join, semi-join and "GroupBy-Join" queries.

State of the Art

3

Contents

In this chapter, we present the state of the art of the domain of treating database queries on parallel, distributed, grid and cloud systems where we review parallel algorithms, presented in the literature, for evaluating the join on such systems. Data skew may degrade the performance of parallel algorithms, so we present the different types of data skew. We also review several algorithms for treating "Group-By Join" queries on parallel architectures.

3.1 Join algorithms for homogeneous parallel architectures

Join operation is one of the most widely used operations in relational database systems, but it is also a heavily time consuming operation. For this reason it was a prime target for parallelization. The main idea behind parallel join algorithms is to:

1. Distribute tuples of the relations to be joined among system's processors using *hash partitioning* or *range partitioning* (DeWitt and Gray 1992) based on the values of the join attribute. The aim of this redistribution phase is to send tuples having the same join attribute value to the same processing node.
2. Then, compute the join of local fragments. To this end, we can use the serial *Nested loops join algorithm*, *sort-merge algorithm* or *hash based join algorithm*.

Three basic parallel algorithms for treating join queries on parallel systems were presented in the literature: *Grace-hash* (Kitsuregawa et al. 1983), *Hybrid hash join* (Schneider and DeWitt 1990) and *Sort-merge join* (Schneider and DeWitt 1989). However, the performance of these algorithms degrades in the presence of data skew mainly *Attribute Value Skew* (AVS) and may produce highly skewed join result sizes on the processors known as *Join Product Skew* (JPS). In the rest of this section, we will review these basic algorithms. Then, we present the different kinds of data skew. Finally, we review parallel join algorithms for treating data skew on homogeneous parallel systems.

3.1.1 Basic parallel join algorithms

In the following subsections, we will briefly review *Grace-hash* (Kitsuregawa et al. 1983), *Hybrid hash join* and *Sort-merge* algorithms. We consider that we want to find the result of the join of two relations R and S. We also consider that S is the inner join relation (the smaller relation used generally to build the hash table) and R is the outer join relation (used generally to probe the hash table).

Grace-hash join algorithm

This is a three-phase algorithm that proceeds as follows (Kitsuregawa et al. 1983):

1. partition S tuples into N buckets, using a hash function $h_1()$ applied on the join attribute value of each tuple. These buckets are then distributed over the processors using a hash function $h_2()$;
2. apply the same hash function $h_1()$ on the tuples of relation R to partition R into N buckets. Then, distribute these buckets using the same hash function $h_2()$;
3. join the respective matching buckets to find $R \bowtie S$.

N is chosen to be very large in order to reduce the chance that the size of any partitioned bucket will exceed the memory capacity of the processor used to compute the join. If the formed buckets are much smaller than

the memory capacity, then several buckets are combined in the join phase to form more optimal size buckets.

Hybrid-hash join algorithm

Hybrid-hash join algorithm (Schneider and DeWitt 1990) is similar to *Grace-hash join* algorithm. However, in the first phase of *Hybrid-hash join* algorithm, the first partitioned bucket of S is kept in memory and the other $N-1$ buckets are distributed over the processors as in Grace-hash join algorithm. In the second phase, the tuples of R hashed to the first bucket are used to immediately probe the in-memory bucket of S and the other $N-1$ buckets are distributed over the processors as in Grace-hash join algorithm. The join of the remaining $N-1$ buckets is performed as in Grace-hash join algorithm.

Sort-merge algorithm

In Sort-merge algorithm (Schneider and DeWitt 1989), the tuples of R and S are redistributed over the processors using a hash function applied on the values of the join attribute. After the redistribution phase, the serial sort-merge algorithm (section 2.1.2) is applied on each processor to compute $R \ltimes S$.

3.1.2 Data skew in parallel architectures

Research has shown that the join operation is parallelizable with near-linear speed-up on parallel systems such as Shared Nothing machines[1] but only under ideal balancing conditions : data skew may have disastrous effects on the performance of parallel algorithms on such architectures (Bamha and Hains 2000; 1999, Seetha and Yu December 1990, DeWitt et al. 1992, Mourad et al. 1994).

Walton (Walton et al. 1991) has classified data skew into two categories : (1) Intrinsic skew and (2) Partition skew.

1. *Intrinsic skew* : also known as *Attribute Value Skew* (AVS) occurs when attribute values are not distributed uniformly. This means that some attribute values appear with frequencies much higher than the others. The AVS is a data property and does not change between algorithms. However, in order to benefit from parallelism, the used algorithms must follow distribution techniques that balance the load of different processors even in the presence of AVS.
2. *Partition skew* : this kind of skew appears in parallel join algorithms when the load of different processors is not uniform. Some types of partition skew can occur even when the input data are uniformly distributed. Partition skew can be divided into four categories :
   (a) *Tuple Placement Skew* (TPS) : the initial distribution of tuples varies between the processors.

[1] Shared Nothing machines : a distributed architecture where each processor has its own memory and own disks.

(b) *Selectivity Skew* (SS): this occurs when the selectivity of selection predicates varies between processors.

(c) Redistribution Skew (RS): this occurs when the number of tuples received by each processor after redistribution is not uniform.

(d) *Join Product Skew* (JPS): this occurs when the join result size is not uniform on all the processors. The JPS may occur even if the input relations do not suffer from AVS.

Remark 1 *In practice, the imbalance of the data related to the use of the hash functions can be due to*:

- *a bad choice of the hash function used. This imbalance can be avoided by using the hashing techniques presented in the literature making it possible to distribute evenly the values of the join attribute with a very high probability (Carter and Wegman April 1979),*
- *an intrinsic data imbalance which appears when some values of the join attribute appear more frequently than others. By definition a hash function maps tuples having the same join attribute values to the same processor. There is no way for a clever hash function to avoid load imbalance that results from these repeated values (DeWitt et al. 1992).*

The algorithms presented above, for treating join operation, are inefficient due to the following reasons:

1. The communication cost, in these algorithms, is very high because all the tuples of the relations are redistributed between processors. Some of these tuples may not even contribute in the result of the join operation. This may also induce high disk input/output whenever distributed data does not fit in memory.
2. These algorithms cannot solve the problem of data skew because data redistribution is generally based on hashing data into buckets and hashing is known to be inefficient in the presence of high frequencies (Bamha 2005, Schneider and DeWitt 1989, Seetha and Yu December 1990).
3. The size of the join result on different processors may be highly skewed (JPS) which degrades the performance of parallel algorithms when treating complex queries.

So, finding load balancing mechanisms for join computing on parallel and distributed systems was and is still one of the main interests of database research community.

3.1.3 Parallel join algorithms for treating data skew

Several parallel algorithms were presented to handle data skew while treating join queries on parallel database systems. We will review some of these algorithms and give their drawbacks.

Bucket Spreading Parallel Hash Algorithm (BSJ)

This algorithm, presented in (Kitsuregawa and Ogawa 1990), proposes a dynamic bucket allocation mechanism for evaluating join operations on *Super Database Computer* (SDC). SDC is a Shared Nothing architecture where nodes are connected through a highly functional Omega interconnection network. This network allows to evenly partition and distribute a bucket over the system machines.

The algorithm proceeds as follows:

1. the tuples of both relations are partitioned into buckets using a hash function,
2. then each formed bucket is evenly partitioned into subbuckets which are distributed over the processors through the Omega network. This subbucket size balancing task is performed by the Omega network without any intervention of the processors,
3. a *bucket size tuning* task is performed by a master processor in order to form buckets having approximately same sizes. This step does not need to be executed on each processor because the subbuckets are flatly distributed. Hence, tuning on one processor is representative to tuning buckets on all processors. Finally, the tuned buckets are evenly allocated to the processors to compute the join. However, before computing the join, each processor must gather the fragments (subbuckets) of its assigned buckets which are stored on other nodes.

This algorithm is not efficient because the tuples of both relations are redistributed twice. In addition, the Flat bucket distribution step necessitates the employment of an expensive Omega network which does not scale well with the rest of the system (Hua and Lee 1991). Finally, the Join Product Skew (JPS) is not treated which may degrade the performance of the algorithm while treating multi-join queries.

Tuple Interleaving Parallel Hash Join (TIJ)

TIJ algorithm (Hua and Lee 1991) is based on BSJ algorithm. However here, instead of using an Omega interconnection network to partition each bucket uniformly over all the processors, they use a software control technique known as *tuple interleaving strategy*. In this algorithm, each processor partitions both relations into p buckets using a hash function where p is greater than the number of processors N. During this step, the tuples of each bucket are spread uniformly across the N processors in the following manner: tuple i of each bucket is sent to processor of index $(((i-1)\ mod\ N)+1)$. Using this tuple interleaving strategy, the N subbuckets related to each bucket should have the same size. Then, a coordinating processor is used to tune the size of subbuckets in order to form N uniform buckets and assigns each bucket to a processor. After that, the processors exchange the tuples of the local subbuckets as directed in the mapping information received from the coordinator processor. After redistribution, each processor performs the join of the received tuples.

In this algorithm, all the processors stay inactive during the bucket tuning phase performed by the coordinator node. In addition, it doesn't treat the JPS problem.

Adaptive Load Balancing Parallel Hash Join (ABJ)

This algorithm, also presented in (Hua and Lee 1991), is formed of four phases as follows:

1. **Split phase**: the tuples of both relations are partitioned, in parallel, into a large number of buckets. After that, each created bucket is statically allocated to a computing node as in Grace Hash Join algorithm.
2. **Partition Tuning phase**: this phase is divided into two steps:
   - **Bucket Retaining stage**: for each relation, each processor chooses from the received buckets only n buckets such that: $\sum_{j=1}^{n} |B_{ij}| \leq \frac{|A|}{N}$ and $\sum_{j=1}^{n+1} |B_{ij}| > \frac{|A|}{N}$ where $|A|$ is the size of the treated relation, $|B_{ij}|$ is the size of bucket j on processor i and N is the number of processors. The remaining buckets are allocated in the next stage.
   - **Bucket Relocating stage**: Each processor i sends the value of $\sum_{j=1}^{n} |B_{ij}|$ and the size of the excess buckets to a coordinator node, which in its turn reallocates these buckets to underflow processors. After that, the coordinator node broadcasts this information to all the nodes and the excess buckets are sent to their destinations.
3. **Bucket Tuning phase**: small buckets are combined to form more optimal size join buckets on each processor.
4. **Join Phase**: the join of the buckets is performed on each processor.

The performance of ABJ algorithm degrades in the presence of highly skewed data because a disk overflow may occur in the split phase. Moreover, redistributing the skewed buckets takes longer time than the other buckets which increases the total communication cost. In addition, the excess buckets must be distributed two times. So, to overcome these drawbacks, Hua et al. have presented the ABJ+ algorithm (Hua and Lee 1991).

Extended Adaptive Load Balancing Parallel Hash Join (ABJ+)

ABJ+ algorithm proceeds in four phases as follows:

1. **Split phase**: the tuples of both relations are partitioned into small subbuckets which are written into local disks.
2. **Partition tuning**: each processor sends the total size of the subbucket to a coordinator node, which in its turn computes the sum of the received data from all the processors. Then, it allocates the buckets to the computing processors as follows:

- it sorts the buckets in decreasing order according to their size.
- buckets are allocated to processors in the sorted order, where each bucket is assigned to the processor with the largest matching sub-bucket. The coordinator node stops assigning buckets to a processor i when it satisfies the following condition: $\sum_{j=1}^{n} |B_{ij}| \leq \frac{|A|}{N}$ and $\sum_{j=1}^{n+1} |B_{ij}| > \frac{|A|}{N}$ where $|A|$ is the size of the treated relation, $|B_{ij}|$ is the size of bucket j on processor i and N is the number of processors. This process is executed for each bucket. After that, the allocation information is broadcasted to all the processors which use this information to redistribute the buckets.

3. **Bucket Tuning phase**: small buckets are combined to form more optimal size join buckets on each processor.
4. **Join Phase**: the join of the buckets is performed on each processor.

This algorithm is not adequate in the presence of highly skewed data because all tuples related to a skewed value of the join attribute are sent to the same processor. In addition, it does not solve the problem of JPS.

Sampling based Load Balancing join algorithms

In (DeWitt et al. 1992), *simple range*, *weighted range partitioning*, *virtual processor partitioning - round robin* and *virtual processor partitioning - processor scheduling* algorithms were presented to evaluate join operation in the presence of skewed data. These algorithms are based on two ideas: (a) *virtual processor partitioning* where the number of created partitions is much greater than the number of processors in the system and (b) *sampling* used to compute statistical information about the values of the join attribute.

In *virtual processor partitioning - round robin* algorithm, the inner relation is sampled. Then, a range partitioning vector is created using these samples. The number of partitions defined by this vector is a multiple of the number of processors. After that, the tuples of build relation are partitioned into buckets using the range partitioning vector. These buckets are distributed over the processors using round-robin partitioning and an in-memory hash table corresponding to the received tuples is created on each processor. Finally, the tuples of the probing table are redistributed using the same range partitioning vector and join is computed on each processor.

Virtual processor partitioning - processor scheduling algorithm is the same as *virtual processor partitioning - round robin* algorithm, but instead of using round robin allocation in allocating buckets to processors, the LPT (Largest Processing Time) heuristic scheduling algorithm (Graham 1969) is used.

These algorithms are better than the other presented algorithms because using sampling avoid the need of a linear traversal of all tuples of the relations. It also treats the AVS of the build relations better than ABJ+

algorithm, but AVS is ignored in the probe relations. In addition, it does not treat JPS.

Drawbacks of the presented algorithms

Algorithms presented in this section are not efficient for treating join operation on parallel systems due to the following reasons:

- they cannot be scalable (and thus cannot guarantee linear-speedup) because their routing decisions are generally performed by a coordinator processor while the other processors are idle,
- they cannot solve the load imbalance problem as they base their routing decisions on incomplete or statistical information. In addition, they generally induce high communication costs during data redistribution since all the tuples are redistributed and not necessarily only relevant data,
- they cannot solve data skew problem because data redistribution is generally based on hashing data into buckets and data hashing is known to be inefficient in the presence of high frequencies (DeWitt et al. 1992).

OSFA-Join: Optimal Symmetric Frequency Adaptive Join algorithm

Optimal skew-handling parallel algorithms for evaluating join on homogeneous Shared Nothing (SN) machines were presented in (Bamha and Hains 1999; 2000, Bamha 2005). These algorithms override the drawbacks of the algorithms stated earlier because the creation of communication templates, used for data redistribution among processors, is based on using *fully distributed histograms* that hold complete data-distribution information. This step is jointly performed in parallel by all processors, *each one not necessarily computing the list of its own messages, so as to balance the overall process*. This makes these algorithms scalable because we do not have idle processors waiting for a coordinator node. In addition, the communication cost is minimal because only tuples that participate in the join operation are redistributed. These algorithms also avoid the problem of Join Product Skew (JPS) while balancing the load of different processors even in the presence of Attribute Value Skew (AVS).

To compute the join of two relations, R and S, *OSFA-Join* proceeds as follows:

1. Each processor i creates, in parallel, the local *histograms* denoted by $Hist(R_i)$ and $Hist(S_i)$ related to partitions R_i and S_i of R and S respectively. The *histograms* hold the local frequency related to each value of the join attribute, i.e., the number of local tuples holding this value for the join attribute. These *histograms* are created under the form of a balanced tree which is a data structure that maintains an ordered set of data to allow efficient search and insert operations.
2. The local *histograms* of R and S are distributed over the processors using a hash function in order to find the global frequency of each join attribute value. Each processor i merges the received entries

to form the global distributed *histograms* denoted by $Hist_i(R)$ and $Hist_i(S)$ in parallel. Thus, $Hist_i(R)$ and $Hist_i(S)$ denote the histogram of R and S respectively.

3. Each processor i computes the intersection of $Hist_i(R)$ and $Hist_i(S)$ to form the *histogram* of $R \bowtie S$. This *histogram*, $Hist_i(R \bowtie S)$, holds the join attribute values that will appear in the join result with the frequency related to each value.
4. Now, all the processors participate in creating the communication templates used to redistribute only tuples that participate effectively in the join. So, each processor i creates the templates for each join attribute value $v \in Hist_i(R \bowtie S)$ as follows:
   - if the frequency of v in R and S are smaller than a certain threshold frequency f_o, then it is considered that tuples related to v do not have any effect on AVS and JPS. So, these tuples are redistributed by hashing.
   - if the frequency of v in R is greater or equal to f_o and is also greater than the frequency of v in S, then they consider that v needs special treatment because it may cause AVS or JPS. So, tuples of R related to v are evenly partitioned over all processors and that of S are duplicated.
   - if the frequency of v in S is greater or equal to f_o and is also greater than the frequency of v in R, then tuples of S related to v are evenly partitioned over all processors and that of R are duplicated.

   The value of f_o is set to $p * log(p)$, where p is the number of processors.
5. Tuples of R and S participating effectively to the join result are redistributed according to the communication templates.
6. Each processor computes the join of the received tuples.

In the contrary to algorithms presented in (DeWitt et al. 1992), the mechanism used in creating the communication templates in *OSFA-Join* allows to balance the load of different processors even in the presence of AVS in R and S. In addition, the size of the join result in all the processors is approximately the same, so it avoids JPS.

Bamha and Hains have proved, using the BSP cost model (section 2.2.2), that *histogram* management has a negligible cost when compared to the provided efficiency gains in reducing communication costs and avoiding load imbalance. They have also proved, using the same model, that these algorithms have a near-linear speed-up performance on homogeneous SN architecture. This was also assured by a series of experimental results.

3.2 Parallel processing of *simple aggregate* and *GroupBy-Join* queries

The input tables of *simple aggregate* and *GroupBy-Join* queries may rapidly grow every day especially in OLAP systems (Datta et al. 1998). Moreover,

the output of these queries must be obtained in a reasonable processing time. For these reasons, parallel processing of queries involving aggregate functions, group-by and/or join operations results in huge performance gain, especially in the presence of parallel DBMS (PDBMS). However, the use of efficient parallel algorithm in PDBMS is fundamental in order to obtain an acceptable performance (Bamha and Hains 2000; 1999, Mourad et al. 1994, Seetha and Yu December 1990).
In the following subsections, we present parallel algorithms already presented in the literature for treating *simple aggregate* and *GroupBy-Join* queries in parallel.

3.2.1 Parallel processing of simple aggregate queries

Two simple parallel algorithms for processing *simple aggregate* queries on shared nothing architecture were presented in (Shatdal and Naughton 1995). The first one is *Two Phase Algorithm* and the second is *Repartitioning Algorithm*.

Two Phase Group By Algorithm

As its name states, this algorithm is formed of two phases. In the first phase, the aggregate function is applied on the local tuples of each processor. Then, in the second phase, the resulting tuples are redistributed using a hash function applied on the Group By attributes among the processors. Hence, after this partitioning step, aggregate results having same values of Group By attributes are found on the same processor. So, a global application of the aggregate function is performed on each processor in order to obtain the final result.

Repartitioning Group By Algorithm

In this algorithm, the local tuples on each processor are partitioned on the Group By attributes using a hash function. After this distribution step, the aggregate function is applied based on the designated grouping. We can see that in this algorithm, the aggregate function is applied only one time, but all the tuples of the base relation are redistributed over the processors.

The implementation results of these algorithms showed that the two phase algorithm performs better when the number of groups is low. However, repartitioning algorithm overrides the other algorithm when the number of groups is high, because it avoids the double execution of the aggregate function. The effect of data skew of the performance of both algorithms was also studied, mainly *selectivity skew* and *tuple placement skew*. In the presence of high selectivity skew, the two phase algorithm proves to have a better performance thanks to the local aggregation in the first phase which allows us to evenly distribute the load of different processors in the second phase. This is not the case for the second algorithm, because after the distribution phase the load of the different processors is skewed

because the hash function is applied on the tuples of the skewed base relation.
However, the second algorithm has a better performance in the presence of tuple placement skew, because in the first phase of the two phase algorithm, processors with low number of tuples must wait for the skewed processors to finish processing the first phase before starting the global aggregation phase.

3.2.2 Parallel processing of "GroupBy-Join" Queries

We have seen in section 2.1.1 of chapter 2, that aggregate functions can be applied on the result of the join of two or more relations. In traditional algorithms that treat such queries, join operation is performed in the first step and then the group-by operation (Chaudhuri and Shim 1994, Yan and Larson 1994). But the response time of these queries may be significantly reduced if the group-by operation is performed before the join (Chaudhuri and Shim 1994), because group-by reduces the size of the relations and thus minimizing the join and data redistribution costs.
Several optimization techniques were introduced in the literature in order to generate the query execution plan with the lowest processing costs (Yan and Larson 1994; 1995, Chaudhuri and Shim 1994, Gupta et al. 1995). Their aim is to study the necessary and sufficient conditions that must be satisfied by the relational query in order to be able to push the Group By past join operation and to find when this transformation helps in decreasing the execution time.

However, we can divide "GroupBy-Join" queries into two categories depending on the group-by and join attributes as follows:

1. if the join attributes are part of the group-by attributes, then we can apply the group-by operation before the join,
2. if the join and group-by attributes are different, then the join operation must be applied before the group-by operation.

In the following subsections, we review the parallel algorithms for evaluating both categories of "GroupBy-Join" queries. We consider that the two relations to be joined are *R* and *S* and that the aggregate function is applied on an attribute of R.

Evaluating Group By before Join operation

Three basic parallel algorithms for executing group-by operations before join operations are presented in (Taniar and Rahayu 2006). These algorithms can be applied when the join and group-by attributes are the same.

- **Early Distribution Scheme**:
  In this algorithm, tuples of base relations are redistributed using a hash function, in a first phase, based on the group-by/Join attributes before applying the join and group-by operations. In the second phase, the aggregate function is evaluated based on the designated grouping, and then the result is joined with the other

table (S). The communication cost in this algorithm is high because all the tuples of the base relations are redistributed. However, the cost of its join operation is reduced because the group-by operation is performed before the expensive join operation.

- **Early Group By with Partitioning Scheme**
  This is a three-phase algorithm which are: *local grouping phase*, *distribution phase* and *final grouping and join phase*.
  In the first phase, the group-by and aggregate function are applied on the local tuples of relation R on each processor. Then, in the distribution phase, the aggregation result and the tuples of S are redistributed over the processors using a hash function applied on the group-by/join attribute. Finally, in the third phase, the global aggregation step is performed on the temporary aggregate results received by each processor, then joined with the tuples of S.
  Applying the group-by operation before the distribution and the join operations helps in reducing the volume of exchanged data. But in this algorithm, all the tuples of the group-by results are redistributed even if they do not contribute in the join result. This is a drawback, because in some cases only few tuples of relations formed of million of tuples contribute in the join operation, thus the distribution of all these tuples is useless.

- **Early Group By with Replication Scheme**
  In this algorithm, the aggregate function is partially applied on each processor. After that, the resulting tuples of this aggregation step are replicated on all the processors. Then, the global aggregation step is applied and finally the join of the aggregation result and the tuples of S is computed. We can see that, in this algorithm, we do not redistribute the tuples of S. However, the replication of the aggregate result introduces high communication and disk Input/Output costs and limits the scalability of the algorithm.

Evaluating Group By after Join operation

In (Jiang et al. 1999, Taniar et al. 2000) three parallel algorithms are proposed for evaluating "GroupBy-Join" queries when the join and Group-By attributes are different. In the following three subsections, we present these algorithms and we give their drawbacks.

- **Join Partition Method (JPM)**
  This algorithm is formed of four phases (fig.3.1). The first phase is the *data partitioning phase* where the two relations to be joined are partitioned into N buckets by applying a partitioning method such as range partitioning on the join attribute (N is the number of processors). After that, the buckets will be redistributed on the different processors.
  In the second phase, *join and local aggregate*, the join operation is performed, in parallel on each processor, using any sequential join algorithm. After that, the aggregate operation is applied locally on

each processor.
The third phase is a *re-distribution phase* where a partitioning method is applied on the group-by attribute in order to send tuples having the same values of this attribute to the same processor.
The final phase is the *global aggregation phase* where each processor applies the aggregate and group-by operations on the received tuples in order to find the final result.

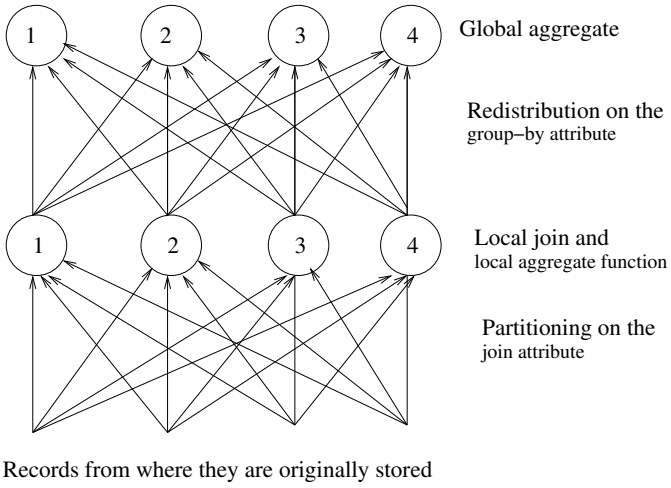

FIG. 3.1 – *Join partition method (JPM) (Taniar et al. 2000).*

The main drawback of this algorithm lies in applying the partition or hash functions on the join attribute of the base relations. This may cause a skew in the load of different processors if we have a skew in the distribution of the values of the join attribute (Bamha 2005, Schneider and DeWitt 1989, Seetha and Yu December 1990). We may also have a skew in the result of the join (JPS) which also affects the performance of this algorithm because the join result is the input of the local aggregation step.

– **Aggregate Partition Method (APM)**
This is a two phase algorithm (fig. 3.2) where in the first phase, *data partitioning and broadcasting*, the table that contains the group-by attribute will be partitioned into N buckets by applying partitioning method on this attribute. After that, these buckets are distributed on the processors and the second table is broadcasted to all processors.
The second phase is the *join and aggregate phase* where each processor performs the join of the received fragments of the first relations with the entire second table. After that, the aggregate and group-by operations are computed on each processor.

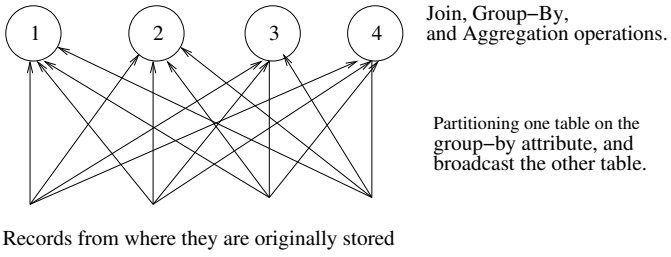

FIG. 3.2 – *Aggregation partition method (APM) (Taniar et al. 2000).*

In this algorithm, we do not need to redistribute the join result in order to apply the group-by and aggregate function because all tuples having the same value of the group-by attribute will be found on the same processor after the redistribution of the first phase. But it introduces other expensive costs, especially communication costs because the relation that does not contain the group-by attribute must be fully replicated on all the processors which makes the algorithm inextensible. In addition, it may suffer from high input/output costs due to hash table overflow in the second phase.
The performance tests presented in (Taniar et al. 2004) show that the performance of the *aggregate partition method* is better than that of the *join partition method* when the number of produced joins is big, but not when it is relatively small. The speed-up tests also show that both algorithms have a non-linear acceleration and that skew has a great impact on their performance. This algorithm suffers also from scalability due to full data replication.

– **Hybrid Partition Method (HPM)**:
In this method, presented in (Jiang et al. 1999), the N processors are divided into m logical clusters, each containing N/m processors. In the first step, the table that contains the group-by attribute is partitioned over the different clusters using the *APM* method. In the second step, within each cluster, the received fragment of the first table and the broadcasted table are partitioned by the join attributes using the *JPM* method.
In this algorithm, the table that contains the group-by attribute is purely partitioned on all the processors. This is not the case for the second table which is to some degree replicated (Taniar et al. 2000), hence the communication cost stays high. In addition, it does not solve the problem of data skew because the partitioning method is applied, as in *JPM*, on the tuples of the base relations which may be highly skewed.

These algorithms suffer from the same drawbacks stated in section 3.1.2 related to parallel join algorithms. In addition, "Group-by Join" algorithms fully materialize the intermediate results of the join operation. This is a significant drawback because the size of the result of this operation is generally large. In addition, the Input/Output cost in these algorithms is very high where it is reasonable to assume that the output relation cannot fit in the main memory of every processor. So, it must be reread from disk in order to evaluate the aggregate function.

3.3 Adaptive join algorithms for heterogeneous parallelism

Today, with the rapid development of network technologies, query processing on distributed systems that may connect heterogeneous pools of machines is gaining the interest of database and scientific computing communities due to their processing power and memory capacity. But, the

heterogeneity of these systems has introduced new challenges which were not present in parallel systems such as shared nothing machines, because in order to benefit effectively from the power of distributed systems no processor must be idle while other processors are overloaded (Karatza and Hilzer 2002). Hence, the actual characteristics of resources such as CPU power, Input/Output speed, connection speed, available memory, etc. must be taken into consideration while assigning jobs to the nodes. It is known that in parallel systems where machines are usually homogeneous, workload imbalance may be due to uneven load distribution, but in heterogeneous distributed systems it may be due to load distribution which is not proportional to the actual capabilities of each machine (Gounaris 2005). Moreover, in multi-user systems the capacity of a machine may highly vary from one instant to another.

Algorithms used for join processing on homogeneous parallel systems may not be effective on heterogeneous systems even if they efficiently handle the problem of data skew because they use a static strategy in load allocation. Hence, in order to obtain an acceptable performance on such systems an adaptive or a dynamic load distribution strategy must be used (Karatza and Hilzer 2002). This strategy must also take in consideration the power of each machine and react with the system state.

Several adaptive algorithms are presented in the literature for treating join operations on heterogeneous distributed systems. These algorithms try to balance the load of the processors throughout all the join processing phases by dynamically adapting the size of tasks assigned to each processor according to its actual performance capabilities. The difference between most of the adaptive algorithms lies in the manner they treat *monitoring*, *assessment* and *response* which are (Paton et al. 2009):

- *monitoring*: collecting information about the progress state of the query or the system's state,
- *assessment*: analyzing the monitored information to know if there is a need to rebalance the load,
- *response*: if balancing load is needed, then necessary reactions are determined to address the problem.

In the rest of this section, we present some of these algorithms.

3.3.1 Expanding Hash-based Join Algorithms

Zhang et al. (Zhang et al. 2004) proposed three adaptive hash-based algorithms for evaluating join operations on such environments where a non-static redistribution strategy is used. In these algorithms, they addressed the problem of buffer overflow while building the hash tables by dynamically allocating additional machines. The algorithms start by creating the hash table of the build relation on a specific number of nodes. During this phase, if the memory of a node is used up, additional nodes will be employed in order to maintain the buckets in memory. The experimental results showed that the performance of the algorithms

significantly degrades in the presence of skewed data. This is because they use hash functions for assigning join attribute values to the computing nodes. However, it is known that hash functions are not efficient in the presence of skewed data. In addition, the communication cost is high because all tuples of both relations are redistributed even those which do not participate in the final join result.

3.3.2 DITN: Data In The Network

Raman et al. presented in (Raman et al. 2005) an adaptable dynamic algorithm for treating queries on a DITN architecture. This architecture is built as a prototype on top of IBM's Web-sphere Information Integrator. It is formed of a CPU-Wrapper whose task is not to wrap data sources, but non-dedicated heterogeneous compute nodes known as co-processors. These co-processors are not expected to store data, instead relations are stored on a shared storage system. So, co-processors access directly this storage device to treat its assigned tasks.
To compute $T_1 \bowtie T_2 \bowtie \ldots \bowtie T_k$, T_i's are logically divided into partitions such that $T_i = T_i^1 \cup T_i^2 \cup \cdots \cup T_i^{p_i}$. So, the join is equivalent to $(T_1^1 \cup T_1^2 \cup \cdots \cup T_i^{p_1}) \bowtie \ldots \bowtie (T_k^1 \cup T_k^2 \cup \cdots \cup T_k^{p_k})$ and each component of this join is assigned to a co-processor. In order to handle the failure or overload of a co-processor, the CPU-Wrapper keeps track of when each co-processor finishes its work-unit. If a co-processor takes a long time, then the CPU-Wrapper will consider that it has failed or is overloaded. So, the CPU-Wrapper reassigns the work-unit of this co-processor to the fastest processor that has already finished its task. Then, when one of these two co-processors finishes treating the work-unit, the CPU-Wrapper cancels the treatment of the other co-processor.

The drawback of this algorithm lies in the use of a central disk storage unit. However, they claim that this structure will not cause a bottleneck on the network because its bandwidth is much larger than the bandwidth of query processing operators such as sort, join or even scan.

3.3.3 Redundant data maintenance algorithm

This dynamic algorithm (Paton et al. 2009) is based on the idea of replicating each hash table on three processors chosen randomly. During hash table building or probing, whenever a tuple is to be hashed, it is sent to the two most lightly loaded processor of the three assigned nodes for the relevant bucket. An extra attribute is stored for each tuple indicating the index of the second processor that holds its replica. During probing phase, each tuple is sent to two of the three processors associated with its buckets. The replica on the most lightly processor is called *primary* and on the second most lightly one is called *secondary*. At each node, while probing, if a tuple matches an entry in the hash table, then the join result is only generated if it is a primary probe unless the matching tuple is stored only on the other two nodes.

When a processor is overloaded, the join operation is performed on the two other processors without the need to redistribute the buckets of the loaded one. However, the drawback of this algorithm is that an additional cost is added to maintain replicas of all hash table entries and to replicate the tuples of both relations on two processors.

3.4 Query processing on the grid

Several projects for building grid systems (the Globus toolkit [2], the Europeans DataGrid project [3], the Grid Physics Network project (GriPhyN) [4], Particle Physics Data Grid (PPDG) [5], etc.) have developed softwares and services to address basic grid functionalities such as: security, authentication, data management and access, and resource discovery. This made query processing on the grid feasible, but new challenges are also introduced which were not present in parallel systems such as Shared Nothing machines. The main challenges are due to the following facts:

- Grid systems usually connect heterogeneous pools of autonomous resources through different and unstable bandwidths. Hence, to effectively benefit from the power of the grid, the actual characteristics of resources such as CPU power, Input/Output speed, connection speed, available memory, etc. must be taken into consideration while assigning jobs to the nodes. This issue was not treated in parallel algorithms for join processing on parallel systems where the main aim of such algorithms is to balance the work load of the homogeneous machines forming the parallel system. However, in grid architectures workload imbalance may also be due to load distribution which is not proportional to the actual capabilities of each machine (Gounaris 2005, Yang et al. 2005, Gounaris et al. 2005).
- Grid sites that belong to different organizations are usually connected through the WAN. Hence, communication between nodes of remote sites is a main barrier for implementing scalable algorithms.
- Grid is an autonomous and volatile environment. Hence, computing or data nodes may suddenly become unavailable. Thus, fault tolerance is a main issue that must be reviewed in existing parallel join algorithms.

Therefore, to benefit from the processing power and storage capacity of grid systems, these issues must be handled by the employed parallel algorithms.

In this section, we will review the algorithms presented in the literature for processing relational queries on the grid. These algorithms can be divided into two categories: (a) static distributed query processing algorithms and (b) adaptive ones.

[2] www.globus.org
[3] http ://eu-datagrid.web.cern.ch/eu-datagrid/
[4] www.griphyn.org
[5] www.ppdg.net

3.4.1 Static distributed query processing algorithms

In (Mach and Schikuta 2007), the performance of the three well known *Hash join, Nested Loop* and *Sort Merge Join* parallel algorithms (presented in section 3.1) is tested on a *Static Simplified Grid Organization*. The tests showed that *Hash join* algorithm outperforms the other two algorithms when the size of the buffers of deployed nodes is greater than ten percent the size of the smaller relation. However, they assumed that the performance of the nodes does not drop below a given value during query execution and that the availability of each node is guaranteed. These two conditions are generally unrealistic in grid architectures. In addition, hash join algorithms cannot solve the problem of data skew, because data redistribution is based on hashing data into bucket. Moreover, this algorithm induces high communication cost since all the tuples are redistributed over the network and not necessarily only relevant ones.

An algorithm for evaluating join on data grid where the relations to be joined are fully replicated on several machines is presented in (Yang et al. 2005). In this algorithm, a well known relation reduction technique is used in order to reduce the redistributed volume of data where only tuples that effectively participate in the join are redistributed. In addition, the execution nodes are selected using an edge-weight-minimum-matching-algorithm where the grid infrastructure is considered as a weighted complete bigraph. This strategy helps in decreasing the exchange time of data between processors, but their algorithm can only be applied if the relations to be joined are fully replicated. This represents a serious problem for scalability.

OGSA-DQP (Lynden et al. 2009) is a service-based *Distributed Query Processor* on the grid. It is based on OGSA-DAI [6] which is a service-based middleware that supports access, sharing, management and coordinated use of heterogeneous physical data sources on the grid by providing a uniform service interface to grid databases (Antonioletti et al. 2005). OGSA-DQP extends OGSA-DAI by adding two services: *DQP coordinator service* and *DQP evaluator service*. The DQP coordinator service receives a perform document from the client. The query is compiled to produce a partitioned query plan. Each partition will be executed on a DQP evaluator. The evaluators read the data from OGSA-DAI data services and invoke needed analysis services for processing the query. In OGSA-DQP, both pipelined and partitioned parallelism are used to evaluate queries which help in decreasing the query processing time.

The main drawbacks of these algorithms are that:

- they do not follow strategies to rebalance the load between processors during query execution. This is an important issue because, as we mentioned earlier, the performance of some grid nodes may highly vary during query processing,
- they do not treat data skew problems which may degrade the performance of parallel processing.

[6] OGSA-DAI: The Open Grid Service Architecture - Data Access and Integration.

– they induce high communication and disk input/output costs during redistribution phase, since all input relations are redistributed across the network and not necessarily only relevant data.

3.4.2 Adaptive distributed query processing algorithms

To avoid the drawbacks of existing grid join algorithms, adaptive distributed query processing has been introduced. This category of approaches follow adaptive strategies to rebalance the load during query processing between processors if one (or more) processor becomes overloaded or unavailable.

Adaptive Grid Query Evaluation Service (AGQES)

In (Gounaris et al. 2005), a service oriented adaptive query processing architecture is introduced for evaluating queries on the grid. This architecture, which is an extension of OGSA-DQP, helps to adapt to changes in the performance of grid nodes where their load is dynamically rebalanced during intra-operation execution. In this architecture, each AGQES (Adaptive Grid Query Evaluation Service) combines a *query engine*, and three adaptive components: *Monitoring Event Detector, Diagnoser* and *Responder*. The local query engine of each AGQES sends to the *Monitoring Event Detector* component low-level monitoring information describing the processing cost of a tuple. In its turn, the *Monitoring Event Component* groups the received information and notifies the *Diagnoser* if the change in the performance is higher than a certain threshold. The *Diagnoser* collects the information from the *Monitoring Event Detectors* and if workload imbalance is detected, the *Responder* is notified. In addition, the *Diagnoser* proposes a new workload distribution vector $(w_1, w_2, \ldots, w_n)$, where w_i is the number of tuples sent to processor p_i and n is the number of processors. When the *Responder* receives a notification it decides whether to apply the proposed workload distribution vector or not depending on the progress of execution. If distribution policy must be notified, then the concerned evaluators that produce data and the diagnosers are informed.

The redistribution of data to rebalance the load follows a cache based strategy which is mainly employed to assure fault-tolerance as described in (Smith and Watson 2005). In this fault-tolerance technique, data exchanged between two nodes are cached on the source node until they are fully processed by the destination node. So in AGQES, the assigned load of an overloaded processor is transferred to other processors using the cached data on the source processors which were already sent to the overloaded one.

These approaches, and other adaptive ones presented in section 3.3, are very sensitive to the problem of data skew which may degrade the performance of parallel and distributed algorithms when evaluating join operations. In addition, during join computation, all the tuples of input

relations are redistributed even if they do not participate in the join result. This induces high communication and disk Input/Output costs whenever the exchanged data does not fit in the memory.

3.5 Map-Reduce model and join operation

Map-Reduce model is mainly used to process homogeneous datasets. This model is not quite adequate for evaluating join operations where heterogeneous datasets need to be merged (Pike et al. 2005). However, it can still be used for evaluating such queries using a *homogenization* process (Yang et al. 2007). In this process, to compute the join $R \bowtie S$ of two relations R and S, a map/reduce process is applied on R and another one on S. It adds a tag representing the data source (i.e R or S) into each tuple. It also extracts the common join attribute of R and S. The output of these two processes is homogeneous. So, a final map/reduce process is executed to merge entries having the same values of the join attribute but different data sources tags. This process is not efficient due to the high map-reduce communication costs and the extra disk space used to store the intermediate homogeneous results (Yang et al. 2007). To override these problems an extension of Map-Reduce model called Map-Reduce-Merge model was presented in (Yang et al. 2007). The user of this model must implement three functions *Map*, *Reduce* and *Merge* having the following signatures:

map: $(k_1, v_1)_\alpha \longrightarrow list(k_2, v_2)_\alpha$
reduce: $(k_2, list(v_2))_\alpha \longrightarrow (k_2, list(v_3))_\alpha$
merge: $((k_2, list(v_3))_\alpha, (k_3, list(v_4))_\beta) \longrightarrow list(k_4, v_5)_\gamma$ where α, β and γ represent dataset lineages.

The map and reduce functions in this model are similar to those of Map-Reduce model. The main difference is in the reduce function where the output is a list of key/value instead of just values.
To compute $R \bowtie S$, we apply a Map-Reduce process on R and another one on S. Here α and β are equal because we need to compute an equi-join. The merge function reads the result of the two reducers and combines each pair of couples having the same values of k_2 and k_3.

Hash based algorithms presented in (Yang et al. 2007) are inefficient in the presence of skewed data. In addition, they involve high communication costs since all the tuples of input relations are redistributed across the network.

3.6 Conclusion

In this chapter, we reviewed:
- parallel algorithms for treating join operations and "GroupBy-Join" queries on SN homogeneous systems,

- adaptive parallel algorithms for treating join operations on distributed heterogeneous and grid systems,
- map-reduce-merge cloud computing model for treating join operations.

We also stated their advantages and drawbacks.

In the following chapters, we propose and present our parallel algorithms based on *fully distributed histograms* usage. These algorithms override the drawbacks of algorithms seen in this chapter.

For the case of "GroupBy-Join" queries, we partially apply the group by operation and aggregate function before evaluating the join even when the join and group by attributes are distinct. In addition, we do not fully materialize the join result. This helps us to highly reduce the communication and disk input/output costs.

We also present a dynamic frequency adaptive parallel algorithm for treating join queries on heterogeneous distributed systems. This algorithm follows a two-step *static* and *dynamic* load balancing strategy of different nodes throughout join processing. A pipelined version of this algorithm is also presented for treating complex join queries. In addition, we propose a fault tolerant variant of this algorithm for treating join queries on the grid.

A skew insensitive parallel algorithm based on map-reduce-merge model for treating semi-join operations is also proposed.

In all our algorithms, we use *fully distributed histograms* to determine join attribute values that appear in the join result. Then, only tuples of input relations associated to these values are redistributed over the nodes. This helps to reduce the communication costs to a minimum, in addition to the number of join buckets (and therefore join computation time). Deploying these histograms also provides us with a detailed frequency distribution information of join attribute values. This helps in treating the effect of Attribute Value Skew (AVS) and avoid the Join Product Skew (JPS).

For scalability issue and to guarantee perfect balancing properties during all the join computation steps, communication templates and data redistribution are performed jointly by all the processors, each one is in charge of data redistribution of a subset of the join attribute values and not necessarily its *own values*. This avoids the slowdown of coordinator processors during tasks generation and reallocation in existing dynamic parallel join algorithms.

4 New Approach for evaluating "GroupBy-Join" Queries in Distributed Architectures

Contents

In this chapter, we are interested in treating SQL queries involving join, group-by operation and aggregate functions. Such queries, known as "GroupBy-Join" queries, are fairly common in many decision support applications. In these queries, the aggregate function allows us to obtain summarized data for each group of tuples based on a designated grouping. However, the size of the input relations is usually very large. So, the parallelization of these queries is highly recommended in order to obtain a desirable response time. Several parallel algorithms that treat such queries have been presented in the literature. However, their most significant drawbacks are that they are very sensitive to data skew and involve expensive communication and Input/Output costs in the evaluation of the join operation.
In this chapter, we present two algorithms that overcome these drawbacks because:

- they evaluate "GroupBy-Join" queries without the need to materialize the join result. Thus, they reduce the communication cost and the needed Input/Output access operations to the disks.

- we use a load assignment technique based on exact data distribution information and not statistical ones. This allows us to avoid load imbalance even in the presence of highly skewed data.

The performance of these algorithms is analyzed using the scalable and portable BSP (Bulk Synchronous Parallel) cost model which predicts a near optimal linear speedup even for highly skewed data. These performance predictions were also validated by the practical test results.

4.1 Introduction

In this chapter, we give a detailed presentation of our algorithms: *GAJFA-Join* (Group-by After Join Frequency Adaptive-Join) published in (Hassan and Bamha 2007) and *GBJFA-Join* (Group-by Before Join Frequency Adaptive-Join) published in (Hassan and Bamha 2006) for treating "GroupBy-Join" queries. *GAJFA-Join* algorithm is used for evaluating "GroupBy-Join" queries when group-by and join attributes are different and *GBJFA-Join* for the case where the join attributes are part of the group-by attributes. Our algorithms override the problems of the algorithms presented in the literature (section 3.2.2) for evaluating "GroupBy-Join" queries. Our main contribution is that, in these algorithms, we do not need to materialize the join operation result, in the contrary to traditional algorithms where the join operation is evaluated first and then the group-by and aggregate functions (Yan and Larson 1994). *GAJFA-Join* and *GBJFA-Join* algorithms are also insensitive to AVS and JPS and their communication and disk Input/Output costs are highly reduced.

In these algorithms, we partially evaluate the aggregate function before redistributing the tuples even when the group-by and join attributes are different. This helps in reducing the cost of data redistribution. We also use distributed histograms of both relations in order to find the tuples that participate in the result of the join operation. Thus, we decrease the Input/Output cost during the build and probe phases of computing the join. It is proved in (Bamha and Hains 2005; 1999), using the BSP model, that the histogram management has a negligible cost when compared to the gain it provides in reducing the communication cost.
In traditional algorithms, all the tuples of the output of the join operation are redistributed using a hashing function. In the contrary, in our algorithms, we only redistribute the result of the semi-joins which are, in general, very small compared to the size of input relations. Using semi-join in parallel and distributed machines to evaluate "GroupBy-Join" queries helps in reducing the amount of data transferred over the network, and therefore in decreasing the communication cost (Chen and Yu 1993, Stocker et al. 2001). The performance of *GAJFA-Join* and *GBJFA-Join* algorithms is analyzed using the BSP cost model (section 2.2.2) which predicts for these algorithms a near linear speedup even for highly skewed data. The results of the practical tests assure these performance predictions and prove that it outperforms the traditional algorithms for evaluating such queries. We will present *GAJFA-Join* algorithm in section 4.2 and *GBJFA-Join* in section 4.3.

4.2 The GAJFA-Join algorithm: Evaluating "GroupBy-Join" queries when group-by and join are different

In this section, we present a detailed description of a new parallel algorithm used to evaluate the "GroupBy-Join" queries when the group-by attributes are different from the join attributes. We assume that the relation R (resp. S) is partitioned among the processors by horizontal

fragmentation and that the fragments R_i for $i \in \mathcal{P} = \{1, ..., p\}$ are almost of the same size on every processor, i.e. $|R_i| \simeq \frac{|R|}{p}$ where p is the number of processors.
For simplicity of description, we consider that the query has only one join attribute x and that the group-by attribute set consists of one attribute y of R and another attribute z of S. We also assume that the aggregate function f is applied on the values of the attribute u of S. So, the query can be expressed in SQL as:

SELECT $R.y, S.z, f(S.u)$
FROM R, S
WHERE $R.x = S.x$
GROUP BY $R.y, S.z$;

However, our algorithm can also be used if one or both of $R.y$ and $S.z$ is not included in the query (cf. remark 2 on page 57).

In the rest of this chapter, we use the following notations for each relation $T \in \{R, S\}$:

- T denotes a relation,
- $|T|$ denotes the size (expressed in bytes or number of pages) of T,
- $||T||$ denotes the number of tuples of T,
- h denotes a hash function,
$$h: \begin{array}{l} T \longrightarrow \mathbb{N} \\ t \longmapsto n \end{array}$$
where t is a tuple of T and n is the node's index that t is hashed to,
- T_i denotes the fragment of T placed on node i,
- $c^i_{r/w}$ denotes the cost of reading/writing a page of data from/to the disk of processor i,
- $AGGR^w_{f,u}(T)$ [1] is the result of applying the aggregate function f on the values of the attribute u of every group of T tuples having identical values of the group-by attributes w. $AGGR^w_{f,u}(T)$ is formed of a list of tuples (v, f_v) where f_v is the result of the aggregate function of the group of tuples having value v for the attribute w (w may be formed of more than one attribute),
- $AGGR^w_{f,u}(T_i)$ denotes the result of applying the aggregate function on the attribute u of relation T_i,
- $AGGR^w_{f,u,i}(T)$ is processor i's fragment of the result of applying the aggregate function on T,
- $AGGR^w_{f,u}(T)(v)$ is the result f_v of the aggregate function of the group of tuples having value v for the group-by attribute w in relation T,
- $AGGR^w_{f,u}(T_i)(v)$ is the result f_v of the aggregate function of the group of tuples having value v for the group-by attribute w in sub-relation T_i,
- $Hist^w(T)$ denotes the histogram[2] of relation T with respect to the attribute w, i.e. a list of pairs (v, n_v) where $n_v \neq 0$ is the number of tuples of relation T having the value v for the attribute w. We can

[1] $AGGR^w_{f,u}(T)$ is implemented as a balanced tree (B^+-tree): a data structure that maintains an ordered set of data to allow efficient search and insert operations (cf. Appendix A page 158).

[2] Histograms are implemented as balanced trees (B^+-tree).

see that $Hist^w(T) = AGGR^w_{count,w}(T)$. The histogram is often much smaller and never larger than the relation it describes,
- $Hist^w(T_i)$ denotes the histogram of fragment T_i,
- $Hist^w_i(T)$ is processor i's fragment of the histogram of T,
- $Hist^w(T)(v)$ is the frequency of value v in relation T,
- $Hist^w(T_i)(v)$ is the frequency of value v in sub-relation T_i.

GAJFA-Join algorithm proceeds in six phases. We will give an upper bound of the execution time of each superstep using BSP cost model. The notation $O(\dots)$ hides only small constant factors: they depend only on the program implementation but neither on data nor on the BSP machine parameters.
A running example will be used in order to illustrate the different phases of the algorithm. We consider that relations R and S are partitioned on 3 processors.

Algorithm 4: GAJFA-Join algorithm steps

In Parallel (on each processor) $i \in [1, p]$ **do**
1► Create the local histogram $Hist^{x,y}(R_i)$ of relation R_i and, on the fly, create $Hist'^{x}(R_i)$ which holds the frequency of each value of the attribute x in $Hist^{x,y}(R_i)$ (Algo. 5);
▷ Create the local histogram $AGGR^{x,z}_{f,u}(S_i)$ of relation S_i and, on the fly, create $Hist'^{x}(S_i)$ which holds the frequency of each value of the attribute x in $AGGR^{x,z}_{f,u}(R_i)$;
2► Create $\overline{Hist}^{x,y}(R_i)$ holding tuples of $Hist^{x,y}(R_i)$ that participate to the join result (Algo. 6);
▷ Create $\overline{AGGR}^{x,z}_{f,u}(S_i)$ holding tuples of $AGGR^{x,z}_{f,u}(S_i)$ that participate to the join result;
3► Create communication templates for only tuples participating to final join result,
4► Redistribute $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ as indicated in the communication templates;
5► Create, $AGGR^{y,z}_{f,u}\big((R \bowtie S)_i\big)$, the result of applying locally the aggregate function and the join result on each processor i (Algo. 9);
6► Redistribute $AGGR^{y,z}_{f,u}\big((R \bowtie S)_i\big)$ using a hash function applied on (y, z) to compute globally the aggregate function;
EndPar

Phase 1: **Creating local histograms**
In this phase, the local histograms $Hist^{x,y}(R_i)(i = 1, ..., p)$ of blocks R_i are created, in parallel, by a scan of the fragment R_i on each processor i in time of the order $O(c^i_{r/w} \times \max_{i=1,...,p} |R_i| + \max_{i=1,...,p} ||R_i||)$.
Figure 4.1 shows an example of such local histograms where for each entry $(x_j, y_j, freq_j)$ of $Hist^{x,y}(R_i)$: $freq_j$ represents the number of tuples of R_i having values x_j and y_j for the attributes $R.x$ and $R.y$ respectively.
In addition, the local fragments $AGGR^{x,z}_{f,u}(S_i)(i = 1, ..., p)$ of blocks S_i are also created, in parallel on each processor i, by applying the aggregate function f on every group of tuples having identical values of the couple of attributes (x, z) in time of the order

$\mathbf{Hist^{x,y}(R_1)}$

X	Y	Freq
1	1	100
1	3	300
1	7	200
1	10	50
1	15	40
3	3	40
3	4	100
3	7	100
4	1	150
4	3	100
4	4	200
4	10	50
6	2	100
6	3	50
6	12	50

$\mathbf{Hist^{x,y}(R_2)}$

X	Y	Freq
1	3	200
1	10	100
2	3	200
2	4	50
2	5	50
2	8	100
2	10	100
2	13	20
2	15	30
2	20	50
4	10	100
4	15	200
7	2	100
7	3	50
7	10	13
7	23	100

$\mathbf{Hist^{x,y}(R_3)}$

X	Y	Freq
1	3	100
1	10	50
1	13	20
1	16	30
1	18	100
1	20	50
1	25	50
4	5	80
4	10	80
4	12	100
4	14	100
4	20	50
4	25	50
7	2	150
7	3	140

FIG. 4.1 – *An example of $Hist^{x,y}(R_i)$ of local blocks R_i.*

$O(c^i_{r/w} \times \max_{i=1,...,p} |S_i| + \max_{i=1,...,p} ||S_i||)$. In this algorithm the aggregate function may be MAX, MIN, SUM or $COUNT$. For the aggregate function AVG a similar algorithm that merges the $COUNT$ and the SUM algorithms is applied.
Figure 4.2 represents $AGGR^{x,z}_{f,u}(S_i)$ related to the running example where the *SUM* aggregate function is applied.

In this algorithm, we only redistribute the tuples of $Hist^{x,y}(R_i)$ and

$\mathbf{AGGR^{x,z}_{f,u}(S_1)}$

X	Z	SUM
1	2	200
1	10	150
1	15	300
3	10	50
3	13	600
3	15	200
4	1	200
4	4	140
4	12	250
4	13	600
4	14	100
4	18	20
4	20	100
4	21	200
11	4	150
11	10	20
11	18	200
11	20	100

$\mathbf{AGGR^{x,z}_{f,u}(S_2)}$

X	Z	SUM
1	10	200
1	12	130
3	12	100
3	15	200
4	12	50
4	13	100
4	14	50
4	20	210
4	25	300
4	26	100
4	27	140
4	30	50
4	31	20
4	34	30
4	35	200
4	39	10
4	40	12
4	48	10
12	20	60
12	21	60

$\mathbf{AGGR^{x,z}_{f,u}(S_3)}$

X	Z	SUM
1	12	200
1	18	150
4	12	100
4	13	50
4	20	100
4	25	150
4	30	125
4	33	50
2	10	30
2	15	40
2	21	100
2	25	150
2	30	200
13	1	220
13	10	140
13	15	200
13	20	250
13	25	300
13	29	350

FIG. 4.2 – *An example of $AGGR^{x,z}_{f,u}(S_i)$ related to S_i.*

$AGGR^{x,z}_{f,u}(S_i)$ that participate effectively in the join result. These tuples are determined in phase 2, but we need first to compute the frequency of each value of the attribute x in $Hist^{x,y}(R_i)$ and $AGGR^{x,z}_{f,u}(S_i)$. So, while creating $Hist^{x,y}(R_i)$ (resp. $AGGR^{x,z}_{f,u}(S_i)$), we also create, on the fly, their local histo-

grams $Hist'^{x}(R_i)$ (resp. $Hist'^{x}(S_i)$) with respect to x. Thus, $Hist'^{x}(R_i)$ and $Hist'^{x}(S_i)$ hold respectively the frequency of each value of the attribute x in $Hist^{x,y}(R_i)$ and $AGGR^{x,z}_{f,u}(S_i)$ for $i = 1, ..., p$. Hence, they can be written as:

$$\begin{cases} Hist'^{x}(R_i) &= Hist^{x}\big(Hist^{x,y}(R_i)\big), \\ Hist'^{x}(S_i) &= Hist^{x}\big(AGGR^{x,z}_{f,u}(S_i)\big). \end{cases}$$

In fact, the difference between $Hist^{x}(R_i)$ and $Hist'^{x}(R_i)$ is that $Hist^{x}(R_i)$ holds the frequency of each value of the attribute x in relation R_i (i.e., for each value v of the attribute x, we find the number of tuples of R_i having value v of x), while in $Hist'^{x}(R_i)$ we count tuples having the same values of the attributes (x, y) only once. Figures 4.3 and 4.4 represent $Hist'^{x}(R_i)$ and $Hist'^{x}(S_i)$ related to $Hist^{x,y}(R_i)$ and $AGGR^{x,z}_{f,u}(S_i)$ shown in figures 4.1 and 4.2 respectively.

We use *Algorithm 5* to create $Hist^{x,y}(R_i)$ and $Hist'^{x}(R_i)$. A similar algorithm is used to create $AGGR^{x,z}_{f,u}(S_i)$ and $Hist'^{x}(S_i)$, but instead of computing the frequency of (x, y) to create $Hist^{x,y}(R_i)$, we apply here the aggregate function on (x, z) to create $AGGR^{x,z}_{f,u}(S_i)$.

In principle, this phase costs:

$$Time_{phase1} = O\big(c^{i}_{r/w} \times \max_{i=1,...,p}(|R_i| + |S_i|) + \max_{i=1,...,p}(||R_i|| + ||S_i||)\big).$$

$\mathbf{Hist'^{x}(R_1)}$

X	Freq
1	5
3	3
4	4
6	3

$\mathbf{Hist'^{x}(R_2)}$

X	Freq
1	2
2	8
4	2
7	4

$\mathbf{Hist'^{x}(R_3)}$

X	Freq
1	7
4	6
7	2

FIG. 4.3 – *An example of local histograms* $Hist'^{x}(R_i)$.

$\mathbf{Hist'^{x}(S_1)}$

X	Freq
1	3
3	3
4	8
11	4

$\mathbf{Hist'^{x}(S_2)}$

X	Freq
1	2
3	2
4	14
12	2

$\mathbf{Hist'^{x}(S_3)}$

X	Freq
1	2
4	6
2	5
13	6

FIG. 4.4 – *An example of local histograms* $Hist'^{x}(S_i)$.

Remark 2 *If the group-by attribute R.y is not included in the query, then we only compute* $Hist^{x}(R_i)$ *on each processor i. And in this case, we replace* $Hist^{x,y}(R_i)$ *and* $Hist'^{x}(R_i)$ *by* $Hist^{x}(R_i)$ *in all the remaining steps of the algorithm. A similar treatment also applies if S.z is not included in the query.*

Phase 2: Local semi-joins computation

In order to minimize the communication cost, only tuples of $Hist^{x,y}(R)$

Algorithm 5: Computing $Hist^{x,y}(R_i)$ and $Hist'^{x}(R_i)$

```
▷ Par (on each node in parallel) i = 1, ..., p
  /* Create two B+-trees to store histogram's entries. */
  ▷ Hist^{x,y}(R_i) = NULL ;
  ▷ Hist'^{x}(R_i) = NULL ;
  ▷ for every tuple t of R_i do
    ▷ if Hist^{x,y}(R_i)(t.x, t.y) = 0 then
      ▷ Insert a new tuple (t.x, t.y, 1) into Hist^{x,y}(R_i) ;
      ▷ freq_1 = Hist'^{x}(R_i)(t.x) ;
      ▷ if freq_1 ≠ 0 then
        ▷ Increment the frequency of t.x in Hist'^{x}(R_i) ;
      ▷ else
        ▷ Insert a new tuple (t.x, 1) into Hist'^{x}(R_i) ;
      ▷ endif
    ▷ else
      ▷ Increment the frequency of (t.x, t.y) in Hist^{x,y}(R_i) ;
    ▷ endif
  ▷ endfor
▷ endPar
```

and $AGGR^{x,z}_{f,u}(S)$ that will be present in the join result are redistributed. These tuples are effectively the result of the following local semi-joins:

$$\begin{cases} \overline{Hist}^{x,y}(R_i) & = & Hist^{x,y}(R_i) \ltimes AGGR^{x,z}_{f,u}(S) \quad \text{and} \\ \overline{AGGR}^{x,z}_{f,u}(S_i) & = & AGGR^{x,z}_{f,u}(S_i) \ltimes Hist^{x,y}(R). \end{cases}$$

For scalability issue, we do not duplicate $AGGR^{x,z}_{f,u}(S)$ and $Hist^{x,y}(R)$ on all the processors. So, to compute $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$, we use the following steps.

a: **Computing global partitioned histograms**

First of all, we need to evaluate the global partitioned histograms $Hist'^{x}_i(R)$ and $Hist'^{x}_i(S)$ (fig. 4.5) for $i \in \mathcal{P}$. Hence, the tuples of $Hist'^{x}(R_i)$ and $Hist'^{x}(S_i)$ are redistributed by applying a hash function on the values of the attribute x. This allows us to send the tuples of $Hist'^{x}(R)$ and $Hist'^{x}(S)$ having the same value of x to the same processor. Here, we use the hash function $(x \ mod \ 3) + 1$.

b: **Creating** $Hist'^{x}_i(R \bowtie S)$

After that we create, in parallel on each processor i, $Hist'^{x}_i(R \bowtie S)$ (fig. 4.6). $Hist'^{x}_i(R \bowtie S)$ holds for each value v of the join attribute x, such that $v \in Hist'^{x}_i(R) \cap Hist'^{x}_i(S)$, the couple $\left(v, Hist'^{x}_i(R)(v) \times Hist'^{x}_i(S)(v)\right)$.

Now, we know the values of x that will be present in the result of the join operation. So, to reduce the communication costs, only tuples of $Hist^{x,y}(R_i)_{i\in\mathcal{P}}$ and $AGGR^{x,z}_{f,u}(S_i)_{i\in\mathcal{P}}$ having values of $x \in Hist'^{x}_i(R \bowtie S)$ are redistributed for further treatment.

Algorithm 6: Computing the local semi-joins $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ on each processor i.

In Parallel (on each processor) $i \in [1, p]$ **do**

a► Hash the local histogram $Hist'^{x}(R_i)$ to create global histogram fragment's $Hist'^{x}_i(R)$ on each processor i;

▷ Hash the local histogram $Hist'^{x}(S_i)$ to create global histogram fragment's $Hist'^{x}_i(S)$ on each processor i;

b► Merge $Hist'^{x}_i(R)$ and $Hist'^{x}_i(S)$ to create join histogram fragment's, $Hist'^{x}_i(R \bowtie S)$, by intersecting $Hist'^{x}_i(R)$ and $Hist'^{x}_i(S)$ on each processor i;

▷ Compute $\overline{Hist}'^{x[i]}(R_j) = Hist'^{x[i]}(R_j) \cap Hist'^{x}_i(R \bowtie S)$, where $Hist'^{x[i]}(R_j)$ is the partition of $Hist'^{x}(R_j)$ sent to processor i from processor j ($j \in \{1, \ldots, n\}$);

▷ Compute $\overline{Hist}'^{x[i]}(S_j) = Hist'^{x[i]}(S_j) \cap Hist'^{x}_i(R \bowtie S)$, where $Hist'^{x[i]}(S_j)$ is the partition of $Hist'^{x}(S_j)$ sent to processor i from processor j ($j \in \{1, \ldots, p\}$);

c► Send each partition $\overline{Hist}'^{x[i]}(R_j)$ (resp. $\overline{Hist}'^{x[i]}(S_j)$) from processor i to processor j ($j \in \{1, \ldots, p\}$);

▷ Create $\overline{Hist}^{x,y}(R_i) = Hist^{x,y}(R_i) \cap (\cup_{j=1}^{p} \overline{Hist}'^{x[j]}(R_i))$;

▷ Create $\overline{AGGR}^{x,z}_{f,u}(S_i) = AGGR^{x,z}_{f,u}(R_i) \cap (\cup_{j=1}^{p} \overline{Hist}'^{x[j]}(S_i))$;

Endpar

$Hist'^{x}_1(R)$

X	Freq
3	3
6	3

$Hist'^{x}_2(R)$

X	Freq
1	14
4	12
7	6

$Hist'^{x}_3(R)$

X	Freq
2	8

$Hist'^{x}_1(S)$

X	Freq
3	5
12	2

$Hist'^{x}_2(S)$

X	Freq
1	7
4	28
13	6

$Hist'^{x}_3(S)$

X	Freq
2	5
11	4

FIG. 4.5 – *Global histograms $Hist'^{x}_i(R)$ and $Hist'^{x}_i(S)$ example.*

$\mathbf{Hist'^{x}_1(R \bowtie S)}$

X	Freq
3	15

$\mathbf{Hist'^{x}_2(R \bowtie S)}$

X	Freq
1	98
4	336

$\mathbf{Hist'^{x}_3(R \bowtie S)}$

X	Freq
2	40

FIG. 4.6 – *Partitions of the histogram of $R \bowtie S$ example.*

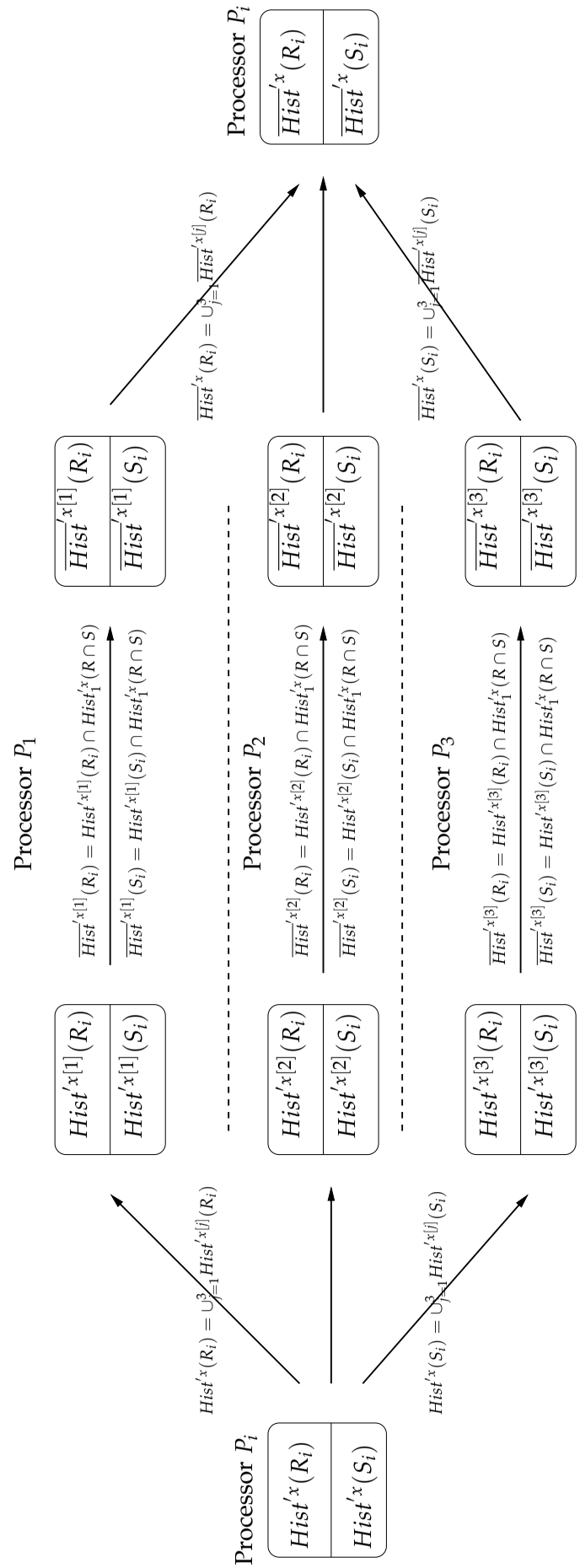

FIG. 4.7 – $\overline{Hist'}^{x[j]}(R_i)$ *and* $\overline{Hist'}^{x[j]}(S_i)$ *computation example.*

c: Creating $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$.

Now, we compute on each processor j, $\overline{\overline{Hist}}^{'x[j]}(R_i)$ and $\overline{\overline{Hist}}^{'x[j]}(S_i)$ for $i \in \mathcal{P}$ (cf. figure 4.7), where:

$$\begin{cases} \overline{\overline{Hist}}^{'x[j]}(R_i) &= Hist^{'x[j]}(R_i) \cap Hist'^{x}_{j}(R \bowtie S) \quad \text{and} \\ \overline{\overline{Hist}}^{'x[j]}(S_i) &= Hist^{'x[j]}(S_i) \cap Hist'^{x}_{j}(R \bowtie S). \end{cases}$$

Here, $Hist^{'x[j]}(R_i)$ is the partition of $Hist'^{x}(R_i)$ which was sent to processor j from processor i in step *(a)* (this also applies on $Hist^{'x[j]}(S_i)$).

After computing $\overline{\overline{Hist}}^{'x[j]}(R_i)$ and $\overline{\overline{Hist}}^{'x[j]}(S_i)$, we perform a communication step where each processor j sends each fragment $\overline{\overline{Hist}}^{'x[j]}(R_i)$ and $\overline{\overline{Hist}}^{'x[j]}(S_i)$ to processor i. After redistribution, each processor i merges the received tuples in order to create $\overline{Hist}'^{x}(R_i)$ which represents the values of x that appear in the join result. Now, we can compute, on each processor i in parallel, $\overline{Hist}^{x,y}(R_i)$ (fig. 4.8) and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ (fig. 4.9). However, we do not need to materialize $AGGR^{x,z}_{f,u}(S)$ and $Hist^{x,y}(R)$ because $\overline{Hist}^{x,y}(R_i)$ is simply $Hist^{x,y}(R_i) \cap \big(\bigcup_{j=1}^{\mathcal{P}} \overline{\overline{Hist}}^{'x[j]}(R_i)\big) = Hist^{x,y}(R_i) \cap \overline{Hist}'^{x}(R_i)$ and similarly $\overline{AGGR}^{x,z}_{f,u}(S_i) = AGGR^{x,z}_{f,u}(S_i) \cap \big(\bigcup_{j=1}^{\mathcal{P}} \overline{\overline{Hist}}^{'x[j]}(S_i)\big) = AGGR^{x,z}_{f,u}(S_i) \cap \overline{Hist}'^{x}(S_i)$.

$\overline{\mathbf{Hist}}^{\mathbf{x,y}}(\mathbf{R_1})$

X	Y	Freq
1	1	100
1	3	300
1	7	200
1	10	50
1	15	40
3	3	40
3	4	100
3	7	100
4	1	150
4	3	100
4	4	200
4	10	50

$\overline{\mathbf{Hist}}^{\mathbf{x,y}}(\mathbf{R_2})$

X	Y	Freq
1	3	200
1	10	100
2	3	200
2	4	50
2	5	50
2	8	100
2	10	100
2	13	20
2	15	30
2	20	50
4	10	100
4	15	200

$\overline{\mathbf{Hist}}^{\mathbf{x,y}}(\mathbf{R_3})$

X	Y	Freq
1	3	100
1	10	50
1	13	20
1	16	30
1	18	100
1	20	50
1	25	50
4	5	80
4	10	80
4	12	100
4	14	100
4	20	50
4	25	50

FIG. 4.8 – $\overline{Hist}^{x,y}(R_i)$ *($i \in \{1,2,3\}$) example.*

The cost of this phase is:

$$\begin{aligned} Time_{phase2} = O\Big(& \max_{i=1,\ldots,p} ||Hist^{x,y}(R_i)|| + \max_{i=1,\ldots,p} ||AGGR^{x,z}_{f,u}(S_i)|| + \\ & \min\big(g \times |Hist^{x}(R)| + ||Hist^{x}(R)||, g \times \tfrac{|R|}{p} + \tfrac{||R||}{p}\big) + \\ & \min\big(g \times |Hist^{x}(S)| + ||Hist^{x}(S)||, g \times \tfrac{|S|}{p} + \tfrac{||S||}{p}\big) + l\Big), \end{aligned}$$

where g is the BSP communication parameter and l the cost of a barrier of synchronization.

We recall (cf. proposition 1 of Appendix A) that, in the above equation,

$\overline{\mathbf{AGGR}}^{\mathbf{x,z}}_{\mathbf{f,u}}(\mathbf{S_1})$

X	Z	SUM
1	2	200
1	10	150
1	15	300
3	10	50
3	13	600
3	15	200
4	1	200
4	4	140
4	12	250
4	13	600
4	14	100
4	18	20
4	20	100
4	21	200

$\overline{\mathbf{AGGR}}^{\mathbf{x,z}}_{\mathbf{f,u}}(\mathbf{S_2})$

X	Z	SUM
1	10	200
1	12	130
3	12	100
3	15	200
4	12	50
4	13	100
4	14	50
4	20	210
4	25	300
4	26	100
4	27	140
4	30	50
4	31	20
4	34	30
4	35	200
4	39	10
4	40	12
4	48	10

$\overline{\mathbf{AGGR}}^{\mathbf{x,z}}_{\mathbf{f,u}}(\mathbf{S_3})$

X	Z	SUM
1	12	200
1	18	150
4	12	100
4	13	50
4	20	100
4	25	150
4	30	125
4	33	50
2	10	30
2	15	40
2	21	100
2	25	150
2	30	200

FIG. 4.9 – $\overline{AGGR}^{x,z}_{f,u}(S_i)$ *(*$i \in \{1,2,3\}$*) example.*

the terms:

$$\min\left(g \times |Hist^x(R)| + ||Hist^x(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\right),$$

and

$$\min\left(g \times |Hist^x(S)| + ||Hist^x(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}\right),$$

represent the necessary time to compute the global histograms, $Hist^x_{i=1,...,p}(R)$ and $Hist^x_{i=1,...,p}(S)$ respectively, starting from the local histograms $Hist^x(R_i)$ and $Hist^x(S_i)$ $(i = 1, ..., p)$. However, the number of tuples of $Hist'^x(R_i)$ and that of $Hist^x(R_i)$ are equal, what differs is only the value of the frequency attribute in these histograms. So, $|Hist'^x(R_i)| = |Hist^x(R_i)|$ (this also applies to $Hist'^x(S_i)$ and $Hist^x(S_i)$). Hence, the above cost still holds for computing $Hist'^x_{i=1,...,p}(R)$ and $Hist'^x_{i=1,...p}(S)$.

During semi-join computation, we store on each processor i, an extra information called $index(v) \in \{LF, HF^R, HF^S\}$ for each value $v \in Hist'^x_i(R \ltimes S)$. This information allows us to decide if, for a given value v, the frequencies of tuples of $Hist^{x,y}(R)$ and $AGGR^{x,z}_{f,u}(S)$ having the value v are greater (resp. lesser) than a threshold frequency f_0. It also permits us to choose dynamically the probe and the build relation for each value v of the join attribute. This choice highly reduces the global redistribution cost. In this algorithm, by evaluating $AGGR^{x,z}_{f,u}(S)$, we partially apply the aggregate function on the attribute u of S and thus reducing the volume of data. This also applies to $Hist^{x,y}(R)$, where all tuples having the same values of (x, y) are represented by a single tuple, but we will still consider that the frequencies of some tuples of $AGGR^{x,z}_{f,u}(S)$ and $Hist^{x,y}(R)$ having a value v of the attribute x is high. So, in order to balance the load between all the processors, these tuples must be evenly redistributed.

In the rest of this chapter, we use the threshold frequency: $f_0 = p \times log(p)$. On each processor i, for a given value $v \in Hist_i'^{x}(R) \cap Hist_i'^{x}(S)$,

- the value $index(v) = LF$, means that the frequency of tuples of relations $Hist_i^{x,y}(R)$ and $AGGR_{f,u,i}^{x,z}(S)$ associated to value v are less than the threshold frequency (i.e. $Hist_i'^{x}(R)(v) < f_0$ and $Hist_i'^{x}(S)(v) < f_0$),
- the value $index(d) = HF^S$, means that $Hist_i'^{x}(S)(v) \geq f_0$ and $Hist_i'^{x}(S)(v) > Hist_i'^{x}(R)(v)$,
- the value $index(d) = HF^R$, means that $Hist_i'^{x}(R)(v) \geq f_0$ and $Hist_i'^{x}(R)(v) \geq Hist_i'^{x}(S)(v)$.

Note that unlike the algorithms presented in (Shatdal and Naughton 1995, Taniar et al. 2000) where both relations R and S are redistributed, we will only redistribute $Hist^{x,y}(R_i) \ltimes AGGR_{f,u}^{x,z}(S)$ and $AGGR_{f,u}^{x,z}(S_i) \ltimes Hist^{x,y}(R)$ to find the final result. This will highly reduce the communication costs. It also helps to reduce the number of join buckets and thus the disk access costs. At the end of this phase, we will divide the semi-joins $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}_{f,u}^{x,z}(S_i)$ on each processor i into three sub-histograms in the following way:

$$\overline{Hist}^{x,y}(R_i) = \bigcup_{m \in \{HASH,PAR,DUP\}} \overline{Hist}^{(m)x,y}(R_i)$$

and

$$\overline{AGGR}_{f,u}^{x,z}(S_i) = \bigcup_{m \in \{HASH,PAR,DUP\}} \overline{AGGR}_{f,u}^{(m)x,z}(S_i)$$

where:

- All the tuples of $\overline{Hist}^{(PAR)x,y}(R_i)$ (resp. $\overline{AGGR}_{f,u}^{(PAR)x,z}(S_i)$) are associated to values v such that $index(v) = HF^R$ (resp. $index(v) = HF^S$),
- All the tuples of $\overline{Hist}^{(DUP)x,y}(R_i)$ (resp. $\overline{AGGR}_{f,u}^{(DUP)x,z}(S_i)$) are associated to values v such that $index(v) = HF^S$ (resp. $index(v) = HF^R$),
- All the tuples of $\overline{Hist}^{(HASH)x,y}(R_i)$ and $\overline{AGGR}_{f,u}^{(HASH)x,z}(S_i)$ are associated to values v such that $index(v) = LF$, i.e. the tuples associated to values which occur with frequencies less than a threshold frequency f_0 in both relations R and S.

Tuples of $\overline{Hist}^{(PAR)x,y}(R_i)$ and $\overline{AGGR}_{f,u}^{(PAR)x,z}(S_i)$ are associated to high frequencies for the join attribute. These tuples have an important effect on Attribute Value Skew (AVS) and Join Product Skew (JPS). So, we will use an appropriate redistribution technique in order to efficiently treat the effect of AVS and avoid the problem of JPS. To this end, tuples related to each join attribute value in $\overline{Hist}^{(PAR)x,y}(R_i)$ (resp. $\overline{AGGR}_{f,u}^{(PAR)x,z}(S_i)$) will be evenly partitioned over the processors. In order to obtain a valid join result, we are obliged to duplicate associate tuples of $\overline{AGGR}_{f,u}^{(DUP)x,z}(S_i)$ (resp $\overline{Hist}^{(DUP)x,y}(R_i)$) on the p processors. The tuples of relations $\overline{Hist}^{(HASH)x,y}(R_i)$ and $\overline{AGGR}_{f,u}^{(HASH)x,z}(S_i)$ (are associated to very low frequencies for the join attribute) have no effect neither on AVS nor JPS. These tuples will be redistributed using a hash function.

Phase 3: Creating the communication templates

Skewed attribute values (those having high frequencies) are also those which may cause join product skew in standard join algorithms. To avoid

the slowdown usually caused by AVS and the imbalance of the size of local joins, we use an appropriate treatment for high attribute frequencies as follows:

3.a We partition the histogram $Hist'^{x}(R \bowtie S)$ into two sub-histograms: $Hist^{(HF)'x}(R \bowtie S)$ and $Hist^{(LF)'x}(R \bowtie S)$ in the following manner:
- the values $v \in Hist^{(HF)'x}(R \bowtie S)$ are associated to high frequencies of the join attribute (i.e. $index(v) = HF^R$ or $index(v) = HF^S$),
- the values $v \in Hist^{(LF)'x}(R \bowtie S)$ are associated to low frequencies of the join attribute (i.e. $index(v) = LF$),

this partition step is performed in parallel, on each processor i, by a local traversal of the histogram $Hist'^{x}_i(R \bowtie S)$ in time:

$$Time_{3.a} = O\Big(\max_{i=1,\ldots,p} ||Hist'^{x}_i(R \bowtie S)||\Big).$$

3.b Communication templates for high frequencies:
We create the communication template: *the list of messages which constitutes the relations' redistribution*. This step is performed jointly by all processors, each one not necessarily computing the list of its own messages, so as to balance the overall process. So, each processor i computes a set of necessary messages related to the values v it owns in $Hist_i^{(HF)'x}(R \bowtie S)$. Communication template is derived by applying algorithm 7 on the tuples of relations $\overline{Hist}^{(PAR)x,y}(R)$ which is mapped to multiple nodes. We also apply the same algorithm to compute the communication template of $\overline{AGGR}^{(PAR)x,z}_{f,u}(S)$, but we replace $Hist'^{x}(R)$ by $Hist'^{x}(S)$.

Algorithm 7: Communication templates for $\overline{Hist}^{(PAR)x,y}(R)$.

▷**for each** couple $(v, n_v) \in Hist'^{x}(R)$ **do**
 ▷ **if** $(n_v \mod p = 0)$ **then**
  ▷ each processor j will hold a block of size $block_j(v) = \dfrac{n_v}{p}$ of tuples associated to value v;
 ▷ **else**
  ▷ Pick a random value j_0 between 0 and $(p-1)$;
  ▷ **if** $\Big($processor's index j is between j_0 and $j_0 + (n_v \mod p)\Big)$ **then**
   ▷ the processor of index j will hold a block of size: $block_j(v) = \lfloor \dfrac{n_v}{p} \rfloor + 1$;
  ▷ **else**
   ▷ the processor of index j will hold a block of size: $block_j(v) = \lfloor \dfrac{n_v}{p} \rfloor$;
  ▷ **endif**
 ▷ **endif**
▷**endfor**

In the above algorithm, $\lfloor x \rfloor$ is the largest integral value not greater than x and $block_j(v)$ is the number of tuples of value v that processor j should own after redistribution of the fragments T_i of relation T. We have chosen to use a random value of j_0 for each value v in order to avoid sending

the additional tuples to the same set of processors (these whose index is between j_0 and $block_j(v) = \lfloor \frac{n_v}{p} \rfloor + 1$).
The absolute value of $Rest_j(v) = Hist_j(T)(v) - block_j(v)$ determines the number of tuples of value v that processor j must send (if $Rest_j(v) > 0$) or receive (if $Rest_j(v) < 0$). For $v \in Hist_i^{(HF)'x}(R \bowtie S)$, processor i owns a description of the layout of tuples of value v over the network. It may therefore determine the number of tuples of value v which every processor must send/receive. This information constitutes the communication templates. This step is thus completed in time:

$$Time_{3.b} = O\Big(||Hist^{(HF)'x}(R \bowtie S)||\Big).$$

The tuples associated to low frequencies (i.e. tuples having $v \in Hist_i^{(LF)'x}(R \bowtie S)$) have no effect neither on the AVS nor on the JPS. These tuples are simply mapped to processors using a hash function and thus, no communication template computation is needed.

The creation of the communication templates has, therefore, taken the sum of the above two steps:

$$Time_{phase3} = O\big(\max_{i=1,\dots,p} ||Hist_i^{'x}(R \bowtie S)|| + ||Hist^{(HF)'x}(R \bowtie S)||\big).$$

Phase 4: Data redistribution
4.a Redistribution of tuples having $v \in Hist_i^{(HF)'x}(R \bowtie S)$:

Each processor i holds, for each one of its local values $v \in Hist_i^{(HF)'x}(R \bowtie S)$, the non-zero communication volumes it prescribes as a part of communication template: $Rest_j(v) \neq 0$ for $j = 1, \dots, p$. This information will take the form of *sending orders* forwarded to their target processors in a first superstep, followed by the actual redistribution superstep where each processor obeys all the received orders.
Each processor i, first splits the processors indices j into two groups: those for which $Rest_j(v) > 0$ and those for which $Rest_j(v) < 0$. This is done by a sequential traversal of the $Rest_j(v)$ array.
Let α (resp. β) be the number of j's where $Rest_j(v)$ is positive (resp. negative) and $Proc(k)_{k=1,\dots,\alpha+\beta}$ the array of processor indices for which $Rest_j(v) \neq 0$ in the manner that: $Rest_{proc(j)}(v) > 0$ *for* $j = 1, \dots, \alpha$ and $Rest_{proc(j)}(v) < 0$ *for* $j = 1, \dots, \alpha + \beta$. A sequential traversal of $Proc(k)_{k=1,\dots,\alpha+\beta}$ determines the number of tuples that each processor j will send. The sending orders related to a value v are computed using algorithm 8. Figure 4.10 gives an example of the value *Rest* associated to a value of the join attribute and the corresponding sending orders.

The maximal complexity of this algorithm is: $O\Big(||Hist^{(HF)'x}(R \bowtie S)||\Big)$, because for a given v, no more than $(p-1)$ processors can send data and each processor i is in charge of redistribution of tuples having $v \in Hist_i^{(HF)'x}(R \bowtie S)$. For each processor i and $v \in Hist_i^{(HF)'x}(R \bowtie S)$, all the order_to_send$(j, i, \dots)$ are sent to processor j when $j \neq i$ in time:

$$O\big(g \times |Hist^{(HF)'x}(R \bowtie S)| + l\big).$$

Thus, this step costs:

$$Time_{phase4.a} = O\big(g \times |Hist^{(HF)'x}(R \ltimes S)| + ||Hist^{(HF)'x}(R \ltimes S)|| + l\big).$$

Algorithm 8: Sending orders for join attribute value v.

▷ $i := 1$;
▷ $j := \alpha + 1$;
▷ **while** $(i \leq \alpha)$ **do**
  ▷ n_tuples = min($Rest_{proc(i)}(v)$, $-Rest_{proc(j)}(v)$);
  ▷ order_to_send(Proc(i),Proc(j),v,n_tuples);
  ▷ $Rest_{proc(i)}(v)$:= $Rest_{proc(i)}(v)$ - n_tuples;
  ▷ $Rest_{proc(j)}(v)$:= $Rest_{proc(j)}(v)$ + n_tuples;
  ▷ **if** $Rest_{proc(i)}(v) = 0$ **then** $i := i + 1$; **endif**
  ▷ **if** $Rest_{proc(j)}(v) = 0$ **then** $j := j + 1$; **endif**
▷ **endwhile**

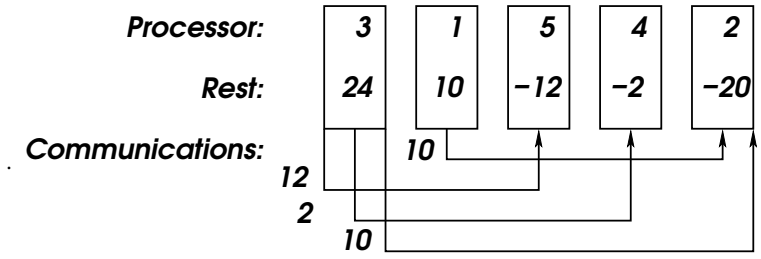

FIG. 4.10 – *Sending orders as a function of Rest values.*

4.b Redistribution of tuples with values $v \in Hist_i^{(LF)'x}(R \ltimes S)$:
Tuples of $\overline{Hist}^{(HASH)x,y}(R_i)$ and $\overline{AGGR}_{f,u}^{(HASH)x,z}(S_i)$ (i.e. tuples having $v \in Hist_i^{(LF)'x}(R \ltimes S)$) are associated to low frequencies. So, they have no effect neither on the AVS nor the JPS. These relations are redistributed using a hash function.

At the end of steps 4.*a* and 4.*b*, each processor i, has local knowledge of how the tuples of semi-joins $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}_{f,u}^{x,z}(S_i)$ will be redistributed. Redistribution is then performed in time:

$$Time_{phase4.b} = O\Big(g \times \big(|\overline{Hist}^{x,y}(R_i)| + |\overline{AGGR}_{f,u}^{x,z}(S_i)|\big) + l\Big).$$

Thus, the total cost of the redistribution phase is the sum of the costs of the above two steps:

$$Time_{phase4} = O\Big(g \times \max_{i=1,\ldots,p} \big(|\overline{Hist}^{x,y}(R_i)| + |\overline{AGGR}_{f,u}^{x,z}(S_i)| + |Hist^{(HF)'x}(R \ltimes S)|\big) + ||Hist^{(HF)'x}(R \ltimes S)|| + l\Big).$$

We mention that, we only redistribute the tuples of the semi-joins

$\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$, where $|\overline{Hist}^{x,y}(R_i)|$ and $|\overline{AGGR}^{x,z}_{f,u}(S_i)|$ are generally very small compared to $|R_i|$ and $|S_i|$. In addition, $|Hist'^{x}(R \bowtie S)|$ is generally very small compared to $|Hist^{x,y}(R)|$ and $|AGGR^{x,z}_{f,u}(S)|$. Thus, we highly reduce the communication cost.

Algorithm 9: Join and local aggregate function computation.

▷ **Par** (on each node in parallel) $i = 1, \ldots, p$
▷ $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$ = NULL;
/* Create a B-tree to store histogram's entries*/
▷ **for each** tuple t of relation $\overline{Hist}^{(PAR)x,y}(R_i)$ **do**
▷ $freq = \overline{Hist}^{(PAR)x,y}(R_i)(t.x, t.y)$;
▷ **for each** entry $(t.x, z, v_1) \in \overline{AGGR}^{(DUP)x,z}_{f,u}(S)(t.x, z)$ **do**
▷ $v_2 = AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z)$;
▷ **if** $v_2 \neq 0$ **then**
▷ Update $AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z) = f(F_f(freq, v_1), v_2)$;
▷ **else**
▷ Insert a new tuple $(t.y, z, F_f(freq, v_1))$ into the histogram $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$;
▷ **endif**
▷ **endfor**
▷ **endfor**
▷ **for each** tuple t of relation $\overline{AGGR}^{(DUP)x,z}_{f,u}(S_i)$ **do**
▷ $f_v = \overline{AGGR}^{(DUP)x,z}_{f,u}(t.x, t.z)$;
▷ **for each** entry $(t.x, y, v_1) \in \overline{Hist}^{(PAR)x,y}(R)$ **do**
▷ $v_2 = AGGR^{y,z}_{f,u}((R \bowtie S)_i)(y, t.z)$;
▷ **if** $v_2 \neq 0$ **then**
▷ Update $AGGR^{y,z}_{f,u}((R \bowtie S)_i)(y, t.z) = f(F_f(v_1, f_v), v_2)$;
▷ **else**
▷ Insert a new tuple $(y, t.z, F_f(t.f_v, v_1))$ into the histogram $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$;
▷ **endif**
▷ **endfor**
▷ **endfor**
▷ **for each** tuple t of relation $\overline{Hist}^{(HASH)x,y}(R_i)$ **do**
▷ $freq = \overline{Hist}^{(HASH)x,y}(R_i)(t.x, t.y)$;
▷ **for each** entry $(t.x, z, v_1) \in \overline{AGGR}^{(HASH)x,z}_{f,u}(S_i)$ **do**
▷ $v_2 = AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z)$;
▷ **if** $v_2 \neq 0$ **then**
▷ Update $AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z) = f(F_f(freq, v_1), v_2)$;
▷ **else**
▷ Insert a new tuple $(t.y, z, F_f(freq, v_1))$ into the histogram $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$
▷ **endif**
▷ **endfor**
▷ **endfor**
▷ **endpar**

Phase 5: **local computation of the aggregate function**

At this step, each processor has partitions of $\overline{Hist}^{x,y}(R)$ and $\overline{AGGR}^{x,z}_{f,u}(S)$. The tuples of $\overline{Hist}^{(PAR)x,y}(R_i)$, $\overline{Hist}^{(DUP)x,y}(R_i)$, $\overline{Hist}^{(HASH)x,y}(R_i)$ can be joined with the tuples of $\overline{AGGR}^{(DUP)x,z}_{f,u}(S_i)$, $\overline{AGGR}^{(PAR)x,z}_{f,u}(S_i)$, $\overline{AGGR}^{(HASH)x,z}_{f,u}(S_i)$ respectively. But, the frequencies of tuples of $\overline{Hist}^{(PAR)x,y}(R_i)$ and $\overline{AGGR}^{(PAR)x,z}_{f,u}(S_i)$ are, by definition, greater than the corresponding (matching) tuples in $\overline{Hist}^{(DUP)x,y}(R_i)$ and $\overline{AGGR}^{(DUP)x,z}_{f,u}(S_i)$ respectively. So, we will choose $\overline{Hist}^{(PAR)x,y}(R_i)$ and $\overline{AGGR}^{(PAR)x,z}_{f,u}(S_i)$ as the *build* relations and $\overline{Hist}^{(DUP)x,y}(R_i)$ and $\overline{AGGR}^{(DUP)x,z}_{f,u}(S_i)$ as *probe* relations. Hence, we need to duplicate the probe relations to all processors in time:

$$Time_{phase5.a} = O\Big(g \times \big(|\overline{Hist}^{(DUP)x,y}(R)| + |\overline{AGGR}^{(DUP)x,z}_{f,u}(S)|\big) + l\Big).$$

Now, using *Algorithm 9*, we are able to compute the local aggregate function, on each processor, without the necessity to fully materialize the intermediate results of the join operation. In this algorithm, we denote the aggregate function by F and we use the following binary function where $\mathfrak{D}$ is the domain of the aggregate attribute:

$$F_f : \mathbb{N} \times \mathfrak{D} \longrightarrow \mathfrak{D}$$

such that:

$$F_f(n,v) = \begin{cases} n \times v & \text{if} \quad f \in \{SUM, COUNT\} \\ v & \text{if} \quad f \in \{MIN, MAX, AVG\} \end{cases}$$

In this algorithm, we create on each processor i, the relation $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$ that holds the local results of applying the aggregate function on every group of tuples having the value of the couple of attributes (y,z). $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$ has the form (y,z,v) where y and z are the group-by attributes and v is the result of the aggregate function.

Figure 4.11 shows an example of applying this algorithm on processor 1. For simplicity of presentation, we use histograms $\overline{Hist}^{x,y}(R_1)$ and $\overline{AGGR}^{x,z}_{f,u}(S_1)$ having small sizes.
The cost of applying this algorithm is:

$$\begin{aligned} Time_{phase5.b} = O\Big(c^i_{r/w} \times \max_{i=1,\ldots,p} \big(&|\overline{Hist}^{(PAR)x,y}(R_i) \ltimes \overline{AGGR}^{(DUP)x,z}_{f,u}(S)| + \\ &|\overline{Hist}^{(DUP)x,y}(R) \ltimes \overline{AGGR}^{(PAR)x,z}_{f,u}(S_i)| + \\ &|\overline{Hist}^{(HASH)x,y}(R_i) \ltimes \overline{AGGR}^{(HASH)x,z}_{f,u}(S_i)|\big)\Big). \end{aligned}$$

So, the total cost of this phase is:

$$\begin{aligned} Time_{phase5} = O\Big(&g \times \big(|\overline{Hist}^{(DUP)x,y}(R)| + |\overline{AGGR}^{(DUP)x,z}_{f,u}(S)|\big) + l + \\ &c^i_{r/w} \times \max_{i=1,\ldots,p} \big(|\overline{Hist}^{(PAR)x,y}(R_i) \ltimes \overline{AGGR}^{(DUP)x,z}_{f,u}(S)| + \\ &|\overline{Hist}^{(DUP)x,y}(R) \ltimes \overline{AGGR}^{(PAR)x,z}_{f,u}(S_i)| + \\ &|\overline{Hist}^{(HASH)x,y}(R_i) \ltimes \overline{AGGR}^{(HASH)x,z}_{f,u}(S_i)|\big)\Big). \end{aligned}$$

Phase 6: **global computation of the aggregate function**

In this phase, a global application of the aggregate function is carried

$\overline{\mathbf{Hist}}^{\mathbf{x,y}}(\mathbf{R_1})$

X	Y	$Freq$
1	1	100
1	3	300
1	3	200
1	3	100
2	3	200
2	4	50
2	5	50
3	3	40
3	4	100
3	7	100
4	1	150
4	3	100
4	4	200
4	10	50
4	10	100
4	15	200
4	5	80
4	10	80
4	12	100
4	14	100
4	20	50
4	25	50

$\overline{\mathbf{AGGR}}^{\mathbf{x,z}}_{\mathbf{f,u}}(\mathbf{S_1})$

X	Z	SUM(U)
1	2	200
1	10	150
1	15	300
1	10	200
1	12	130
1	12	200
1	18	150
2	10	30
2	15	40
2	21	100
2	25	150
2	30	200
3	10	50
4	1	200
4	4	140
4	12	250
4	12	50
4	13	100
4	14	50
4	12	100
4	13	50
4	20	100

$\mathbf{AGGR}^{\mathbf{y,z}}_{\mathbf{f,u}}((\mathbf{R} \ltimes \mathbf{S})_\mathbf{1})$

Y	Z	Freq * SUM(U)
1	2	$100 \times 200 + \ldots$
1	10	$100 \times 150 + 100 \times 200 + \ldots$
1	15	$100 \times 300 + \ldots$
1	12	$100 \times 130 + 100 \times 200 + 150 \times 250 + 150 \times 50 + \ldots$
1	18	$100 \times 150 + \ldots$
3	2	$300 \times 200 + 200 \times 200 + 100 \times 200 + \ldots$
3	10	$300 \times 150 + 300 \times 200 + 200 \times 150 + 200 \times 200 + 100 \times 150 + 100 \times 200 + \ldots$
3	15	$300 \times 300 + 200 \times 300 + 100 \times 300 + \ldots$
3	12	$300 \times 130 + 300 \times 200 + 200 \times 130 + 200 \times 200 + 100 \times 130 + 100 \times 200 + \ldots$
3	18	$300 \times 150 + 200 \times 150 + 100 \times 150 + \ldots$
...		

FIG. 4.11 – *An example of applying the join operation and aggregate function.*

out. For this purpose, every processor redistributes the local aggregation results, $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$, using a common hashing function. The input attributes of the hashing function are y and z. After hashing, every processor applies the aggregate function on the received messages in order to compute the global result $AGGR^{y,z}_{f,u}(R \ltimes S)$.

$AGGR^{y,z}_{f,u}(R \ltimes S)$ is formed of three attributes. The first two are the group-by attributes (y and z) and the third is the result of applying the aggregate function. The time of this step is:

$$Time_{phase6} = O\Big(\min \big(g \times |AGGR^{y,z}_{f,u}(R \ltimes S)| + ||AGGR^{y,z}_{f,u}(R \ltimes S)||, \\ g \times \frac{|R \ltimes S|}{p} + \frac{||R \ltimes S||}{p}\big) + l\Big).$$

In this step, we apply the same result used to redistribute the histograms (cf. proposition 1 of Appendix A) in redistributing $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$.

The global cost of evaluating the "GroupBy-Join" queries in this algorithm is the sum of redistribution cost and local computation of aggregate function. It is of the order:

$$
\begin{aligned}
Time_{total} = O\Big(& c^{i}_{r/w} \times \max_{i=1,\dots,p}(|R_i| + |S_i|) + \max_{i=1,\dots,p}(||R_i|| + ||S_i||) + \\
& \min\big(g \times |Hist^{x}(R)| + ||Hist^{x}(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\big) + \\
& \min\big(g \times |Hist^{x}(S)| + ||Hist^{x}(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}\big) + \\
& ||Hist^{(HF)'x}(R \bowtie S)|| + \max_{i=1,\dots,p} ||AGGR^{x,z}_{f,u}(S_i)|| + \\
& g \times \max_{i=1,\dots,p}\Big(|\overline{Hist}^{x,y}(R_i)| + |\overline{AGGR}^{x,z}_{f,u}(S_i)| + |Hist^{(HF)'x}(R \bowtie S)|\Big) + \\
& g \times \big(|\overline{Hist}^{(DUP)x,y}(R)| + |\overline{AGGR}^{(DUP)x,z}_{f,u}(S)|\big) + \\
& \min\big(g \times |AGGR^{y,z}_{f,u}(R \bowtie S)| + ||AGGR^{y,z}_{f,u}(R \bowtie S)||, g \times \frac{|R \bowtie S|}{p} + \\
& \frac{||R \bowtie S||}{p}\big) + c^{i}_{r/w} \times \max_{i=1,\dots,p}\big(|\overline{Hist}^{(PAR)x,y}(R_i) \bowtie \overline{AGGR}^{(DUP)x,z}_{f,u}(S)| + \\
& |\overline{Hist}^{(DUP)x,y}(R) \bowtie \overline{AGGR}^{(PAR)x,z}_{f,u}(S_i)| + \max_{i=1,\dots,p} ||Hist^{x,y}(R_i)|| + \\
& |\overline{Hist}^{(HASH)x,y}(R_i) \bowtie \overline{AGGR}^{(HASH)x,z}_{f,u}(S_i)|\big) + l\Big).
\end{aligned}
$$

Remark 3 *In the traditional algorithms, the aggregate function is applied on the output of the join operation. The sequential evaluation of the "groupBy-Join" queries requires at least the following lower bound:*

$$bound_{inf_1} = \Omega\big(c^{i}_{r/w} \times (|R| + |S| + |R \bowtie S|)\big).$$

Parallel processing with p processors requires therefore:

$$bound_{inf_p} = \frac{1}{p} \times bound_{inf_1}.$$

Using our approach, the evaluation of the "GroupBy-Join" queries when the join attributes are different from the group-by attributes has an optimal asymptotic complexity when:

$$max\big(|\overline{Hist}^{(DUP)x,y}(R)|, |\overline{AGGR}^{(DUP)x,z}_{f,u}(S)|, |Hist^{(HF)'x}(R \bowtie S)|\big) \leq c^{i}_{r/w} \times max\big(\frac{|R|}{p}, \frac{|S|}{p}, \frac{|R \bowtie S|}{p}\big),$$

this is due to the fact that the local join results have almost the same size and all the terms in $Time_{total}$ are bounded by those of $bound_{inf_p}$. This inequality holds if we choose a threshold frequency f_0 greater than p (which is the case for our threshold frequency $f_0 = p \times log(p)$).

4.3 The GBJFA-Join algorithm: Evaluating "GroupBy-Join" queries when join attributes are part of group-by attributes

In this section, we present a detailed description of our parallel algorithm *GBJFA-Join* used to evaluate "GroupBy-Join" queries when the join attributes are part of the group-by attributes. We assume that the relation R (resp. S) is partitioned among the processors by horizontal fragmentation and that the fragments R_i for $i = 1, ..., p$ are almost of the same size on each processor, i.e. $|R_i| \simeq \frac{|R|}{p}$ where p is the number of processors. For simplicity of description, we consider that the query has only one join attribute x and that the group-by attribute set consists of x, an attribute y of R and another attribute z of S . We also assume that the aggregate function f is applied on the values of the attribute u of S. So, the treated query is the following:

Select $R.x, R.y, S.z, f(S.u)$
From R, S
Where $R.x = S.x$
Group By $R.x, R.y, S.z$;

This algorithm can also be used if one or both of $R.y$ and $S.z$ is not included in the query.

In the rest of this section, we present the algorithm which proceeds in four phases. We also give an upper bound of the execution time of each step.

Phase 1: Creating local histograms
In this phase, the local histograms $Hist^x(R_i)_{i=1,...,p}$ (resp. $Hist^x(S_i)_{i=1,...,p}$) of blocks R_i (resp. S_i) are created, in parallel, by a scan of the fragment R_i (resp. S_i), on processor i, in time $c^i_{r/w} \times \max_{i=1,...,p} |R_i| + \max_{i=1,...,p} ||R_i||$ (resp. $c^i_{r/w} \times \max_{i=1,...,p} |S_i| + \max_{i=1,...,p} ||S_i||$) where $c^i_{r/w}$ is the cost of writing/reading a page of data from disk.

In addition, the local fragments $AGGR^{x,z}_{f,u}(S_i)_{i=1,...,p}$ of blocks S_i are created, on the fly, while scanning relation S_i in parallel, on each processor i, by applying the aggregate function f on every group of tuples having identical values of the couple of attributes (x, z). At the same time, the local histograms $Hist^{x,y}(R_i)_{i=1,...,p}$ are also created.
In principle, this phase costs:

$$Time_{phase1} = O\big(c^i_{r/w} \times \max_{i=1,...,p}(|R_i| + |S_i|) + \max_{i=1,...,p}(||R_i|| + ||S_i||)\big).$$

Phase 2: Creating the histogram of $R \ltimes S$
The first step, in this phase, is to create the histograms $Hist^x_i(R)$ and $Hist^x_i(S)$. To this end, each processor i partitions the local histograms $Hist^x(R_i)$ and $Hist^x(S_i)$ into p partitions $Hist^{x[j]}(R_i)(j = 1, \ldots, p)$ and $Hist^{x[j]}(S_i)(j = 1, \ldots, p)$ using a hash function. We can see that $Hist(R_i) = \cup_{j=1}^{p} Hist^{x[j]}(R_i)$ and $Hist(S_i) = \cup_{j=1}^{p} Hist^{x[j]}(S_i)$. Then, each

processor i sends to each other processor j a fragment $Hist^{x[j]}(R_i)$ (resp. $Hist^{x[j]}(S_i)$) of $Hist^x(R_i)$ (resp. $Hist^x(S_i)$). After redistributing these partitions, each processor i merges the messages it received to constitute $Hist_i^x(R)$ and $Hist_i^x(S)$. While merging, each processor i also retains a trace of the network layout of the values v of the attribute x in its $Hist_i^x(R)$ (resp. $Hist_i^x(S)$): this is nothing but the collection of messages it has just received. This information will help in forming the communication templates in phase 3.

The cost of redistribution and merging step is (cf. proposition 1 of Appendix A):

$$Time_{phase2.a} = O\Big(\min\big(g \times |Hist^x(R)| + ||Hist^x(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\big) + \min\big(g \times |Hist^x(S)| + ||Hist^x(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}\big) + l\Big),$$

where g is the BSP communication parameter and l the cost of a barrier of synchronization.

We recall that, in the above equation, for a relation $T \in \{R, S\}$, the term $\min(g \times |Hist^x(T)| + ||Hist^x(T)||, g \times \frac{|T|}{p} + \frac{||T||}{p})$ is the necessary time to compute $Hist^x_{i=1,\ldots,p}(T)$ starting from the local histograms $Hist^x(T_i)_{i=1,\ldots,p}$.

The histogram $Hist_i^x(R \bowtie S)$ [3] is then computed on each processor i. $Hist_i^x(R \bowtie S)$ holds for each value v of the join attribute x belonging to both $Hist_i^x(R)$ and $Hist_i^x(S)$ the couple $(v, Hist_i^x(R)(v) \times Hist_i^x(S)(v))$ in time:

$$Time_{phase2.b} = O\Big(\max_{i=1,\ldots,p} \big(\min(||Hist_i^x(R)||, ||Hist_i^x(S)||)\big)\Big).$$

The total cost of this phase is:

$$\begin{aligned} Time_{phase2} =& Time_{phase2.a} + Time_{phase2.b} = \\ & O\Big(\min\big(g \times |Hist^x(R)| + ||Hist^x(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\big) \\ & + \min\big(g \times |Hist^x(S)| + ||Hist^x(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}\big) \\ & + \max_{i=1,\ldots,p} \big(\min(||Hist_i^x(R)||, ||Hist_i^x(S)||)\big) + l\Big). \end{aligned}$$

Phase 3: Data redistribution

In order to reduce the communication cost, only tuples of $Hist^{x,y}(R)$ and $AGGR^{x,z}_{f,u}(S)$ that will be present in the join result will be redistributed. These are the tuples of the semi-joins: $\overline{Hist}_i^{x,y}(R) = Hist_i^{x,y}(R) \ltimes Hist^x(R \bowtie S)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i) = AGGR^{x,z}_{f,u}(S_i) \ltimes Hist^x(R \bowtie S)$ for $i = 1, \ldots, p$. $\overline{Hist}_i^{x,y}(R)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ are computed in step *(3.a)* without the need to duplicate $Hist^x(R \bowtie S)$ on all the processors. Then, they are redistributed in step *(3.b)* to compute the final result.

[3] The size of $Hist(R \bowtie S) \equiv Hist(R) \cap Hist(S)$ is generally very small compared to $|Hist(R)|$ and $|Hist(S)|$ because $Hist(R \bowtie S)$ contains only values that appears in both relations R and S.

3.a Computing $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$:
To this end, we first compute on each processor j the intersections:

$$\overline{Hist}^{x[j]}(R_i) = Hist^{x[j]}(R_i) \cap Hist_j(R \bowtie S)$$

and

$$\overline{Hist}^{x[j]}(S_i) = Hist^{x[j]}(S_i) \cap Hist_j(R \bowtie S)$$

for $i = 1, \ldots, p$, where $Hist^{x[j]}(R_i)$ (resp. $Hist^{x[j]}(S_i)$) is the fragment of $Hist^x(R_i)$ (resp. $Hist^x(S_i)$) which was sent by processor i to processor j in the second phase. The cost of this step is:

$$O(\sum_i ||Hist^{x[j]}(R_i)|| + \sum_i ||Hist^{x[j]}(S_i)||).$$

We recall that, $\sum_i ||Hist^{x[j]}(R_i)|| \leq \min(||Hist^x(R)||, \frac{||R||}{p})$, and $\sum_i ||Hist^{x[j]}(S_i)|| \leq \min(||Hist^x(S)||, \frac{||S||}{p})$. Thus, the total cost of this computation step is:

$$O\Big(\min\big(||Hist^x(R)||, \frac{||R||}{p}\big) + \min\big(||Hist^x(S)||, \frac{||S||}{p}\big)\Big).$$

Now, each processor j sends each fragment $\overline{Hist}^{x[j]}(R_i)$ (resp. $\overline{Hist}^{x[j]}(S_i)$) to processor i ($i = 1, \ldots, p$ and , $j \neq i$). The cost of sending these fragments from each processor j to all the other processors is at most of the order:

$$O\Big(g \times \Big(\min\big(|\overline{Hist}^x(R)|, \frac{|R|}{p}\big) + \min\big(|\overline{Hist}^x(S)|, \frac{|S|}{p}\big)\Big)\Big).$$

On the other hand, each processor i receives $\sum_j |\overline{Hist}^{x[j]}(R_i)| + \sum_j |\overline{Hist}^{x[j]}(S_i)|$ pages of data from the other processors. In fact, $|\overline{Hist}^x(R_i)| = \sum_j |\overline{Hist}^{x[j]}(R_i)|$. Thus, the cost of receiving these fragments, on each processor i, is of the order:

$$O\Big(g \times \big(|\overline{Hist}^x(R_i)| + |\overline{Hist}^x(S_i)|\big)\Big).$$

Therefore, the total cost of this communication stage is at most:

$$O\Big(g \times \Big(\min\big(|\overline{Hist}^x(R)|, \frac{|R|}{p}\big) + \min\big(|\overline{Hist}^x(S)|, \frac{|S|}{p}\big)\Big) + l\Big),$$

since $\sum_i ||\overline{Hist}^{x[j]}(R_i)|| \leq \min(||\overline{Hist}^x(R)||, \frac{||\overline{R}||}{p})$ and $\sum_i ||\overline{Hist}^{x[j]}(S_i)|| \leq \min(||\overline{Hist}^x(S)||, \frac{||\overline{S}||}{p})$.

Remark 4 *$\cup_j \overline{Hist}^{x[j]}(R_i)$ is simply the intersection of $Hist^x(R_i)$ and the histogram $Hist^x(R \bowtie S)$ which will be noted:*

$$\overline{Hist}^x(R_i) = \cup_j \overline{Hist}^{x[j]}(R_i) = Hist^x(R_i) \cap Hist^x(R \bowtie S).$$

Hence $\overline{Hist}^x(R_i)$ is only the restriction of the fragment of $Hist^x(R_i)$ to values which will be present in the join of the relations R and S (this also applies to $\overline{Hist}^x(S_i)$).

Now, each processor i evaluates $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$, where

$$\overline{Hist}^{x,y}(R_i) = Hist^{x,y}(R_i) \cap \overline{Hist}^{x}(R_i)$$

and

$$\overline{AGGR}^{x,z}_{f,u}(S_i) = AGGR^{x,z}_{f,u}(S_i) \cap \overline{Hist}^{x}(S_i).$$

The cost of this step is of the order:

$$O\Big(\max_{i=1,\ldots,p} \big(||Hist^{x,y}(R_i)|| + ||AGGR^{x,z}_{f,u}(S_i)|| \big) \Big),$$

which is the necessary time to traverse all the tuples of $Hist^{x,y}(R_i)$ and $AGGR^{x,z}_{f,u}(S_i)$ and access $\overline{Hist}^{x}(R_i)$ and $\overline{Hist}^{x}(S_i)$ respectively, on each processor i.

Therefore, the total cost of step *3.a* is of the order:

$$\begin{aligned} Time_{phase3.a} = O\Big(& \min\big(||Hist^{x}(R)||, \frac{||R||}{p}\big) + \min\big(||Hist^{x}(S)||, \frac{||S||}{p}\big) + \\ & g \times \Big(\min\big(|\overline{Hist}^{x}(R)|, \frac{|R|}{p}\big) + \min\big(|\overline{Hist}^{x}(S)|, \frac{|S|}{p}\big) \Big) + \\ & \max_{i=1,\ldots,p} \big(||Hist^{x,y}(R_i)|| + ||AGGR^{x,z}_{f,u}(S_i)|| \big) + l \Big). \end{aligned}$$

3.b. Redistributing the tuples of $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$:
Now, each processor i distributes the tuples of $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$. After distribution, all the tuples of $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ having the same values of the join attribute x are stored on the same processor. So, each processor i merges the blocks of data received from all the other processors in order to create $\overline{Hist}^{x,y}_i(R)$ and $\overline{AGGR}^{x,z}_{f,u,i}(S)$.
The cost of distributing and merging the tuples is of the order (cf. proposition 1 of Appendix A):
$Time_{phase3.b} =$

$$\begin{aligned} O\Big(& \min\big(g \times |\overline{Hist}^{x,y}(R)| + ||\overline{Hist}^{x,y}(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\big) \\ & + \min\big(g \times |\overline{AGGR}^{x,z}_{f,u}(S)| + ||\overline{AGGR}^{x,z}_{f,u}(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}\big) + l \Big), \end{aligned}$$

where the terms:

$$\min\big(g \times |\overline{Hist}^{x,y}(R)| + ||\overline{Hist}^{x,y}(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\big)$$

and

$$\min\big(g \times |\overline{AGGR}^{x,z}_{f,u}(S)| + ||\overline{AGGR}^{x,z}_{f,u}(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}\big)$$

represent the necessary time to compute $\overline{Hist}^{x,y}_i(R)$ and $\overline{AGGR}^{x,z}_{f,u,i}(S)$ starting from $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ respectively.

The total cost of the redistribution phase is:

$$
\begin{aligned}
Time_{phase3} = O\Big(& \min\big(g \times |\overline{Hist}^{x,y}(R)| + ||\overline{Hist}^{x,y}(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\big) + \\
& \min\big(g \times |\overline{AGGR}^{x,z}_{f,u}(S)| + ||\overline{AGGR}^{x,z}_{f,u}(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}\big) + \\
& \min\big(||Hist^{x}(R)||, \frac{||R||}{p}\big) + \min\big(||Hist^{x}(S)||, \frac{||S||}{p}\big) + \\
& \max_{i=1,\ldots,p}\big(||Hist^{x,y}(R_i)|| + ||AGGR^{x,z}_{f,u}(S_i)||\big) + l\Big).
\end{aligned}
$$

We mention that we only redistribute $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ and their sizes are generally very small compared to $|R_i|$ and $|S_i|$ respectively. In addition, the size of $|Hist^{x}(R \bowtie S)|$ is generally very small compared to $|Hist^{x}(R)|$ and $|Hist^{x}(S)|$. Thus, we highly reduce the communication cost.

Phase 4: **Global computation of the aggregate function**
In this phase, we compute the global aggregate function on each processor using algorithm 10. In this algorithm, $AGGR^{x,y,z}_{f,u,i}(R \bowtie S)$ holds the final result on each processor i. The tuples of $AGGR^{x,y,z}_{f,u,i}(R \bowtie S)$ have the form (x, y, z, v) where v is the result of the aggregate function.

The cost of this phase is: $O(\max_{i=1,\ldots,p} ||AGGR^{x,y,z}_{f,u,i}(R \bowtie S)||)$, because the combination of the tuples of $\overline{Hist}^{x,y}_{i}(R)$ and $\overline{AGGR}^{x,z}_{f,u,i}(S)$ is performed to generate all the tuples of $AGGR^{x,y,z}_{f,u,i}(R \bowtie S)$.

Algorithm 10: Join and local aggregate function computation.

▷ **Par** (on each node in parallel) $i = 1, ..., p$
 ▷ $AGGR^{x,y,z}_{f,u,i}(R \bowtie S)$ = NULL /*Create a B-tree to store histogram's entries.*/
 ▷ **for each** tuple t of relation $\overline{Hist}^{x,y}_{i}(R)$ **do**
  ▷ $freq = \overline{Hist}^{x,y}_{i}(R)(t.x, t.y)$
  ▷ **for each** entry $(t.x, z, v_1) \in \overline{AGGR}^{x,z}_{f,u,i}(S)(t.x, z)$ **do**
   Insert a new tuple $(t.x, t.y, z, F_f(freq, v_1))$ into $AGGR^{x,y,z}_{f,u,i}(R \bowtie S)$;
  EndFor
 EndFor
EndPar

Tab. 4.1 – *Computing resource characteristics*

Cluster ID	CPU Speed (GHz)	CPUs per node	Cores per CPU	Memory (GB)	Storage
3	2.33	2	2	4	160 GB / SATA

The global cost of evaluating the "GroupBy-Join" queries, when the join attribute is part of the group-by attribute, using *GBJFA-join* algorithm is of the order:

$$
\begin{aligned}
Time_{total} = O\Big(& c^{i}_{r/w} \times \max_{i=1,\ldots,p}(|R_i| + |S_i|) + \max_{i=1,\ldots,p}(||R_i|| + ||S_i||) + \\
& + \min(g \times |Hist^{x}(R)| + ||Hist^{x}(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}) \\
& + \min(g \times |Hist^{x}(S)| + ||Hist^{x}(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}) \\
& + \min\left(g \times |\overline{Hist}^{x,y}(R)| + ||\overline{Hist}^{x,y}(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\right) \\
& + \min\left(g \times |\overline{AGGR}^{x,z}_{f,u}(S)| + ||\overline{AGGR}^{x,z}_{f,u}(S)||, g \times \frac{|S|}{p} + \frac{||S||}{p}\right) \\
& + \max_{i=1,\ldots,p}\left(||Hist^{x,y}(R_i)|| + ||AGGR^{x,z}_{f,u}(S_i)||\right) \\
& + \max_{i=1,\ldots,p} ||AGGR^{x,y,z}_{f,u,i}(R \bowtie S)|| + l\Big).
\end{aligned}
$$

4.4 Performance Evaluation

In order to validate our theoretical predictions, we implemented *GAJFA-Join* and *GBJFA-Join* algorithms on a Shared Nothing machine formed of 30 processors using MySQL-5.0.26-log database server and MPI-2 as a Message Passing Interface for communications. The characteristics of the nodes is shown in table 4.1. In the sensitivity analysis presented in this section, the performance of *GAJFA-Join* algorithm used when the Group By and join attributes are different is compared to the *Join Partition Method (JPM)* presented in section 3.2.2. In the second case, i.e. when the join attribute is part of the Group By attributes, we compared the performance of *GBJFA-Join* algorithm to that of *Early Distribution Scheme (EDS)* presented in section 3.2.2.

The skew in the frequencies of the *Join Attribute Values (AVS)* is an important factor that may affect the performance of the parallel database algorithms. Hence, to study the effect of the AVS on the performance of both algorithms, we used the *Zipf* (Zipf 1949, Christodoulakis 1984, Lu and Tan 1992) model to determine the frequencies of the join attribute values. Using this model, in a relation R with a domain $\{1, 2, \ldots, D\}$ of distinct values for its join attribute, the i^{th} distinct value has a number of tuples given by the following expression:

$$\|v_i\| = \frac{\|R\|}{i^z \times \sum_{j=1}^{D} \frac{1}{j^z}}$$

where z is the skew factor. When $z = 0$ the distribution is uniform and when $z \geq 1$ it becomes highly skewed.

We have measured the JPS as the percentage of the maximum deviation of the local join result size with respect to the average over all processors.

4.4.1 Speed-up test

To perform the speed-up test of both algorithms, we varied the number of processors from 1 to 30.

Group By and Join attributes are different

We used two relations formed of 3×10^6 and 2×10^6 tuples and the *Zipf* factor was fixed to 0.6 and 1 respectively. The final result was formed of approximately 731×10^3 groups. Figure 4.12 shows that our algorithm outperforms the *Join Partition Method* (JPM) even for this low value of skewness.
The difference in the performance of the two algorithms is due to the following reasons:

- In JPM algorithm all the tuples of the base relations are redistributed even if they do not participate in the join operation. In the contrary, in our algorithm, we partially apply the aggregate functions on the base relations in the first phase, hence decreasing their sizes. After that only the tuples of the aggregated relations that effectively appear in the join result will be redistributed. So, the communication cost is highly reduced.
- In *JPM*, the result of the join operation is materialized before applying the aggregate and the group-by operations which results in high input/output costs. This is not the case in our algorithm where, in phase 5, we locally compute the aggregate function on each processor without fully materializing the intermediate results of the join operation.

In the figure 4.12, we can see that *GAJFA-Join* algorithm has a super-linear speedup. This is due to the fact that increasing the number of nodes also increases the size of the accumulated caches. So, the data buckets to be joined can fit into caches, especially that in *GAJFA-Join* algorithm only the tuples that are present in the join result are redistributed between the processors. This results in decreasing the access costs to the disks and thus the query computation time.

To compute the final result, in both algorithms, we send the tuples of the intermediate join result having the same values of the group by attributes to the same processor. Thus, the sizes of the final result on each processor is similar in both algorithms. However, as we stated above, an important point in *GAJFA-Join* algorithm is that we partially apply the aggregate function before join computation and that we do not materialize the join result. This can be seen in table 4.2 that shows the average size of join result of both algorithms. This helps in decreasing the communication and disk input/output costs.

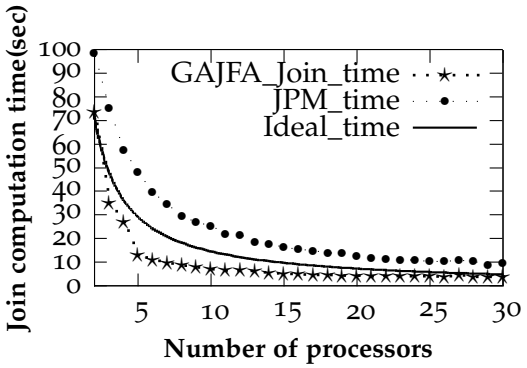

FIG. 4.12 – *Speedup performance of GAJFA-Join algorithm.*

Number of processors	5	10	15	20	25	30
GAJFA-join average join result	147609	73889	49276	36965	29573	24645
JPM average join result	3258419	1629209	1086139	814604	651683	543069

TAB. 4.2 – *GAJFA-join average size of intermediate join results (expressed in number of tuples).*

Join attribute is part of the Group By attributes

Here, we used two relations formed of 10^6 and 4×10^6 tuples and the *Zipf* factor was also fixed to 0.6. The final result was formed of 88524 groups. As we can see in figure 4.13, our algorithm outperforms EDS algorithm even for this low value of skewness.

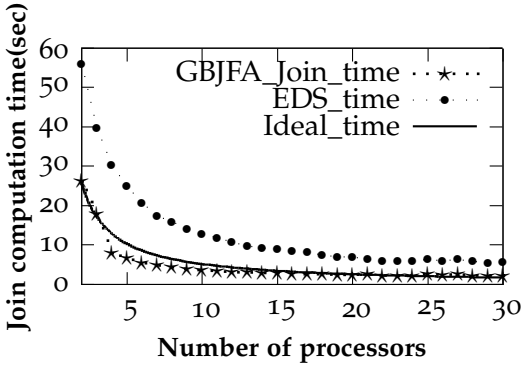

FIG. 4.13 – *Speedup performance of GBJFA-join algorithm.*

Table 4.3 shows that the size of the intermediate join result size in GBJFA-Join algorithm is very small compared to that of EDS algorithm.

4.4.2 The Effect of Attribute Value Skew (AVS) test

In this test, we study the effect of the AVS on the performance of the algorithms. So, we fixed the number of processors to 30 and we varied the skew factor in one relation between 0 and 1.8. We can see in figures 4.14 and 4.15 that the performance of our algorithms is not affected by

Number of processors	5	10	15	20	25	30
GBJFA-join average join result	17704	8852	5901	4426	3540	2950
EDS average join result	800023	400011	266674	200005	160004	133337

TAB. 4.3 – *GBJFA-join average size of intermediate join results (expressed in number of tuples).*

the AVS factor, which is not the case of JPM and EDS algorithms whose performance rapidly degrades when the AVS factor is increased.

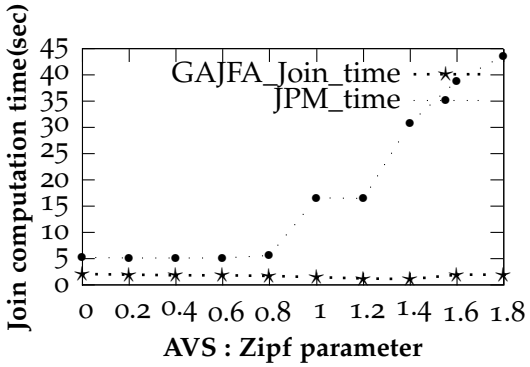

FIG. 4.14 – *The effect of AVS on GAJFA-Join algorithm.*

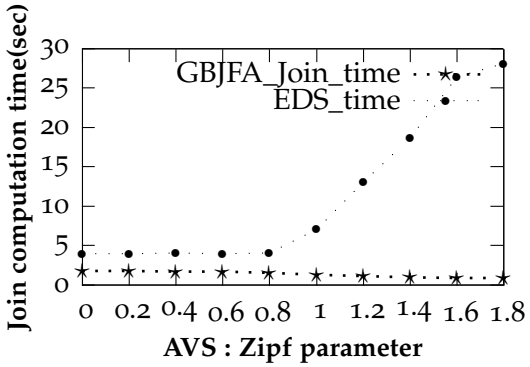

FIG. 4.15 – *The effect of AVS on GBJFA-Join algorithm.*

4.5 Conclusion

In this chapter, we presented *GBJFA-Join* and *GAJFA-Join* algorithms used to compute "GroupBy-Join" queries in a distributed architecture. *GAJFA-Join* algorithm is used when the group-by and join attributes are not the same and *GBJFA-Join* algorithm when they are the same. These algorithms can be used efficiently to reduce the execution time of the query, because we do not materialize the costly join operation which is a necessary step in all the other algorithms presented in the literature for treating this type of queries, thus reducing the Input/Output cost. They also help us to balance the load of all the processors even in the presence of AVS and

to avoid the JPS which may result from computing the intermediate join results. In addition, the communication cost is highly reduced owing to the fact that only histograms and the results of semi-joins are redistributed across the network where their size is very small compared to the size of input relations.

The BSP cost analysis and the implementation results assure that our algorithms have a near optimal linear complexity even for highly skewed data.

5 Efficient Scalable Evaluation of Join Queries on Heterogeneous Distributed Systems

Contents

The fast development of network technologies made the execution of parallel programs on distributed systems that connect heterogeneous machines feasible. However, we still face some challenges: Workload imbalance, in such environment, may not only be due to uneven load distribution among machines, as in parallel systems, but also due to distribution that is not adequate with the characteristics of each machine. In this chapter, we present *DFA-Join*: a new parallel join algorithm for heterogeneous distributed architectures based on a dynamic data distribution and task allocation. This makes it insensitive to data skew and ensures perfect balancing properties during all stages of join computation. The performance of this algorithm is analyzed using the scalable and portable BSP (Bulk Synchronous Parallel) cost model. We show that *DFA-Join* algorithm guarantees optimal complexity and near linear speed-up while

highly reducing the communication cost. A pipelined version of this algorithm, called *PDFA-Join*, is also proposed to evaluate complex queries involving multi-join operations on heterogeneous distributed systems.

5.1 Introduction

In this chapter, we propose and present the performance results of *DFA-Join* algorithm: a Dynamic Frequency Adaptive parallel algorithm to evaluate join operations on heterogeneous distributed systems. In this algorithm, we balance the load between processors in a manner that no processor, in the system, may be idle while other processors are overloaded even in the presence of highly skewed data. To this end, we use a two-step (static and dynamic) load assignment approach. In the first static step, each processor receives a number of buckets whose total join size is proportional to its actual capacity. Then, during the join phase, overloaded processors forward some of the non-treated buckets in their local buffers to underloaded processors. This combination of static and dynamic approach allows us to efficiently handle the effect of AVS, on heterogeneous systems, in a manner to get approximately the same processing time on all processors. We can also control the join product skew using a threshold value, where an idle processor does not accept additional load if its local join result size is greater then the given threshold. In addition, we use fully distributed histograms to find the tuples that participate in the result of the join operation. Then, only these tuples are redistributed, thus the communication cost is highly decreased. It is proved in (Bamha and Hains 2005; 1999), using the BSP cost model, that histogram management has a negligible cost when compared to the gain it provides in reducing the communication cost and balancing load between processors. The performance of our algorithm is analyzed using the BSP cost model which shows that it guarantees optimal performance even for highly skewed data on heterogeneous distributed multi-processor architectures. These results were confirmed by a series of tests.

We also present, in this chapter, *PDFA-Join* algorithm: a Pipelined version of *DFA-join* algorithm for treating complex queries leading to multi-joins. We also analyze the performance of *PDFA-Join* algorithm using BSP cost model.

DFA-Join algorithm is published in (Hassan and Bamha 2009b). *PDFA-Join* algorithm is published in (Hassan and Bamha 2008) and an extended and revised version of this paper is published in (Hassan and Bamha 2009a).

5.2 The DFA-join Algorithm: Evaluating Join Queries on Heterogeneous Distributed Systems

In this section, we describe in detail the steps of DFA-join (Dynamic Frequency Adaptive join) (algorithm 11): a new parallel algorithm for processing join operation on heterogeneous distributed systems formed of p processors having different characteristics of memory, disk Input/Output speed, CPU power, etc. We assume that the relations to be joined (R and S) are partitioned among processors by horizontal fragmentation. To ensure the extensibility of the algorithm, the processors are partitioned into disjoint sets. Each set (group) of processors has a desi-

gned local coordinator node responsible of balancing the load between processors in the group. If a set of processors finishes its assigned tasks before the other sets, then it may ask them to transfer to it some of their load.

Algorithm 11: Parallel DFA-Join computation steps to evaluate the join of R and S.

In Parallel (on each processor) $i \in [1, p]$ **do**
1► Create the local histogram $Hist^x(R_i)$ of relation R_i and, on the fly, hash the tuples of relation R_i into different buckets according to the values of the join attribute,
▷ Create the local histogram $Hist^x(S_i)$ of relation S_i and, on the fly, hash the tuples of relation S_i into different buckets according to the values of the join attribute,
2► Hash the local histogram, $Hist^x(R_i)$, to create global histogram's fragment, $Hist_i^x(R)$, of relation R on each processor i,
▷ Hash the local histograms, $Hist^x(S_i)$, to create global histogram's fragment, $Hist_i^x(S)$, of relation S on each processor i,
▷ Merge $Hist_i^x(R)$ and $Hist_i^x(S)$ to create join histogram's fragment, $Hist_i^x(R \bowtie S)$, by intersecting $Hist_i^x(R)$ and $Hist_i^x(S)$ on each processor i,
3► The coordinator node determines the load of each node based on the global join result size and the node's processing capacity,
▷ Create communication templates for only tuples participating to final join result,
▷ Filter generated buckets to create tasks to be executed on each processor according to its capacity,
4► Exchange data bucket according to communication templates,
5► Execute join tasks (of each bucket) on each processor, and store the join result on local disk.
Loop until no task to execute
▷ Ask a local head node for jobs from overloaded processors,
▷ Steal a job from a designated processor and execute it,
▷ Store the join result on local disk.
End Loop
EndPar

DFA-Join Algorithm can be divided into the following five phases where we use the same notations used in chapter 4 (page 54).

Phase 1. Creating local histograms:
In this phase, we create in parallel, on each processor i, the local histogram $Hist^x(R_i)$ (resp. $Hist^x(S_i)$) $(i = 1, \ldots, p)$ of block R_i (resp. S_i) by a linear traversal of R_i (resp. S_i) in time $\max_{i=1,\ldots,p}(c^i_{r/w} \times |R_i|) + \max_{i=1,\ldots,p}(\gamma_i \times ||R_i||)$ (resp. $\max_{i=1,\ldots,p}(c^i_{r/w} \times |S_i|) + \max_{i=1,\ldots,p}(\gamma_i \times ||S_i||)$) where $c^i_{r/w}$ is the cost to read/write a page of data from/to disk on processor i and γ_i is the time needed for executing one operation on processor i.
The cost of this phase is:

$$Time_{phase1} = O\big(\max_{i=1,\ldots,p} c^i_{r/w} \times (|R_i| + |S_i|) + \max_{i=1,\ldots,p} \gamma_i \times (||R_i|| + ||S_i||)\big).$$

During this step, the fragments of R_i (resp. S_i) are partitioned into mul-

tiple buckets using a hash function. If the generated buckets cannot fit into the available memory, then they are written to disks and the above process can be applied on each bucket without any memory shortage.

In practice, the extra cost related to the creation of local histograms is negligible because these histograms are computed on the fly during the step of the generation of relation's buckets.

Phase 2. Computing global histogram fragments:

In order to minimize local join's computation and communication costs, only tuples that participate in the final join result will be redistributed. These are the tuples of the local semi-joins $\overline{R_i} = R_i \ltimes S$ (resp. $\overline{S_i} = S_i \ltimes R$) on each processor i. To compute these semi-joins, we proceed by applying the following steps:

a. Creating $Hist_i^x(R)$ and $Hist_i^x(S)$ of relations R and S:

We firstly compute the fragments of global histograms $Hist_i^x(R)$ and $Hist_i^x(S)$ by redistributing the tuples of the computed local histograms using a hash function. We use the following strategy for distributing the tuples in order to respect the processing capacity of each processor. Firstly, $Hist^x(R_i)$ (resp. $Hist^x(S_i)$) are partitioned, in parallel on each processor, into multiple number of buckets using a hash function. The number of used buckets is bigger than the number of employed processors. Then, these buckets are distributed over processors according to their processing capacities where processors with higher capabilities receive more buckets than other processors.

We note by $Hist^{x[j]}(R_i)$ (resp. $Hist^{x[j]}(S_i)$) the fragment of $Hist^x(R_i)$ (resp. $Hist^x(S_i)$) that will be sent to processor j from processor i. It is important to mention here that the same hash function must be used for partitioning $Hist_i^x(R)$ and $Hist_i^x(S)$, and that created buckets holding the same index for both histograms must be sent to the same processor.

As we stated in remark 1 of page 33, using hash functions to redistribute data may cause load imbalance on processors. However, this does not arise here owing to the fact that histograms contain only distinct values of the join attribute.

The cost of this step is:

$$Time_{phase2.a} = O\Big(\min\Big(\max_{i=1,\dots,p} \omega_i \times p \times \big(g \times |Hist^x(R)| + \gamma_i \times ||Hist^x(R)||\big), \max_{i=1,\dots,p} \omega_i \times \big(g \times |R| + \gamma_i \times ||R||\big)\Big) + \min\Big(\max_{i=1,\dots,p} \omega_i \times p \times \big(g \times |Hist^x(S)| + \gamma_i \times ||Hist^x(S)||\big), \max_{i=1,\dots,p} \omega_i \times \big(g \times |S| + \gamma_i \times ||S||\big)\Big) + l\Big),$$

where ω_j is the fraction of the total volume of data assigned to processor j such that $\omega_j = \frac{\frac{1}{\gamma_j}}{\sum_{k=1}^{p} \frac{1}{\gamma_k}}$, γ_j is the execution time of one operation on processor j, g is the BSP communication parameter and l the cost of synchronization (you can review proposition 1 of appendix A for the proof of

this cost).

b. Creating $Hist_i^x(R \bowtie S)$ on each processor i:

Now, we can easily create $Hist_i^x(R \bowtie S)$ in time of order:

$$Time_{phase2.b} = O\Big(\max_{i=1,\ldots,p}\big(\gamma_i \times \min(||Hist_i^x(R)||, ||Hist_i^x(S)||)\big)\Big).$$

While creating $Hist_i^x(R \bowtie S)$, we also store for each value $v \in Hist_i^x(R \bowtie S)$ an extra information $index(v) \in \{LF, HF\}$ such that:

$$\begin{cases} index(v) = HF & \text{if } Hist^x(R)(v) \geq f_o \text{ or } Hist^x(S)(v) \geq f_o, \\ index(v) = LF & \text{elsewhere (i.e. values associated to low frequencies).} \end{cases}$$

In this algorithm, the value of f_o is set to $p_o \times log(p_o)$, where p_o is the maximum number of processor in a cluster.
During this step, each processor i also retains a trace of the network layout of values v in its $Hist_i^x(R)$ (resp. $Hist_i^x(S)$). This will be useful in the phase of creating the communication templates.

c. Finding join attribute values of $Hist^{x[j]}(R_i)$ and $Hist^{x[j]}(S_i)$ participating to the join result on each processor j:

Now, we need to determine, on each processor j, the values v of the join attribute in each partition $Hist^{x[j]}(R_i)$ (resp. $Hist^{x[j]}(S_i)$) that will be present in the join result.
So, we compute, on each processor j, the intersections:

- $\overline{Hist}^{x[j]}(R_i) = Hist^{x[j]}(R_i) \bigcap Hist_j^x(R \bowtie S)$ for $i = 1, \ldots, p$ and
- $\overline{Hist}^{x[j]}(S_i) = Hist^{x[j]}(S_i) \bigcap Hist_j^x(R \bowtie S)$ for $i = 1, \ldots, p$,

in time of order:

$$O\Big(\max_{j=1}^{p}\big(\gamma_j \times \sum_{i=1}^{p} ||Hist^{[j]x}(R_i)||\big) + \max_{j=1}^{p}\big(\gamma_j \times \sum_{i=1}^{p} ||Hist^{x[j]}(S_i)||\big)\Big).$$

However, due to applying in step *a*, a hashing function which partitions the buckets in a manner that takes into consideration the processing capacities of each machine, we have:

$$\begin{cases} \sum_{i=1}^{p} ||Hist^{x[j]}(R_i)|| & \leq & \min(\omega_j \times p \times ||Hist^x(R)||, \omega_j \times ||R||), \\ \sum_{i=1}^{p} ||Hist^{x[j]}(S_i)|| & \leq & \min(\omega_j \times p \times ||Hist^x(S)||, \omega_j \times ||S||). \end{cases}$$

Thus, the total cost of this step is:

$$\begin{aligned} Time_{phase2.c} \;=\; & O\Big(\min\big(\max_{i=1}^{p}\omega_i \times \gamma_i \times p \times ||Hist^x(R)||, \max_{i=1}^{p}\omega_i \times \gamma_i \times ||R||\big) \\ & + \min\big(\max_{i=1}^{p}\omega_i \times \gamma_i \times p \times ||Hist^x(S)||, \max_{i=1}^{p}\omega_i \times \gamma_i \times ||S||\big)\Big). \end{aligned}$$

d. Distributing $\overline{Hist}^{x[j]}(R_i)$ and $\overline{Hist}^{x[j]}(S_i)$:

In this step, each processor j sends each fragment $\overline{Hist}^{x[j]}(R_i)$ (resp. $\overline{Hist}^{x[j]}(S_i)$) to processor i. So, the number of data pages received by each processor i from the other processors is:

$$\sum_{j\neq i, j=1}^{p} |\overline{Hist}^{x[j]}(R_i)| + \sum_{j\neq i, j=1}^{p} |\overline{Hist}^{x[j]}(S_i)|.$$

Hence, the cost of this communication step is at most of the order:

$$O(\sum_{j=1}^{p} g \times |\overline{Hist}^{x[j]}(R_i)| + \sum_{j=1}^{p} g \times |\overline{Hist}^{x[j]}(S_i)|),$$

owing to the fact that:

$$\begin{cases} Hist^x(R_i) & = & \bigcup_j Hist^{x[j]}(R_i) & \text{and} \\ |Hist^x(R_i)| & = & \sum_j |Hist^{x[j]}(R_i)| & \geq & \sum_j |Hist^{x[j]}(R_i) \bigcap Hist^x(R \ltimes S)| \end{cases}$$

(this also applies to S), we have $|Hist^x(R_i)| \geq \sum_j |\overline{Hist}^{x[j]}(R_i)|$ (resp. $|Hist^x(S_i)| \geq \sum_j |\overline{Hist}^{x[j]}(S_i)|$).
So, the cost of this step is at most:

$$Time_{phase2.d} = O\Big(g \times (\max_{i=1}^{p} |Hist^x(R_i)| + \max_{i=1}^{p} |Hist^x(S_i)|) + l\Big).$$

e. Determining tuples of R and S that appear in join result:

Finally, the semi-joins $\overline{R}_i = R_i \ltimes S$ and $\overline{S}_i = S_i \ltimes R$ are computed on each processor i. The tuples of $R_i \ltimes S$ (resp. $S_i \ltimes R$) are those whose join attribute values belong to $\overline{Hist}^x(R_i)$ (resp. $\overline{Hist}^x(S_i)$). Thus, $R_i \ltimes S$ (resp. $S_i \ltimes R$) can be computed, in parallel on each processor i, by a linear traversal of R_i (resp. S_i) and consulting $\overline{Hist}^x(R_i)$ (resp. $\overline{Hist}^x(S_i)$) in time of order: $O(\max_i \gamma_i \times ||R_i|| + \max_i \gamma_i \times ||S_i||)$.
During semi-join computation step, relation $\overline{R}_i$ (resp. $\overline{S}_i$) is divided, on the fly, into two sub-relations: $\overline{R}_i^{(HF)}$ and $\overline{R}_i^{(LF)}$ ($\overline{R}_i = \overline{R}_i^{(HF)} \cup \overline{R}_i^{(LF)}$) where:

- tuples of $\overline{R}_i^{(HF)}$ (resp. $\overline{S}_i^{(HF)}$) are associated to values v such that $index(v) = HF$.
- tuples of $\overline{R}_i^{(LF)}$ (resp. $\overline{S}_i^{(LF)}$) are associated to values v such that $index(v) = LF$.

Thus, the global cost of this phase is the sum of the above five steps:
$Time_{phase2} =$

$$\begin{aligned} O\Big(& \min\Big(\max_{i=1,\ldots,p} \omega_i \times p \times (g \times |Hist^x(R)| + \gamma_i \times ||Hist^x(R)||), \\ & \max_{i=1,\ldots,p} \omega_i \times (g \times |R| + \gamma_i \times ||R||)\Big) + \\ & \min\Big(\max_{i=1,\ldots,p} \omega_i \times p \times (g \times |Hist^x(S)| + \gamma_i \times ||Hist^x(S)||), \\ & \max_{i=1,\ldots,p} \omega_i \times (g \times |S| + \gamma_i \times ||S||)\Big) + \\ & \max_{i=1,\ldots,p} \gamma_i \times ||R_i|| + \max_{i=1,\ldots,p} \gamma_i \times ||S_i|| + l\Big). \end{aligned}$$

Phase 3. Creating the communication templates:

In heterogeneous systems, the actual capacity of each machine must be taken into account while assigning data or tasks to each processor in order to achieve an acceptable performance. In addition, available capacities of machines in multi-user systems may rapidly change after load assignment, and the state of an overloaded processor may rapidly become underloaded during join computation. Thus, to benefit from the

processing power of such systems, we must not have idle processors while others are overloaded throughout all the join computation phase.

To this end, we use a two-step load assignment approach,

- the first step is a static one in which a coordinator node assigns to each processor a load proportional to its actual capacities,
- the second step is a dynamic one and is executed throughout the join computation phase. When a processor finishes join processing of its assigned buckets, it asks its local coordinator node to forward to it some of untreated buckets on other processors. To ensure the extensibility of the algorithm, processors are partitioned into disjoint sets of processors. Each set of processors has a designed local coordinator node. Load is first balanced inside each set of processors and whenever a set of processors finishes its assigned tasks, it asks the head node of other set of processors for additional tasks.

The combination of static and dynamic approaches allows us to reduce the join processing time because, in parallel and distributed systems, the total executing time is the time taken by the slowest processor to finish its task.

3.a. Static load assignment step:

In the static load assignment step we compute, in parallel on each processor i, the size of the join of all tuples related to values v that belong to $Hist_i^x(R \bowtie S)$. This is simply the sum of the frequencies $Hist_i^x(R \bowtie S)(v)$ for all values v of the join attribute in $Hist_i^x(R \bowtie S)$. This value is computed by a linear traversal of $Hist_i^x(R \bowtie S)$ in time: $O(\max_{i=1,\ldots,p} \gamma_i \times ||Hist_i^x(R \bowtie S)||)$. After that, all processors send the value $\sum_{v \in Hist_i^x(R \bowtie S)} Hist_i^x(R \bowtie S)(v)$ to a designated coordinator node in time of order $O(p \times g + l)$. The coordinator node, in its turn, calculates the total number of tuples in $R \bowtie S$ ($||R \bowtie S||$) by respectively computing the sum of values received from all the processors.
Now, the coordinator node uses the value of $||R \bowtie S||$ and the information related to the available capacities to assign to each processor i a join volume ($vol_i \times ||R \bowtie S||$) proportional to its resources where the value of vol_i is determined by the head node depending on the actual capacity of each processor i such that $\sum_{i=1}^{p} vol_i = 1$.

3.b. Communication templates creation:

Communication templates are list of messages that constitute the relations redistribution. Owing to the fact that, attribute values v which may lead to attribute value skew (AVS) (those having high frequencies : $index(v) = HF$) are also those which may cause join product skew (JPS) in standard hash algorithms. These values need a special treatment. However tuples associated to values with low frequencies (i.e. $index(v) = LF$) do not have effect neither on AVS nor on JPS. So, these tuples will be simply hashed into buckets in their source processors using a hash function and

their treatment will be postponed to the dynamic phase.

To avoid the effect of AVS and JPS, we partition the histogram $Hist_i^x(R \bowtie S)$, on each processor i, into two sub-histograms: $Hist_i^{(HF)x}(R \bowtie S)$ and $Hist_i^{(LF)x}(R \bowtie S)$ such that:

- $v \in Hist_i^{(HF)x}(R \bowtie S)$ if $index(v) = HF$ i.e. join attribute value with high frequency in relation R or S,
- $v \in Hist_i^{(LF)x}(R \bowtie S)$ if $index(v) = LF$ i.e. join attribute value with low frequency in both relations R and S.

This partitioning step is performed, while computing $\sum_v Hist_i^x(R \bowtie S)(v)$ in step 3.a, in order to avoid reading the histogram two times.

We can start now by creating the communication templates which are computed, in a first time, on each processor i for only values v in $Hist_i^x(R \bowtie S)$ such that the total join size related to these values is inferior or equal to $vol_i \times ||R \bowtie S||$ *starting from the value that generates the highest join result and so on*. Each processor creates order messages for data redistribution as follows: For each value v in $Hist_i^x(R \bowtie S)$, processor i creates communicating messages $order\_to\_send(j, i, v)$ asking each processor j that holds tuples of R or S having values v for the join attribute to send them to it. If the processing capacity of a processor i does not allow it to compute the join result associated to all values v of the join attribute in $Hist_i^x(R \bowtie S)$ [1], then it will not ask the source processors j holding the remaining values v to redistribute their associated tuples but to partition them into buckets using a hash function and save them locally for further join processing step in the dynamic phase. Hence, it sends an *order_to_save(j,v)* message for each processor j holding tuples having values v of the join attribute.
For very special cases where the size of the join result related to a join attribute value v is greater than $vol_i \times ||R \bowtie S||$, processor i sends to each processor j holding tuples related to v an $order\_to\_partition(j, i, rel\_index, vol, v)$ message. Here, rel_index represents the relation with the highest frequency of v and vol represents the number of tuples related to v that processor i can receive. These messages will be treated by the destination nodes in the redistribution phase.
If $Proc(v)$ is the number of processors holding tuples having value v for the join attribute, then we need to create $\sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v)$ on each processor i. Thus, the maximal complexity of creating the communication templates is of the order:

$$O(\max_{i=1,\ldots,p} \gamma_i \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v)).$$

After creating the communication templates, on each processor i, messages: $order\_to\_send(j, i, .)$, $order\_to\_save(j, .)$ and $order\_to\_partition(j, ., ., .)$ are sent to their destination processors j when $j \neq i$ in time:

$$O\Big(\max_{i=1,\ldots,p} g \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) \times |order\_messages| + l\Big),$$

[1] This is the case if $vol_i \times ||R \bowtie S|| < \sum_v Hist_i(R \bowtie S)(v)$ on processor i.

where $|order\_message|$ is the maximum size of the above three order messages.
So, the total cost of this step is:

$$Time_{phase3.b} = O\Big(\max_{i=1,\ldots,p} \gamma_i \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) + \max_{i=1,\ldots,p} g \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) \times |order\_messages| + l \Big).$$

3.c. Task generation step:

After creating the communication templates, each processor i obeys the communication order messages that it has just received. So, tuples that must be sent to each processor are partitioned into multiple number of buckets greater than p using a hash function. Hashing data assigned to a processor into multiple number of buckets facilitates task reallocation in the join phase (phase 5) from overloaded to idle processors. In addition, each processor i partitions tuples whose join attribute value is indicated in the $order\_to\_save()$ messages or those of $\overline{R}_i^{(LF)x}$ (resp. $\overline{S}_i^{(LF)x}$) into buckets using the same hash function on all the processors. However, these buckets will be kept for the moment in their source processors and their redistribution and join processing operations will be postponed till the dynamic phase.
In the presence of $order\_to\_partition(j, i, rel\_index, vol, .)$ messages, processor j partitions tuples of relation rel_index associated to v into p_o buckets using Round-Robin partitioning. Then, processor j sends to processor i one or more buckets of the p_o buckets whose size is bounded by vol. The treatment of the remaining buckets is postponed until an idle processor asks for additional tasks. However, to obtain a valid join result, a copy of all the tuples related to v of the other relation must be sent to processor i and to each processor that will treat one (or more) of these p buckets.
The cost of this step is:

$$Time_{phase3.c} = O\big(\max_{i=1,\ldots,p} \gamma_i \times (||R_i|| + ||S_i||) \big).$$

The global cost of this phase is the sum of the above three steps:
$Time_{phase3} =$

$$O\Big(\max_{i=1,\ldots,p} \gamma_i \times ||Hist_i(R \bowtie S)|| + \max_{i=1,\ldots,p} \gamma_i \times (||R_i|| + ||S_i||) + p \times g + l + \max_{i=1,\ldots,p} \gamma_i \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) + \max_{i=1,\ldots,p} g \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) \times |order\_messages| \Big).$$

Phase 4. Redistribution phase:
After the partition step, the buckets are sent to their destination processors. It is important to mention here that only tuples of R and S that effectively participate in the join result will be redistributed. So, each processor i receives a fragment $\overline{\overline{R}}_i$ (resp. $\overline{\overline{S}}_i$) of R (resp. S). This fragment is

expected to be balanced using our redistribution strategy. We recall that $\max_i ||\overline{\overline{R_i}}|| \leq vol_i \times ||R \bowtie S||$ and $\max_i ||\overline{\overline{S_i}}|| \leq vol_i \times ||R \bowtie S||$. Therefore, in this algorithm, the communication cost is highly reduced and is of the order:

$$Time_{phase4} = O\Big(g \times \max_{i=1,\ldots,p} (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}|) + l\Big).$$

Phase 5. Join computation phase:

Buckets received by each processor are arranged in a queue. Each processor executes successively the join operation of its waiting buckets. If a processor finishes computing the join related to its local data and the overall join operation is not finished, it sends to the local coordinator node a message asking for more work. Hence, the local coordinator node will assign to this idle processor some of the buckets related to join attribute values that were not redistributed earlier in the static phase. However, if all these buckets are already treated, the head node checks the number of untreated buckets in the queue of the other processors, and it asks the processor that has the maximal number of untreated buckets to forward a part of them to the idle one. The number of sent buckets must respect the capacity of the idle processor. The cost of local join computation is:

$$Time_{local\_join} = O\Big(\max_{i=1}^{p} \big(c^i_{r/w} \times (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}| + vol_i \times |\overline{R} \bowtie \overline{S}|) + t^i_h \times ||\overline{\overline{R_i}}|| + t^i_s \times ||\overline{\overline{S_i}}||\big)\Big),$$

where t^i_s is the time needed to search for an entry in the hash table and t^i_h is the time needed to add an entry to the hash table.

The global cost of join computation of two relations R and S using *DFA-Join* algorithm is:

$Time_{DFA-join} =$

$$\begin{aligned}
O\Big(& \max_{i=1,\ldots,p} c^i_{r/w} \times (|R_i| + |S_i|) + \max_{i=1,\ldots,p} \gamma_i \times (||R_i|| + ||S_i||) + g \times \max_{i=1,\ldots,p} (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}|) + \\
& \max_{i=1,\ldots,p} \big(c^i_{r/w} \times (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}| + vol_i \times |\overline{R} \bowtie \overline{S}|) + t^i_h \times ||\overline{\overline{R_i}}|| + t^i_s \times ||\overline{\overline{S_i}}||\big) + \\
& \min\Big(\max_{i=1,\ldots,p} \omega_i \times p \times \big(g \times |Hist^x(R)| + \gamma_i \times ||Hist^x(R)||\big), \\
& \qquad \max_{i=1,\ldots,p} \omega_i \times \big(g \times |R| + \gamma_i \times ||R||\big)\Big) + \\
& \min\Big(\max_{i=1,\ldots,p} \omega_i \times p \times \big(g \times |Hist^x(S)| + \gamma_i \times ||Hist^x(S)||\big), \\
& \qquad \max_{i=1,\ldots,p} \omega_i \times \big(g \times |S| + \gamma_i \times ||S||\big)\Big) + \\
& \max_{i=1,\ldots,p} \gamma_i \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) + \\
& \max_{i=1,\ldots,p} g \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) \times |order_messages| + l\Big).
\end{aligned}$$

Remark 5 *Sequential evaluation of the join of two relations R and S, on processor i, requires at least the following lower bound (the time to scan the input relations and to store the join result)*:

$$bound_{inf_1} = \Omega\big(c^i_{r/w} \times (|R| + |S| + |R \bowtie S|) + t^i_h \times ||R|| + t^i_s \times ||S||\big).$$

TAB. 5.1 – *Computing resource characteristics*

Cluster ID	CPU Speed (GHz)	CPUs per node	Cores per CPU	Memory (GB)	Storage
1	2.5	2	4	32	320 GB / SATA II
2	2.33	2	1	8	2x300 GB Raido / SATA
3	2.33	2	2	4	160 GB / SATA

Therefore, parallel join processing on p heterogeneous processors requires:

$$bound_{inf_p} = \Omega\Big(\max_{i=1}^{p} \big(c^i_{r/w} \times \omega_i \times (|R| + |S| + |R \bowtie S|) + \omega_i \times (t^i_h \times ||R|| + t^i_s \times ||S||)\big)\Big).$$

DFA-Join algorithm has optimal asymptotic complexity when:

$$\max_{i=1}^{p} \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) \times |order\_messages| \leq \max_{i=1}^{p} \big(c^i_{r/w} \times \omega_i \times \max(|R|, |S|, |R \bowtie S|)\big),$$

this is due to the fact that all other terms in $Time_{DFA-join}$ *are bounded by those of* $bound_{inf_p}$. *The above inequality holds since we have*:

$$\max_{i=1}^{p} \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) \times |order\_messages| \leq \max_{i=1}^{p} \omega_i \times |R \bowtie S|,$$

when the chosen threshold frequency f_o *is greater than* p_o *which is the maximum number of processors in each cluster (this is case for our threshold frequency* $f_o = p_o \times log(p_o)$*).*

5.3 DFA-JOIN PERFORMANCE EVALUATION

We compared the performance of *DFA-Join* algorithm to the standard algorithm based on pure hashing and to *OSFA-Join* (Optimal symmetric frequency adaptive join) algorithm (Bamha 2005).
In the tests, we were interested in comparing the speed-up of the three algorithms and the effect of data skew on their performance. The tests were performed on three heterogeneous clusters of Grid'5000 platform [2] using MySQL-5.0.51a-10 Database server and MPICH2 as Message Passing Interface for communications. The characteristics of the three cluster nodes are presented in table 5.1. The network connectivity is provided by a single *Cisco 6509* switch. This switch provides one gigabit ethernet connection to each cluster node.
To study the effect of data skew on the performance, the frequencies of join attribute values have been taken to follow the *Zipf* (Zipf 1949, Christodoulakis 1984) as it is the case in most database tests. The number of tuples of the i^{th} distinct value of the join attribute in a relation R with a domain

[2] Grid'5000 is a grid research infrastructure formed of 5000 processors distributed over nine sites in France (www.grid5000.fr).

$\{1, 2, \ldots, D\}$ is given by the following expression: $||v_i|| = \frac{||R||}{i^z \times \sum_{j=1}^{D} \frac{1}{j^z}}$ where z is the skew factor. The distribution is considered uniform if $z = 0$ and highly skewed when $z \geqslant 1$.

The JPS (Join Product Skew) was measured as the percentage of the maximum deviation of the local join result size with respect to the average over all processors.

5.3.1 Speed-up test

To perform the speed-up test, we applied the three algorithms on two relations formed of 8×10^6 tuples and 4×10^6 tuples respectively. The zipf skew factor of one the relations was set to 0.6 and to 1 in the other.
The number of processors was varied from 1 to 96 chosen from the 3 heterogeneous clusters where the join result was formed of approximately 271×10^6 tuples. Figures 5.1 and 5.2 show that *DFA-Join* algorithm outperforms *OSFA-Join* algorithm and the standard hash join algorithm. The results also show that *OSFA-Join* algorithm is, in general, faster than the standard hash join algorithm even on heterogeneous systems.
In *DFA-Join* algorithm, processors that finish before the others, ask for more work. This may result in skew in the join result. In order to avoid this, we used a skew deviation threshold percentage and processors that exceed this load do not ask for more tasks even if they are idle while others aren't.

In the tests, we set the JPS threshold value to 20% and we can see in figures 5.3 and 5.4 that the percentage of JPS in *DFA-Join* respected this threshold. We can also see that we have a high JPS in standard hash join which is not the case for *DFA-Join* and *OSFA-Join* algorithms.

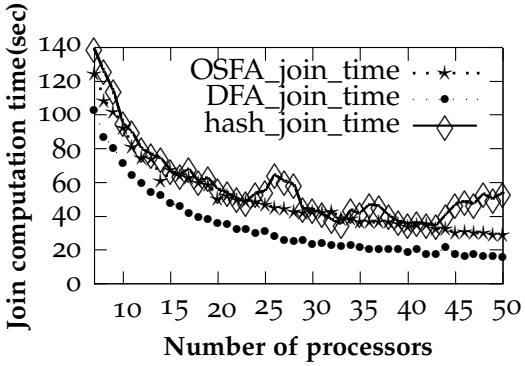

FIG. 5.1 – *DFA-join Speed-up test for number of nodes* $\in \{1, \ldots, 50\}$.

5.3.2 The effect of Attribute Value Skew (AVS) test

To study the effect of AVS on *DFA-Join* algorithm, we fixed the number of processors to 60 and the value of the *Zipf* skew factor was varied from 0 to 1.8. The test was applied on two relations of size 8×10^6 and 4×10^6 tuples respectively. Figure 5.5, shows that *OSFA-Join* and *DFA-Join* algorithms are much more faster than the standard hash join algorithm and that their processing time remains balanced while varying the skew factor.

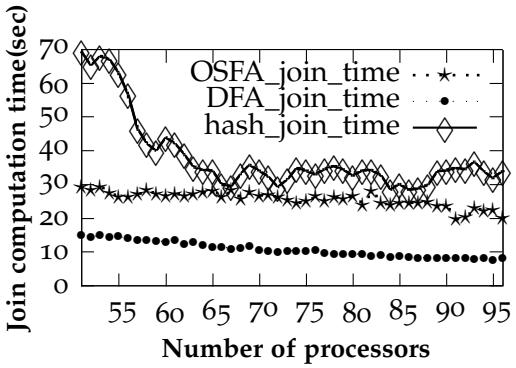

FIG. 5.2 – *DFA-join Speed-up test for number of nodes* $\in \{50, \ldots, 95\}$.

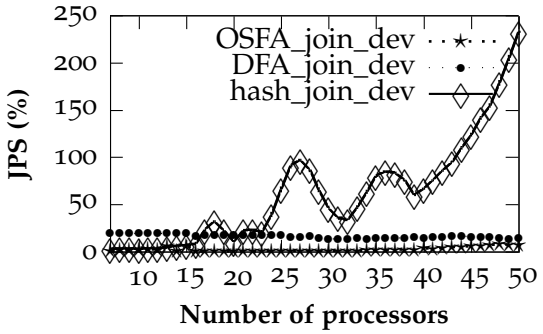

FIG. 5.3 – *DFA-join JPS percentage for number of nodes* $\in \{1, \ldots, 50\}$.

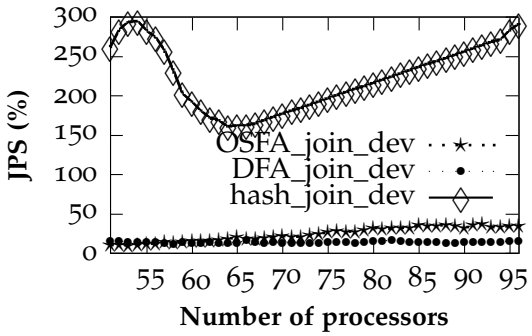

FIG. 5.4 – *DFA-join JPS percentage for number of nodes* $\in \{50, \ldots, 95\}$.

We can also deduce from figure 5.6 that the standard hash join algorithm is very sensitive to the effect of data skew compared to *DFA-Join* and *OSFA-Join* algorithms.

5.3.3 The effect of join selectivity

In this test, we are interested in studying the effect of the join selectivity on the performance of the three algorithms. We applied the three algorithms on two relations formed of 2×10^6 and 4×10^6 tuples where we fixed the number of processor to 54. The selectivity factor [3] was varied

[3]The selectivity factor of $R \bowtie S$ is $\frac{||R \bowtie S||}{||R|| \times ||S||}$.

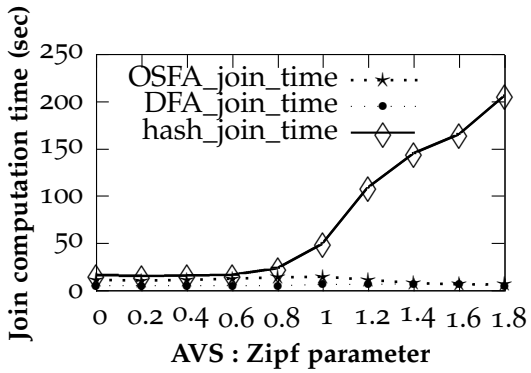

FIG. 5.5 – *The effect of AVS on performance of DFA-join.*

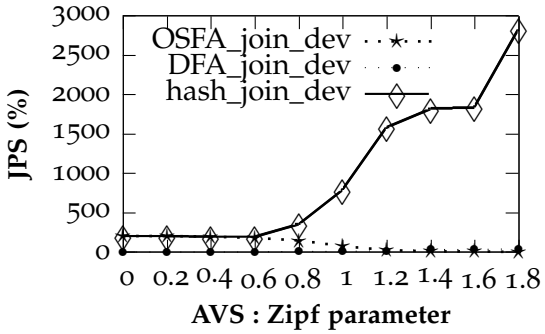

FIG. 5.6 – *The effect of AVS on Join result deviation of DFA-join.*

from 8×10^{-4}% to 6×10^{-3}% generating join result size varying between 64×10^6 and 458×10^6 tuples. Figures 5.7 and 5.8 show that *DFA-Join* and *OSFA-Join* are much more faster than the standard algorithms based on hashing. These figures also show that *DFA-Join* outperforms *OSFA-Join* algorithm. Figures 5.9 and 5.10 confirm that *DFA-Join* and *OSFA-Join* algorithms generate a negligible join skew result which is not the case of the standard hash based algorithm that suffers from high JPS.

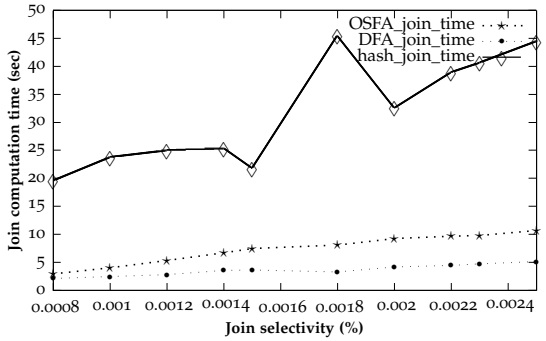

FIG. 5.7 – *The effect of join selectivity on performance of DFA-Join* ($8 \times 10^{-4} \leq$ *selectivity* $\leq 24 \times 10^{-4}$).

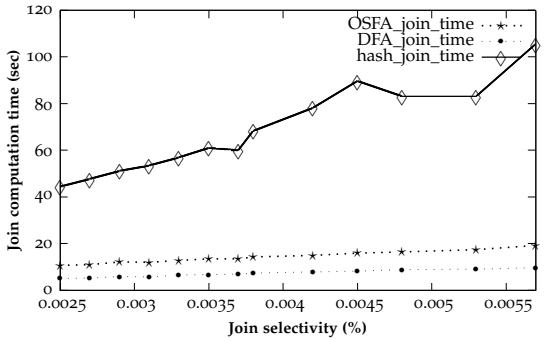

FIG. 5.8 – *The effect of join selectivity on performance of DFA-Join* ($25 \times 10^{-4} \leq$ *selectivity* $\leq 6 \times 10^{-3}$).

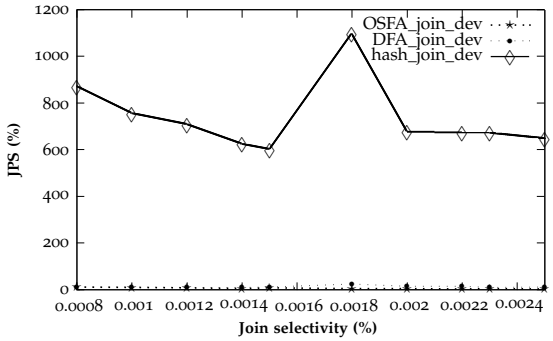

FIG. 5.9 – *The effect of join selectivity on Join result deviation of DFA-Join* ($8 \times 10^{-4} \leq$ *selectivity* $\leq 24 \times 10^{-4}$).

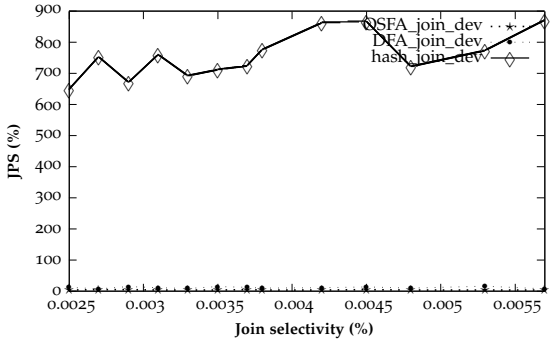

FIG. 5.10 – *The effect of join selectivity on Join result deviation of DFA-Join* ($25 \times 10^{-4} \leq$ *selectivity* $\leq 6 \times 10^{-3}$).

5.4 The PDFA-Join Algorithm: Evaluating Multi-join Queries on Heterogeneous Distributed systems

In the rest of this chapter, we present a pipelined version of *DFA-Join* join algorithm called *PDFA-Join* (Pipelined Dynamic frequency Adaptive join algorithm). The aim of pipelining in *PDFA-Join* is to offer flexible resource allocation and to avoid unnecessary disk input/output for intermediate join result in multi-join queries. We show that *PDFA-Join* algorithm can be applied efficiently in various parallel execution strategies making it possible to exploit not only intra-operator parallelism but also inter-operator

parallelism. These algorithms are used in the objective to avoid the effect of load imbalance due to data skew, and to reduce the communication costs due to the redistribution of the intermediate results which can lead to a significant degradation of the performance.

5.4.1 Limitations of Parallel Execution Strategies in Multi-join Queries

Parallel execution of multi-join queries depends on the execution plan of simple joins that compose it. The main difference between these strategies lies in the manner of allocating the simple joins to different processors and in the choice of an appropriate degree of parallelism (i.e. the number of processors) used to compute each simple join.

Several strategies were proposed to evaluate multi-join queries (Liu and Rundensteiner 2005, Wilschut et al. 1995). They generally depend on the parallel query execution plan. In these strategies intra-operator, inter-operator and pipelined parallelisms can be used. These strategies are divided into four principal categories presented thereafter. We will give for each one its advantages and limitations.

Sequential Parallel execution

Sequential parallel execution is the simplest strategy to evaluate, in parallel, a multi-join query. It does not induce inter-operator parallelism. Simple joins are evaluated one after the other in a parallel way. Thus, at a given moment, one and only one simple join is computed in parallel by all the available processors.

This strategy is very restrictive and does not provide efficient resource allocation due to the fact that a simple join cannot be started until all its operands are entirely available, and whenever a join operation is executed on a subset of processors, all the other processors remain idle until the next join operation. Moreover, this strategy induces unnecessary disk Input/Output because intermediate results are written to disk and not immediately used for the next operations.

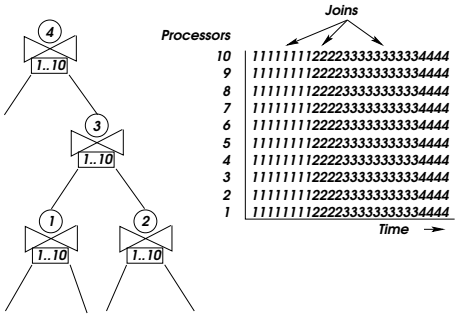

FIG. 5.11 – *Sequential parallel execution.*

The execution time of each join is then the execution time of the slowest processor. Figure 5.11 illustrates an example of a sequential parallel execution. To reach acceptable performance, join algorithms used in this strategy should reduce the load imbalance between all the processors and the number of idle processors must be as small as possible.

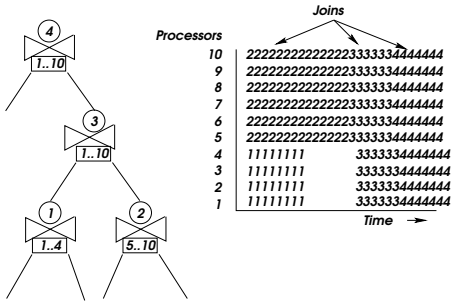

FIG. 5.12 – *Parallel synchronous execution.*

Parallel Synchronous execution

Parallel synchronous execution uses in addition to intra-operator parallelism, inter-operator parallelism (M.-S. Chen and Wu). In this strategy several simple join operations can be computed simultaneously on disjoint sets of processors. Figure 5.12 gives an example of a parallel synchronous execution: processors 1 to 4 compute join 1 while processors 5 to 10 compute join 2. Once these joins are finished, all the processors are used to compute join 3 and then join 4.

The parallel execution time of an operator depends on the degree of parallelism. The execution time decreases by increasing the number of processors until the arrival at a point of saturation (called optimal degree of parallelism) from which increasing the number of processors, increases the parallel execution time (Rahm 1996, M.-S. Chen and Wu). The main difficulty in this strategy lies in the manner of allocating the simple joins to the available processors and in the choice of an appropriate degree of parallelism to be used for each join.

In this strategy, the objective of such allocation is to reduce the latency where the global execution time of all operators should be of the same order. This also applies to the global execution time of each operator in the same group of processors where the local computation within each group must be balanced.

This Strategy combines only intra and inter operator parallelism in the execution of multi-join queries and does not introduce pipelined parallelism and large number of processors may remain idle if are not used in inter-operator parallelism. This constitutes the main limitations of this strategy for flexible resource allocation in addition to unnecessary disk input/output operation for intermediate join result.

Segmented Right-Deep execution

Contrary to a parallel synchronous strategy, a *Segmented Right-Deep execution* (Chen et al. 1992, Liu and Rundensteiner 2005) employs, in addition to intra-operator parallelism, pipelined inter-operator parallelism which is used in the evaluation of the right-branches of the query tree.
An example of a segmented right-deep execution is illustrated in figure 5.13 where all the available processors initially compute join 1 and disjoint sub-sets of processors are used to compute simultaneously joins 2, 3 and 4 using pipelined parallelism. Note that pipelined parallelism can-

not start until the creation of the hash table of the join result of operation 1.

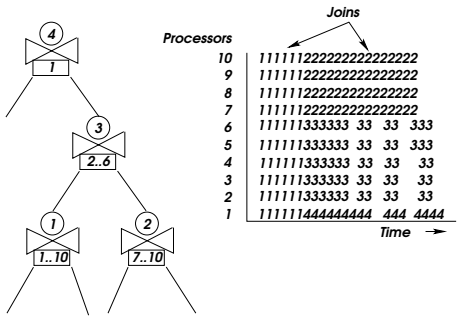

FIG. 5.13 – *Segmented right-deep execution.*

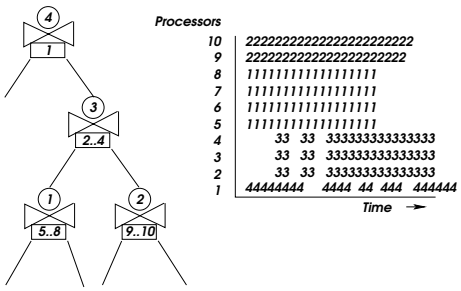

FIG. 5.14 – *Full parallel execution.*

Segmented right-deep execution offers more flexible resource allocation than parallel synchronous execution strategy : many joins can be computed on disjoint sets of processors to prepare hash tables for pipelined joins. Its main limitation remains in the fact that pipelined parallelism cannot be started until all the hash tables are computed. Moreover, no load balancing between processors can be performed whenever pipelined parallelism begins.

Full Parallel execution

Full Parallel execution (Wilschut and Apers 1991, Wilschut et al. 1995, Liu and Rundensteiner 2005) uses inter-operator parallelism and pipelined inter-operator parallelism in addition to intra-operator parallelism. In this strategy, all the simple joins, associated to the multi-join query, are computed simultaneously in parallel using disjoint sets of processors. Inter-operator parallelism and pipelined inter-operator parallelism are exploited according to the type of the query tree.
The effectiveness of such strategy depends on the quality of the execution plans generated during the query optimization phase and on the ability to evenly divide load between processors in the presence of skewed data.

All existing algorithms using this strategy are based on static hashing to redistribute data over the network which makes them very sensitive to data skew. Moreover, pipelined parallelism cannot start until the creation of hash tables of build relations. We recall that all join algorithms used in these strategies require data redistribution of all intermediate join results (and not only tuples participating to the join result) which may induce

a high cost of communication. In addition, no load balancing between processors can be performed when pipelined parallelism begins. This can lead to a significant degradation of performance.

In the following section, we will present *PDFA-Join* (Pipelined Dynamic Frequency Adaptive Join): a new join algorithm which can be used in different execution strategies allowing to exploit not only intra-operator but also inter-operator and pipelined parallelism. This algorithm is proved to induce a minimal cost for communication (only relevant tuples are redistributed over the network), while guaranteeing perfect load balancing properties in a heterogeneous multi-processor machine even for highly skewed data.

5.4.2 Parallelism in Multi-join Queries using PDFA-Join Algorithm

Pipelining was largely studied and successfully implemented in many classical join algorithms, on Shared Nothing (SN) multi-processor machine, in the presence of ideal conditions of load balancing and in the absence of data skew (Liu and Rundensteiner 2005). Nevertheless, these algorithms are generally based on static hash join techniques and are thus very sensitive to AVS and JPS.

The pipelined algorithm introduced in (Bamha and Exbrayat 2003) solves this problem and guarantees perfect load balancing on homogeneous SN machines. However, its performance degrades on heterogeneous multi-processor architectures where the load of each processor may vary in a dynamic and unpredictable way.

We propose to adapt *DFA-Join* to pipelined multi-join queries to solve the problem of data skew and load imbalance between processors on heterogeneous multi-processors architectures during all the stages of join computation.

Detailed Algorithm

In this section, we present a parallel pipelined execution strategy for the multi-join query, $\mathcal{Q} = (R \bowtie_{a_1} S) \bowtie_{b_1} (U \bowtie_{a_2} V)$, given in figure 5.15 (this strategy can be easily generalized to any bushy multi-join query) where R, S, U and V are source relations and a_1, a_2 and b_1 are join attributes.

We will give in detail the execution steps to evaluate the join query $\mathcal{Q}_1 = R \bowtie_{a_1} S$ (the same technique is used to evaluate $\mathcal{Q}_2 = U \bowtie_{a_2} V$). We assume that each relation $T \in \{R, S, U, V\}$ is horizontally fragmented among p processors.

PDFA-Join algorithm (*Algorithm 12*) can be divided into the following five phases.

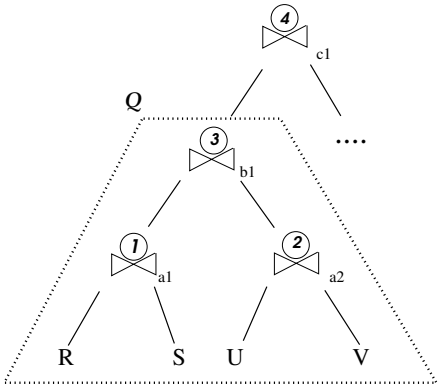

FIG. 5.15 – *Parallel execution of a multi-join query using PDFA-Join algorithm.*

Algorithm 12: Parallel PDFA-Join computation steps to evaluate the join of R and S on attribute a_1 and preparing the next join on attribute b_1.

In Parallel (on each processor) $i \in [1, p]$ **do**
1► Create the local histogram $Hist^{a_1}(R_i)$ of relation R_i and, on the fly, hash the tuples of relation R_i into different buckets according to the values of join attribute a_1 ;
▷ Create the local histogram $Hist^{a_1}(S_i)$ of relation S_i and, on the fly, hash the tuples of relation S_i into different buckets according to the values of join attribute a_1, ;
2► Create global histogram's fragment, $Hist_i^{a_1}(R)$, of relation R on each processor i ;
▷ Create global histogram's fragment, $Hist_i^{a_1}(S)$, of relation S on each processor i ;
▷ Merge $Hist_i^{a_1}(R)$ and $Hist_i^{a_1}(S)$ to create join histogram, $Hist_i^{a_1}(R \bowtie S)$ on each processor i ;
3► Create communication templates for only tuples that appear in join result ;
▷ Filter generated buckets to create tasks to execute on each processor according to its capacity ;
▷ Create local histograms $Hist^{b_1}(R_i \bowtie S_i)$ of join result (of the buckets associated to processor i) on attribute b_1 of the next join using histograms and communication templates (See Algo. 13.) ;
4► Exchange data stored on each bucket according to communication templates ;
5► Execute join tasks (of each bucket) on each processor, and store the join result on local disk ;
Loop until no task to execute
▷ Ask a local head node for jobs from an overloaded processor ;
▷ Steal a job from a designated processor and execute it ;
▷ Store the join result on local disk ;
endloop
endpar

Phase 1. Creating local histograms:
In this phase, we create in parallel, on each processor i, the local histogram $Hist^{a_1}(R_i)$ (resp. $Hist^{a_1}(S_i)$) $(i = 1, \ldots, p)$ of block R_i (resp. S_i) by a linear traversal of R_i (resp. S_i) in time:

$$\max_{i=1,\ldots,p}(c^i_{r/w} \times |R_i|) + \max_{i=1,\ldots,p} \gamma_i \times ||R_i||$$

(resp. $\max_{i=1,\ldots,p}(c^i_{r/w} \times |S_i|) + \max_{i=1,\ldots,p} \gamma_i \times ||S_i||$) where $c^i_{r/w}$ is the cost to read/write a page of data from disk on processor i.
While creating the histograms, tuples of R_i (resp. S_i) are partitioned, on the fly, into N buckets using a hash function in order to facilitate the redistribution phase.
The cost of this phase is:

$$Time_{phase1} = O\Big(\max_{i=1,\ldots,p} c^i_{r/w} \times (|R_i| + |S_i|) + \max_{i=1,\ldots,p} \gamma_i \times (||R_i|| + ||S_i||)\Big).$$

Phase 2. Computing the histogram of $R \bowtie S$:

In this phase, we compute $Hist^{a_1}_i(R \bowtie S)$ on each processor i. This helps in specifying the values of the join attribute that will be present in the join result. So, only tuples of R and S that effectively participate in the join result are redistributed, in a further phase, which allows us to minimize the communication cost. The histogram of $R \bowtie S$ is formed of entries of the form (v, n_v) such that $v \in Hist^{a_1}(R) \cap Hist^{a_1}(S)$ and $n_v = Hist^{a_1}(R)(v) \times Hist^{a_1}(S)(v)$. So, we must first compute the global histograms $Hist^{a_1}_i(R)$ and $Hist^{a_1}_i(S)$ by redistributing the tuples of the local histograms using a hash function that distributes the values of the join attribute in a manner that respects the processing capacity of each processor.
The cost of this step is:

$$\begin{aligned} Time_{phase2.a} = \\ O\Big(&\min\big(\max_{i=1,\ldots,p} \omega_i \times p \times (g \times |Hist^{a_1}(R)| + \gamma_i \times ||Hist^{a_1}(R)||), \\ &\max_{i=1,\ldots,p} \omega_i \times (g \times |R| + \gamma_i \times ||R||)\big) + \\ &\min\big(\max_{i=1,\ldots,p} \omega_i \times p \times (g \times |Hist^{a_1}(S)| + \gamma_i \times ||Hist^{a_1}(S)||), \\ &\max_{i=1,\ldots,p} \omega_i \times (g \times |S| + \gamma_i \times ||S||)) + l\Big). \end{aligned}$$

where ω_i is the fraction of the total volume of data assigned to processor i such that: $\omega_i = (\frac{1}{\gamma_j})/(\sum_{j=1}^{p} \frac{1}{\gamma_j})$, γ_i is the execution time of one instruction on processor i, g is the BSP communication parameter and l the cost of synchronization (section 2.2.2) (the detailed proof of this cost is given in *proposition 1* of *Appendix A*).

Now, we can easily create $Hist^{a_1}_i(R \bowtie S)$ by computing in parallel, on each processor i, the intersection of $Hist^{a_1}_i(R)$ and $Hist^{a_1}_i(S)$ in time of order:

$$Time_{phase2.b} = O\Big(\max_{i=1,\ldots,p} \big(\gamma_i \times \min(||Hist^{a_1}_i(R)||, ||Hist^{a_1}_i(S)||)\big)\Big).$$

While creating $Hist^{a_1}_i(R \bowtie S)$, we also store for each value $v \in Hist^{a_1}_i(R \bowtie S)$ an extra information $index(v) \in \{LF, HF\}$ such that:

$$\begin{cases} index(v) = HF & \text{if } Hist^{a_1}(R)(v) \geq f_o \text{ or } Hist^{a_1}(S)(v) \geq f_o \\ & \quad \text{(i.e. values having high frequencies)} \\ index(v) = LF & \quad \text{elsewhere (i.e. values associated to low frequencies).} \end{cases}$$

The used threshold frequency is $f_o = p \times log(p)$. This information will be useful in the phase of the creation of communication templates.
The total cost of this phase is the sum of $Time_{phase2.a}$ and $Time_{phase2.b}$.

$$Time_{phase2} = O\Big(\min\big(\max_{i=1,\dots,p} \omega_i \times p \times (g \times |Hist^{a_1}(R)| + \gamma_i \times ||Hist^{a_1}(R)||), \max_{i=1,\dots,p} \omega_i \times (g \times |R| + \gamma_i \times ||R||)\big) + \min\big(\max_{i=1,\dots,p} \omega_i \times p \times (g \times |Hist^{a_1}(S)| + \gamma_i \times ||Hist^{a_1}(S)||), \max_{i=1,\dots,p} \omega_i \times (g \times |S| + \gamma_i \times ||S||)\big) + l \Big).$$

We recall that the size of a histogram is, in general, very small compared to the size of base relations.

Phase 3. Creating the communication templates:
To balance the load of all the processors throughout the join processing, we use, as in *DFA-Join* algorithm, a two-step (static then dynamic) load assignment approach which allows us to reduce the join processing time. The steps of this algorithm are the same as that of *phase 3* in *DFA-Join* algorithm. However, during step *3.c* of this algorithm, local histogram of the join result, $Hist^{b_1}(R \bowtie S)$, on attribute b_1 is created directly from $Hist^{a_1}(R_i)$ and $Hist^{a_1}(S_i)$ using Algorithm 13.

Algorithm 13: Join result histogram's creation algorithm on attribute b_1.

▷**Par** (on each node) $i \in [1, p]$ **do**
▷ $Hist^{b_1}(R_i \bowtie S_i)$=NULL ; /* Create an empty B$^+$-tree to store histogram's entries. */
▷**for** each tuple t of each bucket of relation R_i **do**
▷ $freq1 = Hist^{a_1}(S)(t.a_1)$;
▷ **if** $(freq1 > 0)$ (i.e. tuple t will be present in $R \bowtie S$) **then**
▷ $freq2 = Hist^{b_1}(R_i \bowtie S_i)(t.b_1)$;
▷ **if** $(freq2 > 0)$ (i.e value $t.b_1$ is present in $Hist^{b_1}(R_i \bowtie S_i)$) **then**
▷ Update $Hist^{b_1}(R_i \bowtie S_i)(t.b_1) = freq1 + freq2$;
▷ **else**
▷ Insert a new couple $(t.b_1, freq1)$ into the histogram $Hist^{b_1}(R_i \bowtie S_i)$;
▷**endif**
▷**endif**
▷**endfor**
▷**endpar**

Owing to the fact that the access to the histogram (equivalent to a search in a B$^+$-tree) is performed in a constant time, the cost of the creation of the histogram of join result is: $O\big(\max_{i=1,\dots,p} \gamma_i \times ||R_i||\big)$.

The global cost of this phase is:

$$Time_{phase3} =$$

$$O\Big(\max_{i=1,\dots,p} \gamma_i \times ||Hist_i^{a1}(R \bowtie S)|| + \max_{i=1,\dots,p} \gamma_i \times (||R_i|| + ||S_i||) + p \times g + l + \max_{i=1,\dots,p} \gamma_i \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) + \max_{i=1,\dots,p} g \times \sum_{v \in Hist_i^{(HF)x}(R \bowtie S)} Proc(v) \times |order\_messages| \Big).$$

Phase 4. Data redistribution:

According to communication templates, buckets are sent to their destination processors. It is important to mention here that only tuples of R and S that effectively participate in the join result will be redistributed. So, each processor i receives a partition $\overline{\overline{R}}$ (resp. $\overline{\overline{S}}$) of R (resp. S). Therefore, in this algorithm, communication cost is highly reduced and the global cost of this phase is:

$$Time_{phase4} = O\Big(g \times \max_{i=1}^{p} (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}|) + l\Big).$$

Phase 5. Join computation:

Buckets received by each processor are arranged in a queue. Each processor executes successively the join operation of its waiting buckets. The cost of this step of local join computation is:

$$Time_{local\_join} = O\Big(\max_{i=1}^{p} \big(c_{r/w}^i \times (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}| + vol_i \times |\overline{R} \bowtie \overline{S}|) + t_h^i \times ||\overline{\overline{R_i}}|| + t_s^i \times ||\overline{\overline{S_i}}||\big)\Big),$$

where t_s is the time needed to search for an entry in the hash table and t_h is the time needed to add an entry to the hash table.

If a processor finishes computing the join related to its local data and the overall join operation is not finished, it will send to the head node a message asking for more work. Hence, the head node will assign to this idle processor some of the buckets related to join attribute values that were not redistributed earlier in the static phase. However, if all these buckets are already treated, the head node checks the number of non treated buckets in the queue of the other processors and asks the processor that has the maximal number of non treated buckets to forward a part of them to the idle one. The number of sent buckets must respect the capacity of the idle processor.

The global cost of join computation of two relations R and S using *PDFA-Join* algorithm is:

$$Time_{PDFA-Join} =$$

$$O\Big(\max_{i=1,\dots,p} c_{r/w}^i \times (|R_i| + |S_i|) + \max_{i=1,\dots,p} \gamma_i \times (||R_i|| + ||S_i||) + \max_{i=1,\dots,p} (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}|) + \min\Big(\max_{i=1,\dots,p} \omega_i \times p \times (g \times |Hist^{a_1}(R)| + \gamma_i \times ||Hist^{a_1}(R)||), \max_{i=1,\dots,p} \omega_i \times (g \times |R| + \gamma_i \times ||R||)\Big) +$$

$$\min\Big(\max_{i=1,\ldots,p}\omega_i\times p\times(g\times|Hist^{a_1}(S)|+\gamma_i\times||Hist^{a_1}(S)||),$$
$$\max_{i=1,\ldots,p}\omega_i\times(g\times|S|+\gamma_i\times||S||)\Big)+$$
$$\max_{i=1,\ldots,p}\gamma_i\times\sum_{v\in Hist_i^{(HF)x}(R\bowtie S)}Proc(v)+$$
$$\max_{i=1,\ldots,p}g\times\sum_{v\in Hist_i^{(HF)x}(R\bowtie S)}Proc(v)\times|order\_messages|+$$
$$\max_{i=1,\ldots,p}\left(c^i_{r/w}\times(|\overline{\overline{R_i}}|+|\overline{\overline{S_i}}|+vol_i\times|\overline{R}\bowtie\overline{S}|)+t^i_h\times||\overline{\overline{R_i}}||+t^i_s\times||\overline{\overline{S_i}}||)+l\right)$$

Remark 5 on page 91 related to *DFA-Join* algorithm also applies for *PDFA-Join* algorithm. This shows *PDFA-Join* algorithm has an optimal asymptotic complexity.

Discussion

To understand the whole mechanism of *PDFA-Join* algorithm, we compare existing approaches (based on hashing) to our pipelined join algorithm using different execution strategies to evaluate the multi-join query

$$\mathcal{Q}=(R\bowtie_{a_1}S)\bowtie_{b_1}(U\bowtie_{a_2}V).$$

A *Full Parallel* execution of DFA-Join algorithm (i.e. a basic use of DFA-Join where we do not use pipelined parallelism) requires the evaluation of $\mathcal{Q}_1=(R\bowtie_{a_1}S)$ and $\mathcal{Q}_2=(U\bowtie_{a_2}V)$ on two disjoint set of processors, the join results of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are then stored on the disk. The join result of query $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are read from disk to evaluate the final join query $\mathcal{Q}$.

Existing approaches allowing pipelining, start by the evaluation of the join queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$, and then each generated tuple in query $\mathcal{Q}_1$ is immediately used to build the hash table. However, the join result of query $\mathcal{Q}_2$ is stored on the disk.
At the end of the execution of $\mathcal{Q}_1$, the join result of query $\mathcal{Q}_2$ is used to probe the hash table. This induces unnecessary disk input/output. Existing approaches require data redistribution of all intermediate join result (not only relevant tuples) this may induce high communication cost. Moreover, data redistribution in these algorithms is based on hashing which makes them very sensitive to data skew.

In *PDFA-Join* algorithm, we first compute in parallel the histograms of R and S on attribute a_1, and at the same time we compute the histograms of U and V on attribute a_2. As soon as these histograms are available, we generate the communication templates for $\mathcal{Q}_1$ and $\mathcal{Q}_2$ and by the way the histograms of the join results of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ on attribute b_1 are also computed. Join histograms on attribute b_1 are used to create the communication templates for $\mathcal{Q}$ which makes it possible to immediately use the tuples generated by $\mathcal{Q}_1$ and $\mathcal{Q}_2$ to evaluate the final join query $\mathcal{Q}$.

PDFA-Join algorithm achieves several enhancements compared to pipelined join algorithm presented in the literature: During the creation of communication templates, we create on the fly the histograms for the next join, limiting by the way the number of accesses to data (and to the

disks). Moreover, data redistribution is limited to only tuples participating effectively to join result, which highly reduces the communication costs. Dynamic data redistribution in *PDFA-Join* makes it insensitive to data skew while guaranteeing perfect load balance during all the stages of join computation.

PDFA-Join can be used in various parallel strategies, however in the parallel construction of the histograms for source relations, we can notice that the degree of parallelism might be limited by two factors: the total number of processors available, and the original distribution of data. A simultaneous construction of two histograms on the same processor (which occurs when two relations are distributed, at least partially, over the same processors) would not be really interesting compared to a sequential construction. This intra-processor parallelism does not bring acceleration, but should not induce noticeable slowdown: histograms are generally small, and having several histograms in memory would not necessitate swapping. On the other hand, as relations are usually much bigger than the available memory, we have to access them by blocks. As a consequence, accessing one or several relations does not really matter. Our pipeline strategy will really be efficient if different join operators are executed on disjoint (or at least partially disjoint) sets of processors. This brings us to limit the number of simultaneous builds. As a consequence, we have to segment our query trees, similarly to segmented right-deep trees, each segment (i.e. a set of successive joins) being started when the former is over. Once the histograms are produced for both tables, we can compute the communication templates, then distribute data, and finally compute the join. Unfortunately, the computation of the communication templates is the implicit barrier within the execution flow that prohibits the use of long pipeline chains.

5.5 Conclusion

In this chapter, we presented, *DFA-Join*, a scalable parallel join algorithm for heterogeneous architectures. We also presented a pipelined version of *DFA-Join* algorithm called *PDFA-Join* for evaluating complex queries with multi-join on such architectures. These algorithms are based on an efficient dynamic data redistribution approach allowing to highly reduce communication cost and massively reducing the number of join buckets (and thus the global join computation) owing to the fact that these buckets contain only tuples participating effectively to the join results. *DFA-Join* and *PDFA-Join* algorithms guarantee perfect balancing properties on multi-user homogeneous architecture as well as on heterogeneous distributed architectures during all the stages of join computation.

Our experience with join operations and the BSP cost analysis show that the overhead related to distributed histogram[4] management remains very small compared to the gain it provides in reducing communication time and managing join buckets reallocation to balance load between processors. These algorithms proved to have an optimal complexity even for

[4]We recall that histograms are, in general, very small compared to the sizes of base relations.

highly skewed data.

We showed that *PDFA-Join* algorithm can be applied efficiently in various parallel execution strategies offering flexible resource allocation and reducing disks input/output of intermediate join result in the evaluation of multi-join queries. This algorithm achieves several enhancements compared to solutions suggested in the literature by reducing communication costs to only relevant tuples while guaranteeing perfect balancing properties on heterogeneous multi-processor shared nothing architectures even for highly skewed data.

A new approach for evaluating join queries on the grid

6

Contents

Scientific experiments in some research domains such as high energy physics, global climate change, bio-informatics, molecular biology, etc. generate a huge amount of data whose size is in the range of hundreds of megabytes to petabytes. These data are stored on geographically distributed and heterogeneous resources. Researchers who need to analyze and have a fast access to such data are also located all over the globe. Queries executed by these researchers may require the transfer of huge amount of data over the wide area network in a reasonable time. Due to these emerging needs, the grid infrastructure which connects widely geographically distributed and heterogeneous computing and storage resources, was born. In this chapter, we are interested in treating join queries on the grid. We propose a new parallel algorithm allowing to highly reduce communication and disk Input/Output costs. The algorithm is based on a dynamic task allocation strategy which makes it insensitive to data skew and ensures perfect load balancing properties during all the stages of join computation. A cost analysis is also presented to prove the efficiency and scalability of our approach.

6.1 Introduction

The difficulty of processing queries on grid infrastructure is due to several factors:

- The first one, is the heterogeneity of the used machines. In order to benefit effectively from the power of such architecture, no processor must be idle while other processors are overloaded throughout the query evaluation phase. So, the actual characteristics of resources such as CPU power, Input/Output speed, connection speed, available memory, etc. must be taken into consideration while assigning jobs to the nodes.
- The grid is usually a multi-user system, hence the load of a machine may highly vary from one instant to another. And thus, the processing capacity of a node may rapidly degrade during query processing.
- One or more nodes may fail during query processing, hence a recovery process is needed in order to transfer the tasks already assigned to the failed nodes to other available ones.

In order to override these problems, we present *GDFA-Join*: a new Grid Dynamic Frequency Adaptive parallel algorithm for evaluating join operations on the grid. In this algorithm, we use distributed histograms to determine the tuples of input relations that effectively appear in the join result, then only these tuples are redistributed. This helps to highly reduce the communication and disk Input/output costs. However, before redistributing these tuples between the processors, they are partitioned into buckets in a manner that the sizes of the join of all associated buckets are approximately equal even when input data are skewed. The size of these buckets is determined based on several factors such as the size of the join result which can be known using distributed histograms and the processing capacity of the available nodes. After this partitioning step, the buckets are distributed over the processors to compute the join result. The number of buckets received by each processor depends on its processing capacity. When a processor becomes inaccessible or overloaded, its assigned buckets are forwarded to other processors by the data nodes. We also give the cost of each phase of our algorithm. The cost analysis shows that it is scalable and guarantees a highly low communication and disk Input/Output costs even in the presence of skewed data.

6.2 The GDFA-Join Algorithm: Evaluating join queries on the Grid

In this section, we introduce *GDFA-Join* (Grid Dynamic Frequency Adaptive join) algorithm: a new parallel algorithm for evaluating join operations on the grid infrastructure. We assume that the grid connects multiple clusters of heterogeneous machines having different characteristics of memory, disk Input/Output speed, CPU power, etc. Each cluster in the grid (fig. 6.1) has a local coordinator node responsible of balancing

the load of its computing processors. For scalability, we also deploy a hierarchy structure of intermediate coordinator nodes where each one of these nodes is responsible of a set of sites. This hierarchical structure allows us to decrease the coordinating communication costs between the coordinator nodes of different clusters as much as possible.

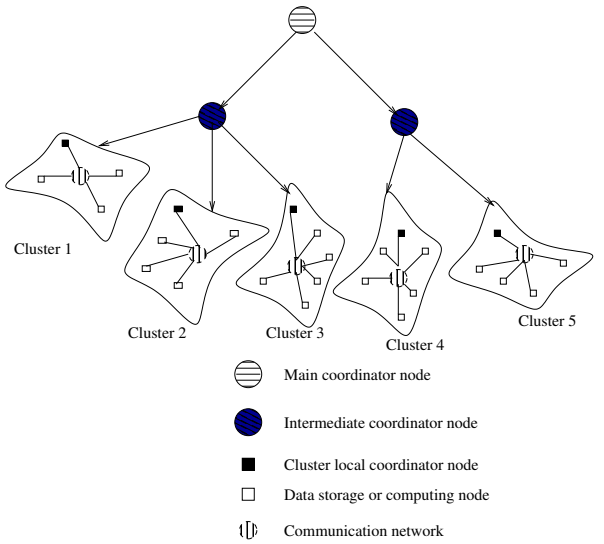

FIG. 6.1 – *GDFA-Join grid architecture.*

We assume that the relations R and S to be joined are partitioned respectively into m and n partitions by horizontal fragmentation i.e. $R = \cup_{i=1}^{m} R_i$ and $S = \cup_{i=1}^{n} S_i$. Data is replicated over several nodes of the grid for fault tolerance and for the sake of ensuring high availability and robustness. This replication also helps in accelerating access time to data.

Algorithm 14 shows the steps to be followed in order to compute the join on grid systems. These steps can be divided into the following five phases. We will give for each phase an upper bound of execution time.

In this algorithm, we use the following notations in addition to the notations introduced in page 54:

- num_{clust}: the number of clusters forming the grid,
- m_p: communication message protocol cost per page of data,
- m_l: communication message latency for one page of data,
- $c^i_{r/w}$: read/write cost of a page of data on local disk on processor i,
- t^i_r: time to read a record from main memory of processor i,
- t^i_w: time to write a record to main memory of processor i,
- t^i_s: time of a simple search in a B$^+$-tree on processor i,
- t^i_h: time to add an entry to a B$^+$-tree on processor i,
- P_T: the list of processors' indexes holding a replication of T,
- $Proc_{Hist^x(R \bowtie S)}$: the list of processors' indexes that participate in computing $Hist^x(R \bowtie S)$,
- $rep_{(T)}$: the number of processors holding a replica of T,
- $bucket\_size$: the size of generated join buckets specified by the coordinator node.

Algorithm 14: GDFA-Join (Grid Dynamic Frequency Adaptive Join) algorithm.

1► On each participating cluster designated by the query parser and optimizer
  ▷ Create, in parallel, the local histograms of R (resp. S) on the assigned nodes holding sub-relation of R (resp. S). On the fly, partition the tuples into buckets using Round-robin method.

2► **When** the local histogram of each partition of R and S is available on a sub-group, $Proc_{Hist^x(R \bowtie S)}$, of processors, **then**
  ▷ Hash the local histograms of R and S over the processors $Proc_{Hist^x(R \bowtie S)}$ to create their distributed global histogram partitions;
  ▷ Create, $Hist^x(R \ltimes S)$, the histogram of $R \ltimes S$, by finding the intersection of the distributed global histograms of R and S;
  ▷ The main coordinator node determines the optimal join buckets size based on the global join size and the processing capacities of available nodes;

3► Each processor in $Proc_{Hist^x(R \bowtie S)}$, in parallel,
  ▷ Divides the join attribute values into sub-groups in a manner that the size of the join of tuples related to each subset is approximately equal to the specified bucket size;
  ▷ Forwards the partitioning information to their destination processors in $Proc_{Hist^x(R \bowtie S)}$;
  ▷ Forwards the partitioning information to processors not in $Proc_{Hist^x(R \bowtie S)}$ which have participated in creating the local histograms and hold a replica of its local input data;

4► Each processor holding a sub-relation of input data, filters a number of generated buckets in step *1* according to its capacity as indicated in the communication templates;

5► **Join computation**:
  ▷ *assigning data buckets to computing nodes*: local cluster coordinator nodes assign data buckets to local processing nodes according to their processing capacities;
  ▷ Exchange data buckets;
  ▷ *Load balancing during join computation due to grid multi-user characteristic*: local coordinator nodes try to balance the load locally during the join computation phase. If this is not possible locally, then processors of other sites are chosen by the intermediate coordinator nodes;

Phase 1. Choosing nodes responsible of relations partitions:

The Grid architecture connects a huge number of nodes. So, we usually have the possibility to assign to each partition of R and S at least one processor that is responsible of creating their local histogram, then preparing the buckets formed of tuples that effectively appear in the join result. This assigning step is carried out by the query parser and optimizer. In this step, some conflicts may arise. For example, if all the processors holding a specific partition S_k of S are already assigned to treat partitions of R, then S_k is immigrated to another processor that is not occupied. Thus, a wise data placement and replication strategy must be used in order to decrease, as much as possible, the need of migrating data between processors.

Globus toolkit provides software services and libraries for grid computing environments (Allcock et al. 2001). Some of these services (GridFTP, the replica catalog and the Replica Management services) are dedicated for data transfer and managing multiple copies of data sets over the grid. The data management service offers tools that allow the user to replicate data over the grid, but it leaves the definition of replication semantics for the user. Thus, in order to decrease as much as possible the migration of relations partitions between processors, we use a data management strategy that avoids, when possible, saving relations that have possible common join attributes on the same processors. This step needs the access to database index and is performed in a constant time.

Phase 2. Creating local histograms of R and S:

In this phase, each processor i holding a fragment R_j of R (resp. S_k of S) partitions it, in parallel, into a multiple number of buckets using Round-robin method. We have chosen to use this partitioning method instead of hash functions in order to create, on each processor, data buckets having approximately the same number of tuples even if the data are highly skewed. We will see, in further phases, the benefit of this partitioning and data replication to accelerate data redistribution and recovery treatment due to possible nodes failure.
On the fly, we also create the local histogram $Hist^x(R_j)$ (resp. $Hist^x(S_k)$) of R_j (resp. S_k) that holds the frequency n_v of each join attribute value v of R_j (resp. S_k).
We can notice that the histogram related to the same partition of R or S is computed, in parallel, on several processors holding a replica of this partition. We have chosen to follow this strategy, instead of evaluating the histogram of each fragment on the fastest processor holding a copy of this fragment, due to the dynamic characteristic of grid systems where a node may suddenly become unavailable or an underloaded processor may become overloaded. When the creation of the histograms related to all the partitions of R and S is terminated on a subgroup, $Proc_{Hist^x(R \bowtie S)}$, of the assigned processors, these processors are informed by the coordinator node to compute the global histogram of $R \bowtie S$. So, we do not need to wait all the used processors to finish computing the local histograms. However, the histogram of each partition of R and S must be available on at least one processor.

The cost of hashing tuples of R_j into buckets and creating its local histogram is of the order:

$$O\Big(\min_{i \in P_{R_j}} \big(|R_j| \times c^i_{r/w} + ||R_j|| \times (t^i_r + t^i_w + t^i_h)\big)\Big),$$

where $|R_j| \times c^i_{r/w}$ represents the cost of loading data from the local disk(s) of processor i and $||R_j|| \times (t^i_r + t^i_w)$ is the cost of partitioning tuples into buckets. The histograms are created, on the fly, during the creation of R_j's data buckets with cost $||R_j|| \times t^i_h$.

Thus, the total cost of partitioning tuples of relation R into buckets and creating the local histogram partitions on processors $Proc_{Hist^x(R \bowtie S)}$ is of the order (the same cost also applies for S):

$$\max_{j=1}^{m}\Big(\min_{i\in P_{R_j}}\big(|R_j|\times c^i_{r/w}+||R_j||\times(t^i_r+t^i_w+t^i_h)\big)\Big).$$

So, the global cost of this phase is:

$$Time_{phase_2}=O\Big(max_{j=1}^{m}\Big(\min_{i\in P_{R_j}}\big(|R_j|\times c^i_{r/w}+||R_j||\times(t^i_r+t^i_w+t^i_h)\big)\Big)+ max_{j=1}^{n}\Big(\min_{i\in P_{S_j}}\big(|S_j|\times c^i_{r/w}+||S_j||\times(t^i_r+t^i_w+t^i_h)\big)\Big)\Big).$$

Phase 3. Creating the global histograms:

Communication cost is one of the factors that affects the total processing time of the join operation in parallel and distributed systems. So, decreasing this cost, as much as possible, results in accelerating the treatment of this operation. To this end, only tuples related to join attribute values that satisfy the join condition must be redistributed. In fact, these are the values that appear in $Hist^x(R \bowtie S)$: the global histogram of $R \bowtie S$. However, to compute the partition $Hist^x_i(R \bowtie S)$ of this global histogram, on processor i, we firstly need to create the distributed global histograms $Hist^x_i(R)$ and $Hist^x_i(S)$ by redistributing the tuples of the local histograms on processors $Proc_{Hist^x(R\bowtie S)}$ using a hash function. So, each processor $i \in Proc_{Hist^x(R\bowtie S)}$ holding a partition of R (resp. S) sends to each processor $j \in Proc_{Hist^x(R\bowtie S)}$ a partition $Hist^{x[j]}(R_i)$ (resp. $Hist^{x[j]}(S_i)$) of $Hist^x(R_i)$ (resp. $Hist^x(S_i)$) such that $Hist^x(R_i)=\cup_{j\in Proc_{Hist^x(R\bowtie S)}} Hist^{x[j]}(R_i)$ (resp. $Hist^x(S_i)=\cup_{j\in Proc_{Hist^x(R\bowtie S)}} Hist^{x[j]}(S_i)$).
The cost of this step is at most (a detailed proof of this cost is given in proposition 2 of *Appendix A*):

$$\begin{aligned}O\Big(&\min\big(|Hist^x(R)|,\frac{|R|}{m}\big)\times(m_p+m_l)+\min\big(|Hist^x(S)|,\frac{|S|}{n}\big)\times(m_p+m_l)+\\ &\min\Big(\frac{m}{m+n}\times\big(|Hist^x(R)|\times m_p+\max_{i\in Proc_{Hist^x(R\bowtie S)}}||Hist^x(R)||\times(t^i_r+t^i_h)\big),\\ &\quad\frac{m}{(m+n)^2}\times\big(|R|\times m_p+\max_{i\in Proc_{Hist^x(R\bowtie S)}}||R||\times(t^i_r+t^i_h)\big)\Big)+\\ &\min\Big(\frac{n}{m+n}\times\big(|Hist^x(S)|\times m_p+\max_{i\in Proc_{Hist^x(R\bowtie S)}}||Hist^x(S)||\times(t^i_r+t^i_h)\big),\\ &\quad\frac{n}{(m+n)^2}\times\big(|S|\times m_p+\max_{i\in Proc_{Hist^x(R\bowtie S)}}||S||\times(t^i_r+t^i_h)\big)\Big)\Big).\end{aligned}$$

Now, we can create $Hist^x_i(R \bowtie S)$ on each processor i of $Proc_{Hist^x(R\bowtie S)}$, in parallel, by computing the intersection $Hist^x_i(R) \cap Hist^x_i(S)$ with a cost of the order:

$$O\Big(\max_{i\in Proc_{Hist^x(R\bowtie S)}}\Big(\min\big(||Hist^x_i(R)||,||Hist^x_i(S)||\big)\times(t^i_r+t^i_s+t^i_h)\Big)\Big).$$

This cost is the necessary time to read each entry of the smallest histogram. And, for each read entry, we need to test if the value of the join attribute appears in the second histogram, and if it is the case, then we must add a new entry to $Hist^x_i(R \bowtie S)$.

During this step, each processor i also retains a trace of the network layout of values v in its $Hist_i^x(R)$ (resp. $Hist_i^x(S)$). This will be useful in the phase of creating the communication templates.

It is important to mention here that using hash functions to distribute the tuples of histograms does not create load imbalance on the processors even if data of the input relations is highly skewed. This is due to the fact that histograms hold for each join attribute value v a unique record (v, n_v) where n_v represents its frequency. In addition, the size of these histograms is very small compared to the size of the input relations and the overhead related to the management of distributed histograms remains very small compared to the gain that they provide in load balancing and reducing communication costs.
When a processor i finishes from computing its assigned partition of $Hist^x(R \bowtie S)$, it sends the sum of the frequencies related to the join attribute values v such that $v \in Hist_i^x(R \bowtie S)$ to the local coordinator node. Then, each local coordinator node sends the sum of the received values to the main coordinator node. At the end of this step, the main coordinator node knows the total number of tuples in $R \bowtie S$. This information is used while creating the communication templates in the next step.

The total cost of this phase is at most:
$Time_{phase_3} =$

$$
\begin{aligned}
O\Big(& \min\big(|Hist^x(R)|, \frac{|R|}{m}\big) \times (m_p + m_l) + \min\big(|Hist^x(S)|, \frac{|S|}{n}\big) \times (m_p + m_l) + \\
& \min\Big(\frac{m}{m+n} \times \big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t_r^i + t_h^i)\big), \\
& \quad \frac{m}{(m+n)^2} \times \big(|R| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||R|| \times (t_r^i + t_h^i)\big)\Big) + \\
& \min\Big(\frac{n}{m+n} \times \big(|Hist^x(S)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(S)|| \times (t_r^i + t_h^i)\big), \\
& \quad \frac{n}{(m+n)^2} \times \big(|S| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||S|| \times (t_r^i + t_h^i)\big)\Big) + \\
& \max_{i \in Proc_{Hist^x(R \bowtie S)}} \Big(\min\big(||Hist_i^x(R)||, ||Hist_i^x(S)||\big) \times (t_r^i + t_s^i + t_h^i)\Big)\Big).
\end{aligned}
$$

Phase 4. Join buckets creation:

In order to solve the problems presented in the introduction, we will use a strategy based on creating buckets formed of tuples that effectively appear in the join result. These buckets are created in a manner that the size of the join of associated buckets of R and S are approximately equal. So, the main coordinator node determines the size of the join buckets *bucket_size* based on the final join result size and the computing characteristics of the available nodes.

4.a *Creating the partitioning schema of the join attribute values* :
To create these data buckets, all the nodes in $Proc_{Hist^x(R \bowtie S)}$ that have evaluated partitions of $Hist^x(R \bowtie S)$ participate in creating the communication templates. So, this step is performed jointly by all processors of $Proc_{Hist^x(R \bowtie S)}$ *each one not necessarily computing the list of its own messages*,

in order to balance the overall process. To this end, each processor i divides the set of join attribute values $v \in Hist^x(R \bowtie S)$ into multiple subsets. And, the size of the join of tuples related to all values v in each subset is approximately equal to the size *bucket_size* of buckets specified by the coordinator node. So, each processor $i \in Proc_{Hist^x(R \bowtie S)}$ executes algorithm 15 to create the communication templates for relation R. A similar algorithm is executed for S.

Algorithm 15: Creating the communication templates related to R on processor i in GFA-Join algorithm

In Parallel (on each processor) $i \in [1, p]$ **do**
▷ **for each** $j \in Proc_{Hist^x(R \bowtie S)}$ **do**
▷ $bucket\_index = 0$;
▷ $size = 0$;
▷ **for each** value $v \in Hist^{x[i]}(R_j)$ **do**
▷ **if** $v \in Hist_i^x(R \bowtie S)$ **then**
▷ **if** $size + Hist_i^x(R \bowtie S)(v) \leq bucket\_size$ **then**
▷ add a message $order\_to\_hash(v, bucket\_index, i)$ to the list of messages of j; /* *Hash the tuples of R associated to v into a bucket of index (bucket_index, i).*/
▷ $size$ += $Hist_i^x(R \bowtie S)(v)$;
▷ **else if** $Hist_i^x(R \bowtie S)(v) \leq bucket\_size$ **then**
▷ $bucket\_index$ += 1;
▷ add a message $order\_to\_hash(v, bucket\_index, i)$ to the list of messages of j; /* *Hash the tuples of R associated to v into a bucket of index (bucket_index, i).*/
▷ $size = Hist_i^x(R \bowtie S)(v)$.
▷ **else**
▷ **if** $size \neq 0$ **then** $bucket\_index$ += 1; **endif**
▷ **if** $Hist_i^x(R)(v) \leq Hist_i^x(S)(v)$ **then**
▷ add $order\_to\_partition(v, bucket\_index, i, \lceil \frac{Hist_i^x(R \bowtie S)(v)}{bucket\_size} \rceil, S\_index)$ to the list of messages of j; /* *Hash the tuples of R associated to v into a bucket of index (bucket_index, i).* */
▷ **else**
▷add $order\_to\_partition(v, bucket\_index, i, \lceil \frac{Hist_i^x(R \bowtie S)(v)}{bucket\_size} \rceil, R\_index)$ to the list of messages of j; /* *Divide the tuples of R associated to v into* $\lceil \frac{Hist_i^x(R \bowtie S)(v)}{bucket\_size} \rceil$ *buckets with indexes between bucket_index and* $bucket\_index + \lceil \frac{Hist_i^x(R \bowtie S)(v)}{bucket\_size} \rceil - 1$. */
▷ $size = 0$;
▷ $bucket\_index$ += $\lceil \frac{Hist_i^x(R \bowtie S)(v)}{bucket\_size} \rceil$;
▷ **endif**
▷ **endif**
▷ **endfor**
▷ **endfor**
EndPar

The global cost of algorithm 15 is at most of the order:

$$O\Big(\big(\sum_{j=1}^{p} ||Hist^{x[i]}(R_j)|| + \sum_{j=1}^{p} ||Hist^{x[i]}(R_j)|| \big)(t_r^i + t_w^i + 2 \times t_s^i) \Big).$$

However, we have:

$$\sum_{j=1}^{p} ||Hist^{x[i]}(R_j)|| \leq \min\big(\frac{m}{m+n} \times ||Hist^x(R)||, \frac{||R||}{m+n}\big),$$

and

$$\sum_{j=1}^{p} ||Hist^{x[i]}(S_j)|| \leq \min\big(\frac{n}{m+n} \times ||Hist^x(R)||, \frac{||S||}{m+n}\big).$$

These inequalities hold, owing to the fact that, in the general case of join attribute values frequency distribution, each processor in $Proc_{Hist^x(R \bowtie S)}$ is responsible of approximately $\frac{||Hist^x(R)||}{m+n}$ (resp. $\frac{||Hist^x(S)||}{m+n}$) join attribute values, and in the same set of processors each join attribute value may appear in at most m (resp. n) processors holding R (resp. S) partitions. But, if the frequency of each join attribute value in R (resp. S) is equal to 1, then each processor will be responsible of $\frac{||R||}{m+n}$ (resp. $\frac{||S||}{m+n}$) values.

Thus, the cost of this algorithm is at most of the order:

$$O\Big(\max_{i \in Proc_{Hist^x(R \bowtie S)}} \min\big(\frac{m}{m+n} \times ||Hist^x(R)||, \frac{||R||}{m+n}\big) \times (t_r^i + t_w^i + 2 \times t_s^i) + \max_{i \in Proc_{Hist^x(R \bowtie S)}} \min\big(\frac{n}{m+n} \times ||Hist^x(S)||, \frac{||S||}{m+n}\big) \times (t_r^i + t_w^i + 2 \times t_s^i)\Big).$$

After that, the list of messages are sent to their destinations to create the join buckets holding only tuples that appear in the join result.
The cost of this communication step is at most:

$$O\Big(\min\big(\frac{m}{m+n} \times |\overline{Hist}^x(R)|, \frac{|\overline{R}|}{m+n}\big) \times (m_p + m_l) + \min\big(\frac{n}{m+n} \times |\overline{Hist}^x(S)|, \frac{|\overline{S}|}{m+n}\big) \times (m_p + m_l) + \max\big(\max_{i=1}^{m} |\overline{Hist}^x(R_i)| \times m_p, \max_{i=1}^{n} |\overline{Hist}^x(S_i)| \times m_p\big)\Big),$$

where $\min\big(\frac{m}{m+n} \times |\overline{Hist}^x(R)|, \frac{|\overline{R}|}{m+n}\big) \times (m_p + m_l)$ (resp. $\min\big(\frac{n}{m+n} \times |\overline{Hist}^x(S)|, \frac{|\overline{S}|}{m+n}\big) \times (m_p + m_l)$) is the cost of sending the list of partition messages to processors of $Proc_{Hist^x(R \bowtie S)}$ holding partitions of R (resp. S). The cost of receiving the messages related to R (resp. S) partitions is of the order: $O(\max_{i=1}^{m} |\overline{Hist}^x(R_i)| \times m_p)$ (resp. $O(\max_{i=1}^{n} |\overline{Hist}^x(S_i)| \times m_p)$). Here, $\overline{Hist}^x(R_i)$ (resp. $\overline{Hist}^x(S_i)$) is the restriction of $Hist^x(R_i)$ (resp. $Hist^x(S_i)$) to values of the join attribute that appear in the join result.

So, the total cost of this step is the sum of the above two costs:

$$Time_{phase4.a} =$$

$$O\Big(\max_{i \in Proc_{Hist^x(R \bowtie S)}} \min\big(\frac{m}{m+n} \times ||Hist^x(R)||, \frac{||R||}{m+n}\big) \times (t_r^i + t_w^i + 2 \times t_s^i) +$$

$$\max_{i \in Proc_{Hist^x(R \bowtie S)}} \min\big(\frac{n}{m+n} \times ||Hist^x(S)||, \frac{||S||}{m+n}\big) \times (t_r^i + t_w^i + 2 \times t_s^i) +$$

$$\min\big(\frac{m}{m+n} \times |\overline{Hist}^x(R)|, \frac{|\overline{R}|}{m+n}\big) \times (m_p + m_l) +$$

$$\min\big(\frac{n}{m+n} \times |\overline{Hist}^x(S)|, \frac{|\overline{S}|}{m+n}\big) \times (m_p + m_l) +$$

$$\max\big(\max_{i=1}^{m} |\overline{Hist}^x(R_i)| \times m_p, \max_{i=1}^{n} |\overline{Hist}^x(S_i)| \times m_p\big)\Big).$$

4.b *Join buckets creation*:
Now, we need to create the following semi-joins: $\overline{R}_j = R_j \ltimes S$ ($j = 1, \ldots, m$) and $\overline{S}_k = S_k \ltimes R$ ($k = 1, \ldots, n$). The fragment $\overline{R}_j$ (resp. $\overline{S}_k$) holds the tuples of R_j (resp. S_k) which participate in the join result. To create these semi-joins, each processor of $Proc_{Hist^x(R \bowtie S)}$ obeys the message orders it has just received. So, for each received message $order\_to\_hash(v, bucket\_index, i)$, the processor saves all tuples having value v of the join attribute to a bucket of index $(bucket\_index, i)$. For messages $order\_to\_partition(v, bucket\_index, \lceil \frac{Hist_i^x(R \bowtie S)(v)}{bucket\_size} \rceil, relation\_index)$, if the relation has the index $relation\_index$, then the tuples will be partitioned over $\lceil \frac{Hist_i^x(R \bowtie S)(v)}{bucket\_size} \rceil$ buckets using Round-robin method. And in this case, the indexes of the buckets are between $bucket\_index$ and $bucket\_index - 1$. If this is not the case (the relation does not have the index $relation\_index$), then all the tuples holding a value v are saved in a bucket of index $bucket\_index$. However, while redistributing the data, this bucket must be sent to each processor that received an associated bucket of the second relation in order to have a valid result. Each coordinator node is responsible of assigning the computing nodes for the data buckets of index $(bucket\_index, i)$ if processor i is in its cluster. This makes our algorithm scalable because we do not have a central coordinator node that manages all the tasks. So, the cost of computing $\overline{R}_j$ on processor i is of the order:

$$O(||R_j|| \times (t_r^i + t_w^i + t_s^i)).$$

(This cost also applies for the partitions of S).
However, in this semi-join computation step, we can benefit from the replication of base relations to reduce its cost. We recall here that in the second phase, each processor i holding a sub-relation of R or S has partitioned its tuples over multiple buckets using the Round-robin method. So, we use this partitioning to decrease the creation time of buckets containing tuples that effectively participate in the join result.
To this end, we will assign to each processor that belongs to a set of processors holding a replica of the same sub-relation of R or S a number of buckets related to this sub-relation. Hence, each processor in $Proc_{Hist^x(R \bowtie S)}$ will forward the list of distribution messages it has received over these processors. To reduce the communication cost, we follow the following strategy to redistribute the communication templates. Each processor i in $Proc_{Hist^x(R \bowtie S)}$ holding a replication R_j of R forwards, in a first step, the communication template to only one processor in P_{R_j} [1]. And then, in a second step, each one of these two processors forwards the communication

[1] we recall that P_{R_j} is the list of processors holding a replication of R_j.

templates to another two processors and so on. Following this method, we only need $\log_2(rep_{(R_j)})$ communication steps where $rep_{(R_j)}$ is the number of processors holding a replication of R_j.
Thus, the communication cost relative to each partition R_j of relation R is at most:

$$O\Big(\max_{j=1}^{m}\Big(\log_2(rep_{(R_j)})\times\big(|\overline{Hist}^x(R_j)|\times(2\times m_p+m_l)\big)\Big)\Big),$$

and to each S_j partition of S is at most:

$$O\Big(\max_{j=1}^{n}\Big(\log_2(rep_{(R_j)})\times\big(|\overline{Hist}^x(S_j)|\times(2\times m_p+m_l)\big)\Big)\Big).$$

Since we used the Round-robin method to create the data buckets in the second phase, each processor i holding a replica of a partition R_j of R will be responsible of approximately $\frac{||R_j||}{rep_{(R_j)}}$. Thus, the cost of creating $\overline{R}_j$ ($\forall j \in \{1,\ldots,m\}$) and partitioning its tuples over the buckets, as specified in the communication templates, is of the order:

$$O\Big(\max_{j=1}^{m}\Big(\max_{i\in P_{R_j}}\big(\frac{||R_j||}{rep_{(R_j)}}\times(t_r^i+t_w^i+t_s^i)\big)\Big)\Big).$$

Similarly, the cost of creating $\overline{S}_j$ ($\forall j \in \{1,\ldots,n\}$) and partitioning its tuples over the buckets, as specified in the communication templates, is of the order:

$$O\Big(\max_{j=1}^{n}\Big(\max_{i\in P_{S_j}}\big(\frac{||S_j||}{rep_{(S_j)}}\times(t_r^i+t_w^i+t_s^i)\big)\Big)\Big).$$

So, the total cost of this semi-join and partitioning step *(4.b)* is of the order:

$$\begin{aligned}O\Big(&\max\Big(\max_{j=1}^{m}\Big(\max_{i\in P_{R_j}}\big(\frac{||R_j||}{rep_{(R_j)}}\times(t_r^i+t_w^i+t_s^i)\big)\Big),\max_{j=1}^{n}\Big(\max_{i\in P_{S_j}}\big(\frac{||S_j||}{rep_{(S_j)}}\times(t_r^i+t_w^i+t_s^i)\big)\Big)\Big)+\\
&\max_{j=1}^{m}\Big(\log_2(rep_{(R_j)})\times\big(|\overline{Hist}^x(R_j)|*(2*m_p+m_l)\big)\Big)+\\
&\max_{j=1}^{n}\Big(\log_2(rep_{(S_j)})\times\big(|\overline{Hist}^x(S_j)|\times(2\times m_p+m_l)\big)\Big)\Big).\end{aligned}$$

The total cost of this phase is at most:

$$\begin{aligned}Time_{phase4}=O\Big(&\max_{i\in Proc_{Hist^x(R\bowtie S)}}\min\big(\frac{m}{m+n}\times||Hist^x(R)||,\frac{||R||}{m+n}\big)\times(t_r^i+t_w^i+2\times t_s^i)+\\
&\max_{i\in Proc_{Hist^x(R\bowtie S)}}\min\big(\frac{n}{m+n}\times||Hist^x(S)||,\frac{||S||}{m+n}\big)\times(t_r^i+t_w^i+2\times t_s^i)+\\
&\min\big(\frac{m}{m+n}\times|\overline{Hist}^x(R)|,\frac{|\overline{R}|}{m+n}\big)\times(m_p+m_l)+\\
&\min\big(\frac{n}{m+n}\times|\overline{Hist}^x(S)|,\frac{|\overline{S}|}{m+n}\big)\times(m_p+m_l)+\\
&\max\big(\max_{i=1}^{m}|\overline{Hist}^x(R_i)|\times m_p,\max_{i=1}^{n}|\overline{Hist}^x(S_i)|\times m_p\big)+\end{aligned}$$

$$\max\Big(\max_{j=1}^{m}\Big(\max_{i\in P_{R_j}}\big(\frac{||R_j||}{rep_{(R_j)}}\times(t_r^i+t_w^i+t_s^i)\big)\Big),$$
$$\max_{j=1}^{n}\Big(\max_{i\in P_{S_j}}\big(\frac{||S_j||}{rep_{(S_j)}}\times(t_r^i+t_w^i+t_s^i)\big)\Big)\Big)+$$
$$\max_{j=1}^{m}\Big(\log_2(rep_{(R_j)})\times\big(|\overline{Hist}^x(R_j)|\times(2\times m_p+m_l)\big)\Big)+$$
$$\max_{j=1}^{n}\Big(\log_2(rep_{(S_j)})\times\big(|\overline{Hist}^x(S_j)|\times(2\times m_p+m_l)\big)\Big)\Big).$$

Phase 5. Join computation phase:

In this step, each coordinator node chooses, from its cluster, the processors with the highest performance and lowest load in order to compute the join of the buckets. It is preferred to choose these processors from the ones that do not hold data buckets related to the query. This is possible due to the huge number of processors in grid architectures. The number of computing processors is chosen based on the join result size and the nodes performance and characteristics.

5.a *Assigning data buckets to computing nodes*:
As stated in phase 4, each coordinator node is responsible of assigning the task of joining a set of buckets distributed over the nodes of the grid to its local computing processors. So firstly, it will determine the number of buckets that can be treated in the local nodes of its cluster based on their computing characteristics and load. Then, each coordinator node asks the nodes holding these buckets to forward them to the chosen processors. However, if the site cannot treat all these buckets, then it will ask its intermediate coordinator node to find nodes on other clusters that can treat the join of the remaining buckets. For scalability, load is first balanced inside each set of processors and whenever a set of processors finishes its assigned tasks, it asks another local coordinator for additional tasks.
Each processor holding partitions of R or S, will forward these buckets to the computing nodes as specified by the coordinator nodes with a total cost of order:

$$O\Big(\max\Big(\max_{j=1}^{m}\big(\frac{|\overline{R}_j|}{rep_{(R_j)}}\times(m_p+m_l)\big),\max_{j=1}^{n}\big(\frac{|\overline{S}_j|}{rep_{(S_j)}}\times(m_p+m_l)\big)\Big)\Big).$$

Each processing node i receives a partition $\overline{\overline{R_i}}$ of $\overline{R}$ and a partition $\overline{\overline{S_i}}$ of $\overline{S}$ relative to its processing capacity such that:

$$||\overline{\overline{R_i}}\bowtie\overline{\overline{S_i}}||\leq vol_i\times||R\bowtie S||.$$

The cost of receiving these buckets is at most:

$$O\Big(\max_{i\in Proc_{Hist^x(R\bowtie S)}}(|\overline{\overline{R_i}}|+|\overline{\overline{S_i}}|)\times m_p\Big).$$

After receiving the buckets, each processor i evaluates their join with a cost of the order:

$$O\Big(\max_{i\in Proc_{Hist^x(R\bowtie S)}}\Big(c_{r/w}^i\times\big(|\overline{\overline{R_i}}|+|\overline{\overline{S_i}}|+vol_i\times|R\bowtie S|\big)+t_h^i\times||\overline{\overline{R_i}}||+t_s^i\times||\overline{\overline{S_i}}||\Big)\Big).$$

So, the total cost of this phase is of the order:
$Time_{phase5} =$

$$O\Big(\max\Big(\max_{j=1}^{m}\big(\frac{|\overline{R}_j|}{rep_{(R_j)}} \times (m_p + m_l)\big), \max_{j=1}^{n}\big(\frac{|\overline{S}_j|}{rep_{(S_j)}} \times (m_p + m_l)\big)\Big) + \max_{i \in Proc_{Hist^x(R \bowtie S)}} \Big(c^i_{r/w} \times \big(|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}| + vol_i \times |R \bowtie S|\big) + t^i_h \times ||\overline{\overline{R_i}}|| + t^i_s \times ||\overline{\overline{S_i}}||\Big) + \max_{i \in Proc_{Hist^x(R \bowtie S)}} (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}|) \times m_p \Big).$$

5.b *Load balancing during join computation*:
The grid is a multi-user system. Hence, during the join computation phase some processors may become over-loaded. If this happens then the local coordinator node tries to balance the load of the computing processors locally if possible, otherwise it will ask for the help from other clusters.

5.c *Failure recovery*:
If a join computing node becomes unavailable, then the coordinator node chooses a processor from its site to replace this node. If this is not possible then it will ask its intermediate coordinator node to propose processing nodes in other site. After that the coordinator node asks the data nodes that have already sent data buckets to the failed node to resend them to the new one.

The global cost of computing the join operation on the grid using *GDFA-Join* algorithm is:

$Time_{GDFA-Join} =$

$$\begin{aligned} O\Big(& max_{j=1}^{m}\Big(\min_{i \in P_{R_j}} \big(|R_j| \times c^i_{r/w} + ||R_j|| \times (t^i_r + t^i_w + t^i_h)\big)\Big) + \\ & max_{j=1}^{n}\Big(\min_{i \in P_{S_j}} \big(|S_j| \times c^i_{r/w} + ||S_j|| \times (t^i_r + t^i_w + t^i_h)\big)\Big) + \\ & \min\big(|Hist^x(R)|, \frac{|R|}{m}\big) \times (m_p + m_l) + \min\big(|Hist^x(S)|, \frac{|S|}{n}\big) \times (m_p + m_l) + \\ & \min\Big(\frac{m}{m+n} \times \big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t^i_r + t^i_h)\big), \\ & \quad \frac{m}{(m+n)^2} \times \big(|R| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||R|| \times (t^i_r + t^i_h)\big)\Big) + \\ & \min\Big(\frac{n}{m+n} \times \big(|Hist^x(S)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(S)|| \times (t^i_r + t^i_h)\big), \\ & \quad \frac{n}{(m+n)^2} \times \big(|S| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||S|| \times (t^i_r + t^i_h)\big)\Big) + \\ & \max_{i \in Proc_{Hist^x(R \bowtie S)}} \Big(\min\big(||Hist^x_i(R)||, ||Hist^x_i(S)||\big) \times (t^i_r + t^i_h + t^i_s)\Big) + \\ & \max_{i \in Proc_{Hist^x(R \bowtie S)}} \min\big(\frac{m}{m+n} \times ||\overline{Hist}^x(R)||, \frac{||\overline{R}||}{m+n}\big) \times (t^i_r + t^i_w + 2 \times t^i_s) + \end{aligned}$$

$$\max_{i \in Proc_{Hist^x(R \bowtie S)}} \min\big(\frac{n}{m+n} \times ||\overline{Hist}^x(S)||, \frac{||\overline{S}||}{m+n}\big) \times (t_r^i + t_w^i + 2 \times t_s^i) +$$
$$\max\big(\max_{i=1}^{m} |\overline{Hist}^x(R_i)| \times m_p, \max_{i=1}^{n} |\overline{Hist}^x(S_i)| \times m_p\big) +$$
$$\max\Big(\max_{j=1}^{m}\Big(\max_{i \in P_{R_j}}\big(\frac{||R_j||}{rep_{(R_j)}} \times (t_r^i + t_w^i + t_s^i)\big)\Big),$$
$$\max_{j=1}^{n}\Big(\max_{i \in P_{S_j}}\big(\frac{||S_j||}{rep_{(S_j)}} \times (t_r^i + t_w^i + t_s^i)\big)\Big)\Big) +$$
$$\max_{j=1}^{m}\Big(\log_2(rep_{(R_j)}) \times (|\overline{Hist}^x(R_j)| \times (2 \times m_p + m_l))\Big) +$$
$$\max_{j=1}^{n}\Big(\log_2(rep_{(S_j)}) \times (|\overline{Hist}^x(S_j)| \times (2 \times m_p + m_l))\Big) +$$
$$\max\Big(\max_{j=1}^{m}\big(\frac{|\overline{R}_j|}{rep_{(R_j)}} \times (m_p + m_l)\big), \max_{j=1}^{n}\big(\frac{|\overline{S}_j|}{rep_{(S_j)}} \times (m_p + m_l)\big)\Big) +$$
$$\max_{i \in Proc_{Hist^x(R \bowtie S)}} (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}|) \times m_p +$$
$$\max_{i \in Proc_{Hist^x(R \bowtie S)}} \Big(c_{r/w}^i \times (|\overline{\overline{R_i}}| + |\overline{\overline{S_i}}| + vol_i \times |R \bowtie S|) + t_h^i \times ||\overline{\overline{R_i}}|| + t_s^i \times ||\overline{\overline{S_i}}||\Big)\Big).$$

Remark 6 *Sequential evaluation of the join of two relations R and S on processor i requires at least the following lower bound (the time to scan the input and to store the join result)*:

$$bound_{inf_1} = \Omega\big(c_{r/w}^i \times (|R| + |S| + |R \bowtie S|) + t_h^i \times ||R|| + t_s^i||S||\big).$$

Therefore, parallel join processing on p heterogeneous processors requires:

$$bound_{inf_p} = \Omega\Big(\max_i \big(c_{r/w}^i \times vol_i \times (|R| + |S| + |R \bowtie S|) + vol_i \times (t_h^i \times ||R|| + t_s^i \times ||S||)\big)\Big).$$

GDFA-Join algorithm has optimal asymptotic complexity since all the terms in $Time_{GDFA-Join}$ *are bounded by those of* $bound_{inf_p}$.

6.3 GDFA-Join Performance Evaluation

One of the main goals of the grid is to allow users to access and to analyze databases owned by different organizations. Thus, it is very probable that these organizations use different database management systems such as IBM, MySQL, Microsoft SQL server, Oracle, etc., or even different database paradigms (relational, XML, object, etc.). So, a middleware is needed to allow users to know the properties of structured data sources and access them. OGSA-DAI is such a service-based middleware that supports access, sharing, management and coordinated use of heterogeneous physical data sources on the grid by providing a uniform service interface to databases exposed to the grid (Antonioletti et al. 2005).
The main services of OGSA-DAI are the Grid Data Service (GDS) and Grid Data Service Factory (GDSF). The GDS represents a client session with a physical data resource, while GDSF service is used to represent the presence of a physical data resource on the grid. GDS services are created

TAB. 6.1 – *Computing resource characteristics*

Cluster ID	CPU Speed (GHz)	CPUs per node	Cores per CPU	Memory (GB)	Storage
1	2.0	2	1	2	80 GB / IDE
2	2.2	2	2	4	2x73 GB Raido / SAS
3	2.6	2	2	4	250 GB / SATA

by the GDSF and any client that needs to interact with the physical data resource has to instantiate a GDS. We propose to use OGSA-DAI as a middleware between database servers and our algorithm in order to allow the potential access to heterogeneous data sources.

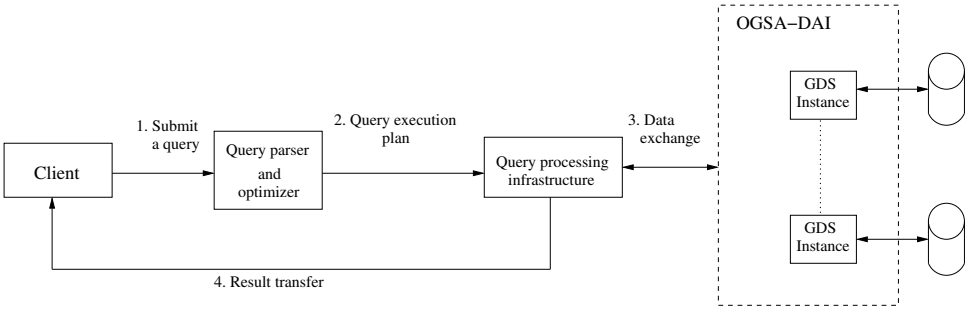

FIG. 6.2 – *Interactions during query execution*

Figure 6.2 shows the interactions that take place during query processing. Firstly, the user submits the query to the *query parser* in order to create the distributed query execution plan. The *Grid Distributed Query Service (GDQS)* presented in (Alpdemir et al. 2004) can be used for this purpose. Then, the query execution plan is sent to the grid nodes that are designated by the query optimizer. To execute queries, processing nodes need to access data distributed over the grid. To this end, OGSA-DAI services are used as described earlier. Finally, the query results are sent to the user using the GridFTP service.

We performed our tests on a heterogeneous set of machines of Grid'5000. The characteristics of these machines are shown in table 6.1. In the performance tests, we divided the nodes into 3 clusters. Each cluster has its local coordinator node and all the sites share a common main coordinator node. We compared the performance of GDFA-Join algorithm with that of osfa-join. We recall here that the performance evaluation presented in section 5.3 of chapter 5 shows that osfa-join algorithm outperforms the basic hash join algorithms.

6.3.1 Speed-up test

In this test, we applied osfa-join and GDFA-Join algorithms on two relations formed of 10^7 tuples and 8×10^6 tuples and the join result is formed of 650×10^6 tuples. The zipf skew factor in both relations was set to 0.5. The maximum number of processors used is 85. Figures 6.3 and 6.4 show

that GDFA-Join outperforms osfa-join algorithm. And figures 6.5 and 6.6 show that both algorithms do not suffer from JPS problems.

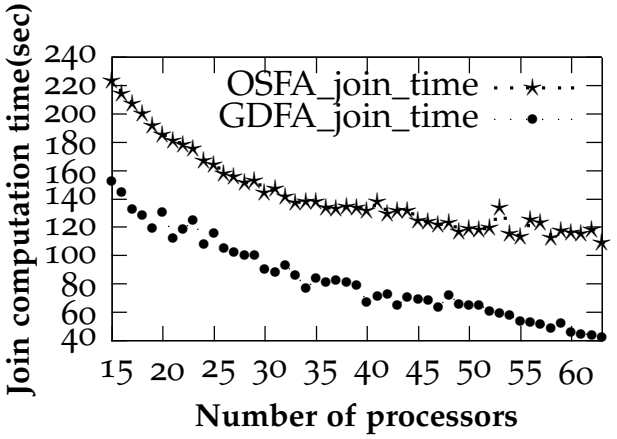

FIG. 6.3 – *GDFA-Join Speed-up test for number of nodes* $\in \{15, \ldots, 63\}$.

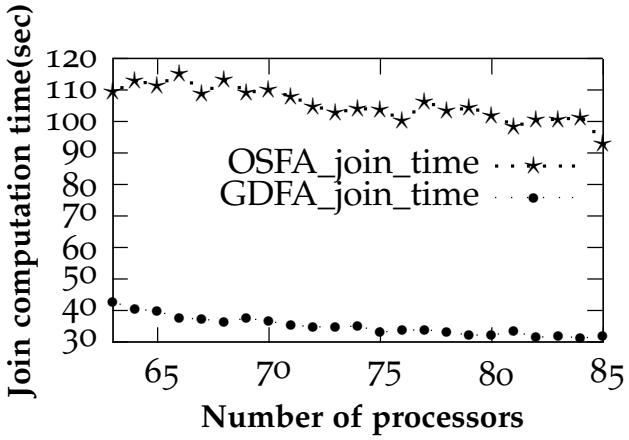

FIG. 6.4 – *GDFA-Join Speed-up test for number of nodes* $\in \{63, \ldots, 85\}$.

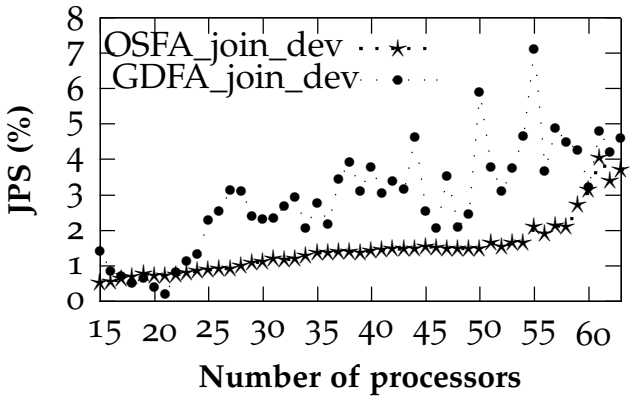

FIG. 6.5 – *GDFA-Join JPS percentage for number of nodes* $\in \{15, \ldots, 63\}$.

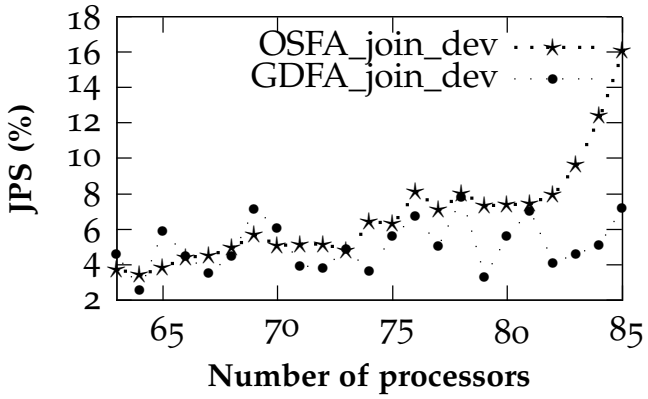

FIG. 6.6 – *GDFA-Join JPS percentage for number of nodes* $\in \{63, \ldots, 85\}$.

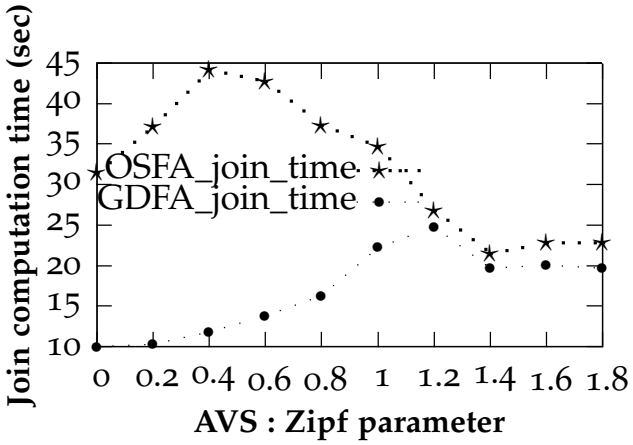

FIG. 6.7 – *The effect of AVS on performance of GDFA-Join.*

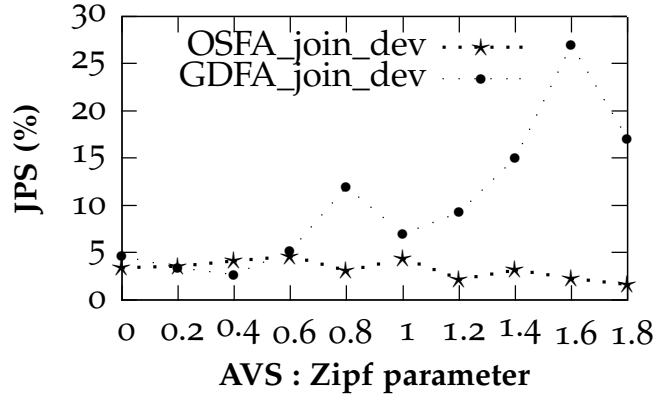

FIG. 6.8 – *The effect of AVS on Join result deviation of GDFA-Join.*

6.3.2 The effect of Attribute Value Skew (AVS) test

The effect of AVS was studied for both algorithms where we fixed the number of processors to 65 and we varied the $zipf$ factor from 0 to 1.8. Figure 6.7 shows that GDFA-Join algorithm outperforms osfa-join algorithm. Figures 6.8 shows that the percentage of JPS in GDFA-Join algorithm is hi-

gher than the JPS percentage in osfa-join algorithm. This is normal since in GDFA-Join algorithm, each node receives a load which is proportional to its processing capacities. This is not the case of osfa-join algorithm where all the nodes receive approximately the same load. In addition, during join computation phase of GDFA-Join algorithm, idle nodes steal data buckets from overloaded ones.

6.4 Conclusion

We have proposed, in this chapter, a new parallel algorithm called GDFA-Join for evaluating join operations on the grid. This algorithm allows us to benefit from data storage and processing capabilities provided by grid architectures where load assigned to each computing processor respects its processing capabilities. The algorithm allows us to optimize the computation time using a load balancing strategy. This strategy balances the load of processing nodes of each cluster, then between the different clusters during all the stages of join computation. In order to guarantee the scalability of our algorithm, we use a hierarchy structure of coordinator nodes. Each coordinator node is responsible of balancing the load of its local processors. And the intermediate nodes balance the load between the clusters. In addition, each processor is in charge of histogram management and communication templates associated to a subset of join attribute values and not necessarily its own data. It is also insensitive to data skew while reducing communication cost since only the tuples that effectively appear in the join result are redistributed across the network.

7 Semi-join Computation on Distributed File Systems Using Map-Reduce-Merge Model

Contents

Semi-join is the most used technique to optimize the treatment of complex relational queries on distributed architectures. However, the overhead related to semi-join computation can be very high due to data skew and to the high cost of communication in distributed architectures. Internet search engines needs to process vast amount of raw data every day. Hence, systems that manage such data should assure scalability, reliability and availability issues with reasonable query processing time. Hadoop and Google's File System are examples of such systems. In this chapter, we present CFA-Semi-Join algorithm: a new Frequency Adaptive algorithm based on Map-Reduce-Merge model and distributed histograms for processing semi-join operations on Cloud systems. A cost analysis of this algorithm shows that our approach is insensitive to data skew while highly reducing communication and disk Input/Output costs.

7.1 Introduction

In this chapter, we are interested in evaluating semi-join operation on Distributed File Systems (DFS). This operation is useful for reducing processing time of queries involving join operations, by means of selecting only relevant data and thereby reducing massively the number of join buckets (and thus join computation time) since these buckets contain only tuples that appear effectively in the join result. Semi-join is also used to reduce the amount of data transferred over the network and therefore the communication costs in distributed architectures (Stocker et al. 2001).

However, parallel semi-join computation is very sensitive to data skew and communication costs. To this end, we propose *CFA-Semi-Join*: a new approach based on distributed histograms [1] and Map-Reduce-Merge (Yang et al. 2007) model for evaluating semi-join operations on Distributed File Systems (DFS). Our approach is insensitive to data skew, owing to the fact that we only redistribute the histogram's entries: the list of distinct couples (v, n_v) where v represents a semi-join attribute value and n_v is the number of tuples having value v for the join attribute. We recall that, on each node, we only have one couple representing each specific value v. Hence, applying hash functions on such couples does not cause load imbalance between processors even in the presence of highly skewed input relations.
A cost analysis shows that our approach induces a low communication cost because we only redistribute histogram's entries (and not input relations data) while guaranteeing scalability and perfect balancing properties during all the stages of semi-join computation.

Our algorithm called *CFA-Semi-Join* (Cloud Frequency Adaptive Semi-Join) is published in (Hassan and Bamha 2010).

7.2 CFA-Semi-Join: A Map-Reduce-Merge based algorithm for computing semi-joins

In this section, we propose a new algorithm called CFA-Semi-Join (Cloud Frequency Adaptive semi-join) based on the Map-Reduce-Merge model presented in Yang et al. (2007) for evaluating semi-join operations on distributed file systems such as Hadoop Distributed File System (HDFS) and GFS. Our approach is insensitive to data skew and is based on an efficient technique allowing to highly reduce communication and disk Input/Output costs.
To compute the semi-join, $R \ltimes S$, of two relations R and S, we assume that the input relations R and S are divided into chunks (splits) of data. These chunks are stored in a DFS. In such systems, each chunk is also replicated on several nodes for reliability issues.

[1] Histograms are implemented as Balanced trees (B^+-trees), and the size of these histograms is very small compared to the sizes of input relations since the histograms contain only the list of distinct join attribute values and their corresponding frequencies.

Throughout this chapter, for a relation $T \in \{R, S\}$, we use the following notations:

- $|T|$: number of pages forming T,
- $||T||$: number of tuples in T,
- T_i: the split(s) of relation T placed on processor i,
- $(BR_i)_{map_j}$: hashed bucket of index i related to R splits placed on mapper j,
- $(BS_i)_{map_j}$: hashed bucket of index i related to S splits placed on mapper j,
- $LHist(BR_i)_{map_j}$: local histogram of $(BR_i)_{map_j}$, i.e. the list of pairs (v, n_v) where v is a semi-join value and n_v its corresponding frequency in relation BR_i on mapper map_j,
- $LHist(BS_i)_{map_j}$: local histogram of $(BS_i)_{map_j}$,
- $GHist(BR_i)$ (resp. $GHist(BS_i)$): global histogram of buckets BR_i (resp. BS_i),
- $Hist(BR_i \bowtie BS_i)$: histogram related to join attribute values that appear in both relations partitions BR_i and BS_i,
- $\overline{LHist}(BR_i)_{map_j}$: the restriction of $LHist(BR_i)_{map_j}$ to join attribute values that appear in the semi-join result,
- $\overline{GHist}(R \bowtie S)$: join attribute values that appear in both R and S,
- $c^i_{r/w}$: read/write cost of a page of data on local disk on processor i,
- t^i_r: time to read a record from main memory of processor i,
- t^i_w: time to write a record to main memory of processor i,
- t^i_s: time of a simple search in a B$^+$-tree on processor i,
- t^i_h: time to add an entry to a B$^+$-tree on processor i,
- m_p: communication message protocol cost per page of data,
- m_l: communication message latency for one page of data,
- v: semi-join attribute value,
- n_v: number of tuples having value v for the semi-join attribute,
- σ: semi-join selectivity factor,
- $NB\_mappers$: number of mapper nodes of each relation,
- $NB\_reducers$: number of reducer nodes of each relation,
- $NB\_mergers$: number of reducer nodes.

Our algorithm can be seen as a multi-pass hierarchical workflow of several map, reduce and merge functions as shown in figure 7.1. It can be divided into the following four phases. We will give for each phase an upper bound of execution time.

1. Map phase:

We have two groups of mappers, the first group is responsible of treating R partitions and the second is responsible of S ones. Each mapper in R (resp. S) group is assigned one or several file splits of R (resp. S). Each mapper reads its assigned splits from the DFS. Tuples of these splits are partitioned into buckets using a hash function applied on the value of the join attribute (Algorithm 17). On the fly, we also compute the frequency n_v of each value of the join attribute v in each bucket. We have chosen to use Balanced B$^+$-trees [2] to store the couples (v, n_v). We refer to these

[2] A balanced tree (B$^+$-tree) is a data structure that maintains an ordered set of data to allow efficient search and insert operations.

B^+-trees as *histograms*. At the end, we have on each mapper a list of local histograms. The partitions of local histograms on all R-mappers (resp. S-mappers) having the same index are designated to the same R-reducer (resp. S-reducer).

Algorithm 16: Semi-join algorithm workflow.

1► Each R-mapper (resp. S-mapper):
  ▷ reads its assigned data splits from the DFS;
  ▷ partitions the read data into buckets and, on the fly, creates the local histogram of each bucket (Algo. 17);
2► Each R-reducer/merger node (resp. S-reducer node) (Algo. 18):
  ▷ reads remotely the local histogram partitions holding its index;
  ▷ creates the distributed global histograms of R (resp. S);
3► Each R-reducer/merger node (Algo. 19):
  ▷ reads remotely the distributed global histograms of S;
  ▷ finds the semi-join attribute values that appear in the final result by finding the intersection of distributed global histograms of R and S;
  ▷ creates the restriction of local histogram received in (3) to values appearing in the final semi-join result;
4► Each R-mapper node:
  ▷ reads remotely the associated local histogram restriction created in (7);
  ▷ merges the read data with the associated input data buckets to find the final semi-join result (Algo. 20);

Algorithm 17: Map function for computing local histogram for a relation $T \in \{R, S\}$.

```
map(String table, const String key) {
 /* table: relation's name, key: join attribute.*/
int hash_function(String);
 ▷ vector < B+-tree > LHist;
 ▷ for each tuple t in the assigned splits of table {
    ▷ hash t into the bucket BT_i where i = hash_function(t.v);
    ▷ if ( join attribute value t.v is already inserted in LHist(BT_i)) then
        increment the frequency relative to t.v in LHist(BT_i);
    ▷ else
        add the couple (t.v, 1) to LHist(BT_i);
      endif
    }
}
```

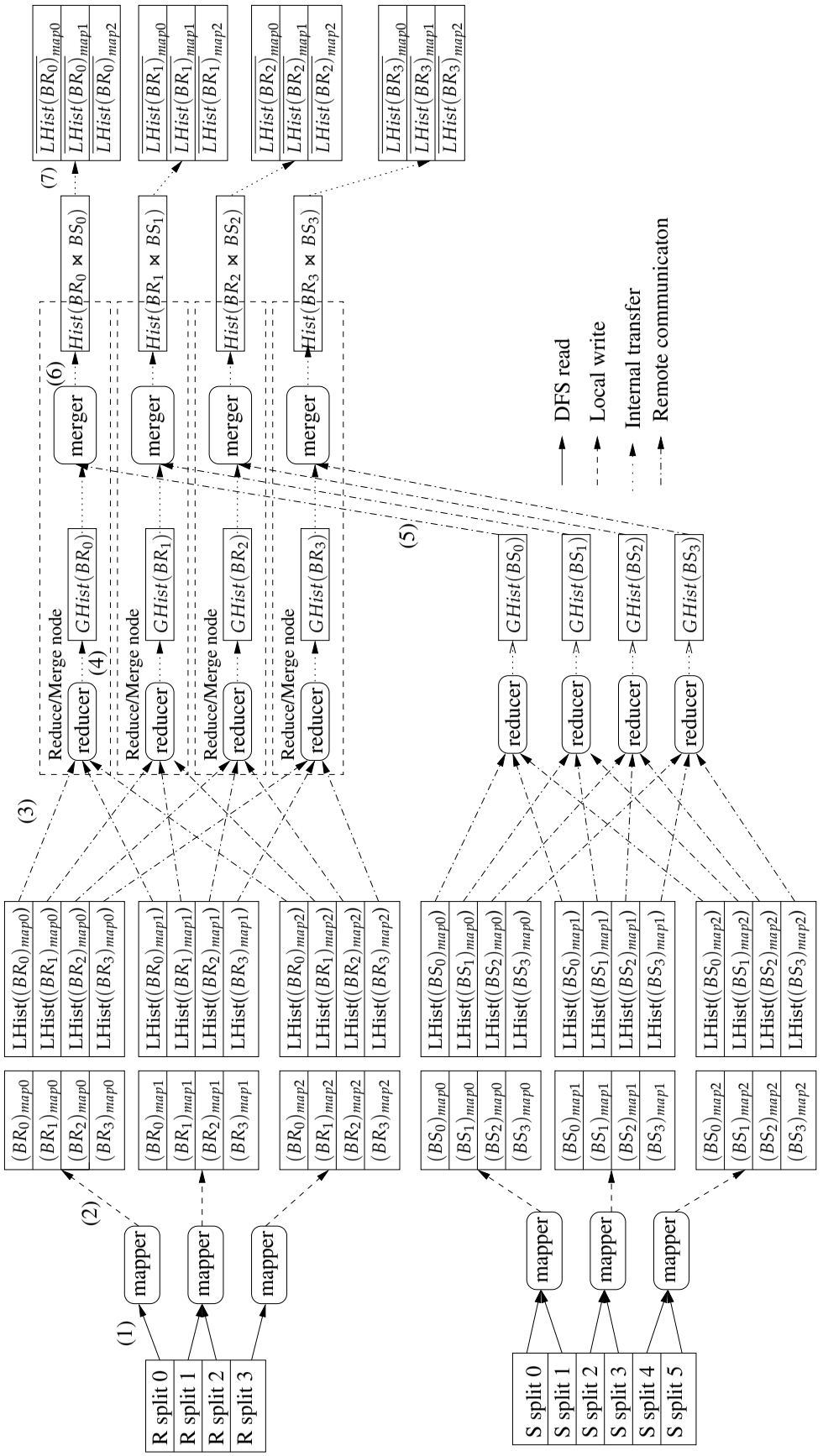

FIG. 7.1 – *Semi-join computation steps in CFA-Semi-Join algorithm.*

The cost of this phase is of the order:

$$Time_{phase1} = O\Big(\max\Big(\max_{i=1}^{NB\_mappers} \big(|R_i| \times c^i_{r/w} + ||R_i|| \times (t^i_r + t^i_w + t^i_h)\big), \max_{i=1}^{NB\_mappers} \big(|S_i| \times c^i_{r/w} + ||S_i|| \times (t^i_r + t^i_w + t^i_h)\big)\Big)\Big).$$

where $|R_i| \times c^i_{r/w}$ (resp. $|S_i| \times c^i_{r/w}$) represents the cost of loading data from the local disk(s) on processor i and $||R_i|| \times (t^i_r + t^i_w)$ (resp. $||S_i|| \times (t^i_r + t^i_w)$) is the cost of hashing tuples into buckets. The histograms are created, on the fly, during the creation of R and S data buckets using Algorithm 17 with cost $||R_i|| \times t^i_h$ and $||S_i|| \times t^i_h$.

2. Reduce phase:

Each R-reducer (resp. S-reducer) reads its designated distributed local histogram partition from each R-mapper (resp. S-mapper). Applying the same hash function on all the mappers while partitioning (v, n_v) allows us to forward all entries having same values of v to partitions of the local histograms holding the same index on all the mappers. Thus, the associated records of local histograms on all mappers are sent to the same reducer. Each reducer computes the sum of the frequencies n_v related to each join attribute value v in order to find its global frequency. We call this structure the distributed global histogram.

Algorithm 18: Reduce function for computing distributed global histogram for a relation $T \in \{R, S\}$.

```
reduce(int reducer_id, vector <B+-tree> LHist(BT_reducer_id)){
  ▷ B+-tree GHist(BT_reducer_id);
  ▷ for each mapper map_i {
     ▷ read remotely LHist(BT_reducer_id)_map_i;
     ▷ for each couple (v, n_v) of LHist(BT_reducer_id)_map_i {
        ▷ if ( v already exists in GHist(BT_reducer_id) with frequency freq_v ) then
                update freq_v += n_v;
        ▷ else
                insert (v, n_v) into GHist(BT_reducer_id);
          endif
     }
  }
}
```

In order to give the cost of Algorithm 18, we study it from both mappers and reducers sides.

a. Local histogram distribution from the mappers side:

The cost of forwarding the local histograms of R from R-mapper i to R-reducers is of the order:

$$O\Big(\max_{i=1}^{NB\_mappers} \sum_{j=0}^{NB\_reducers} |LHist\big((BR_j)_{map_i}\big)| \times (m_p + m_l)\Big).$$

We give an upper bound to this cost using the following four cases:

a.1. Each semi-join attribute value appears in R with a frequency equal to 1:

In this case, we have $||GHist(R)|| = ||R||$. So, if tuples of R are evenly partitioned over the mappers, on each mapper map_i we have:

$$\sum_{j=1}^{NB\_reducers} ||LHist\big((BR_j)_{map_i}\big)|| = \frac{||GHist(R)||}{NB\_mappers} = \frac{||R||}{NB\_mappers}.$$

So, the above forward cost can be written as:

$$O\big(\frac{|GHist(R)|}{NB\_mappers} \times (m_p + m_l)\big) \leq O\big(\frac{|R|}{NB\_mappers} \times (m_p + m_l)\big).$$

The above inequality holds due to the fact that $|GHist(R)| \leq |R|$ since histograms are only formed of couples (v, n_v) where v is the semi-join attribute value and n_v is its corresponding frequency. Thus, the cost to forward relation's R local histograms is of the order:

$$O\Big(\min\big(\frac{|GHist(R)|}{NB\_mappers}, \frac{|R|}{NB\_mappers}\big) \times (m_p + m_l)\Big) \leq O\Big(\min\big(|GHist(R)|, \frac{|R|}{NB\_mappers}\big) \times (m_p + m_l)\Big).$$

a.2. All the semi-join attribute values appear in R with frequencies less than $NB\_mappers$:

In this case, we can partition R into $NB\_mappers$ relations $R^1, R^2, \ldots, R^{NB\_mappers}$ such that the frequency of each semi-join attribute value v in R^j is exactly equal to 1. So, as proved in case a.1, the cost of distributing the local histograms of each partition R^j is of the order:

$$O\Big(\min\big(\frac{|GHist(R^j)|}{NB\_mappers}, \frac{|R^j|}{NB\_mappers}\big) \times (m_p + m_l)\Big).$$

Thus, the cost of forwarding the local histogram partitions of R_i on each mapper i is of order:

$$O\Big(\sum_{j=1}^{NB\_mappers} \min\big(\frac{|GHist(R^j)|}{NB\_mappers}, \frac{|R^j|}{NB\_mappers}\big) \times (m_p + m_l)\Big) \leq O\Big(\min\big(|GHist(R)|, \frac{|R|}{NB\_mappers}\big) \times (m_p + m_l)\Big).$$

This inequality holds owing to the fact that:

$$\begin{cases} \max_j |GHist(R^j)| & \leq & |GHist(R)| \quad \text{and} \\ \sum_i \min(a_i, b_i) & \leq & \min(\sum_i a_i, \sum_i b_i). \end{cases}$$

Remark 7 *In practice, this partitioning step is carried out to only show the validity of the above cost. And whenever a semi-join value appears with a frequency k, a couple (v, k) is sent to its destination processor in one message and not k messages of the $(v, 1)$. This reduces massively the number of messages exchanged over the network.*

a.3. All the semi-join attribute values appear in R with frequencies greater than $NB\_mappers$:

In this case, each mapper i is at most in charge of $||GHist(R)||$ values of semi-join attribute (i.e. all the semi-join attribute values of R appear in R_i), and thus the cost of this step is of the order:

$$O\big(|GHist(R)| \times (m_p + m_l)\big) \leq O\big(\frac{|R|}{NB\_mappers} \times (m_p + m_l)\big).$$

This inequality holds because each semi-join attribute value appears with a frequency greater than $NB\_mappers$ and thus:

$$||GHist(R)|| \leq \frac{||R||}{NB\_mappers}$$

and

$$|GHist(R)| \leq \frac{|R|}{NB\_mappers}.$$

So, the forward cost is of the order:

$$O\Big(\min\big(|GHist(R)|, \frac{|R|}{NB\_mappers}\big) \times (m_p + m_l)\Big).$$

a.4. General case:

The cost of this case can be deduced from cases b and c, where we can divide R into two sub-relations R' and R''. R' holds tuples related to semi-join attribute values with frequencies less than $NB\_mappers$ and R'' those whose frequencies are greater than $NB\_mappers$. So, the cost of forwarding the local histograms of R' is:

$$O\Big(\min\big(|GHist(R')|, \frac{|R'|}{NB\_mappers}\big) \times (m_p + m_l)\Big),$$

and of R'':

$$O\Big(\min\big(|GHist(R'')|, \frac{|R''|}{NB\_mappers}\big) \times (m_p + m_l)\Big).$$

The cost of forwarding R is thus the sum of the above two costs, which is at most of the order:

$$O\Big(\min\big(|GHist(R)|, \frac{|R|}{NB\_mappers}\big) \times (m_p + m_l)\Big).$$

This is due to the fact that:

$$\begin{cases} GHist(R') \cap GHist(R'') & = \emptyset \quad \text{and} \\ \min(a,b) + \min(c,d) & \leq \min(a+c, b+d). \end{cases}$$

So, the cost of relation's R local histogram forward step is at most:

$$O\Big(\min\big(|GHist(R)|, \frac{|R|}{NB\_mappers}\big) \times (m_p + m_l)\Big).$$

b. Receiving local histograms from the reducers side:
The cost of receiving the local histograms by each reducer j is of the order:

$$O\Big(\max_{j=1}^{NB\_reducers} \sum_{i=1}^{NB\_mappers} |LHist\big((BR_j)_{map_i}\big)| \times (m_p)\Big).$$

We give an upper bound to this cost and prove it using the following four cases:

b.1. Each semi-join attribute value appears in R with a frequency equal to 1:

Using an appropriate hash function allowing to evenly partition the distinct semi-join attribute values, we can send to each reducer approximately $\frac{||R||}{NB\_reducers}$ records. Since $||GHist(R)|| = ||R||$ and $|GHist(R)| \leq |R|$, the receiving cost of the local histogram partitions by each reducer i is of the order $\frac{|GHist(R)|}{NB\_reducers} \times m_p$. In addition, in order to create the distributed global histogram, each reducer i merges the received records of local histogram partitions with a cost of the order: $\frac{||GHist(R)||}{NB\_reducers} \times (t_r^i + t_h^i)$.
Hence, the global cost of this step is at most:

$$\begin{aligned} O\big(\frac{|GHist(R)|}{NB\_reducers} \times m_p + \frac{||GHist(R)||}{NB\_reducers} \times (t_r^i + t_h^i)\big) \\ \leq O\big(\frac{|R|}{NB\_reducers} \times m_p + \frac{||R||}{NB\_reducers} \times (t_r^i + t_h^i)\big). \end{aligned}$$

So, the cost on each reducer is at most:

$$\begin{aligned} O(\min\big(&|GHist(R)| \times m_p + ||GHist(R)|| \times (t_r^i + t_h^i), \\ &\frac{|R|}{NB\_reducers} \times m_p + \frac{||R||}{NB\_reducers} \times (t_r^i + t_h^i)\big). \end{aligned}$$

b.2. All the semi-join attribute values appear in R with frequencies less than $NB\_reducers$:

Following the same demonstration method used for the mappers, we divide R into $NB\_reducers$ relations: $R^1, R^2, \ldots, R^{NB\_reducers}$ such that the frequency of each semi-join value v is exactly equal to 1. Thus, the receiving and local histograms' records merging costs on each reducer related to each partition R^i is at most:

$$\begin{aligned} O\Big(\min\big(&\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(R^i)| \times m_p + ||GHist(R^i)|| \times (t_r^i + t_h^i)\big), \\ &\frac{NB\_mappers}{NB\_reducers^2} \times \big(|R^i| \times m_p + ||R^i|| \times (t_r^i + t_h^i)\big)\big)\Big). \end{aligned}$$

So, the total cost on each reducer is:

$$\begin{aligned} O\Big(\sum_{i=1}^{NB\_reducers} &\min\big(\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(R^i)| \times m_p + ||GHist(R^i)|| \times (t_r^i + t_h^i)\big), \\ &\frac{NB\_mappers}{NB\_reducers^2} \times \big(|R^i| \times m_p + ||R^i|| \times (t_r^i + t_h^i)\big)\big)\Big) \\ \leq O\Big(\min\big(&\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(R)| \times m_p + ||GHist(R)|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big), \\ &\frac{NB\_mappers}{NB\_reducers^2} \times \big(|R| \times m_p + ||R|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big)\big)\Big). \end{aligned}$$

b.3. All the semi-join attribute values appear in R with frequencies greater than $NB\_reducers$:

Owing to the fact that hashing histograms is insensitive to data skew, by using an appropriate hash function (remark 1 on page 33), each reducer will be responsible of approximately $\frac{||GHist(R)||}{NB\_reducers}$ semi-join attribute values. So, if each semi-join attribute value appears on each R-mapper, then each reducer receives at most: $NB\_mappers \times \frac{||GHist(R)||}{NB\_reducers}$ local histogram records. Thus, the communication and records merging cost on each reducer is of the order:

$$O\Big(\frac{NB\_mappers}{NB\_reducers}\big(|GHist(R)| \times m_p + ||GHist(R)|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big)\Big),$$

Since each semi-join attribute value appears with a frequency greater than $NB\_reducers$, we have:

$$\begin{cases} |GHist(R)| & \leq & \frac{|R|}{NB\_reducers} & \text{and} \\ ||GHist(R)|| & \leq & \frac{||R||}{NB\_reducers}. \end{cases}$$

And thus,

$$\begin{cases} \frac{NB\_mappers}{NB\_reducers}|GHist(R)| & \leq & \frac{NB\_mappers}{NB\_reducers^2} \times |R| & \text{and} \\ \frac{NB\_mappers}{NB\_reducers}||GHist(R)|| & \leq & \frac{NB\_mappers}{NB\_reducers^2} \times ||R||. \end{cases}$$

So, the communication and records merging cost in this case is at most:

$$\begin{aligned} O(\min\Big(&\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(R)| \times m_p + ||GHist(R)|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big), \\ &\frac{NB\_mappers}{NB\_reducers^2} \times \big(|R| \times m_p + ||R|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big)\Big). \end{aligned}$$

b.4. General case:
The cost for the general case is of the order:

$$\begin{aligned} O(\min\Big(&\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(R)| \times m_p + ||GHist(R)|| \times (t_r^i + t_h^i)\big), \\ &\frac{NB\_mappers}{NB\_reducers^2} \times \big(|R| \times m_p + ||R|| \times (t_r^i + t_h^i)\big)\Big). \end{aligned}$$

The prove is straightforward from cases *b* and *c* where we can follow the same steps of case *a.4*.

So, the total cost of this phase is at most:

$$Time_{phase2} =$$

$$O\Bigg(\max \Bigg(\min\big(|GHist(R)|, \frac{|R|}{NB\_mappers}\big) \times (m_p + m_l) + \min\Big(\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(R)| \times m_p + ||GHist(R)|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big), \frac{NB\_mappers}{NB\_reducers^2} \times \big(|R| \times m_p + ||R|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big)\Big), \min\big(|GHist(S)|, \frac{|S|}{NB\_mappers}\big) \times (m_p + m_l) + \min\Big(\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(S)| \times m_p + ||GHist(S)|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big), \frac{NB\_mappers}{NB\_reducers^2} \times \big(|S| \times m_p + ||S|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big)\Big)\Bigg)\Bigg).$$

3. Distributed global histogram merge phase:
At this step, each merger reads a partition of the distributed global histogram related to R and its associated partition of S. After that, the intersection of these two partitions, $Hist(R \bowtie S)$, is computed as shown in Algorithm 19. $Hist(R \bowtie S)$ holds only the join attribute values that appear in R and S. Now, we need to determine the values of the semi-join attribute that appear in $R \ltimes S$. To this end, each merger reads from each R-mapper the associated partition of the local histogram and computes its intersection with $Hist(R \bowtie S)$ as shown in Algorithm 19.

Remark 8 *In order to optimize the query execution, we can combine each merger with one R-reducer. This helps to decrease communication costs because the partitions of the local histograms of R are already read by the R-reducers. In addition, while computing $Hist(R \bowtie S)$, we read locally the global histogram of R and remotely those of S from S-reducers.*

The global cost of this phase is at most:
$Time_{phase3} =$

$$\begin{aligned} & O\Big(\frac{|GHist(S)|}{NB\_reducers} \times (2 \times m_p + m_l) + \max_{i=1}^{NB\_mergers} \frac{t_r^i + t_h^i + t_s^i}{NB\_reducers} \times \min(||GHist(R)||, ||GHist(S)||) + \max_{i=1}^{NB\_mergers} \sum_{j=1}^{NB\_mappers} ||LHist((BR_i)_{map_j})|| \times \max_{i=1}^{NB\_mergers}(t_r^i + t_s^i + t_h^i)\Big) \\ \leq\ & O\Big(\frac{|GHist(S)|}{NB\_reducers} \times (2 \times m_p + m_l) + \frac{t_r^i + t_h^i + t_s^i}{NB\_reducers} \times \min(||GHist(R)||, ||GHist(S)||) + \frac{NB\_mappers}{NB\_reducers} \times \min(||GHist(R)||, \frac{||R||}{NB\_reducers}) \times \max_{i=1}^{NB\_mergers}(t_r^i + t_s^i + t_h^i)\Big). \end{aligned}$$

The term $\frac{|GHist(S)|}{NB\_reducers} \times (2m_p + m_l)$ is the cost of forwarding and receiving the global histogram partitions from S-reducers to the mergers and

Algorithm 19: Global histogram merge function.

```
merge(int m_id, B+-tree GHist(BR_{m_id}), B+-tree GHist(BS_{m_id})) {
 /* m_id is the identifier of the merger. */
 /* Find Hist(R ⋉ S) which is the intersection of hist_R and hist_S. */
 ▷ vector <B+-tree> LHist(BR_{m_id}) ;
 ▷ B+-tree Hist(BR_{m_id} ⋉ BS_{m_id}) ;
 ▷ read locally GHist(BR_{m_id}) ;
 ▷ read remotely GHist(BS_{m_id}) from the associated S-mapper ;
 ▷ for each couple (v, n_R(v)) in GHist(BR_{m_id}) {
    ▷ if ( v ∈ GHist(BS_{m_id}) with frequency n_S(v) ) then
       Insert (v, n_R(v) × n_S(v)) into Hist(BR_{m_id} ⋉ BS_{m_id}) ;
  }
 /* Find the join attribute values in each Split of R that appear in
    the join result. */
 ▷ for each mapper map_i {
    ▷ for each join attribute value v in LHist(BR_{merger_id})_{map_i}
       ▷ if ( v ∈ Hist(BR_{m_id} ⋉ BS_{m_id}) ) then
          add v to LHist(BR_{merger_id})_{map_i} ;
         endif
  }
}
```

$\min(\frac{||GHist(R)||}{NB\_reducers}, \frac{||GHist(S)||}{NB\_reducers}) \times (t_r^i + t_h^i + t_s^i)$ is the cost of creating the intersection of the global histograms of R and S.
The term $\max_{i=0}^{NB\_reducers} \sum_{j=0}^{n_{R-maps}} ||LHist((BR_i)_{map_j})|| \times (t_r^i + t_s^i + t_h^i)$ is the cost of creating $\overline{LHist}(BR_i)_{map_j}$ on reducer i which holds the semi-join attribute values that appear in the semi-join result of each mapper map_i.

4. Semi-join merge phase:
At this step, we use a group of mergers where each one computes the intersection of a set of hashed buckets of R created by the mappers in the first phase with the associated list of semi-join values that appear in the final semi-join result created in phase 3 (Algorithm 20). The union of the results represents $R \ltimes S$.

Remark 9 *In order to reduce communication costs, this merge phase may be executed on R-mappers of the first phase which already store the buckets of R in their local disks.*

The cost of forwarding the fragments of $\overline{LHist}(BR_i)_{map_j}$ from the mergers to the mappers is:

$$\sum_{j=1}^{NB\_mappers} |\overline{LHist}(BR_i)_{map_j}| \times (m_p + m_l).$$

The cost of receiving these fragments is:

$$\sum_{i=1}^{NB\_mergers} |\overline{LHist}(BR_i)_{map_j}| \times m_p.$$

The cost of computing the final semi-join result on mapper j is:

Algorithm 20: semi-join merge function.

```
merge(int merger_id, vector <data_bucket> (BR[])_merger_id,
                vector <B+-tree> LHist(BR[])_merger_id ) {
  ▷ for each bucket (BR_i)_merger_id on merger merger_id{
    ▷ for each tuple t in (BR_i)_merger_id
        ▷ if ( t.v ∈ LHist(BR_i)_mapper_id ) then
                Write t to the semi-join result buffer;
          endif
    }
}
```

$||R_j|| \times (t_r^j + \sigma \times t_w^j + t_s^j)$ and the cost of writing this result is: $\sigma \times |R_j| \times c_{r/w}^j$.

The global cost of this phase is at most:

$$
\begin{aligned}
Time_{phase4} = O\Big(& \max_{j=1}^{NB\_mappers} \sigma \times |R_j| \times c_{r/w}^j + \max_{j=1}^{NB\_mappers} ||R_j|| \times (t_r^j + \sigma \times t_w^j + t_s^j) + \\
& \max_{i=1}^{NB\_mergers} \sum_{j=1}^{NB\_mappers} |\overline{LHist}(BR_i)_{map_j}| \times (m_p + m_l) + \\
& \max_{j=1}^{NB\_mappers} \sum_{i=1}^{NB\_mergers} |\overline{LHist}(BR_i)_{map_j}| \times m_p \Big) \\
\leq O\Big(& \max_{j=1}^{NB\_mappers} \big(||R_j|| \times (t_r^j + \sigma \times t_w^j + t_s^j) + \sigma \times |R_j| \times c_{r/w}^j\big) + \\
& \frac{NB\_mappers}{NB\_reducers} \times |\overline{GHist}(R \bowtie S)| \times (m_p + m_l) + \\
& \max_i \sigma \times |\overline{R}_i| \times m_p \Big).
\end{aligned}
$$

So, the total cost of this algorithm is the sum of the costs of all phases:
$Time_{CFA-Semi-Join} =$

$$
\begin{aligned}
O\Bigg(& \max\Big(\max_{i=1}^{NB\_mappers} \big(|R_i| \times c_{r/w}^i + ||R_i|| \times (t_r^i + t_w^i + t_h^i)\big), \\
& \quad \max_{i=1}^{NB\_mappers} \big(|S_i| \times c_{r/w}^i + ||S_i|| \times (t_r^i + t_w^i + t_h^i)\big)\Big) + \\
& \max\Big(\min\big(|GHist(R)|, \frac{|R|}{NB\_mappers}\big) \times (m_p + m_l) + \\
& \min\Big(\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(R)| \times m_p + ||GHist(R)|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big), \\
& \quad \frac{NB\_mappers}{NB\_reducers^2} \times \big(|R| \times m_p + ||R|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big)\Big), \\
& \min\big(|GHist(S)|, \frac{|S|}{NB\_mappers}\big) \times (m_p + m_l) + \\
& \min\Big(\frac{NB\_mappers}{NB\_reducers} \times \big(|GHist(S)| \times m_p + ||GHist(S)|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big), \\
& \quad \frac{NB\_mappers}{NB\_reducers^2} \times \big(|S| \times m_p + ||S|| \times \max_{i=1}^{NB\_reducers}(t_r^i + t_h^i)\big)\Big)\Big) +
\end{aligned}
$$

$$\max_{i=1}^{NB\_mergers} \frac{t_r^i + t_h^i + t_s^i}{NB\_reducers} \times \max(||GHist(R)||, ||GHist(S)||) +$$
$$\frac{NB\_mappers}{NB\_reducers} \times \min(||GHist(R)||, \frac{||R||}{NB\_reducers}) \times \max_{i=1}^{NB\_mergers}(t_r^i + t_s^i + t_h^i) +$$
$$\frac{NB\_mappers}{NB\_reducers} \times |\overline{GHist}(R \ltimes S)| \times (m_p + m_l) +$$
$$\max_{j=1}^{NB\_mappers} \left(||R_j|| \times (t_r^j + \sigma \times t_w^j + t_s^j) + \sigma \times |R_j| \times c_{r/w}^j \right).$$

Remark 10 *Using $NB\_mappers$ mappers and $NB\_reducers$ reducers, the computation of $R \ltimes S$ requires at least the following cost*:

$$O(\max_{i=1}^{NB\_mappers} (c_{r/w}^i \times (|R_i| + |S_i| + \sigma \times |R_i|) + ||R_i|| + ||S_i|| +$$
$$(m_p + m_l) \times (|R_i| + |S_i|)) + \tag{7.1}$$
$$m_p \times \frac{|R| + |S|}{NB\_reducers} + \max_{i=1}^{NB\_reducers} \frac{(t_r^i + t_s^i) \times ||R|| + t_h^i \times ||S||}{NB\_reducers}),$$

where $c_{r/w}^i \times (|R_i| + |S_i|)$ is the cost of reading input relations from disk and $c_{r/w} \times \sigma \times |R_i|$ is the cost of writing the semi-join result on the disk of processor i. The term $||R_i|| + ||S_i||$ represents the cost of partitioning the tuples into buckets that will be sent to the reducers. The term

$$(m_p + m_l) \times (|R_i| + |S_i|) + m_p \times \frac{|R| + |S|}{NB\_reducers}$$

represents the cost of communicating data between mappers and reducers. And $\frac{(t_r^i + t_s^i) \times ||R|| + t_h^i \times ||S||}{NB\_reducers}$ is the cost of computing the semi-join result. Our algorithm has optimal asymptotic complexity when the number of reducers and the number of mappers are of the same order since all the terms of the global cost are bounded by those of equation 7.1.

In the contrary to algorithms based on hashing such as those presented in Yang et al. (2007), our approach is insensitive to data skew. This is due to the fact that we apply the hash function on the distributed histograms which hold a single entry for each specific semi-join attribute value and not directly on input relations data.
To compute semi-joins, we don't need to compute the frequencies of the semi-join attribute. Creating balanced tree which only contains the semi-join attribute values is sufficient. However, we have chosen to follow this strategy because it allows us to create efficient communication templates for join queries. These communication templates allow us to create balanced join buckets even in the presence of highly skewed data.

7.3 Conclusion

Map-Reduce based systems offer an easy programming model that automatically manages parallel issues such as load balance and fault tolerance, etc. Join and semi-join processing on such systems is based on hashing all data of input relations which is known to be inefficient in the presence

of data skew. In this chapter, we have introduced a semi-join algorithm based on a Map-Reduce-Merge model and distributed histograms allowing to highly reduce the communication costs owing to the fact that only histograms are redistributed across the network and the sizes of these histograms are very small compared to the sizes of input relations. Distributed histograms computation allows us to avoid unnecessary disk Input/Output and data redistribution while guaranteeing perfect load balancing properties during all the stages of semi-join computation even in the presence of highly skewed data. Each processor is in charge of a subset of semi-join attribute values and not necessarily its own data.
Our experience with distributed histograms and join operation shows that the overhead related to histogram management remains negligible compared to the gain it provides in reducing communication and join computation costs.

Conclusion

8

Ce travail a porté sur le traitement des opérations de base de données telles que les requêtes de regroupement et de jointure sur des architectures distribuées. Dans ce cadre nous avons proposé une variété d'algorithmes basés sur une approche complètement distribuée où chaque processeur s'occupe d'un sous-ensemble des valeurs de l'attribut de jointure et non de ses propres données : À aucune étape du traitement, les données ne se trouvent centralisées sur un seul processeur, ce qui garantit l'extensibilité de ces algorithmes.
Pour équilibrer les charges des différents processeurs, nous nous sommes basés sur l'utilisation des histogrammes *distribués* pour la redistribution des données et la génération des schémas de communications. Le sur-coût lié à l'utilisation de ces histogrammes reste négligeable du fait que les histogrammes distribués sont créés à la volée durant la phase de création des buckets de données (durant l'étape de la phase de redistribution des données) conjointement par tous les processeurs. Ces histogrammes nous aident à avoir une information très détaillée sur la répartition des valeurs de l'attribut de jointure qui participent effectivement dans le résultat de la jointure avec leurs fréquences. Puis, seuls les tuples associés à ces valeurs sont redistribués. Ceci permet, d'une part, de réduire considérablement le nombre de buckets de jointures (et par conséquent le temps global de traitement), et d'autre part, de minimiser les coûts de communication et de lecture/écriture sur les disques lors de la phase de redistribution des données. Les fréquences calculées sont utilisées, lors de la création des schémas de communications, pour attribuer les charges ou les tâches aux différents processeurs tout en respectant leurs puissances de calcul même en présence d'un fort déséquilibre des données. Les schémas de communications créés permettent également d'éviter le déséquilibre du résultat de la jointure. Pour garantir l'extensibilité de nos algorithmes, la tâche de création des schémas de communications est effectuée conjointement par tous les nœuds : chaque nœud s'occupe de la redistribution des données d'un sous-ensemble de valeurs de l'attribut de jointure et non de ses *propres* données. Nous rappelons que, dans le cas des algorithmes conventionnels, l'extensibilité reste limitée puisque la tâche de création des schémas de communications est effectuée par un nœud coordinateur qui s'occupe de l'allocation des buckets de données aux différents nœuds. En d'autres termes, pendant ce temps, tous les autres nœuds sont inactifs.

Dans la première partie de ce travail de thèse, nous avons développé deux algorithmes, GAJFA-Join (Group-by After Join Frequency Adaptive

Join) et GBJFA-Join (Group-by Before Join Frequency Adaptive Join) pour l'évaluation des requêtes de "Group-By Join" dans un environnement *Shared Nothing*. Contrairement aux algorithmes présentés dans la littérature pour l'évaluation de ces requêtes, nos algorithmes évitent de matérialiser le résultat intermédiaire de la jointure tout en traitant de manière très efficace les problèmes de déséquilibre des valeurs de l'attribut de jointure. Dans ces deux algorithmes, on applique partialement l'opération de regroupement ainsi que la fonction d'agrégat avant la jointure même quand les attributs de jointures et du group-by sont distincts. Ceci réduit considérablement le volume des résultats intermédiaires et permet de minimiser le coût de communication entre les nœuds du système.

Dans la deuxième partie de cette thèse, nous avons présenté deux algorithmes, DFA-Join (Dynamic Frequency Adaptive Join) et PDFA-Join (Pipelined Dynamic Frequency Adaptive Join), pour évaluer respectivement la jointure et la multi-jointure sur les systèmes distribués hétérogènes et multi-utilisateur. Un autre algorithme appelé GDFA-Join (Grid Dynamic Frequency Adaptive Join) a été proposé pour l'évaluation de la jointure sur les grilles. Comme pour les algorithmes GAJFA-Join et GBJFA-Join, dans les trois algorithmes DFA-Join, PDFA-Join et GDFA-Join, nous utilisons les informations détaillées sur la distribution de valeurs de l'attribut de jointure sous forme des histogrammes distribués. Comme expliqué avant, ces histogrammes sont utilisés pour créer les schémas de communications où seuls les tuples qui participent effectivement au résultat de la jointure seront redistribués. Donc, les coûts de communication et de lecture/écriture des données sur les disques sont optimisés même en présence d'un fort déséquilibre des valeurs de l'attribut de jointure.

Sur les architectures hétérogènes, les charges des nœuds peuvent varier de manière rapide et imprévisible. Ainsi, afin d'exploiter efficacement la puissance de ces architectures, nous avons utilisé dans les algorithmes DFA-Join et PDFA-Join une technique de rééquilibrage des charges en deux-étapes : l'une *statique* et l'autre *dynamique*. Dans l'étape statique, chaque nœud reçoit une charge proportionnelle à sa capacité de calcul. Puis, dans l'étape dynamique les buckets de données appartenant aux processeurs surchargés sont transférés vers les processeurs les moins chargés durant l'évaluation de la jointure.

Dans l'algorithme GDFA-Join sur la grille, les tuples sont répartis en buckets de telle sorte que les résultats des jointures locales soient à peu près de même taille. Les schémas pour créer ces buckets sont réalisés par tous les nœuds. Cela aide à garantir l'extensibilité des nos algorithmes. Le nœud coordinateur de chaque cluster est responsable de l'attribution des buckets à ses nœuds en respectant leurs puissances de calcul. Durant, l'évaluation de la jointure, l'équilibrage des charges se fait de manière hiérarchique : les charges sont dans un premier temps équilibrées au sein de chaque cluster (ou groupe des processeurs) puis en suite entre les différents clusters. Les trois algorithmes garantissent que la taille du résultat de la jointure dans chaque nœud est proportionnelle à sa capacité de calcul même en présence d'un fort déséquilibre des données. Dans

les grilles, nous profitons de l'espace de stockage disponible pour la réplication des données dans le but, de réduire l'embouteillage lié aux lectures/écritures sur les disques, d'accélérer le temps d'évaluation de la jointure et de traiter de manière efficace les pannes possibles d'un ou de plusieurs nœuds.

Dans la dernière partie de cette étude, nous avons développé l'algorithme CFA-Semi-Join (Cloud Frequency Adaptive Semi-Join) pour l'évaluation des semi-jointures sur les systèmes de fichiers distribués. Cet algorithme présente l'avantage de réduire fortement les coûts de communication tout en traitant de manière très efficace les problèmes de déséquilibre des données.

L'analyse de complexité de nos algorithmes (GAJFA-Join, GBJFA-Join, DFA-Join, PDFA-Join, GDFA-Join et CFA-Semi-Join) et les résultats expérimentaux obtenus montrent que ces algorithmes possèdent une accélération presque linéaire.

Nous envisageons d'adapter l'algorithme PDFA-Join pour l'évaluation des requêtes complexes sur les grilles. Cet algorithme doit être capable d'exploiter non seulement le parallélisme intra-opérateur, mais également le parallélisme inter-opérateur pipeliné. Il doit également être tolérant aux pannes.

Il serait, aussi, intéressant de travailler sur le traitement de requêtes complexes dans les systèmes de fichiers distribués (DFS). Comme dans le cas de semi-jointures, nous pensons que l'utilisation des histogrammes distribués pour créer des schémas de communications apporte la possibilité de réduire le temps d'exécution d'une requête dans les systèmes de fichiers distribués.

Les travaux présentés, dans cette thèse, peuvent être adaptés à d'autres modèles de données (par exemple données XML distribuées) et également à d'autres applications liées à la grille (découverte de services, gestion de ressources distribuées, etc.). Les approches présentées peuvent être étendues à d'autres applications parallèles telles que l'aide à la décision et le data mining où des gros volumes des données sont distribués entre les différents processeurs.

Récemment, des chercheurs motivés par l'extensibilité du système Pair-à-Pair (P2P), ont commencé d'étudier le traitement des requêtes sur cette architecture. Des approches basées sur les algorithmes de hachage ont été présentées dans (Sattler et al. 2004, Huebsch et al. 2003) pour l'évaluation de la jointure sur les systèmes P2P. Nous pensons qu'adopter nos techniques d'équilibrage de charge à cette architecture serait particulièrement prometteur dans l'optimisation du temps d'exécution des requêtes.

Appendix A

Proposition 1: The cost of creating the global histogram fragments, $Hist_i^x(R)_{i=1,\ldots,p}$, on heterogeneous systems starting from the local histograms $Hist^x(R_i)_{i=1,\ldots,p}$ of the fragment R_i of relation R according to some attribute x is of order [1]:

$$O\Big(\min\Big(\max_{i=1,\ldots,p}\omega_i \times p \times \big(g \times |Hist^x(R)| + \gamma_i \times ||Hist^x(R)||\big), \max_{i=1,\ldots,p}\omega_i \times \big(g \times |R| + \gamma_i \times ||R||\big)\Big) + l\Big)$$

if a hash function $\mathcal{H}$ that distributes the distinct values of the attribute x among the p processors in a manner that respects the processing capacity of each machine is used.

In the above equation, γ_i is the time needed for executing one operation on processor i and ω_i is the fraction of the total volume of data that must be assigned to processor i in a manner that respects its available capacity in order to allow all participating processors to terminate their assigned tasks approximately at the same time where ω_i is given by the following formula:

$$\omega_i = \frac{\frac{1}{\gamma_i}}{\sum_{k=1}^{p}\frac{1}{\gamma_k}}.$$

Remark 11 *On homogeneous systems, γ_i has the same value on all the p processors. Thus, $\omega_i = \frac{1}{p}\ \forall i \in \{1,\ldots,p\}$. So, the cost of creating the local histogram fragments $Hist^x(R_i)_{i=1,\ldots,p}$ becomes*:

$$O\Big(\min\Big(g \times |Hist^x(R)| + ||Hist^x(R)||, g \times \frac{|R|}{p} + \frac{||R||}{p}\Big) + l\Big). \tag{A.1}$$

Proof: In order to prove this cost, we will consider four cases where the difference between them lies in the frequency distribution of the attribute x of relation R.

Case 1: In this case, we consider that all the values of the attribute x appear in the relation R with a frequency equal to 1, hence $||Hist^x(R)|| = ||R||$. So if we use a hash function $\mathcal{H}$ on each processor i that distributes the tuples of R in a manner that respects the processing power of each machine, then each processor will receive $\omega_i \times ||Hist^x(R)||$ tuples from all the processors in the system. So, the cost of creating $Hist_i^x(R)$ is of the order:

$$O\Big(\max_{i=1,\ldots,p}\omega_i\big(g \times |Hist^x(R)| + \gamma_i \times ||Hist^x(R)||\big) + l\Big) \leqslant O\Big(\max_{i=1,\ldots,p}\omega_i\big(g \times |R| + \gamma_i \times ||R||\big) + l\Big). \tag{A.2}$$

The term $\omega_i \times g \times |Hist^x(R)|$ is the necessary time for communicating data and the term $\omega_i \times \gamma_i \times ||Hist^x(R)||$ is the time needed to form $Hist_i^x(R)_{i=1,\ldots,p}$ starting from the blocks received by each processor i during the distribution of the local histograms $Hist^x(R_i)_{i=1,\ldots,p}$.
The histogram is formed of couples having the form (v, n_v) where v are the distinct values of attribute x of R and n_v their corresponding frequencies, hence $|Hist^x(R)| \leqslant |R|$ which makes that the above inequality holds.
So, for the case where all the values of the attribute x have a frequency equal to 1 in R, the fragments $Hist_i^x(R)$ of the global histogram can be

[1] This is the generalized form of *proposition 1* presented in Bamha and Hains (2005) for computing $Hist_i^x(R)$ in a Shared Nothing architecture.

computed in time of the order:

$$O\Big(\min\Big(\max_{i=1,\dots,p}\omega_i\times\big(g\times|Hist^x(R)|+\gamma_i\times||Hist^x(R)||\big),\\ \max_{i=1,\dots,p}\omega_i\times\big(g\times|R|+\gamma_i\times||R||\big)\Big)+l\Big)$$
$$\leqslant O\Big(\min\Big(\max_{i=1,\dots,p}\omega_i\times p\times\big(g\times|Hist^x(R)|+\gamma_i\times||Hist^x(R)||\big),\\ \max_{i=1,\dots,p}\omega_i\times\big(g\times|R|+\gamma_i\times||R||\big)\Big)+l\Big).$$

Case 2: Now we consider that the frequency of all the values of the attribute x in R is lower than p. In order to use the cost obtained in *case 1*, relation R must be partitioned into p relations $R^1, R^2, \dots, R^p$ such that the frequency of each value v in relation $R^j (j = 1, \dots, p)$ is exactly equal to 1. So, the cost of creating $Hist_i^x(R^j)$ of each relation R^j for $j = 1, \dots, p$ according to *case 1* is of order:

$$O\Big(\min\Big(\max_{i=1,\dots,p}\omega_i\times\big(g\times|Hist^x(R^j)|+\gamma_i\times||Hist^x(R^j)||\big),\\ \max_{i=1,\dots,p}\omega_i\times\big(g\times|R^j|+\gamma_i\times||R^j||\big)\Big)+l\Big), \tag{A.3}$$

hence, the necessary time for creating $Hist_i^x(R)$ of relation R is:

$$O\Big(\sum_{j=1,\dots,p}\min\Big(\max_{i=1,\dots,p}\omega_i\times\big(g\times|Hist^x(R^j)|+\gamma_i\times||Hist^x(R^j)||\big),\\ \max_{i=1,\dots,p}\omega_i\times\big(g\times|R^j|+\gamma_i\times||R^j||\big)\Big)+l\Big)$$
$$\leqslant O\Big(\min\Big(\max_{i=1,\dots,p}\omega_i\times p\times\big(g\times|Hist^x(R)|+\gamma_i\times||Hist^x(R)||\big),\\ \max_{i=1,\dots,p}\omega_i\times\big(g\times|R|+\gamma_i\times||R||\big)\Big)+l\Big). \tag{A.4}$$

The above inequality holds due to the fact that $Hist_i^x(R)$ can be created by simply fusioning the fragments $Hist_i^x(R^j)_{j=1,\dots,p}$ in time of order:

$$O\Big(\sum_{j=1,\dots,p}\max_{i=1,\dots,p}\omega_i\times\gamma_i\times||Hist_i^x(R^j)||\Big)\leqslant O\big(\max_{i,j}\omega_i\times\gamma_i\times||Hist_i^x(R^j)||\times p\big)$$
$$\leqslant O\big(\max_i\omega_i\times\gamma_i\times||Hist^x(R)||\times p\big)$$

and that $\sum_i \min(a_i, b_i) \leqslant \min(\sum_i a_i, \sum_i b_i)$.

In practice, we do not need to partition relation R into p relations where each processor distributes the couples (v, n_v) related to all values v of the join attribute x using a hash function even if $n_v > 1$. However, we used this technique here to demonstrate the validity of the cost model.

Case 3: In this case, we consider that the frequencies of all the values of the attribute x are higher than p. Hence, each processor i is responsible of treating $\omega_i \times ||Hist^x(R)||$ distinct values of R where ω_i depends on the processing power of each processor, and if each value v of attribute x is found on all the p processors than each processor i will receive a block of data of maximal size $\omega_i \times |Hist^x(R)| \times p$ from all the other processors.

So, if an appropriate hash function is used, each processor receives a block of size $\omega_i \times |Hist^x(R)|$ with a communication cost of order:

$$O\Big(\max_i \big(g \times \omega_i \times |Hist^x(R)| \times p\big) + l\Big) \leqslant O\Big(\max_i \big(g \times \omega_i \times |R|\big) + l\Big). \tag{A.5}$$

This inequality holds due to the fact that $||Hist^x(R)|| \leqslant \frac{||R||}{p}$ because each value v of x appears with a frequency higher than p, thus:

$$\omega_i \times ||Hist^x(R)|| \times p \leqslant \omega_i ||R||.$$

After this step of communication, $||Hist_i^x(R)||$ is created on each processor i by a simple fusion of all the fragments of $Hist^x(R_j)_{j=1,\ldots,p}$ received from all processors j in time of the order: $O\big(\omega_i \times \gamma_i \times ||Hist^x(R)|| \times p\big)$. Thus the cost of creating $Hist_i^x(R)$ is of order:

$$\begin{aligned} O\Big(\min\Big(&\max_{i=1,\ldots,p} \omega_i \times p \times \big(g \times |Hist^x(R)| + \gamma_i \times ||Hist^x(R)||\big), \\ &\max_{i=1,\ldots,p} \omega_i \times \big(g \times |R| + \gamma_i \times ||R||\big)\Big) + l\Big). \end{aligned} \tag{A.6}$$

Case 4: The cost of the general case can be deduced from the previous three cases where R is partitioned into two disjoint sub-relations R' and R'' such that the tuples of R' (resp. R'') are associated with frequencies lower (resp. higher) than p for the join attribute x. Thus, using cases 1 and 2, the creation of the histogram of R' costs:

$$\begin{aligned} O\Big(\min\Big(&\max_{i=1,\ldots,p} \omega_i \times p \times \big(g \times |Hist^x(R')| + \gamma_i \times ||Hist^x(R')||\big), \\ &\max_{i=1,\ldots,p} \omega_i \times \big(g \times |R'| + \gamma_i \times ||R'||\big)\Big) + l\Big), \end{aligned} \tag{A.7}$$

and according to case 3 the creation of the histogram of R'' costs:

$$\begin{aligned} O\Big(\min\Big(&\max_{i=1,\ldots,p} \omega_i \times p \times \big(g \times |Hist^x(R'')| + \gamma_i \times ||Hist^x(R'')||\big), \\ &\max_{i=1,\ldots,p} \omega_i \times \big(g \times |R''| + \gamma_i \times ||R''||\big)\Big) + l\Big). \end{aligned} \tag{A.8}$$

Owing to the fact that $Hist^x(R) = Hist^x(R') \cup Hist^x(R'')$ and that $min(a,b) + min(c,d) \leqslant min(a+c, b+d)$, we can deduce that the cost of creating $Hist_i^x(R)$ is at most of order:

$$\begin{aligned} O\Big(\min\Big(&\max_{i=1,\ldots,p} \omega_i \times p \times \big(g \times |Hist^x(R)| + \gamma_i \times ||Hist^x(R)||\big), \\ &\max_{i=1,\ldots,p} \omega_i \times \big(g \times |R| + \gamma_i \times ||R||\big)\Big) + l\Big). \end{aligned} \tag{A.9}$$

Proposition 2: We consider that relation R (resp. S) is partitioned into m (resp. n) fragments such that $R = \cup_{i=1}^{m} R_i$ (resp. $S = \cup_{i=1}^{n} S_i$).
Let us consider that the local histogram partitions of R are created on m processors of the grid and that of S on n processors according to some attribute A. The cost of creating the distributed global histogram, $Hist_i^x(R)_{i=1,\ldots,m+n}$, on a set of $m+n$ processors ($Proc_{Hist^x(R \bowtie S)}$) starting from the local histograms $Hist^x(R_i)_{i=1,\ldots,m}$ of relation R_i of R is of the order:

$$O\Big(\min\big(|Hist^x(R)| \times (m_p + m_l), \frac{|R|}{m} \times (m_p + m_l)\big) + \min\Big(\frac{m}{m+n} \times \big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t_r^i + t_h^i)\big), \frac{m}{(m+n)^2} \times \big(|R| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||R|| \times (t_r^i + t_h^i)\big)\Big)\Big),$$

if a hash function $\mathcal{H}$ that evenly distributes the distinct values of the join attribute A of the local histograms among the $m+n$ processors is used.

In the above equation, m_p is the communication message protocol cost per page of data and m_l is the communication message latency for one page of data. $c_{r/w}^i$ is the read/write cost of local disk on processor i, t_r^i is the needed time to read a record from its main memory and t_h^i is the time to add an entry to a B^+-tree on processor i.

Proof: In order to prove this cost we will consider four cases from the side of the m processors that forward the local histogram partitions and other four cases from the side of the $m+n$ processors that receive the local histograms to create the global histogram partitions. The difference between the cases lies in the frequency distribution of the attribute A of relation R.

a. Cost of forwarding the local histogram partitions:
The cost of forwarding the local histogram partitions of R from the m processors to the $m+n$ processors of $Proc_{Hist^x(R \bowtie S)}$ is of the order:

$$O\Big(max_i |Hist^x(R_i)| \times (m_p + m_l)\Big). \tag{A.10}$$

We will give an upper bound of this cost to prove the scalability of our approach using the following four cases:

a.1. Each join attribute value appears in R with a frequency equal to 1:
In this case, we have $||Hist^x(R)|| = ||R||$. So, if the tuples of R are evenly partitioned over the m processors, then on each processor i, we have:

$$||Hist^x(R_i)|| = \frac{||Hist^x(R)||}{m} = \frac{||R||}{m}.$$

So, the above forward cost (A.10) can be written as:

$$O\big(\frac{|Hist^x(R)|}{m} \times (m_p + m_l)\big) \leq O\big(\frac{|R|}{m} \times (m_p + m_l)\big).$$

The above inequality holds due to the fact that $|Hist^x(R)| \leq |R|$, since histograms are only formed of couples (v, n_v) where v is the join attribute value and n_v is its corresponding frequency.

Thus, the cost to forward relation's R local histograms is at most:

$$O\Big(\min\big(\tfrac{|Hist^x(R)|}{m} \times (m_p + m_l), \tfrac{|R|}{m} \times (m_p + m_l)\big)\Big) \leq O\Big(\min\big(|Hist^x(R)| \times (m_p + m_l), \tfrac{|R|}{m} \times (m_p + m_l)\big)\Big).$$

a.2. All the join attribute values appear in R with frequencies less than m:
In this case, we can partition R into m relations $R^1, R^2, \ldots, R^m$ such that the frequency of each join attribute value v in R^j ($j = 1, \ldots, m$) is exactly equal to 1. So, the cost of distributing the local histograms of each partition R^j as proved in case *a.1* is of the order:

$$O\Big(\min\big(\frac{|Hist^x(R^j)|}{m} \times (m_p + m_l), \frac{|R^j|}{m} \times (m_p + m_l)\big)\Big).$$

Thus, the cost of forwarding the local histogram partitions of R_i on each processor i is at most:

$$O\Big(\sum_{j=1}^{m} \min\big(\frac{|Hist^x(R^j)|}{m} \times (m_p + m_l), \frac{|R^j|}{m} \times (m_p + m_l)\big)\Big) \leq O\Big(\min\big(|Hist^x(R)| \times (m_p + m_l), \frac{|R|}{m} \times (m_p + m_l)\big)\Big).$$

This inequality holds owing to the fact that:

$$\begin{cases} |Hist^x(R^j)| & \leq |Hist^x(R)| \quad \text{and} \\ \sum_i \min(a_i, b_i) & \leq \min(\sum_i a_i, \sum_i b_i). \end{cases}$$

Remark 12 *In practice, this partitioning step is carried out to only show the validity of the above cost. And whenever a join value appears with a frequency k, a couple (v, k) is sent to its destination processor in one message and not k messages $(v, 1)$. This reduces massively the number of messages exchanged over the network.*

a.3. All the join attribute values appear in R with frequencies greater than m:
In this case, each processor i is at most in charge of $||Hist^x(R)||$ values of join attribute (i.e. all the join attribute values of R appear in R_i). So, the cost of this step is of the order:

$$O\big(|Hist^x(R)| \times (m_p + m_l)\big) \leq O\big(\frac{|R|}{m} \times (m_p + m_l)\big).$$

This inequality holds because each join attribute value appears with a frequency greater than m and thus: $||Hist^x(R)|| \leq \frac{||R||}{m}$ and $|Hist^x(R)| \leq \frac{|R|}{m}$.
So, the forward cost is of the order:

$$O\Big(\min\big(|Hist^x(R)| \times (m_p + m_l), \frac{|R|}{m} \times (m_p + m_l)\big)\Big).$$

a.4. General case:
The cost of this case can be deduced from cases *b* and *c*, where we can divide R into two sub-relations R' and R''. R' holds tuples related to join attribute values with frequencies less than m and R'' those whose frequencies are greater than m. So, the cost of forwarding the local histograms of R' is:

$$O\Big(\min\big(|Hist^x(R')| \times (m_p + m_l), \frac{|R'|}{m} \times (m_p + m_l)\big)\Big),$$

and of R'': $O\Big(\min\big(|Hist^x(R'')| \times (m_p + m_l), \frac{|R''|}{m} \times (m_p + m_l)\big)\Big)$.

The cost of forwarding R is thus the sum of the above two costs which is of the order:

$$O\Big(\min\big(|Hist^x(R)| \times (m_p + m_l), \frac{|R|}{m} \times (m_p + m_l)\big)\Big).$$

This is due to the fact that: $\begin{cases} Hist^x(R') \cap Hist^x(R'') = \emptyset & \text{and} \\ min(a,b) + \min(c,d) \leq \min(a+c, b+d) \end{cases}$
So, the cost of relation's R local histogram forward step is of the order:

$$O\Big(\min\big(|Hist^x(R)| \times (m_p + m_l), \frac{|R|}{m} \times (m_p + m_l)\big)\Big).$$

This also applies to S.

b. The cost of receiving the local histograms:
We will give the cost of receiving and merging the local histogram partitions by the $m+n$ processors from the m processors of R using the following four cases:

b.1. Each join attribute value appears in R with a frequency equal to 1:
Using an appropriate hash function allowing to evenly partition the distinct join attribute values, we can send to each processor approximately $\frac{||R||}{m+n}$ records. Since, $||Hist^x(R)|| = ||R||$ and $|Hist^x(R)| \leq |R|$, the receiving cost of the local histogram partitions by each processor i is of the order $\frac{|Hist^x(R)|}{m+n} \times m_p$. In addition, in order to create the distributed global histogram, each processor i merges the received local histogram partitions' records with a cost of the order: $\frac{||Hist^x(R)||}{m+n} \times (t_r^i + t_h^i)$.
The term $\frac{||Hist(R)||}{m+n} \times t_r^i$ is the needed time to read all the entries of the received partitions of the local histograms. And, $\frac{||Hist(R)||}{m+n} \times t_h^i$ is the needed time to add the new entries to the global histogram of $R \bowtie S$.

Hence, the cost of this step is at most:

$$\begin{aligned} O\big(\frac{|Hist^x(R)|}{m+n} \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} \frac{||Hist^x(R)||}{m+n} \times (t_r^i + t_h^i)\big) \\ \leq O\big(\frac{|R|}{m+n} \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} \frac{||R||}{m+n} \times (t_r^i + t_h^i)\big). \end{aligned}$$

So, the cost on each processor is of the order:

$$\begin{aligned} O\Big(\min\big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t_r^i + t_h^i), \\ \frac{|R|}{m+n} \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} \frac{||R||}{m+n} \times (t_r^i + t_h^i)\big)\Big). \end{aligned}$$

b.2. All the join attribute values appear in R with frequencies less than $m+n$:
Following the same demonstration method used in case (a.2.), we divide R into $m+n$ relations: $R^1, R^2, \ldots, R^{m+n}$ such that the frequency of each join value v is exactly equal to 1. Thus, the receiving and local histograms records merging cost on each processor related to each partition R^j is:

$$\begin{aligned} O\Big(\min\big(\frac{m}{m+n} \times \big(|Hist^x(R^j)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R^j)|| \times (t_r^i + t_h^i)\big), \\ \frac{m}{(m+n)^2} \times \big(|R^j| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||R^j|| \times (t_r^i + t_h^i)\big)\big)\Big). \end{aligned}$$

So, the total cost on each reducer is:

$$O\Big(\sum_{i=1}^{m+n} \min\Big(\frac{m}{m+n} \times \big(|Hist^x(R^j)| \times m_p + ||Hist^x(R^j)|| \times (t_r + t_h)\big), \frac{m}{(m+n)^2} \times \big(|R^j| \times m_p + ||R^j|| \times (t_r + t_h)\big)\Big)\Big)$$
$$\leq O\Big(\min\Big(\frac{m}{m+n} \times \big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t_r^i + t_h^i)\big), \frac{m}{(m+n)^2} \times \big(|R| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||R|| \times (t_r^i + t_h^i)\big)\Big)\Big).$$

(Remark 12 also applies to this step).

b.3. All the join attribute values appear in R with frequencies greater than $m+n$:
If we use an appropriate hash function, each processor will be responsible of approximately $\frac{||Hist^x(R)||}{m+n}$ join attribute values. So, if each join attribute value appears in each R_i ($i \in (1, \ldots, m)$), then each processor of the $m+n$ processors receives at most: $m \times \frac{||Hist^x(R)||}{m+n}$ local histograms records. Thus, the communication and records merging cost on each processor is of the order:

$$O\Big(\frac{m}{m+n}\big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t_r^i + t_h^i)\big)\Big)$$

Since each join attribute value appears with a frequency greater than $m+n$, we have: $||Hist^x(R)|| \leq \frac{||R||}{m+n}$ and $|Hist^x(R)| \leq \frac{|R|}{m+n}$.
So, $\frac{m}{m+n}||Hist^x(R)|| \leq \frac{m}{(m+n)^2} \times ||R||$ and $\frac{m}{m+n}|Hist^x(R)| \leq \frac{m}{(m+n)^2} \times |R|$.
Thus, the communication and records merging cost in this case is of the order:

$$O(\min\Big(\frac{m}{m+n} \times \big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t_r^i + t_h^i)\big), \frac{m}{(m+n)^2} \times \big(|R| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||R|| \times (t_r^i + t_h^i)\big)\Big).$$

b.4. General case:
The cost for the general case is of the order:

$$O(\min\Big(\frac{m}{m+n} \times \big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t_r^i + t_h^i)\big), \frac{m}{(m+n)^2} \times \big(|R| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||R|| \times (t_r^i + t_h^i)\big)\Big).$$

The proof is straightforward from cases *b2* and *b3* where we can follow the same steps of case *a.4*.

So, the total cost of creating the distributed global histogram of R starting from the local ones is of the order:

$$O\Big(\min\big(|Hist^x(R)| \times (m_p + m_l), \frac{|R|}{m} \times (m_p + m_l)\big) + \min\Big(\frac{m}{m+n} \times \big(|Hist^x(R)| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||Hist^x(R)|| \times (t_r^i + t_h^i)\big), \frac{m}{(m+n)^2} \times \big(|R| \times m_p + \max_{i \in Proc_{Hist^x(R \bowtie S)}} ||R|| \times (t_r^i + t_h^i)\big)\Big)\Big).$$

B^+ trees

A B^+ tree is a form of balanced search trees where data are sorted in a manner which allows efficient and rapid lookup, insertion and deletion of data. As shown in figure A.1, a B^+ tree is formed of nodes linked by pointers. A node may be an *internal* or a *leaf* node.

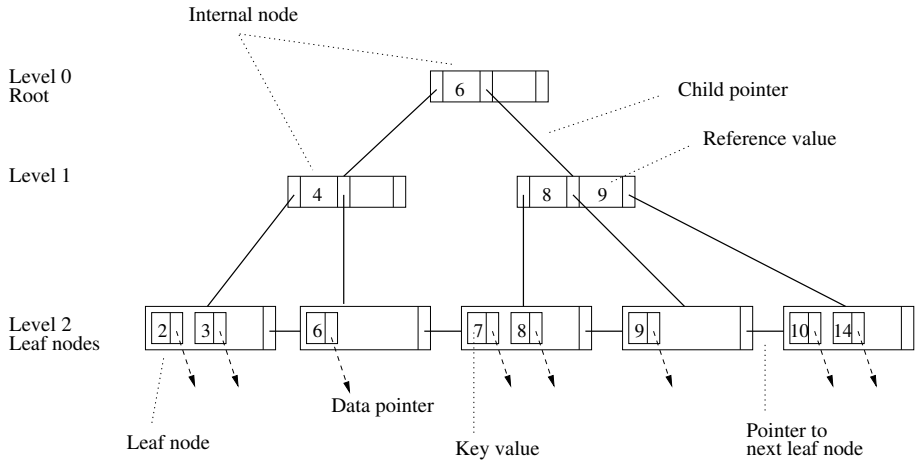

FIG. A.1 – *A B^+ tree example.*

An *internal node*, including the *root* node, holds entries formed of *reference values* and *child pointers*. Each child pointer points to a node holding values less than or equal to the reference value immediately to its right. However, the last pointer points to a node holding values that are greater than the last reference value in the internal node.

A *leaf node* holds entries formed of a *key value* and a *data pointer* to the storage location associated with the key value. An extra pointer in each leaf node points to its immediate sibling node (the node to its right).

We can see that in B^+ trees, all data are stored in the leaf nodes and internal nodes only hold referential data that helps to decrease the lookup time of data stored at the leaves. In addition, the sequential linkage between the leaf nodes allows us to access sequentially these nodes without the need to visit the internal nodes.

Remark 13 *To implement the histograms used in this thesis report, we can replace the data pointers of the leaf nodes by the frequency of the key. In addition, for equi-join queries, we can use B trees. However, B^+ trees are more efficient for other types of θ-join queries.*

A B^+ tree of order m is a balanced tree where :
- each node has k keys such that $m \leq k \leq 2m$ except the root which may have between 1 and $2m$ keys,
- a node is either a leaf or an internal node that has $k+1$ children,
- to store N keys, $1 + \log_{m+1} \frac{N+1}{2}$ levels is needed.

Notations

General notations

- T denotes a relation,
- $|T|$ denotes the size (expressed in bytes or number of pages) of T,
- $||T||$ denotes the number of tuples of T,
- h denotes a hash function,
$$h : T \longrightarrow \mathbb{N}$$
$$t \longmapsto n$$
where t is a tuple of T and n is the nodes index that t is hashed to,
- T_i denotes the fragment of T placed on node i,
- n_v : number of tuples having value v for the join or semi-join attribute,
- $c^i_{r/w}$ denotes the cost of reading/writing a page of data from/to the disk of processor i,
- m_p : communication message protocol cost per page of data,
- m_l : communication message latency for one page of data,
- $c^i_{r/w}$: read/write cost of a page of data on local disk on processor i,
- t^i_r : time to read a record from main memory of processor i,
- t^i_w : time to write a record to main memory of processor i,
- t^i_s : time of a simple search in a B$^+$-tree on processor i,
- t^i_h : time to add an entry to a B$^+$-tree on processor i.

Notations used in *GAJFA-Join*, *GBJFA-Join*, *DFA-Join*, *PDFA-Join* **and** *GDFA-Join* **algorithms** :

- $Hist^w(T)$ denotes the histogram[2] of relation T with respect to the attribute w, i.e. a list of pairs (v, n_v) where $n_v \neq 0$ is the number of tuples of relation T having the value v for the attribute w. The histogram is often much smaller and never larger than the relation it describes,
- $Hist^w(T_i)$ denotes the histogram of fragment T_i,
- $Hist^w_i(T)$ is processor i's fragment of the histogram of T,
- $Hist^w(T)(v)$ is the frequency (n_v) of value v in relation T,
- $Hist^w(T_i)(v)$ is the frequency of value v in sub-relation T_i,
- $Hist^{w[j]}(T_i)$ is the fragment of $Hist^w(T_i)$ sent from processor i to processor j,
- $\overline{Hist}^{w[j]}(T_i)$ is the restriction of $Hist^{w[j]}(T_i)$ to entries related to entries of w that appear in the final join result,
- $\overline{\overline{T_i}}$ is the partition of T received by processor i as indicated by the communication template to compute the join result.

Notations specific to *GAJFA-Join* **and** *GBJFA-Join* **algorithms** :

- $AGGR^w_{f,u}(T)$ [3] is the result of applying the aggregate function f on the values of the attribute u of every group of tuples of T having identical values of the group-by attributes w. $AGGR^w_{f,u}(T)$ is formed of a list of tuples (v, f_v) where f_v is the result of the aggregate function of the group of tuples having value v for the attribute w (w may be formed of more than one attribute),
- $AGGR^w_{f,u}(T_i)$ denotes the result of applying the aggregate function on the attribute u of relation T_i,

[2]Histograms are implemented as balanced trees (B$^+$-tree).

[3]$AGGR^w_{f,u}(T)$ is implemented as a balanced tree (B$^+$-tree) : a data structure that maintains an ordered set of data to allow efficient search and insert operations (cf. Appendix A page 158).

- $AGGR^{w}_{f,u,i}(T)$ is processor i's fragment of the result of applying the aggregate function on T,
- $AGGR^{w}_{f,u}(T)(v)$ is the result f_v of the aggregate function of the group of tuples having value v for the group-by attribute w in relation T,
- $AGGR^{w}_{f,u}(T_i)(v)$ is the result f_v of the aggregate function of the group of tuples having value v for the group-by attribute w in sub-relation T_i.

Notations specific to *GDFA-Join* **algorithms** :

- num_{clust} : the number of clusters forming the grid,
- P_T : the list of processors' indexes holding a replication of T,
- $Proc_{Hist^x(R \bowtie S)}$: the list of processors' indexes that participate in computing $Hist^x(R \bowtie S)$,
- $rep_{(T)}$: the number of processors holding a replica of T,
- *bucket_size* : the size of generated join buckets specified by the coordinator node.

Notations specific to *CFA-Semi-Join* **algorithms** :

- $(BR_i)_{map_j}$: hashed bucket of index i related to R splits placed on mapper j,
- $(BS_i)_{map_j}$: hashed bucket of index i related to S splits placed on mapper j,
- $LHist(BR_i)_{map_j}$: local histogram of $(BR_i)_{map_j}$, i.e. the list of pairs (v, n_v) where v is a semi-join value and n_v its corresponding frequency in relation BR_i on mapper map_j,
- $LHist(BS_i)_{map_j}$: local histogram of $(BS_i)_{map_j}$,
- $GHist(BR_i)$ (resp. $GHist(BS_i)$) : global histogram of buckets BR_i (resp. BS_i),
- $Hist(BR_i \bowtie BS_i)$: histogram related to join attribute values that appear in both relations partitions BR_i and BS_i,
- $\overline{LHist}(BR_i)_{map_j}$: the restriction of $LHist(BR_i)_{map_j}$ to join attribute values that appear in the semi-join result,
- $\overline{GHist}(R \bowtie S)$: join attribute values that appear in both R and S,
- σ : semi-join selectivity factor,
- *NB_mappers* : number of mapper nodes of each relation,
- *NB_reducers* : number of reducer nodes of each relation,
- *NB_mergers* : number of reducer nodes.

References

Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris J. Scheiman. Loggp : Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation. In *SPAA*, pages 95–105, 1995.

Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Efficient data transport and replica management for high-performance data-intensive computing. In *IEEE Mass Storage Conference*, 2001.

Nedim M. Alpdemir, Arijit Mukherjee, Anastasios Gounaris, Norman W. Paton, Paul Watson, Alvaro A. Fernandes, and Desmond J. Fitzgerald. Ogsa-dqp : A service for distributed querying on the grid. *Advances in Database Technology - EDBT 2004*, pages 858–861, 2004. URL `http://www.springerlink.com/content/tvep8uec5hheq8y4`.

Carlos E. R. Alves, Edson Cáceres, Frank K. H. A. Dehne, and Siang W. Song. A cgm/bsp parallel similarity algorithm. In Ana L. C. Bazzan, editor, *WOB*, pages 1–8, 2002.

Mario Antonioletti, Malcolm Atkinson, Rob Baxter, Andrew Borley, Neil P. Chue Hong, Brian Collins, Neil Hardman, Alastair C. Hume, Alan Knox, Mike Jackson, Amy Krause, Simon Laws, James Magowan, Norman W. Paton, Dave Pearson, Tom Sugden, Paul Watson, and Martin Westhead. The design and implementation of grid database services in ogsa-dai : Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4) :357–376, 2005. ISSN 1532-0626. doi : http://dx.doi.org/10.1002/cpe.v17:2/4.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds : A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

M. Bamha. An optimal and skew-insensitive join and multi-join algorithm for ditributed architectures. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA'2005). 22-26 August, Copenhagen, Danemark*, volume 3588 of *Lecture Notes in Computer Science*, pages 616–625. Springer-Verlag, 2005.

M. Bamha and G. Hains. A skew insensitive algorithm for join and multi-join operation on Shared Nothing machines. In *the 11th International Conference on Database and Expert Systems Applications DEXA'2000*, volume 1873 of *Lecture Notes in Computer Science*, London, United Kingdom, 2000. Springer-Verlag.

M. Bamha and G. Hains. An efficient equi-semi-join algorithm for distributed architectures. In *Proceedings of the 5th International Conference on Computational Science (ICCS'2005). 22-25 May, Atlanta, USA*, volume 3515 of *Lecture Notes in Computer Science*, pages 755–763. Springer-Verlag, 2005.

M. Bamha and G. Hains. A frequency adaptive join algorithm for Shared Nothing machines. *Journal of Parallel and Distributed Computing Practices (PDCP), Volume 3, Number 3, pages 333-345*, 1999. Appears also in Progress in Computer Research, F. Columbus Ed. Vol. II, Nova Science Publishers, 2001.

Mostafa Bamha and Matthieu Exbrayat. Pipelining a skew-insensitive parallel join algorithm. *Parallel Processing Letters*, 13(3) :317–328, 2003.

Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. Bsp vs logp. In *SPAA '96 : Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 25–32, New York, NY, USA, 1996. ACM. ISBN 0-89791-809-6. doi : http://doi.acm.org/10.1145/237502.237504.

H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1) :4–24, 1990. ISSN 1041-4347. doi : http://dx.doi.org/10.1109/69.50903.

F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000 : A large scale and highly reconfigurable grid experimental testbed. In *GRID '05 : Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9492-5. doi : http://dx.doi.org/10.1109/GRID.2005.1542730.

J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2) :143–154, April 1979. ISSN 0022-0000.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : a distributed storage system for structured data. In *OSDI '06 : Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL `http://portal.acm.org/citation.cfm?id=1298475`.

Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB '94 : Proceedings of the 20th International Conference on Very Large Data Bases*, pages 354–366, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8.

M.-S. Chen and P. S. Yu. Combining joint and semi-join operations for distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 5(3) : 534–542, 1993. ISSN 1041-4347. doi : http://dx.doi.org/10.1109/69.224205.

Ming S. Chen, Ming L. Lo, Philip S. Yu, and Honesty C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *VLDB '92 : Proceedings of the 18th International Conference on Very Large Data Bases*, pages 15–26, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1-55860-151-1. URL `http://portal.acm.org/citation.cfm?id=645918.672489`.

S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems*, 9(2) :163–186, 1984. ISSN 0362-5915.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6) :377–387, 1970.

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp : towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7) :1–12, 1993. ISSN 0362-1340. doi : http://doi.acm.org/10.1145/173284.155333.

A. Datta, B. Moon, and H. Thomas. A case for parallelism in datawarehousing and OLAP. In *Ninth International Workshop on Database and Expert Systems Applications, DEXA 98*, IEEE Computer Society, pages 226–231, Vienna, 1998.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. In *OSDI ' 04 : Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, 2004. URL `http://labs.google.com/papers/mapreduce.html`.

Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *SCG '93 : Proceedings of the ninth annual symposium on Computational geometry*, pages 298–307, New York, NY, USA, 1993. ACM. ISBN 0-89791-582-8. doi : http://doi.acm.org/10.1145/160985.161154.

Frank Dehne, Wolfgang Dittrich, and David Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *SPAA '97 : Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 106–115, New York, NY, USA, 1997. ACM. ISBN 0-89791-890-8. doi : http://doi.acm.org/10.1145/258492.258503.

Frank K. H. A. Dehne, Wolfgang Dittrich, David A. Hutchinson, and Anil Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory Comput. Syst.*, 35(6) :567–597, 2002.

D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 18th VLDB Conference*, pages 27–40, Vancouver, British Columbia, Canada, 1992.

David J. DeWitt and Jim Gray. Parallel database systems : The future of high performance database systems. *Commun. ACM*, 35(6) :85–98, 1992.

David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1) :44–62, 1990.

Jörn Eisenbiegler, Welf Löwe, and Wolf Zimmermann. Bsp, logp, and oblivious programs. In *Euro-Par '98 : Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 865–874, London, UK, 1998. Springer-Verlag. ISBN 3-540-64952-2.

Michael J. Flynn. Very high-speed computing systems. In *Proceeding of the IEEE, 54(12)*, pages 1901–1909, 1966.

Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78 : Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM. doi : http://doi.acm.org/10.1145/800133.804339.

I. Foster and C. Kesselman. *The Grid : Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

Ian Foster. What is the grid ? - a three point checklist. *GRIDtoday*, 1(6), July 2002. URL `http://www.gridtoday.com/02/0722/100136.html`.

Ian T. Foster. Globus toolkit version 4 : Software for service-oriented systems. In Hai Jin, Daniel A. Reed, and Wenbin Jiang, editors, *NPC*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.

Ian T. Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. *CoRR*, abs/0901.0131, 2009.

Frédéric Gava. External Memory in Bulk Synchronous Parallel ML. *Scalable Computing : Practice and Experience*, 6(4) :43–70, 2005.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03 : Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press. ISBN 1581137575. doi : 10.1145/945445.945450. URL `http://dx.doi.org/10.1145/945445.945450`.

Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The qrqw pram : accounting for contention in parallel algorithms. In *SODA '94 : Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 638–648, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics. ISBN 0-89871-329-3.

Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write PRAM model : Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 28(2) :733–769, 1999. URL `citeseer.ist.psu.edu/gibbons96queueread.html`.

Anastasios Gounaris. Resource aware query processing on the grid. Thesis report, University of Manchester, Faculty of Engineering and Physical Sciences, 2005.

Anastasios Gounaris, Jim Smith, Norman W. Paton, Rizos Sakellariou, Alvaro A. A. Fernandes, and Paul Watson. Adapting to changing resource performance in grid query processing. In *Data Management in Grids, First VLDB Workshop, DMG 2005, Trondheim, Norway, September 2-3, 2005, Revised Selected Papers*, volume 3836 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2005. ISBN 3-540-31212-9.

R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17 :416–429, 1969.

Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 358 – 369, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-379-4.

hadoop. Apache hadoop. `http://hadoop.apache.org/core/`.

Jiawei Han and Micheline Kamber. *Data Mining : Concepts and Techniques*. Morgan Kaufmann, 2000. ISBN 1-55860-489-8.

Lilian Harada and Masaru Kitsuregawa. Dynamic join product skew handling for hash-joins in shared-nothing database systems. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 246–255. World Scientific Press, 1995. ISBN 981-02-2220-3.

Mohamad Al Hajj Hassan and Mostafa Bamha. An efficient pipelined parallel join algorithm on heterogeneous distributed architectures. In *Communications in Computer and Information Science, Volume 47*, pages 119–133. Springer Verlag, 2009a.
Extended and revised version of "ICSOFT 2008 Best papers".

Mohamad Al Hajj Hassan and Mostafa Bamha. Pipelined parallelism in multi-join queries on heterogeneous shared nothing architectures. In José Cordeiro, Boris Shishkov, Alpesh Ranchordas, and Markus Helfert, editors, *ICSOFT (PL/DPS/KE)*, pages 127–134. INSTICC Press, 2008. ISBN 978-989-8111-51-7.
Selected to appear to "ICSOFT Best papers Book".

Mohamad Al Hajj Hassan and Mostafa Bamha. An efficient parallel algorithm for evaluating join queries on heterogeneous distributed systems. In *HiPC 2009 : Proceedings of the 16th international conference on high performance computing*. IEEE Computer Society Press, 2009b.

Mohamad Al Hajj Hassan and Mostafa Bamha. Semi-join computation on distributed file systems using map-reduce-merge model. In *Proceedings of the 25th Symposium On Applied Computing*. ACM, 22 - 26 March 2010. To appear.

Mohamad Al Hajj Hassan and Mostafa Bamha. Parallel processing of 'group-by join' queries on Shared Nothing machines. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT'06)*, volume 1, pages 301–307, Setubal, Portugal, 11-14 September 2006. INSTICC press. ISBN 972-8865-69-4.
Also appears in 'ICSOFT Best papers Book', 2007.

Mohamad Al Hajj Hassan and Mostafa Bamha. An optimal evaluation of groupby-join queries in distributed architectures. In *Proceedings of the third International Conference on Web Information Systems and Technologies (WEBIST 2007)*, volume IT, pages 246–252, Barcelona, Spain, 3 - 6 March 2007. INSTICC press.

K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. of the 17th International Conference on Very Large Data Bases*, pages 525–535, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann. ISBN ISBN 1-55860-150-3.

Kien A. Hua, Wallapak Tavanapong, and Honesty C. Young. A performance evaluation of load balancing techniques for join operations on multicomputer database systems. In *ICDE '95 : Proceedings of the Eleventh International Conference on Data Engineering*, pages 44–51, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-6910-1.

Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *VLDB '2003 : Proceedings of the 29th international conference on Very large data bases*, pages 321–332. VLDB Endowment, 2003. ISBN 0-12-722442-4.

Yi Jiang, Kevin H. Liu, and Clement H. C. Leung. Parallel algorithms for queries with aggregate functions in the presence of data skew. In *HiPC '99 : Proceedings of the 6th International Conference on High Performance Computing*, pages 207–211, London, UK, 1999. Springer-Verlag. ISBN 3-540-66907-8.

H. Karatza and R. Hilzer. Load sharing in heterogeneous distributed systems. In *Proceedings of the 2002 Winter Simulation Conference*, pages 489–496, 2002.

M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash : A new, robust, parallel hash join method for skew in the super database computer (SDC). In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Very Large Data Bases : 16th International Conference on Very Large Data Bases, August 13–16, 1990, Brisbane, Australia*, pages 210–221, Los Altos, CA 94022, USA, 1990. Morgan Kaufmann Publishers. ISBN 1-55860-149-X.

M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1) :63–74, 1983. ISSN 0288-3635.

Ralf Lämmel. Google's mapreduce programming model — revisited. *Sci. Comput. Program.*, 68(3) :208–237, 2007. ISSN 0167-6423. doi : http://dx.doi.org/10.1016/j.scico.2007.07.001.

Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Efficient processing of ad-hoc top-k aggregate queries in olap. Technical report, UIUCDCS-R-2005-2596, Department of Computer Science, UIUC, June 2005.

Weifa Liang and Maria E. Orlowska. Computing multidimensional aggregates in parallel. *Informatica (Slovenia)*, 24(1), 2000.

Bin Liu and Elke A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB '05 : Proceedings of the 31st international conference on Very large data bases*, pages 829–840, Endowment, 2005. VLDB.

Welf Löwe, Wolf Zimmermann, and Jörn Eisenbiegler. On linear schedules of task graphs for generalized logp-machines. In *Euro-Par '97 : Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 895–904, London, UK, 1997. Springer-Verlag. ISBN 3-540-63440-1.

H. Lu and K.-L. Tan. Dynamic and load-balanced task-oriented database query processing in parallel systems. In *Proceedings of Advances in Database Technology (EDBT '92)*, volume 580 of *LNCS*, pages 357–372, Berlin, Germany, 1992. Springer. ISBN 3-540-55270-7.

Steven Lynden, Arijit Mukherjee, Alastair C. Hume, Alvaro A. A. Fernandes, Norman W. Paton, Rizos Sakellariou, and Paul Watson. The design and implementation of ogsa-dqp : A service-based distributed query processor. *Future Gener. Comput. Syst.*, 25(3) :224–236, 2009. ISSN 0167-739X. doi : http://dx.doi.org/10.1016/j.future.2008.08.003.

P. S. Yu M.-S. Chen and K.-L. Wu. Scheduling and processor allocation for the execution of multi-join queries. In *International Conference on Data Engineering*.

Werner Mach and Erich Schikuta. Parallel database join operations in heterogeneous grids. *Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2007)*, 0 :236–243, 2007. doi : http://doi.ieeecomputersociety.org/10.1109/PDCAT.2007.79.

A. N. Mourad, R. J. T. Morris, A. Swami, and H. C. Young. Limits of parallelism in hash join algorithms. *Performance evaluation*, 20(1/3) :301–316, May 1994. ISSN 0166-5316.

Norman W. Paton, Jorge Buenabad Chávez, Mengsong Chen, Vijayshankar Raman, Garret Swart, Inderpal Narang, Daniel M. Yellin, and Alvaro A. A. Fernandes. Autonomic query parallelization using non-dedicated computers : an evaluation of adaptivity options. *VLDB J.*, 18(1) :119–140, 2009.

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data : Parallel analysis with sawzall. 2005. URL `http://labs.google.com/papers/sawzall-sciprog.pdf`.

Erhard Rahm. Dynamic load balancing in parallel database systems. In *In Proc. of EURO-PAR*, pages 37–52. Springer, 1996.

Vijayshankar Raman, Wei Han, and Inderpal Narang. Parallel querying with non-dedicated computers. In *VLDB '05 : Proceedings of the 31st international conference on Very large data bases*, pages 61–72. VLDB Endowment, 2005. ISBN 1-59593-154-6.

K. Sattler, E. Buchmann, and K. Böhm. A physical query algebra for dht-based p2p systems. In *In 6th Workshop on Distributed Data and Structures (WDAS'2004*, pages 8–9, 2004.

D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor data-base machines. In *International Conference On Very Large Data Bases (VLDB '90)*, pages 469–480, Palo Alto, Ca., USA, 1990. Morgan Kaufmann Publishers, Inc. ISBN 0-55860-149-X.

Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data, Portland, Oregon, United States, May 1989*, pages 110–121, New York, NY, USA, 1989. ACM Press.

M. Seetha and P. S. Yu. Effectiveness of parallel joins. *IEEE, Transactions on Knowledge and Data Enginneerings*, 2(4) :410–424, December 1990.

A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2) :104–114, 1995. ISSN 0163-5808.

Ambuj Shatdal and Jeffrey F. Naughton. Processing aggregates in parallel database systems. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1994.

D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.

Jim Smith and Paul Watson. Fault-tolerance in distributed query processing. In *IDEAS '05 : Proceedings of the 9th International Database Engineering & Application Symposium*, pages 329–338, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2404-4. doi : http://dx.doi.org/10.1109/IDEAS.2005.29.

Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. Integrating semi-join-reducers into state-of-the-art query processors. In *Proceedings of the 17th International Conference on Data Engineering*, pages 575 – 584. IEEE Computer Society, 2001.

D. Taniar, Y. Jiang, K.H. Liu, and C.H.C. Leung. Aggregate-join query processing in parallel database systems,. In *Proceedings of The Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region HPC-Asia2000*, volume 2, pages 824–829. IEEE Computer Society Press, 2000.

David Taniar and J. Wenny Rahayu. Parallel processing of 'groupby-before-join' queries in cluster architecture. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid, Brisbane, Qld, Australia*, pages 178–185. IEEE Computer Society, 2001.

David Taniar and Wenny Rahayu. Parallel "groupby-before-join" query processing for high performance parallel/distributed database systems. In *AINA '06 : Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, pages 693–700, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2466-4-01. doi : http://dx.doi.org/10.1109/AINA.2006.256.

David Taniar, Rebecca Boon-Noi Tan, C. H. C. Leung, and K. H. Liu. Performance analysis of "groupby-after-join" query processing in parallel database systems. *Inf. Sci. Inf. Comput. Sci.*, 168(1-4) :25–50, 2004. ISSN 0020-0255. doi : http://dx.doi.org/10.1016/j.ins.2003.09.029.

David Taniar, Clement H. C. Leung, J. Wenny Rahayu, and Sushant Goel. *High Performance Parallel Database Processing and Grid Databases*. John Wiley & Sons, 2008a. ISBN 978-0-470-10762-1.

David Taniar, Clement H. C. Leung, Wenny Rahayu, and Sushant Goel. *High Performance Parallel Database Processing and Grid Databases*. Wiley Publishing, 2008b. ISBN 0470107626, 9780470107621.

L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, August 1990.

C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Barcelona*, 1991.

Rajeev Wankar. Grid computing with globus : An overview and research challenges. *IJCSA*, 5(3) :56–69, 2008.

Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS '91 : Proceedings of the first international conference on Parallel and distributed information systems*, pages 68–77, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press. ISBN 0-8186-2295-4.

Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel evaluation of multi-join queries. *SIGMOD Rec.*, 24(2) :115–126, 1995. ISSN 0163-5808. doi : http://doi.acm.org/10.1145/568271.223803.

J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. New algorithms for parallelizing relational database joins in the presence of data skew. *IEEE Trans. on Knowl. and Data Eng.*, 6(6) :990–997, 1994. ISSN 1041-4347. doi : http://dx.doi.org/10.1109/69.334888.

Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 89–100, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-5400-7.

Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *VLDB '95 : Proceedings of the 21th International Conference on Very Large Data Bases*, pages 345–357, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-379-4.

Donghua Yang, Jianzhong Li, and Qaisar Rasool. Join algorithm using multiple replicas in data grid. In *WAIM*, pages 416–427, 2005.

Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge : simplified relational data processing on large clusters. In *SIGMOD '07 : Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi : http://doi.acm.org/10.1145/1247480.1247602.

Hansjörg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser environments. In *Proceedings of the sixteenth international conference on Very large databases*, pages 186–197, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc. ISBN 0-55860-149-X.

Xi Zhang, Tahsin Kurc, Tony Pan, Umit Catalyurek, Sivaramakrishnan Narayanan, Pete Wyckoff, and Joel Saltz. Strategies for using additional resources in parallel hash-based join algorithms. In *HPDC '04 : Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 4–13, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-2175-4. doi : http://dx.doi.org/10.1109/HPDC.2004.34.

G. K. Zipf. Human Behavior and the Principle of Least Effort : An Introduction to Human Ecology. *Reading, MA, Adisson-Wesley*, 1949.

Mohamad AL HAJJ HASSAN

Parallélisme et équilibrage de charges dans le traitement de la jointure sur des architectures distribuées

Résumé :

L'émergence des applications de bases de données dans les domaines tels que le data warehousing, le data mining et l'aide à la décision qui font généralement appel à de très grands volumes de données rend la parallélisation des algorithmes des jointures nécessaire pour avoir un temps de réponse acceptable. Une accélération linéaire est l'objectif principal des algorithmes parallèles, cependant dans les applications réelles, elle est difficilement atteignable : ceci est dû généralement d'une part aux coûts de communications inhérents aux systèmes multi-processeurs et d'autre part au déséquilibre des charges des différents processeurs. En plus, dans un environnement hétérogène multi-utilisateur, la charge des différents processeurs peut varier de manière dynamique et imprévisible.
Dans le cadre de cette thèse, nous nous intéressons au traitement de la jointure et de la multi-jointure sur les architectures distribuées hétérogènes, les grilles de calcul et les systèmes de fichiers distribués. Nous avons proposé une variété d'algorithmes, basés sur l'utilisation des histogrammes distribués, pour traiter de manière efficace le déséquilibre des données, tout en garantissant un équilibrage presque parfait de la charge des différents processeurs même dans un environnement hétérogène et multi-utilisateur. Ces algorithmes sont basés sur une approche dynamique de redistribution des données permettant de réduire les coûts de communication à un minimum tout en traitant de manière très efficace le problème de déséquilibre des valeurs de l'attribut de jointure.
L'analyse de complexité de nos algorithmes et les résultats expérimentaux obtenus montrent que ces algorithmes possèdent une accélération presque linéaire.
Mots clés : Systèmes de gestion de bases de données parallèles, Jointures parallèles, Multi-jointure, Les Grilles de calcul, Les systèmes de fichiers distribués, Déséquilibre des données, Équilibrage dynamique de charges.

Parallelism and load balancing in the treatment of the join on distributed architectures

Résumé :
The appeal of parallel processing becomes very strong in applications which require ever higher performance and particularly in applications such as : data-warehousing, decision support, On-Line Analytical Processing (OLAP) and more generally DBMS. A linear speed-up is the main objective of parallel algorithms. However, in real applications, it's not obvious to reach this objective due to the high communication cost in parallel and distributed systems and to the possible skew in the charge of different processors. In addition, on heterogeneous multi-user architectures, the load of each processor may highly vary in a dynamic and unpredictable way.
In this thesis, we are interested in treating the join and multi-join queries on distributed multi-user heteregeneous systems, grid systems and distributed file systems. We have proposed several algorithms based on using distributed histograms. These algorithms are based on a dynamic data distribution and task allocation which makes them insensitive to data skew and ensure perfect balancing properties during all stages of join computation even on heteregeneous multi-user environment. The complexity analysis of our algorithms and the experimental results show that they have a near-linear speedup.
Keywords : Parallel Database Management Systems, Parallel joins, Multi-join, Grid systems, Distributed File Systems, Data skew, Dynamic load balancing.

PUCVL
Pôle Universités
Centre Val de Loire

LIFO

UNIVERSITE D'ORLEANS