Big Data Warehouse : model de distribution des cubes de données à la volée

Yassine Ramdane

▶ To cite this version:

Yassine Ramdane. Big Data Warehouse : model de distribution des cubes de données à la volée. Databases [cs.DB]. Université de Lyon, 2019. English. NNT : 2019LYSE2099 . tel-02506735

HAL Id: tel-02506735
https://theses.hal.science/tel-02506735v1

Submitted on 12 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre NNT : 2019LYSE2099

THESE de DOCTORAT DE L'UNIVERSITÉ DE LYON

Opérée au sein de

L'UNIVERSITÉ LUMIÈRE LYON 2

École Doctorale : ED 512 Informatique et Mathématiques

Discipline : Informatique

Soutenue publiquement le 21 novembre 2019, par :

Yassine RAMDANE

Big Data Warehouse :

Modèle de distribution de données dans des cubes à la volée.

Devant le jury composé de :

Salima HASSAS, Professeure des universités, Université Claude Bernard Lyon 1, Présidente

François PINET, Directeur de Recherches, Inst. Nat. Rech.en Sces et Tech. Environmt et Agricult., Rapporteur

Olivier TESTE, Professeur des universités, Université Toulouse 2, Rapporteur

Vargas-Solar GENOVEVA, Chargé de recherche, CNRS, Examinateur

Abdelmouneem REZGUI, Maître de conférences, Illinois State University, Examinateur

Omar BOUSSAID, Professeur des universités, Université Lumière Lyon 2, Directeur de thèse

Nadia KABACHI, Maître de conférences, Université Claude Bernard Lyon 1, Co Directrice

Contrat de diffusion

Ce document est diffusé sous le contrat *Creative Commons* « Paternité – pas d'utilisation commerciale – pas de modification » : vous êtes libre de le reproduire, de le distribuer et de le communiquer au public à condition d'en mentionner le nom de l'auteur et de ne pas le modifier, le transformer, l'adapter ni l'utiliser à des fins commerciales.

UNIVERSITÉ LUMIÈRE LYON 2
ÉCOLE DOCTORALE INFORMATIQUE ET MATHÉMATIQUES

Thèse de Doctorat

En Informatique

Presentée par
Yassine Ramdane

Thème

Big Data Warehouse : Model de Distribution des Données dans des Cubes à la Volée

Sous la direction de :
M. Omar Boussaid and Mme Nadia Kabachi

Soutenue le 21 November 2019
devant la Commission d'Examen

Jury

President :	**Professeur François Pinet,** Institue IRSTEA - Clermont-Ferrand, France.
Rapporteurs :	**Professeur Olivier Teste,** Université Toulouse 2 Jean Jaurès, UT2C, IRIT.
	Professeur François Pinet, Institue IRSTEA - Clermont-Ferrand, France.
Examinateurs :	**Professeure Salima Hassas,** Laboratoire LIRIS-CNRS/université Lyon 1
	Professeur Omar Boussaid, Université Lumière Lyon 2, ERIC.
	Char. de Rech. Genoveva Vargas-Solar, Université de Grenoble, CNRS.
	MCF, Abdelmounaam Rezgui, Université de Illinios, USA.
	MCF Nadia Kabachi, Université de Claude Bernard Lyon 1, ERIC.

Abstract

Partitioning and distributing of the data have been widely used in sharing nothing systems, more particularly in the distributed systems that used the MapReduce paradigm, such as Hadoop ecosystem and Spark. They have been used for many purposes, such as load balancing, skipping to load unnecessary data partitions and for guiding the physical design of distributed databases or data warehouses. To do analysis with data warehouses, usually, we used OLAP queries. An OLAP query is a complex query that contains several cost operations, such as the star join, the projection, filtering, and aggregate functions. In this thesis, we propose different static and dynamic approaches of partitioning and load balancing of the data, to improve the performances of distributed big data warehouses over Hadoop cluster. We have proposed different static and dynamic schemes of a big data warehouse over a cluster of homogeneous nodes, which can help the distributed system to enhance the executing time of OLAP query operations, such as star join operation, scanning tables, and Group-By operation. We have proposed four approaches: The first approach, is a new data placement strategy which able to help a query processing system to perform a star join operation in only one MapReduce cycle, without a shuffle phase; In the second contribution, we propose different partitioning and bucketing techniques to skip loading some HDFS blocks and to enhance the parallel treatment of the distributed system, based on a workload-driven model; In the third approach, we propose a novel physical design of distributed big data warehouse over Hadoop cluster, such as we combine between our first data-driven approach and the second workload-driven solution; The fourth contribution has been developed to improve Group-by and aggregate functions, by using a dynamic method, which able to define on the fly the best partitioning scheme of the reducer inputs. To evaluate our approaches, we have conducted some experiments on different cluster sizes, using different data warehouses volumes where the fact table has more than 28 billions of records. We have used the TPC-DS benchmark, a Hadoop-YARN platform, a Spark engine, and Ray and Hive system. Our experiments show that our methods outperform the state-of-the-art approaches in many aspects, especially on the OLAP query execution time.

***Keywords* :** Big Data Warehouse, Partition, Bucket, Load Balancing, Star Join operation, Skipping to load Unnecessary HDFS Blocks, MapReduce, Mappers, Reducers, Shuffle, Multi-agent system.

Résumé

Le partitionnement et la distribution des données ont été largement utilisés dans les systèmes "shared nothing systems", plus particulièrement dans les systèmes distribués qui utilisent le paradigme MapReduce, tels que Hadoop et Spark. Ils ont été utilisés pour l'équilibrage des charges de données, pour éviter le chargement des partitions inutiles et pour guider la conception physique des bases de données et des entrepôts de données distribuées. Pour effectuer des analyses sur les données entreposées, nous utilisons généralement des requêtes OLAP. Une requête OLAP est une requête complexe contenant plusieurs opérations coûteuses, telles que la jointure en étoile, la projection, le filtrage et les opérations d'agrégats. Dans cette thèse, nous proposons différentes approches statiques et dynamiques de partitionnement et d'équilibrage des charges des données sur un cluster Hadoop, afin d'améliorer les performances des entrepôts de données distribuées. Nous avons proposé différents schémas statiques et dynamiques d'un entrepôt de données volumineux distribué sur un cluster de n?uds homogènes ; ce qui peut aider le système distribué à améliorer le temps d'exécution des opérations d'une requête OLAP. Nous avons proposé quatre approches : la première est une nouvelle stratégie de placement de données, pour permettre à un système de traitement des requêtes d?exécuter la jointure en étoile en un seul cycle MapReduce, sans la phase de shuffle. Dans la deuxième contribution, nous proposons différentes techniques de partitionnement et de bucketing pour éviter le chargement inutile de certains blocs de données HDFS et pour améliorer le traitement parallèle, en utilisant un modèle piloté par une charge de requêtes. Dans la troisième approche, nous proposons une nouvelle conception physique d'un entrepôt de données volumineux distribué sur un cluster Hadoop, en combinant la première approche basée sur les données et la seconde solution qui repose sur une charge de requêtes. La quatrième contribution a été développée pour améliorer les fonctions de groupement (*Group-By*) et d'agrégation en utilisant une approche dynamique capable de définir à la volée le meilleur schéma de partitionnement des reducers. Pour évaluer nos approches, nous avons fait des expérimentations avec différentes tailles de *clusters*, en utilisant différents volumes d'entrepôts de données, et où la table des faits contient plus de 28 milliards d'enregistrements. Nous avons utilisé le benchmark TPC-DS, la plate-forme Hadoop-YARN, le moteur d'exécution Spark, le système Ray et Hive. Les résultats expérimentaux obtenus montrent que nos méthodes sont plus performantes que les approches existantes sur plusieurs aspects, notamment en terme du temps d'exécution des requêtes OLAP.

Mots clés **:** Entrepôt de données massives, Partition, *Bucket*, Equilibrage des charges, Jointure en étoile, requêtes OLAP *MapReduce*, *Mappers*, *Reducers*, *Shuffle*, Système multi-agents.

Dedications

In the thesis of my dearest mother,
To my dear father,
To my dear wife,
To my dear son Amir,
To my dear daughters Asma, Aya, and Zahra,
To my brothers and sisters,
To all my family and friends.

Acknowledgments

Soyons reconnaissants aux personnes qui nous donnent du bonheur; elles sont les charmants jardiniers par qui nos âmes sont fleuries.
Marcel Proust (1871-1922).

I want to give great thanks to my advisors: professor Omar Boussaid and the associate-professor Nadia Kabachi, for introducing me to research and helping me find an interesting problem to work on. Omar and Nadia have always been a source of encouragement and inspiration since the early days of writing the idea on a white board.

I thank professor François Pinet for having done me the honor to preside over the jury of my thesis. I also appreciate the members of the jury, professor Olivier Teste, the professor Salima Hassas, the associate-professor Genoveva Vargas-Solar and the associate-professor Abdelmounaam Rezgui.

I would like to give thanks to the director of ERIC laboratory professor Jerome Darmont and the professor Fadila Bentayeb, the team leader of SID/ERIC laboratory, for providing me all support during my thesis.

I thank all the teachers and managers of the ERIC laboratory and university Lumiere Lyon 2 for all the efforts made for the success and development of the university. A big thank to my colleagues at ERIC laboratory for the pleasant atmosphere of work during the three years of the thesis. I would also like to thank all my friends for making my life enjoyable. I wish I could thank each of you by name individually, but the list will be too long.

Finally, I thank my parents and my wife for giving me their unconditional encouragement and support.

Contents

Table of Contents

List of Figures **iv**

List of tables **vi**

1 General Introduction **1**
1.1 Context 1
1.2 Objectives and contributions 6
1.3 Organization of the thesis 8

2 State of the art **10**
2.1 Introduction 10
2.2 Background 11
2.2.1 Data warehouse design and OLAP query 12
2.2.2 Hadoop ecosystem 14
2.2.2.1 MapReduce paradigm 14
2.2.2.2 Hadoop Data File System (HDFS) 15
2.2.2.3 YARN resource manager 16
2.2.3 Apache Spark 16
2.2.4 Apache Hive and Spark SQL 18
2.2.5 Storage formats supported in HDFS 18
2.3 Partitioning and load balancing of the data 20
2.3.1 Static techniques of PLB of data 21
2.3.1.1 HadoopDB 21
2.3.1.2 Hadoop++ 23
2.3.1.3 CoHadoop 23
2.3.1.4 HaoLap (Hadoop based OLAP) 24
2.3.1.5 Cogset 25
2.3.1.6 Partitioning scheme for OLTP workload 26

2.3.1.7 Max skipping HDFS blocks 26
2.3.1.8 BlinkDB 27
2.3.2 Dynamic techniques of PLB of the data 27
2.3.2.1 Amoeba 28
2.3.2.2 AdaptDB 28
2.3.2.3 SWORD 28
2.3.2.4 Locality aware paritioning in parallel DBMS 29
2.3.2.5 Adaptive MapReduce using Situation-Aware Mappers 30
2.3.2.6 The Partition cost model for load balancing in MapReduce 31
2.3.2.7 LEEN approach 32
2.3.2.8 Partition Tuning-based Skew Handling (PTSH) Approach 33
2.3.2.9 SCID partitioning 34
2.3.2.10 Load balancing reducer loads using multi-agent system 34
2.4 Join Algorithms in the MapReduce Paradigm 35
2.4.1 Repartition join 35
2.4.2 Broadcast and Semi-join 36
2.4.3 Multi-way join 36
2.4.4 Optimize star join operation 37
2.4.4.1 Decrease disk spill and network cost in star join operation 37
2.4.4.2 Single-scan approach 38
2.4.5 Bucket-map join and Sort-merge-bucket join 38
2.5 Conclusion 39

3 Optimize Star Join Operation for an OLAP Query 40
3.1 Introduction 40
3.2 Optimize star join operation approach 43
3.2.1 Notations 43
3.2.2 Construct the buckets 45
3.2.2.1 Selecting #B 45
3.2.2.2 Selecting the bucketed key 46
3.2.2.3 Building the buckets 49
3.2.3 Placement of the Buckets 52
3.2.4 Query Transformation 53
3.2.5 Scalability of our approach using NewKey method 55
3.3 Experiments 56

3.3.1 Experimental Setup 56
3.3.1.1 Data generation 57
3.3.1.2 Implementation 58
3.3.2 Results 59
3.3.3 Discussion 65
3.4 Conclusion 67

4 Skipping Loading Unnecessary Data Blocks and Load balancing 69
4.1 Introduction 69
4.2 PBS approach 71
4.2.1 Problem Formulation 73
4.2.2 Selecting the Frequent Predicates Attributes 73
4.2.3 Creating the Reference Table 75
4.2.4 Partitioning and Distributing the Data Warehouse 81
4.2.5 Scalability of the Approach 82
4.3 Experiments 82
4.3.1 Experiment Setup 83
4.3.2 Generation of Data and the Workload 83
4.3.3 Implementation 84
4.3.4 Experiment Results 86
4.3.5 Discussion 89
4.4 Conclusion 92

5 SkipSJoin: A New Physical Design for Distributed Big Data Warehouse on Hadoop Cluster 93
5.1 Introduction 93
5.2 SkipSJoin Approach 95
5.2.1 Formalization 95
5.2.2 Selecting #B and Bkey 98
5.2.2.1 Selecting $\#B$ 98
5.2.2.2 Adding Bkey column to the fact table 98
5.2.2.3 Adding Bkey column to the dimension tables 99
5.2.3 Selecting the Frequent Attributes 99
5.2.4 Building the Partitions and the Buckets 100
5.2.5 Placement of the Buckets 101
5.3 Experiments 101
5.3.1 Experimental Setup 101
5.3.1.1 Generation of the data and the workload 102
5.3.1.2 Implementation of SkipSJoin 102

5.3.2 Results 103
5.3.3 Discussion 105
5.4 Conclusion 106

6 SGMAS: Optimize Group-By Operation Using Multi-agent System 107
6.1 Introduction 107
6.2 Problem Explanation and Motivation 111
6.3 Our Proposal SGMAS 113
6.3.1 Preliminary Concepts and Notations 114
6.3.2 Architecture of our system 116
6.3.2.1 Knowledge Base (KB) Size 118
6.3.2.2 Equilibrium Measurement 119
6.3.2.3 Avoid Infeasible Calculus 120
6.3.2.4 Architecture Detail 121
6.4 Experiments 126
6.4.1 Experiment Setting 126
6.4.1.1 Implementation 127
6.4.1.2 Preparation the data and OLAP workload 130
6.4.2 Performance Evaluation 131
6.5 Conclusion 135

7 General Conclusion and Perspectives 136
7.1 Conclusion 136
7.2 Perspectives 139

Bibliographie 142

List of Figures

2.1 Example of an OLAP query 13
2.2 Example of data warehouse tables in star schema [91] 13
2.3 YARN architecture [116] 17
2.4 Example of Spark architecture with some components of the Hadoop ecosystem [72]. 19
2.5 HadoopDB architecture [7] 22
2.6 Sessionization without colocation vs. with colocation [35] 24
2.7 A PREF partitioned Database [125] 30
2.8 Adaptive techniques in SAMs approach [117] 31
2.9 LEEN Architecture [73] 33

3.1 The steps of our approach 44
3.2 Building a new dimension with FKey method 51
3.3 Building a new dimension and its buckets with NewKey method . . 52
3.4 Example of how to perform star join with our approach(using NewKey method) 54
3.5 Illustrative example to explain how to deal with DW updates with our method NewKey 56
3.6 Queries runtime in $C1$ with $DW1$ (#B=180). 62
3.7 Queries runtime in $C2$ with $DW1$ (#B=180). 62
3.8 Queries runtime in $C2$ with $DW2$ (#B=630). 62
3.9 Impact of #B on the query execution time in $C1$ with $DW1$ 63
3.10 Impact of #B on the query execution time in $C2$ with $DW1$ 63
3.11 Impact of #B on the query execution time in $C2$ with $DW2$ 63
3.12 Impact of #B on the size of dimensions with DW1 64
3.13 Impact of #B on the size of dimensions with DW2 64

4.1 The different steps of PBS approach 72
4.2 Execution time of the workload W using the three databases DB_CSV, DB_DP and DB_PBS with two modes. 88
4.3 Queries performances with DB_DP and DB_PBS approaches . . 89

5.1 The steps of building SkipSJoin 96
5.2 Runtime of the queries 104
5.3 Execution time of W with different approaches 105

6.1 Distribution of the fragments (key, list(values)), before and after shuffle phase with default hash-partitioning method 112
6.2 Smart distribution of the *fragments* over the reducers in a small-cluster of our system 116
6.3 Different observations captured in sequence times of reducer inputs, using blind hash-partitioning method. 117
6.4 How the system store the previous joint actions of the history queries execution in KB 119
6.5 Example of distributing 5 fragments by 3 AMs upon 2 reducers . . 121
6.6 Queries runtime in different approaches with DW1 132
6.7 Queries runtime in different approaches with DW2 133
6.8 Load balancing of the reducers with DW1 134
6.9 Load balancing of the reducers with DW2 134

List of tables

3.1 Characteristics of DW tables 58
3.2 Notations for the compared approaches 60
3.3 Selected queries 61
3.4 Characteristics of the six selected queries 61

4.1 Examples of queries 75
4.2 Characteristics of data warehouse tables 85
4.3 The queries used in the workload and their use frequency 85
4.4 Reference table RT 87

5.1 Notations for the approaches 104
5.2 Reference table RT 104

6.1 Π_q^e structure 125
6.2 Approach notations 129
6.3 Characteristics of the DW tables 131
6.4 Query tranformation 132

Chapter 1

General Introduction

Without big data, you are blind and deaf
and in the middle of a freeway.
Geoffrey Moore

1.1 Context

PROCESSING massive data is a big challenge for the companies; at the same time, Hadoop becomes a standard platform to process Big Data. Nowadays, many big companies such as Yahoo, Facebook, eBay, and so on, use this platform to store and manage their vast data. The five keys [71] to make Big data a huge business are Volume (have colossal size), Velocity (rapidity of the treatment), Variety (different data sources: text, graph, and so on), Veracity (the quality of the data), and the Value (the business benefit). In our context, we focus on studying how to improve the performances of a distributed big data warehouse over a Hadoop cluster. More precisely, we examine how to optimize OLAP queries processing in a distributed big data warehouse over the Hadoop platform.

A Big Data Warehouse (BDW) can be seen as a massive database that is designed for query and analysis rather than for transaction processing. The size of a data warehouse is between hundreds of gigabytes and hundreds of petabytes or more. A BDW can be modeled using a star schema, snowflake schema, and constellation schema [78]. The most used is the star schema. A star schema consists of one fact table and several independent dimension tables. To do analysis with BDW, usually, we used OLAP queries. OLAP queries are typically expensive queries that take a long time to be executed over distributed BDW. In distributed systems such as Hadoop and Spark, improving OLAP query runtime is not a trivial task. OLAP query is composed of several operations and clauses. Usually, it takes the form of "Select... Function()... From... Where... Join-Predicates...Filters...Group By...". In the MapReduce (MR) paradigm, each clause can be executed in the map phase or in the reduce phase, and each operation needs one or several MR iterations for being performed with a considerable amount of data shuffled among the Datanodes of the cluster. For example, in some cases, the star join operation needs n-1 or 2(n-1) MR cycles [24] where n is the number of tables used by the OLAP query. Some other issues could be addressed when running OLAP queries on distributed BDW, such as scanning unnecessary data partitions, unbalanced in the mapper and the reducer loads, data skew distribution, and so on.

To enhance the performances of an OLAP query and speed up their processing on the cluster, the administrators of the distributed system applied some strategies over the distributed BDW to overcome the above issues. The most known strategy used is the partitioning and the load balancing (or distributing) of the BDW over the cluster' nodes.

There are two types of partitioning (fragmentation): vertical and horizontal partitioning. While vertical partitioning technique is used generally to improve join operation in Relational Database Management System (DBMS) [8, 14, 100] and in NoSQL DBMS [22], Horizontal Partitioning (HP) has widely served for many purposes, especially on the parallel DBMS [29, 79, 92, 125] and distributed systems such as Hadoop MapReduce and Spark [33, 35, 37, 80, 82, 108, 110, 115]. The HP techniques have used to skip loading irrelevant data partition in distributed BDW [29, 79, 92, 103, 108], to improve join operation [9, 14, 15, 20, 22, 24, 33, 35,

41, 75, 82, 96, 127], for load balancing and handling data skew in homogeneous and heterogeneous environment [16, 27, 37, 40, 73, 80, 88, 89, 98, 110, 117, 120, 125], and to guide the physical design of distributed databases or BDWs [7, 11, 19, 106, 115]. In the HP, we can use hash-partitioning or range-partitioning method, and we can combine between the two processes. Hash-partitioning is the most useful method in the current version of some frameworks such as Hive, Spark SQL, and Pig. With this technique, we can partition and bucket the DW tables as we need. We will use this technique widely in this thesis.

Generally, in a distributed system, partitioning the tables is followed by spreading or load balancing of the fragments created over the cluster nodes. The objective is to parallelize the treatment and to speed up the query processing. We can distinguish two categories of Partitioning and Load Balancing (PLB for short) of the data [42] used in the distributed systems: static and dynamic. In static PLB technique or off-line PLB, we partition and distribute the BDW before the treatment (i.e., pre-processing). This category requires the prior knowledge of DW scheme and some other information about the treatment. Alternatively, in dynamic technique, the PLB of the data may execute online and at the moment of the treatment, namely, when a user launches the query, the system elaborates on the fly the physical plan of the query by partitioning and distributing the data over the cluster' nodes.

Both static and dynamic PLB techniques are based on three models: the data-driven model, the workload-driven, and the hybrid model. In data-driven model, the PLB do independent of the used query workload, whereas in the workload-driven model the PLB is based only on the workload used, and in the hybrid approach we combine the two models. Each model has its advantages and its limitations as we will show in the next chapters of this thesis. We should note that our physical design for a distributed BDW proposed in this thesis, which based on the different techniques of the PLB of the data, helps the query optimizer to make the best physical plan of the query processing.

There are numerous technologies in distributed systems; the most known is the parallel DBMS and the systems that used the MR paradigm (such as Hadoop,

Spark, Tez, and so on). Both technologies are robust and high-performance computing platforms. The parallel DBMSs outperform the MR system in some applications [128]; however, MR frameworks provide a more fault tolerance model than parallel DBMSs. Moreover, the MR of the Hadoop system is much scalable than all commercial parallel DBMS. Also, DBMSs require that data conform to a well-defined schema, whereas MR permits data to be in any arbitrary format, that is, the MR programmer is free to structure their data in any manner. Furthermore, in parallel DBMS, and since the programmers only need to specify their goal in a high-level language, they not burden by the underlying storage details, such as indexing options and join strategies. In this thesis, we propose new approaches using Hadoop platform with Spark as an execution engine which also based on the MR paradigm with more flexibility. We should note that MapReduce is complementary to DBMSs, not a competing technology.

In distributed systems, we can distinguish four sources of imbalance [42] that can slow down the query processing: (1) imbalance due to partition sizes of the split inputs of the BDW tables, stored as data blocks in HDFS. The split inputs are the HDFS chunks scanned and uploaded by the mapper tasks. Unbalanced in the distribution of the split inputs can drastically degrade the performances of the parallel processing of a query being executed on a distributed system (particularly, in the first map tasks); (2) imbalance due to partition sizes of the reducer inputs. Generally, an OLAP query is performed in multiple MR iterations or Spark stages. Hence, balancing the split inputs is not enough to achieve high performances. So, the system should balance the reducer loads to enhance query processing; (3) imbalance due to heterogeneous of nodes (obviously, machines with high performances are faster than computers with low performances); and finally (4) imbalance due to computations. In some scientific domain, some partitions take more execution time than others although they may have the same tuples' numbers, e.g., using a nonlinear function in the reducer phase. In this thesis, we tackle the issues (1) and (2) on a cluster of homogeneous nodes using the Hadoop ecosystem.

In the Hadoop ecosystem, scanning a large number of HDFS blocks of a large DW table is a costly task that can slow down the query processing. We can skip

loading some unnecessary data blocks if we partition or index some DW tables with the appropriate predicate attributes used in the query's filters. However, the way of selecting the candidates' attributes remains a challenging task. In this thesis, we show different approaches that tackle this problem, and we propose some techniques to overcome this issue. In other words, we show how to manage the metadata table of a distributed BDW, persisted in the memory by the Namenode of the cluster.

As we have explained earlier, while the data split inputs of distributed BDW tables can be balanced by using a static technique of the PLB of the data (based on data-driven or workload-driven model) since we know in advance the load balancing decision. However, some operations of an OLAP query such as Group by and aggregate functions, which performed in the reduce phase, cannot be optimized without using a dynamic PLB technique, because we cannot collect some relevant information that can help the system to balance the reducer loads, only at the query runtime. Namely, we must pick up some relevant information on the fly to make good partition scheme of reducer inputs.

We can distinguish two categories of the approaches that deal with balancing reducer loads problem. These that balance the reducer inputs in advance [27, 37, 40, 73, 98, 110, 120] using, for example, sampling method or on-line strategic, and those attempt to balance the reducer loads after start performing the reducer function [16, 80], namely, after the shuffle phase. Some works of these categories [40, 73] have changed the default synchronization mechanism of the MapReduce paradigm, using a monitoring system or Multi-Agent System (MAS), to balance on the fly the reducers loads. Whatever the technique used to balance the reducer loads, the balancer algorithm used must be executed in real-time and should not affect the query runtime itself.

In this thesis, we survey these dynamics strategies, and we propose a new smart strategy based on MAS, to balance on the fly the reducer loads without change the default mechanism of the MR paradigm, which can improve the Group-By execution time.

1.2 Objectives and contributions

The main objective of our work is to propose static and dynamic techniques of PLB of data to improve OLAP query performances in distributed BDW, modelized in a star schema. We close to use horizontal fragmentation for all our proposals. These methods are based on data-driven, workload-driven, and hybrid model (data + workload-driven). The methods proposed to revolve around four main axes:

1. Propose an approach based on a data or schema-driven model, using a static technique of PLB of the data, for balancing the data split inputs of distributed BDW over a cluster of homogeneous nodes.
2. Propose an approach based on a workload driven model, using a static technique of PLB of data, to manage the metadata of the DW tables and avoid loading some unnecessary data blocks.
3. Propose a hybrid approach (combine between data-driven and workload-driven strategies) to build a new physical design of distributed BDW over Hadoop cluster (combine between approaches 1 and 2).
4. Propose an approach based on a data-driven model, using dynamic PLB technique, to balance on the fly the reducer loads of an OLAP query being executed on a cluster of nodes, using Multi-Agent System (MAS).

In our contributions, we take into account several parameters such as DW size, skew distribution of the values of the primary keys and the foreign keys of the fact and dimension tables, and the cluster's nodes characteristics (number of CPU cores, memory size, and disc size, and so on). We tune these parameters to make a good partition scheme of the split inputs and the reducers loads, and to minimize the overhead in the distributed system which can enhance the performances of the OLAP workload.

The objective of the first contribution is to optimize the star join operation. This operation is the most expensive in an OLAP query and can create a bottleneck

in the network due to the amount of data shuffled. Usually, this operation requires multiple MR iterations to be performed, which can significantly slow down query processing. We propose a heuristic technique of PLB of data, in which we can execute this operation in only one MR iteration or Spark stage. Also, we show how to improve this operation without considering the workload used. Namely, our approach is based solely on the data-driven model. With this solution, many issues have been tackled such as (1) the partitions' number problem (i.e., number of the fragments or the buckets), should be selected to partition the DW tables; (2) the appropriate attribute selected to hash-partition the fact and the dimension tables; and (3) how to build and spread the buckets of the fact table and the dimensions to minimize data skew distribution and to enhance the parallel treatment.

The objective of the second proposal is to skip loading unnecessary HDFS blocks when scanning BDW tables, and avoid obtaining big partitions of some tables, which can disrupt the parallel processing. In this contribution, we trade-off between the number of table's partitions created and their sizes. We propose a PLB technique based on the query workload, namely, using a stable workload. In this approach, we combine between partitioning and bucketing technique such that, we partition some DW tables by the frequent predicate attributes used in the queries filters, and we bucketed some other tables by some other columns (e.g., bucketed table by its primary key).

In the third proposal, we combine between the two first approaches (data-driven and workload-driven model). So, we propose a new physical design for distributed BDW, in which we can perform the star join operation in a single Spark stage, and the system can skip scanning some irrelevant HDFS blocks of some DW tables.

The fourth contribution is different to the precedent ones. In this approach, we propose a new dynamic technique of PLB of data, using MAS, to balance on the fly the reduce loads. The main objective of this solution is to optimize Group by clause of an OLAP query and aggregate functions. This operation also incurs a high rate of shuffle and degrade the parallel treatment. So, balancing the intermediate results is also a mandatory requirement to improve this operation and the query execution time. The main key of this proposal is that we don't make any

changes about the default mechanism of MR paradigm as some approaches have done in [37, 40, 73]. In our method, we collect on the fly some relevant information through the history execution of the OLAP queries, which can help the system to elaborate a good partition scheme of the reducer loads. In our proposal, we tackle the problem of data skew distribution in the reducer phase. Our solution based on a data-driven model, such we do not assume any assumption about the query workload used.

We propose an approach to spread smartly the clusters (keys, list(values)), produced by the mapper outputs (after execute filtering and the star join operation), over the partitions of the reducers, using a knowledge base component, that can help the system to make the best load balancing decision. Moreover, to implement our MAS, we have used Ray [85] framework, which is a hybridization between Bulk-synchronous parallel systems such as MapReduce and Apache Spark [123], and the actor model for asynchronous tasks such as Akka [101].

1.3 Organization of the thesis

The thesis is organized into seven chapters.

Chapter 2 presents the state of the art of static and dynamic techniques used to improve DW performances in a distributed system such as the Hadoop ecosystem and Spark. We start by background about the distributed system technologies such as Hadoop, Spark, MapReduce, and so on, and we present the different storage formats supported by Hadoop which can use to split and storage the DW tables, such as the column format Parquet and ORC. Then, we show the different static and dynamic techniques of the PLB of the data, existed in the literature. We detail some join algorithms such as shuffle hash join, broadcast hash join, and other algorithms such as multi-way join and trie join, and hyper join. We also show different physical design proposed which deal to the problem of data skipping, load balancing, and optimize OLAP query processing. In the dynamic technique, we focus particularly on the works that deal about data skew and balancing the

reducers loads.

Chapter 3 exposes the problem of optimizing star join operation of an OLAP query, the complexity of this operation. Then, we detail our data placement strategy to optimize this operation based on the foreign keys of the fact table, primary keys of dimensions, and the physical characteristics of the cluster's nodes.

In Chapter 4, we propose our partitioning and bucketing techniques to skip loading unnecessary HDFS block through a stable query workload. Such that, we propose generals rules to speed up OLAP query processing in a distributed system such as Spark and Hive, through the attributes used in the filters of an OLAP workload.

In Chapter 5, we present our physical design of distributed BDW, such we combine between the data placement proposed in chapter 3, and our skipping technique proposed in chapter 4 to avoid scanning some unnecessary data partition of some DW tables.

Chapter 6 exposes the problem of balancing reducer inputs and handling data skew to improve Group by operation and aggregate functions, namely, we detail our heuristic strategy which based on the cooperative agents to balance on the fly the clusters(key, list(values)), produced after executing filters and star join operation of an OLAP query.

Note that, in the previous chapters, i.e., from chapter 3 to chapter 6, we detail our contributions with an experiments section, where we show how to implement the different methods proposed, and how to evaluate them using a numerous approaches of the state-of-the-art.

We conclude the thesis in Chapter 7, where we summarize the different proposed approaches, their advantages and their limitations, and finally, we donate our perspectives and the future works.

Chapter 2

State of the art

Big data is at the foundation of all of the megatrends
that are happening today, from social to mobile
to the cloud to gaming.
Chris Lynch.

2.1 Introduction

Partitioning and distribution techniques applied to manage a BDW upon the cluster' nodes have been widely studied in shared nothing system [9, 11, 14, 16, 20, 22, 24, 29, 33, 35, 40, 73, 75, 79, 80, 92, 108, 110, 115, 117, 125], such as parallel DBMS and the Hadoop ecosystem. They have used for many purposes, such as load balancing, for skipping to load unnecessary data partitions, and for guiding the physical design of DW. As we have seen earlier in chapter 1, the DW can be modeled by three models. The most used in the literature is the star-schema with one fact table and many dimension tables. Data warehouses are dedicated to the analysis and decision-making applications. This analysis is often performed through OLAP queries. An OLAP query is composed of many operations such

as projection, filtering, star join, grouping, and aggregate functions. Performing such queries on distributed BDW, often take a very high response time that is not acceptable to decision-makers who require reasonable response time to meet decision-making needs. So the administrator must use different techniques for the partitioning and load balancing of the data to decrease this time.

Executing an OLAP query with MapReduce paradigm requires many iterations and multiple shuffle phases, and incur high communication cost. Also, in distributed BDW, the problem of disc I/O and CPU overhead must be taken into account, especially with a complex OLAP query. So, almost all the proposed works that tackle OLAP query optimization in distributed BDW over Hadoop cluster focused on minimizing the costs cited earlier. As we have presented in the previous chapter, we can distinguish two types of the PLB of data that can use to optimize the query processing over a distributed BDW: static and dynamic techniques. Before detailing the different approaches that deal the problem of the PLB of the data and the different join algorithms in the distributed systems, we present some background about data warehouse concepts and design and the star join schema, OLAP query, the different distributed systems technologies such as Hadoop system, MapReduce paradigm, Spark engine, and the resource manager YARN. Moreover, we present some storage formats supported by HDFS, used to store the relational DW tables.

2.2 Background

We present in this section the data warehouse architecture, star join schema, the structure of OLAP queries, the feature of Hadoop platform, MapReduce paradigm, Spark engine, Hive, and Spark SQL query processing system, the different storage formats supported in HDFS, more precisely column storage format, and we explain the execution mechanism of the resource manager Hadoop-YARN.

2.2.1 Data warehouse design and OLAP query

Data warehouse systems have become a key component of the corporate information system architecture, in which they play a crucial role in building business decision support systems [78]. Nowadays, data warehousing technologies are successfully used in many industries, including retail, manufacturing, financial services, banking, telecommunication, healthcare, and so on. We can consider a data warehouse as a large database system with additional functionality; however, some well optimizing techniques, such as indexing, data partitioning, materialized view, and query optimization, are applied differently in the data warehouse systems. Moreover, some other operations, such as data cleaning, data refreshment, and multidimensional and parallel query optimization, are specific to the data warehouses only. Usually, the size of a data warehouse is between hundreds of gigabytes and hundreds of petabytes.

A data warehouse can be modeled using a star schema that is the most commonly used, snowflake schema, and constellation schema. A star schema consists of a central table, namely fact table, referencing several dimension tables, thus resembling a star. Figure 2.2 shows a data warehouse architecture in star schema.

The content of a DW is analyzed by Online Analytical Processing (OLAP) applications to discover trends, patterns of behaviors, and anomalies as well as for finding hidden dependencies between data. OLAP queries are typically expensive queries that take a long time to be performed over distributed Big Data Warehouses (BDW). In distributed BDW, improving OLAP query processing is not a trivial task. An OLAP query is composed of several clauses, and it usually takes the form of "*Select..Function()..From..Where...Join-Predicates...Filters...GROUP BY..*". Each clause is performed in the map phase or in the reduce phase, and each operation may need one or several MR iterations or Spark stages to be executed, with a considerable amount of data shuffled among the data nodes. Figure 2.1 shows an example of OLAP query extracted from TPC-DS [48] benchmark. To improve the OLAP query runtime, several techniques of Partitioning and distributing applied to improve DW performances as we will present in this thesis.

```
SELECT d_year, i_brand_id, SUM(ss_sales_price)
FROM  date_dim, item, store_sales
WHERE date_dim.d_date_sk = store_sales.ss_sold_date_sk
AND store_sales.ss_item_sk = item.i_item_sk
AND i_manufact_id = 128
GROUP BY d_year, i_brand_id;
```

Figure 2.1: Example of an OLAP query

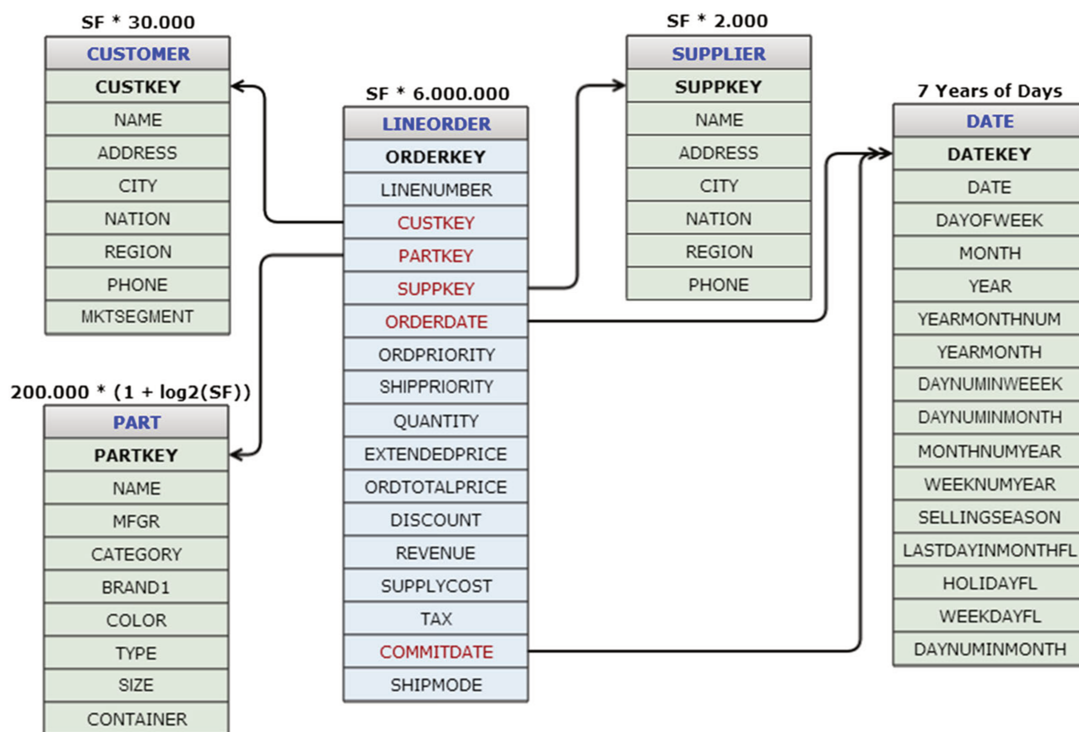


Figure 2.2: Example of data warehouse tables in star schema [91]

2.2.2 Hadoop ecosystem

Hadoop [60] traces back to 2004 when Google published the second of a pair of papers describing two of the key ideas behind its search success. The first detailed the Google File System [38], or GFS, as a way of distributing data across hundreds or thousands of inexpensive computers. To glean insights from that data, a second tool, called MapReduce [30], which is a programming model that enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity nodes [121].

Yahoo is the main contributor to the open-source HDFS, an open-source version of Googles GFS [38], together with an open-source version of MapReduce [30], and a SQL-like query interface called Pig [90]. After that, Apache Hadoop becomes support other SQL interfaces like Hive-QL [112] and Spark SQL [13]. The first version of Hadoop is able only to execute one framework at once time with MR paradigm, until 2013, where Vavilapalli et al. [116] propose YARN module, a new resource manager for Hadoop, that support not only MapReduce engine but other flexible engines like Spark [123], Tez [102], and so on.

In the following, we give some details about the MapReduce paradigm, Hadoop Data File System (HDFS), and the resource manager YARN.

2.2.2.1 MapReduce paradigm

In recent years the MR framework [30, 31] has emerged as one of the most widely used parallel computing platforms for processing data on terabyte and petabyte scales. MapReduce is substantially different from previously analyzed models of parallel computation because it interleaves parallel and sequential computation [76]. In MR programming paradigm, the basic unit of information is a (key; value) pair where each key and each value are binary strings. The input to any MR algorithm is a set of (key; value) pairs. Operations on a set of pairs occur in three stages: the map stage, the shuffle stage, and the reduce stage. In the map stage,

the mapper μ takes as input a single (key; value) pair and produces as output any number of new (key; value) pairs. During the shuffle stage, the underlying system that implements MR sends all of the values that are associated with an individual k key to the same machine. In the reduce stage, the reducer p takes all of the values associated with a single key k and outputs a multi-set of (key; value) pairs with the same k key. Note that the map and reduce stage perform in parallel over the cluster' nodes.

Many modern parallel engines like Spark, Tez, and so on, adapt the global mechanism of MR with more flexibilities. In the version V-1.x of Hadoop, we can only use MR, from version V-2.x, Hadoop supports other execution engines. The standard Hadoop MR can slow down the query processing since it requires to save intermediate results in the disc for each MR cycle. To overcome this issue, a new execution engine like Spark and Tez works in-memory and exploit better the node' resources (CPU cores and memory). In some case, these engines improve application runtime up to 100 times than Hadoop MR. In this thesis, we finish using the Spark engine, which is adapted and used by many companies in the world [70]. In Sect. 2.2.2, we detail the features of this engine.

2.2.2.2 Hadoop Data File System (HDFS)

The Hadoop Distributed File System (HDFS) [21, 104] is a distributed file system designed to store massive data sets and to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. In HDFS, files can be written only once, and updates of existing files are not allowed. HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode (multi NameNode from HDFS V-2.x), a master server that manages the file system namespace and regulates access to files by clients. Also, there are several DataNodes, usually one per node in the cluster, which manages storage attached to the nodes that they run on.

HDFS is designed to reliably store very large files across machines (nodes) in a large cluster. It stores each file as a sequence of blocks; The blocks of a file are replicated for fault tolerance. The block size and replication factor (by default 3) are configurable per file. In Hadoop V-1.x, the size of the block was 64 MB; From Hadoop V-2.x, the size of the block become 128 MB. HDFS support different storage formats [12, 94, 114], and each block can split in many chunks. We will explain some of these storage formats in Sect. 2.2.3.

2.2.2.3 YARN resource manager

As we have noted earlier, Hadoop V-1.x [34] Supports MapReduce processing model only and does not support non-MR tools. From V-2.x [116], YARN module is added to Hadoop distributed system, which allows Hadoop working with MR as well as other distributed computing models like Apache Spark [66], Apache Tez [69], and Apache HBase [61] processing engine. Hadoop-yarn architecture separates resource management functions from the programming model. Namely, MapReduce is just one of the applications running on top of YARN. Figure 2.3 shows the global architecture of Hadoop YARN. YARN controls resource management, scheduling, and security when we run applications on it. By default, Hadoop-YARN uses MapReduce (MR) paradigm (see Fig. 2.3). When a user submits an application, the RM (i.e., YARN) allocates to the Application Master (AM), which represent the main of the application, a container in any slave-node for being performed. The AM handle the job's tasks which run on the other containers of the Node-Managers. In this thesis, all our contributions are developed using the Spark engine on Hadoop-YARN platform. In the next section, we detail Spark features.

2.2.3 Apache Spark

Apache Spark [66] is an open-source framework for big data processing. It has emerged as the next generation big data processing engine. In Wikibon site [70],

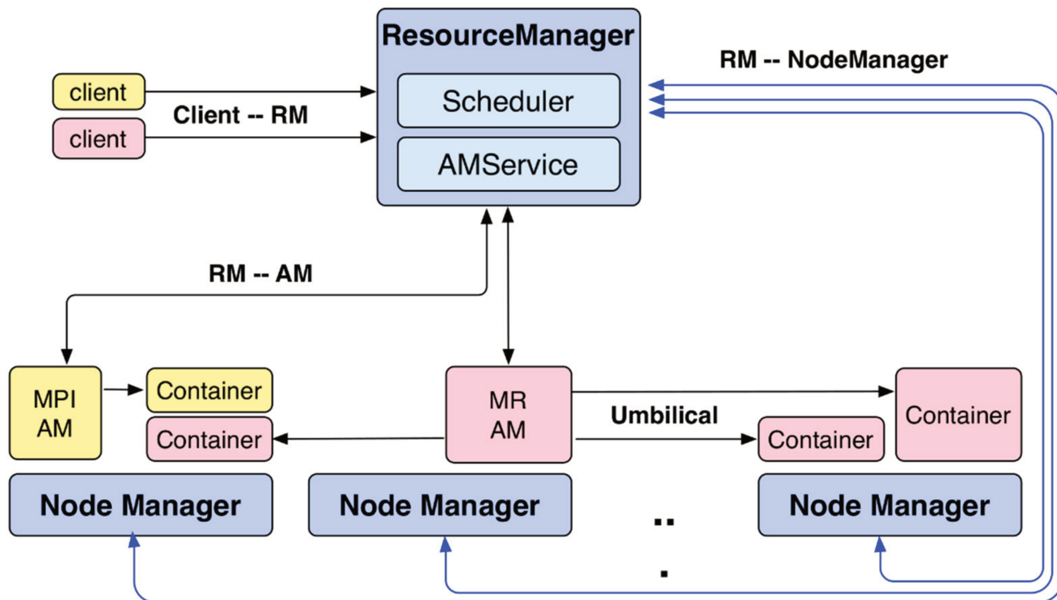


Figure 2.3: YARN architecture [116]

some analysts predict that Apache Spark will account for one third (37%) of all the big data spending in 2022. Spark [123] maintains Hadoop MapReduce's linear scalability and fault tolerance, but extends it in a few important ways: (1) it uses a new concept of resilient distributed datasets (RDDs [124]) which make it faster for processing large volume of data leveraging distributed memory; (2) it is easier to program with its APIs in Scala, Java, Python, and R; (3) it supports a variety of compute-intensive tasks, including interactive queries, streaming, machine learning, and graph processing.

We can run a Spark application on three cluster's RM: YARN, Mesos, and Spark standalone [68]. In Hadoop YARN, we can submit a Spark application on two deploy modes, YARN-client and YARN-cluster mode. In client-mode, the driver component of spark job will run on the machine from which job submitted, this mode is suitable when the job submitting machine is within or near to the cluster infrastructure. In this case, there is no high network latency of data move-

ment for final result generation between cluster platform and driver. In contrast, in cluster-mode, the driver component of spark job will not run on the local machine from which job is submitted but with "Application Master". In this case, network disconnection between driver and cluster infrastructure reduces. In this thesis, all our experiences with Apache Spark done on YARN with client-mode.

2.2.4 Apache Hive and Spark SQL

Apache Hive [62] is a data warehouse system built on top of Apache Hadoop that facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in various databases and file systems. Hive [112, 113] supports queries expressed in a SQL-like declarative language - HiveQL, which are compiled into MapReduce jobs that are executed using Hadoop.

Hive manages the metadata of the DW tables stored in HDFS. Hive uses MySQL or Hbase to store these metadata. In this thesis, we use Spark SQL [13] as query processing which has Catalyst optimizer and Hive to manage the metadata. Spark SQL can use HiveMetastore to get the metadata of the data stored in HDFS. This metadata enables Spark SQL to do better optimization of the queries that it executes (we use Spark as a processing engine).

Figure 2.4 shows an example of Spark architecture with some component of the Hadoop ecosystem [72].

2.2.5 Storage formats supported in HDFS

Nowadays, choosing the optimal file format in Hadoop is one of the essential factors to improve performance for big data processing. We can distinguish two types of storage formats supported in HDFS (see the works of Ahmed, S. et al. [12] and Plase, D. et al. [94]): row-based storage format like text/CSV, SequenceFile, and Avro; and column-based storage format like Parquet, RCFile and ORC (Optimized

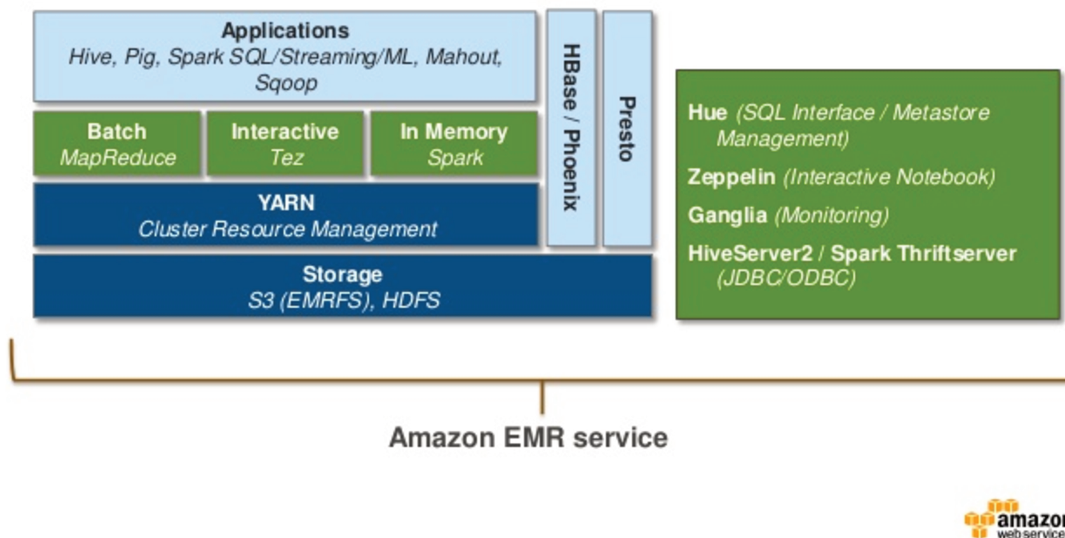


Figure 2.4: Example of Spark architecture with some components of the Hadoop ecosystem [72].

Row Columnar) storage format. Each one has some advantages and limitations as we will show below. Note that almost all of these storage formats are used to store relational DW tables on computer clusters and designed for systems using the MapReduce framework. Moreover, with these formats, we can split data blocks which can speed up parallel processing[1].

Row-based storage format such as text/CSV, SequenceFile are unrecommended formats in Big data processing, especially when using OLAP workload. Since the system must load in memory, all attributes of the DW table when scanning HDFS blocks, another deficiency with this type of format is that storage data with their require a considerable amount of space in disk since they lack to data compression approach. Apache Avro [51] also is row-based binary serialization format; however, it supports block compression. The advantage of Avro format is that stores

[1]Unlike JSON and XML files, where we cannot split data with them, something that can limit the scalability and the parallelism of a distributed system.

metadata with the data but also allow the specification of an independent schema for reading the file. Moreover, Avro format is suitable in OLTP workload where there are more data transactions. Note that the most compatible platform with this format is Kafka [77].

Column-based storage format such as Apache Parquet [64] and ORC [63] are a suitable format for distributed BDW since only the attributes solicited by an OLAP query are loaded into memory, which can speed up the query processing. Parquet is a columnar data storage format [118]; it uses binary data stored in the form of columns. It is very effective and provides the best query performance on the Hadoop platform. Like Avro format, Parquet is supported by Apache Spark framework for the storage of BDW. ORC is also an optimized column-based storage format but has built-in indexes to speed up query processing. It performs better without compression than other data formats in Hive and Spark. It consumes less execution time for reading, writing, and accessing data. This format works well on Hive. Both formats employ compression to reduce the size of files. They support several compression codecs, including Snappy, GZIP, deflate, and BZIP2. With these formats, we reduce the size of the data on the disk and consequently the I/O and CPU resources required to deserialize data.

For more detail, see the works of [12, 94, 114]. In our experiments, we have using Parquet format, which is more compatible with Apache Spark.

2.3 Partitioning and load balancing of the data

In this section, we detail some static and dynamic techniques for the partitioning and load balancing (PLB) of the data in the MapReduce paradigm.

2.3.1 Static techniques of PLB of data

Numerous works used the static techniques of partitioning and load balancing of data in the distributed systems. Data placement and guiding the physical design of the databases and data warehouses over a cluster' nodes have a significant impact on improving query processing in a distributed system. Applied static techniques of the PLB of the data requires in general prior knowledge of DB or DW scheme, the workload used, and the type of the treatment that we want to do (e.g., type of join operation). In static techniques, we make in advance a partitioning scheme of the DW over the cluster' nodes to speed up query processing (generally, we balance the split inputs of the database). In other words, the query optimizer can create efficient query plan through this static physical design, without needs to repartition and redistribute the DW's tables again during the first map stage. Static techniques are based on two models: a data-driven model [7, 15, 18, 19, 33, 35, 115, 88, 89, 106] and a workload-driven model [11, 14, 29, 92, 108].

In this subsection, we detail some works that use the static techniques of PLB to manage the data split inputs over the Hadoop ecosystem and the parallel DBMS.

2.3.1.1 HadoopDB

HadoopDB [7] is an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Figure 2.5 shows the architecture of HadoopDB. The basic idea behind HadoopDB is to connect multiple single-node database systems using Hadoop as the task coordinator and network communication layer. By using the MapReduce framework, queries are parallelized. So, each SQL query is transformed to a sequence of MR jobs. We can install PostgreSQL or MySQL in database system or other flexible framework such as MonetDB [19].

HadoopDB is much better than naïve Hadoop in both scalability and efficiency. Moreover, with this platform, join operation can perform in the map phase without a shuffle phase. However, the results are still not satisfactory for DW applications,

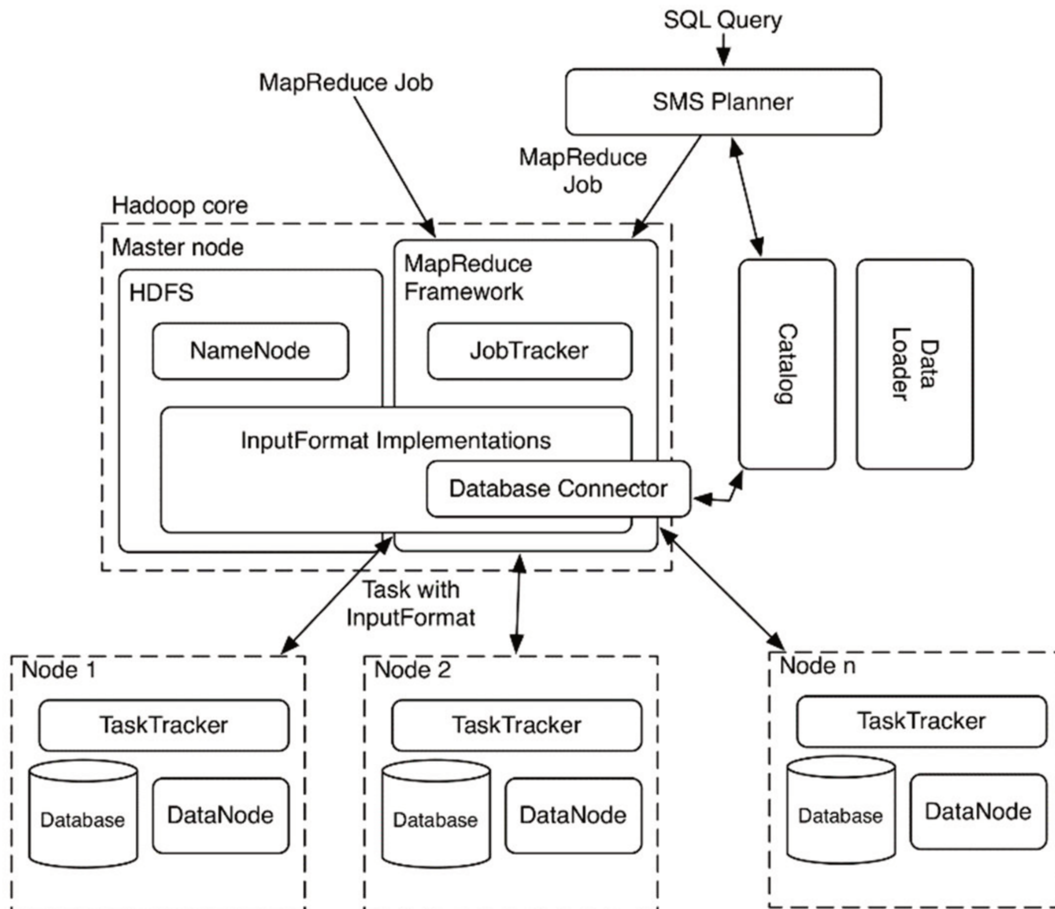


Figure 2.5: HadoopDB architecture [7]

especially in a star join operation with several tables, HadoopDB can lose its performance advantages. If we replicate each dimension table to all the database nodes, this will incur a high space cost, especially with large dimensions. Furthermore, HadoopDB stores the dimensions in a local DBMS and hence disrupts the dynamic scheduling and fault tolerance of Hadoop, and the system takes a long time to pick up some primary keys and join them with the foreign keys of the fact table.

2.3.1.2 Hadoop++

Hadoop++ [33] is an improvement to Hadoop, where we can co-partition related files using a Trojan index. Hadoop++ don't replace the Hadoop framework at all when integrating index Trojan index. Trojan indexes are built on data partitions, which contain meta-data about each of these partitions such as, data set size, index size, and the number of records. Note that Trojan Indexes are created at data load time and so have no penalty at query execution time. The main advantage of Hadoop++ is that we can join two tables in the map phase without a shuffle phase; however, in Hadoop++, we can only co-partition two tables on their join key. This drawback makes this approach not suitable for OLAP query when joining several tables (generally, one fact table and several dimension tables).

2.3.1.3 CoHadoop

CoHadoop [35] is an extension of Hadoop++ [33]. With CoHadoop, we can co-partition many log files on their join key. Moreover, CoHadoop places automatically the blocks files that have the same join key in the same node. With CoHadoop data placement, the query optimizer, such as HiveQL and Spark SQL, exploits this fact to generate efficient query plans. In other words, with CoHadoop physical design, the system can perform join operation (of log files) locally, in map phase, and without shuffle phase. The experiences show that CoHadoop enhances log processing better than Hadoop, Hadoop++, and HadoopDB. Figure 2.6 shows an example of data loading in CoHadoop.

Although CoHadoop is efficient in a small cluster, however, when the cluster is large, co-locating log files will be problematic, such that if one or many nodes crashes, the system needs to postpone the user transactions to launch the co-locating algorithm. This can harm the system. So adapting an offline process to gather some chunks that have the same join key in the same node would be better in some cases, see the work of Nonava [89]. Another drawback of CoHadoop is that it targeted nodes randomly for every new key (a key is the join condition).

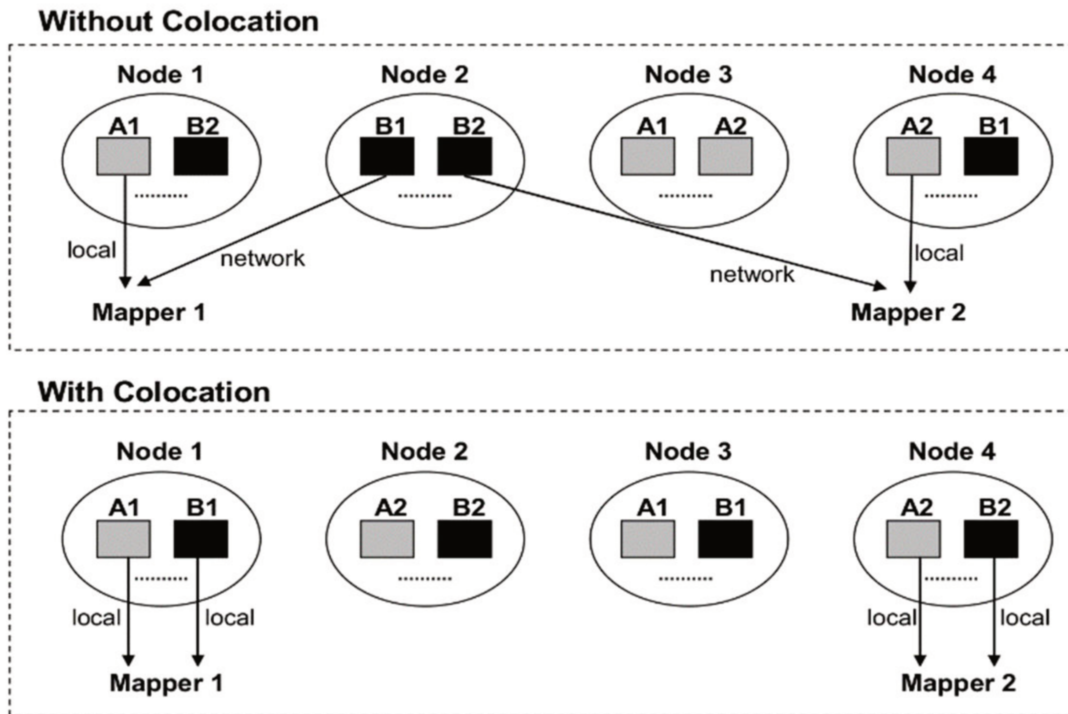


Figure 2.6: Sessionization without colocation vs. with colocation [35]

This problem can create an imbalance in the cluster's nodes. To overcome this issue, CoHadoop++ [88] has extended CoHadoop method to manage better the load balancing of the distributed system. Unfortunately, we should note that both methods of the data placement (i.e., [35, 88]) are not efficient to enhance OLAP workload performance, especially in the star join operation.

2.3.1.4 HaoLap (Hadoop based OLAP)

Haolap [106] adapts MOLAP model rather ROLAP to improve OLAP processing by using a new storage technique to maintain the multidimensional views. In Haolap, the cube is divided in chunks to speed up roll up and drill down operations. Haolop focused on improving selection, data filtering, and data aggregation of an

OLAP query. Haolap uses MOLAP model and may outperform some other MOLAP and ROLAP approaches (such as HadoopDB [7] and Hive [112]) in term of query execution time[2], however in BDW, and since the workload may change periodically, this may take overhead the memory due to the size of cubes maintained. Another drawback of Haolop is that it keeps dimensions in meta-data table and the fact table in HDFS, this is not a realistic solution since some dimension in BDW may have a big size, and the Namenode cannot maintain in memory this huge metadata.

2.3.1.5 Cogset

Cogset [115] is a hybridization between the parallel DBMS and the Hadoop ecosystem as HadoopDB [7], however instead to use DB systems such as MySQL or PostgresSQL, Cogset has adapted only Hadoop system through a library of interface adapters. The idea of Valvåg et al. [115] is to benefice from the efficiency of the parallel DBMS, and the fault tolerance and load-balanced parallel processing of Hadoop MapReduce. Cogset ensures co-locating files which can process join operation locally (log files), The experiments done by Valvåg et al. revealed that Cogset outperform native Hadoop (without considering the current storage format such as Parquet and ORC), HadoopDB [7], and some commercial parallel database system (e.g., Vertica [49]), in some MapReduce tasks such as joining two tables, full scan table, and some aggregate functions.

The scalability of Cogset remains questionable since it has tested in a cluster of 25 nodes, Moreover, for sophisticated star join queries with several dimensions, Cogset still needs multiple MapReduce cycles to perform star join operation, and the chained declustering [44] distributing strategy used cannot ensures collocating the tuples that contain the foreign keys of a fact table and the primary keys of the dimension tables in the same node, as our partitioning scheme can do, see our contribution in Chapt. 3.

[2]Since it avoids the join operation which considers as a heavy task when big data are involved [107]

2.3.1.6 Partitioning scheme for OLTP workload

Curino et al. [29] have developed Schism project, a novel graph-based, workload-driven partitioning system for transactional workloads, designed for disk-based shared-nothing databases. The authors use static partitioning technique which based on a stable workload, and their approach can reduce significantly the cross-machine transactions for OLTP workloads in parallel databases systems. Schism tackles two main issues: (1) distributed transactions and (2) temporal workload skew. To overcome these drawbacks, Pavlo et al. [92] have developed Horticulture system, which is a scalable tool to generate database designs for stored procedure-based parallel OLTP systems automatically. We should note that the goal of the partitioning for OLTP workload is not the same as in OLAP workload, such that in large-scale analytical applications (i.e., data warehouses) the objective is to distribute data across nodes to maximize intra-query parallelism; in contrast, to achieve high consistency criteria in OLTP workload, we need multi-node coordination to perform the transactions. In our thesis, we focus on enhancing OLAP query performance in a distributed big data warehouse over a Hadoop cluster.

2.3.1.7 Max skipping HDFS blocks

Sun et al. [108] have proposed a new algorithm to tackle the problem of Balanced MaxSkip Partitioning. The bottom-up clustering method used by the authors, which based on the hyper-graph technique, can max skip loading the tuples of the blocks through a given workload, and can also parallelize the treatment since they study the balancing constraint of the block sizes. Although the approach used by Sun et al. can achieve max level of skipping loading HDFS blocks, however, they use a sophisticated clustering algorithm (based on Graph theory) that usually takes a long time to be performed by the system. Moreover, the authors make some assumptions to the occurrence rate of the same filters used in the workload. In this thesis, we use a smooth technique based on data mining measures to skip loading some unnecessary HDFS blocks, without making any assumptions about the filters used in the stable workload.

2.3.1.8 BlinkDB

BlinkDB [11] is a project to improve the ad-hoc and OLAP workload execution time in distributed big data warehouses. AGARWAL et al. [11] have used a sampling method to response swiftly a complex query on a big data environment. BlinkDB allows the users to introduce SQL query and perform it online (not as HaLoap [106] project, which requires to pre-build the cube in advance). BlinkDB is more adaptable for a stable workload (i.e. the set of columns used in *Where* and *Group By* clauses remain fairly stable over time), which allows selecting the best sample that gives best accurate results. The main drawback of BlinkDB is that it can provide inconsistent results due to the sampling method used.

2.3.2 Dynamic techniques of PLB of the data

In the previous subsections, we have seen some static data placement strategies in the parallel DBMS systems and the Hadoop ecosystem. Before detail some approaches that used the dynamic techniques of the PLB of the data, we should distinguish between two categories of these techniques: In the first category, the system can incrementally rebuild a new data placement of a database according to the change of the workload [79, 82, 103, 125] and the techniques of the second category are used to partition and distribute the data partitions, at the moment of the query processing [16, 27, 37, 40, 73, 80, 98, 110, 117, 120]. Note that the methods of the first category produce a new permanent scheme of the database or the DW and are suitable much more in parallel DBMS since the transfer of HDFS data blocks is a difficult operation in the current versions of the Hadoop ecosystem. The partitioning algorithms of the second category produce temporary schemes, which usually, are served to reorganize the intermediate results and load balancing the reducer loads, for a query being executed over a distributed system.

In the following, we detail some dynamic approaches cited earlier.

2.3.2.1 Amoeba

Amoeba [103] used Hyper-partitioning technique to avoid scanning some unnecessary HDFS blocks based on the query workload, something that can speed up query processing. Amoeba approach, versus to some state-of-the-art methods [29, 92, 108], doesn't assume any assumption about the query workload used (Ad-hoc or OLAP workload), it adapts the partition scheme incrementally as more and more queries are observed (frequent queries). The main drawback of Amoeba is that it is not suitable to enhance join operation, which is considered the most costly task in analytic queries.

2.3.2.2 AdaptDB

To improve Ameoba [103] approach, Lu et al. [82] have proposed a new partitioning scheme called AdaptDB. AdaptDB repartitions datasets automatically to get the best performance on frequent join queries. The authors have developed a new algorithm called hyper-join that based on two techniques: two-phase partitioning and smooth repartitioning. The challenge of the hyper-join method is not only minimized the network communication cost during join operation but finding optimal splits to partition the data warehouse tables, something that allows reducing the total amount of disk I/O (when scanning HDFS blocks). Although AdaptDB is suitable for the three DW schemas, namely star schema, snowflake schema, and constellation schema, however, this scheme still needs many MapReduce iterations to perform the star join operation which can increase the query processing time.

2.3.2.3 SWORD

Kumar et al. [79] have proposed a new data placement strategy and replica mechanism in the cloud computing to improve OLTP workload runtime, and to minimize the resources consumed in executing the OLTP workload. For this, the authors focus on handling the query span (the query span is the average number of ma-

chines involved in the execution of a query). Compared to the previous works (e.g., Schism approach [29] and Pavlo et al. [92]), in SWORD project, the authors have developed a heuristic algorithm to handle the workload change and to rebuild a new database scheme automatically, according to the current workload used, without interrupting the users' transactions. Unfortunately, we should be noted that the method used in SWORD is suitable in parallel DBMS and we cannot apply straightforward in Hadoop technology due to the mechanism storage of HDFS.

2.3.2.4 Locality aware paritioning in parallel DBMS

Zamanian et al. [125] have proposed a novel partitioning scheme called Predicate-Based Reference Partition (PREF) that allows to co-partition sets of tables based on given join predicates. Their approach is based on a hybrid model (i.e., data-driven and workload driven models). The authors propose two partitioning schemes: in the first one, the system partitioned the database according to the schema of tables and the data, and to build the second scheme, the system takes into account also the query workload used to reduce the search space when scanning the partitions. PREF is designed for analytical workloads (not suitable for OLTP workload) and work well for simple schema DW with uniformly distributed data (such as TPC-H) and a complex schema with skewed data (such as TPC-DS [48]). An example of how PREF partition database is illustrated in Fig. 2.7.

The main objective of PREF architecture is to **maximize data-locality** while **minimizing data-redundancy**, something that look for in our proposals (see next chapters), such that it can ensure local join between several tables, by partitioning these tables on their join key and duplicate some tuples over the cluster's nodes. PREF approach, and contrary to some static methods[29, 92], it can rebalance the partitions of the table automatically according to the workload used. The experiments done by Zamanian et al. revealed that PREF is more efficient in equi-join, anti-join, and outer join operation, however, in some case, either it may need to the full replicate a table (such in some sophisticated star join query), or it is obliged to perform the join remotely (such in Cartesian join and Theta join), this makes the generality of the approach remains questionable.

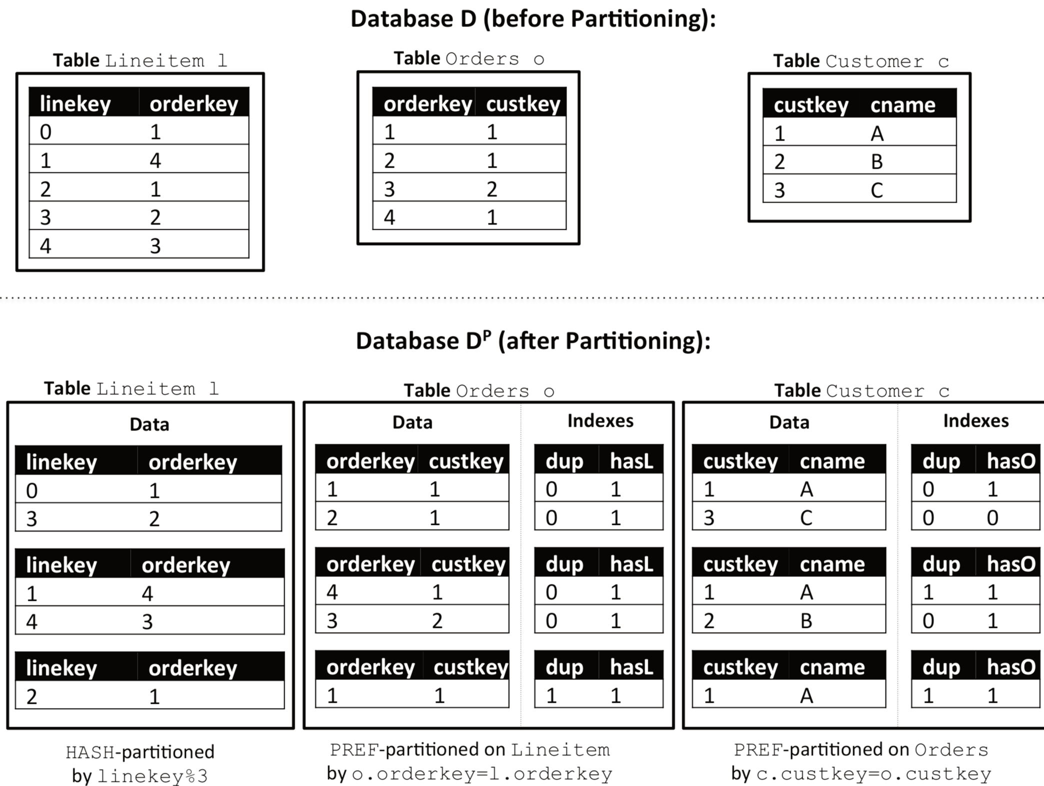


Figure 2.7: A PREF partitioned Database [125]

Moreover, the technique used in PREF solution to duplicate the tuples over the cluster's nodes is implemented in a parallel relational DBMS and is not allowed in the current version of the Hadoop ecosystem, due to the mechanism of write once and read many of HDFS.

2.3.2.5 Adaptive MapReduce using Situation-Aware Mappers

Usually, in the MapReduce paradigm, the mapper tasks execute independently; namely, any of the mappers know nothing about what the other mappers do. This situation can slow down the mapper tasks due to the imbalance in the mapper

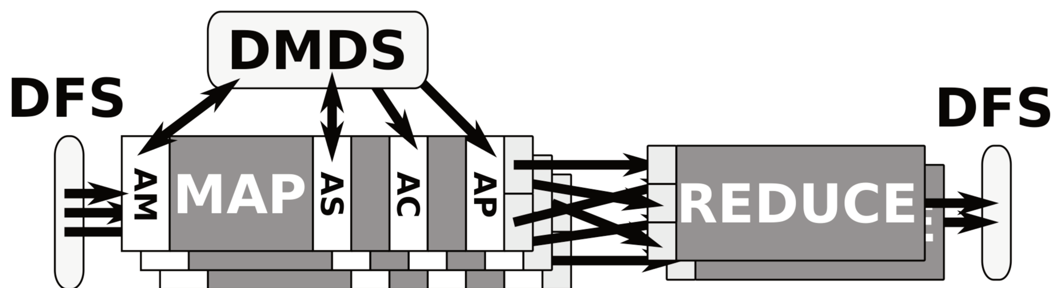


Figure 2.8: Adaptive techniques in SAMs approach [117]

loads. To enhance the load balancing of the data split inputs and the mapper outputs, Vernica et al. [117] have proposed a novel solution denoted SAMs (Situation Aware Mappers), which allows the mappers to be aware of the task loads through a transactional distributed meta-data store (DMDS) component. This component shares the metadata between the mappers and helps the system to make a global load balancing decision of the mapper inputs and outputs. The authors propose multiple adaptive techniques (see Fig. 2.8) to improve the load balancing of the split inputs, improve the local aggregation in the combine phase, and rebalanced the mapper outputs based an on-line sampling method (to pick up the frequent mapper keys). The two main issues of these techniques (i.e., algorithms) are: (1) these algorithms can take more time than the mapper tasks itself since its perform at runtime, especially with massive data, and this obviously increases the job execution time, and (2) the size of DMDS component can increase significantly in a big data environment and maintaining all the metadata become a problematic.

2.3.2.6 The Partition cost model for load balancing in MapReduce

Gufler et al. [40] have developed a novel method to balance on the fly the reducer loads. The idea of the authors is waiting until all or the majority of the mappers finish executing their tasks, to collect sufficient information that helps the optimizer system to elaborate good partitioning scheme of the reducer inputs. The authors present two load balancing approaches, fine partitioning, and dynamic

fragmentation, based on the partition cost model. Moreover, in their approach, they consider also non-linear reducer functions used in some scientific data domain, where we can obtain unbalanced in partition runtime although that the partitions have the same number of tuples. In this thesis, we don't tackle this problem since the tuples of any data warehouse table have the same size.

Furthermore, though this approach is suitable to enhance Group By operation and aggregate functions of an OLAP query, however, changing the default synchronizing mechanism adopted in this approach can overload the memory buffer of the intermediate results, and the system may crash. Also, if the mapper tasks are fort unbalanced, we obtain stragglers in the map phase and the job runtime increases. Moreover, in some aggregate functions of an OLAP queries, we cannot split roughly the reducer partitions, since this enforce the system, in many cases, to add another MapReduce cycle to finish performing the aggregate function.

2.3.2.7 LEEN approach

Data locality issue has been widely studied in some static partitioning approaches used on sharing nothing systems [7, 14, 33, 35, 89, 115]. The main aim of these solutions is to enhance the map tasks runtime. Ibrahim, S., et al. [73] have tackled the problem of data locality in the reduce phase to limit the data shuffling. The authors have proposed a novel method, called LEEN, to balance the reducer loads and to decrease the amount of data transferred in the shuffle phase. LEEN remedies two issues when distributing the pairs (key, values) over the reducing partitions: the first is the frequency variation of the keys produced by the mappers and the second is the inconsistency in the key's distribution. Although LEEN approach can improve the data locality along with fair distribution, however, it has the same drawback as the solution of Gufler et al. [40], such that the authors embrace an asynchronous map and reduce scheme to keep a track on all the intermediate keys' frequencies and key's distributions. Figure 2.9 shows the architecture of LEEN. Chen et al. [27] also have developed CLP approach to enhance data locality; however, instead of to use on the fly distribution as LEEN, the authors adopt pre-processing phase to gather the frequent keys used, by using

sampling technique.

Figure 2.9: LEEN Architecture [73]

2.3.2.8 Partition Tuning-based Skew Handling (PTSH) Approach

Almost all of the approaches existing in the literature [16, 27, 37, 40, 73, 98, 120], that balance the reducer loads, perform the reduce phase in one MapReduce cycle. Gao et al. [37], due to the specific problem treated (association rule mining (ARM) on healthcare data), they adopt two phases MapReduce to execute the reducer function. Although PTSH approach can handle the data skew effectively, however, it can increase the shuffled data due to the two MapReduce iterations for the reduce phase. Moreover, PTSH treats a specific problem and may not work well in other domains.

2.3.2.9 SCID partitioning

Zhuo et al. [110] have proposed the SCID algorithm (i.e., Splitting and Combination algorithm for skew Intermediate Data blocks) to balance the intermediate results and enhance the reducer function runtime. SCID uses a sampling method to predict the frequent keys produced in the map phase. Note that although this algorithm uses sampling technique such as [120] and [98] which can affect to the load balancing decision, it is efficient even with Big Data since it distributes smartly the remainder mapper keys that haven't considered in the sampling algorithm. The defect of this method that we can be noticed is that the time of sampling can slow down job processing when the size of the sampler increase.

2.3.2.10 Load balancing reducer loads using multi-agent system

In the previous approaches, namely the works of [27, 37, 40, 73, 98, 120], the authors focus on balancing the reducer loads in advance using different partitioning techniques. Baert et al. [16], contrary to the earlier methods, their algorithm balances the reducer loads after the shuffle phase, using multi-agents system. The authors inspired their work from Known et al. [80] approach, and their technique is much suitable in the case of heavy data skew in the reducers; The drawback of this method is it may degrade the system performances and slow down job processing, since the system must halt the reducer tasks to trigger the re-balance algorithm. Moreover, we cannot apply this technique in all aggregate functions, e.g., standard deviation and variance function, since the system needs to create more MapReduce cycles to complete calculating the aggregate function. We will propose in this thesis (see chapter 6) a new approach based on a multi-agents system which able to balance on the fly the reducer loads, without changing the default mechanism of the MapReduce paradigm.

2.4 Join Algorithms in the MapReduce Paradigm

We have seen in the previous section (Sect. 2.3), the different data placement strategies (i.e., static and dynamic PLB techniques) and the various physical design of the data warehouses and databases in parallel DBMS and distributed systems such as Hadoop MapReduce and Spark. These physical designs or schemes help the execution engine of a distributed system to perform the SQL queries and other analytic applications swiftly. In OLAP queries, join is the most expensive operation and can involve high network cost due to the shuffle phase, especially in a star join with several dimension tables. In this section, we expose some algorithms that can enhance join and star join operation in distributed systems.

Almost all of the existing join algorithms in the literature rely on dynamic techniques of partitioning and load balancing of data, e.g., on-line partitioning and load balancing, such as repartition (or hash-shuffle), hash-broadcast join [20], multi-way join [9, 127], triejoin [75] and to perform star join operation such the works of [24, 96]. Few are based on static techniques, such as trojan join [33] and star join on HadoopDB [7]and JOUM [15]. This kind of algorithms requires prior knowledge of the table schema and join conditions. In the following, we detail some join algorithms.

2.4.1 Repartition join

Repartition join [20] is the same as a partitioned sort-merge join in the parallel RDBMS literature [32]. The join in this algorithm is executed in the reduce phase. This algorithm, which also called hash-shuffle join, is not suitable when using larges tables due to the amount of data to be shuffled. Variant of the repartition join are used in Pig [90], Hive [112], and Spark SQL [13]. Executing star join with this type of join may need up to 2(n-1) MapReduce cycles [96], where n is the number of tables involved in the query. Hash-shuffle join is the worst join algorithm in the MapReduce paradigm that you should avoid to use it.

2.4.2 Broadcast and Semi-join

In the case where we have a small table R joined with a large table L, the system can use broadcast join [20], such that it spread the table R across the nodes where each partition L will join with some records of R on the join key. This type of algorithm requires that R table can fit in memory without the overhead. The broadcast join is performed in the map phase and and is implemented in several query processing systems like Pig [90], Hive [112], and Spark SQL [13]; moreover, it is suitable in star join and OLAP queries if the dimensions of the data warehouses are small enough to fit in memory; otherwise, the system needs to combine between repartition join and broadcast join to perform the star join operation.

In the previous algorithm, many records of the table broadcasted (i.e., R) may not be referenced by any records of the large table partitioned and are not used by the join. To avoid sending all tuples of R across the nodes, Blanas et al. [20] have implemented the Semi-join algorithm on MapReduce. The issue of this technique is that it needs to scan L table to check the join keys values, before broadcasting the subsets of R, which can increase the join runtime significantly. To decrease this time, a preprocessing phase can be done to determine the subset of R that will join with L.

2.4.3 Multi-way join

Usually, joining several tables in a distributed system needs multi MapReduce iterations (e.g., in shuffle join, the system requires two MapReduce cycles to join three tables). Afrati and Ulmane [9] have proposed multi-way join algorithm in MapReduce environment which able to join several tables in one MapReduce job, with low communication cost. Although this algorithm is executed on the fly and not require any prior knowledge about the data warehouse placement and scheme, however, the amount of data to be shuffle is not negligible, especially in star join when joining larges tables. Another alternative of multi-way join has been proposed to improve the not equi-join operation. Zhang et al [127] have developed

a multi-way algorithm for Theta-join queries using MapReduce Paradigm, where they have considered other operators such as $(<, >)$ in the join condition function. Other dynamic partitioning techniques have also been proposed to enhance the performance of the two previous algorithms (namely the works in [9, 127]), such that the approaches of Tao et al. [109] in cloud environment, Zhang, C., et al. [126], and Myung et al. [87].

2.4.4 Optimize star join operation

As we have explained before star join is a ubiquitous and expensive operation in OLAP queries. It increases the disk spill when scanning fact and dimensions tables, rise the communication cost due to the shuffle phase and requires many MapReduce cycles to perform it. To remedy these issues, many works have been proposed [24, 41, 96, 109, 126, 129, 130]. In this subsection, we explain two recent works, namely the approaches of Brito et al. [24] and Purdilă et al. [96].

2.4.4.1 Decrease disk spill and network cost in star join operation

To enhance star join operation in cloud computing, Brito et al. [24] have proposed two efficient algorithms: the first called Spark Bloom-Filtered Cascade Join (SBFCJ), which use Bloom filter [111], to avoid scanning some unnecessary records of the fact table and the second called Spark Broadcast Join (SBJ). The two algorithms are complementary, although SBJ requires that the dimension fit in the executor memory of Spark. Moreover, this approach uses RDD and is developed at the moment of producing Spark SQL by Armbrust et al. [13]. We should note that Spark SQL is based on the Catalyst optimizer and usually gives better results than Brito et al. solution.

2.4.4.2 Single-scan approach

Purdilă et al. [96] have developed a novel dynamic algorithm called Single-scan which can perform a star join operation in two MapReduce iterations. Single-scan is much better than Bloom filter technique [24] and works well with small and large dimensions, without any assumption about memory size. In the first MapReduce pass, Single-scan adds filters to all dimensions, and in the second pass, it scans the fact table and performs join between the fact and the dimension tables. You should note that although the method proposed by the authors has reduced the number of MapReduce iterations for the star join operation, however, their approach still needs the shuffle phase, since it doesn't take into account the data locality factor.

2.4.5 Bucket-map join and Sort-merge-bucket join

Hive and Spark SQL use two efficient join algorithms that can perform the join operation in map side, which are: Bucket-map join and sort-merge-bucket join [52, 53]. The two algorithms are suitable for large tables. Bucket-map join needs that the tables are bucketed with the join column and the number of the buckets of each table is multiple or divisible to the other table. Moreover, it requires that the tables are not sorted. The main drawback of this algorithm is it overhead the memory in some cases. To overcome this issue, Hive-QL and Spark SQL use Sort-Merge-Bucket (SMB) join which requires that all tables must bucket by the same join key and all tables have the same number of the buckets. These two algorithms can improve the performances of the join operation drastically. We can notice that the major problem with these algorithms is that the tables need to be bucketed in the same way how the SQL joins writes. We will provide more details about these algorithms in our contributions (see chapter 3 and 5).

In this thesis, we adopt SMB join to improve star join operation runtime on a distributed big data warehouses.

2.5 Conclusion

We have seen in this section the different static and dynamic techniques of the partitioning and the load balancing of the data in shared nothing systems (i.e., both parallel DBMS and distributed technologies such as Hadoop MapReduce and Spark), and the different join algorithms existing in the literature. We have tackled three main issues living in distributed big data warehouses and both OLTP and OLAP workload which is: scanning unnecessary data partitions or data blocks, optimize the join operation, more particularly the star joins task, and enhancing the reduce functions (such as Group By operation and aggregates functions). All our contributions in this thesis treated the issues cited earlier.

In the first contribution, we proposed a novel data placement strategy [1, 3, 4] of distributed big data warehouses over Hadoop cluster. This static PLB technique allows performing star join operation in only one Spark stage (or one MapReduce iteration) whatever the size of the fact and the dimensions tables. We can consider our contribution as an extension of the approaches of [33] and [9] to enhance star join operation. The second contribution [2] is a generalization of Sun et al. [108] approach, such we propose some methods (partitioning and bucketing techniques) to avoid scanning some unnecessary HDFS blocks whatever the OLAP workload used and to optimize the parallel treatment in the distributed systems. The third contribution [6] is a hybridization between the first approach [4] and the second [2], where we propose a novel physical design for distributed big data warehouses over Hadoop cluster to perform the scan, the filtering and the star join operation swiftly.

The three previous contributions, namely the approaches [1, 2, 3, 4, 6], are based on a static PLB of the data; to improve Group By and aggregate functions, we need to use a dynamic PLB technique as we have done in the fourth approach. So, our contribution [5] optimizes Group By task using a multi-agent system. This approach is similar to the Gufler et al. [40] approach, such that we distribute on the fly the fragments produced by the mappers (after execute scanning, filtering and star join of an OLAP query) over the reducers. However, the main key of our solution is that we keep the default mechanism of the MapReduce paradigm. In the following chapters, we detail these contributions.

Chapter 3

Optimize Star Join Operation for an OLAP Query

Information is the oil of the 21st century,
and analytics is the combustion engine.
Peter Sondergaard

3.1 Introduction

Improving OLAP query performance in a distributed system such as Hadoop and Spark is a challenging task. An OLAP query is composed of several operations, such as projection, filtering, join, and grouping. The star join operation is the most expensive one and usually involve considerable communication cost. The common method used to decrease the network traffic for the star join operation is to co-partition some tables of a data warehouse on their join key. However, this operation still requires many MapReduce cycles in existing data warehouses partitioning schemes [18, 24, 96, 115]. In this chapter, we present the main part of our data warehouse physical design over a Hadoop cluster. This part allows

performing filtering, projection and the star join operation, locally and in only one Spark stage, without a shuffle phase. We propose two methods called "FKey" and "NewKey" based on a data mining technique to guide our physical design of a BDW. Our partitioning and distribution scheme helps the query's optimizer (e.g., Catalyst optimizer of Spark SQL) to make an efficient physical query plan, such that the system does not need to re-partition and re-distributing the partitions of the data warehouse tables again to perform the star join operation.

A Data Warehouse (DW) can be seen as a huge database that is designed for query and analysis rather than for transaction processing. The size of a DW is between hundreds of gigabytes and hundreds of petabytes. A commonly used model in the data warehouses is the star schema. A star schema consists of one fact table and several dimension tables.

In the literature, many studies have tackled the problem of the partitioning and load balancing (PLB) of the data warehouses over Hadoop cluster. As we have seen in section 2.3, we can distinguish two types of techniques, static and dynamic. While static technique, used usually to balance the split inputs [7, 11, 14, 18, 108]. In dynamic techniques, the system performs the PLB algorithm at the moment of query processing [16, 40, 73, 82, 110, 125], that is, it elaborates the partition scheme on the fly. Note that making a good PLB of a BDW over a cluster can help the query optimizer to make an efficient query plan, which can speed up the query processing.

Apache Hadoop [60] uses different techniques for the PLB of the data to enhance query performances. However, the random distribution of Hadoop blocks may slow down the query processing, especially with the OLAP query when joining several tables. The join operation is the most expensive one and often involves a high communication cost. In some cases, a star joins operation will need $n-1$ or $2(n-1)$ MapReduce cycles [96], where n is the number of tables involved in the query. To reduce the MapReduce iterations' number, minimize the disk spill, and decrease the network communication cost when performing a star join operation, some solutions have been proposed [41, 24, 96]. However, **to the best of our knowledge, none of the previous proposals is able to perform a star join**

operation in only one Spark stage (or one MapReduce cycle), without a shuffle phase, as our approach can do.

In this chapter, we propose a new data placement strategy for a distributed BDW over a Hadoop cluster (see our contributions in [1, 3, 4]). We present two methods called Fkey and NewKey, to guide the physical design of a distributed BDW over a Hadoop cluster. Our static PLB technique allows performing the star join operation in **only one Spark stage**, **without a shuffle phase**, based solely on a data-driven model, to avoid the workload update issue. Fkey method based on two measures: the density and the skewness, the second method, i.e. NewKey, based on an efficient algorithm of the data mining called **balanced K-means algorithm [83]**. In our contribution, we take into account: the size of the DW, the distribution of the foreign and primary keys of the fact and dimensions tables, and the characteristics of the cluster nodes. We have exploited the **Sort-Merge-Bucket (SMB)** join algorithm of Spark SQL to execute star join operation correctly. We have implemented and deployed our approach using Scala language of Spark over a cluster of homogeneous nodes, Hadoop-YARN platform, Spark engine, Hive system to manage the metadata of the DW tables, and the TPC-DS benchmark.

The key points of our contribution are:

1. We propose a new data placement strategy which allows performing the star join operation in **a single Spark stage without a shuffle phase**, by exploiting the algorithm SMB join of Spark SQL.
2. We propose a heuristic technique for the PLB of the data to balance evenly the data split inputs of a distributed BDW over a cluster of homogeneous nodes, which can decrease the data skew of the mapper tasks and improve the parallel treatment for executing an OLAP query.

In Section 3.2, we detail our data placement strategy and the two methods Fkey and NewKey of our approach, We present our experiments in Section 3.3, and we conclude in Section 3.4.

3.2 Optimize star join operation approach

For the two methods proposed of our approach, namely FKey and NewKey, we start by building horizontal fragments of the fact and dimensions tables, using hash-partitioning technique, then we distribute these fragments evenly over the cluster nodes, in order to perform filtering, projection, and the star join operation in only one Spark stage without a shuffle phase, which can improve the OLAP query runtime significantly. We assume that we know in advance the data warehouse size and the characteristics of the cluster (i.e., the number of CPU cores and the memory size of each data node). We can divide our approach into two phases: (1) we construct the buckets of the fact and the dimension tables and (2) we place the buckets that have the same join key in the same node as far as possible. Before detailing the two methods, we define some notations used in this chapter.

3.2.1 Notations

We assume that we have a star schema DW, denoted $E = \{F, D1, D2, ..., Dk\}$, such as F is the fact table and $Dd, d \in 1..k$ are the dimension tables. We denote by $FK = \{fk_1, fk_2, ..., fk_k\}$ the set of the foreign keys in F correspond to the dimensions $Dd, d \in 1..k$, and $PK = \{pk_1, pk_2, ..., pk_k\}$ is the set of the primary keys of these dimensions. We denote by fk_m the key selected from FK, to build the buckets of F and all $Dd, d \in 1..k$, by using FKey method, and nk the new key added in F and all $Dd, d \in 1..k$, to build the buckets of all tables of E, by using NewKey method. We denote by $\#B$ the number of the buckets should be built. Note that to build $\#B$ buckets of a table $T \in E$, using any of its key k (of integer type), we use hash-partitioning technique. We denote by $BF = \{BF_0, BF_1, ..., BF_{\#B-1}\}$ the set of the buckets of F and by $BDd = \{BDd_0, BDd_1, ..., BDd_{\#B-1}\}$ the set of the buckets of Dd, where $d \in 1..k$. We denote also by $SBF = \{\|BF_0\|, \|BF_1\|, ..., \|BF_{\#B-1}\|\}$ and $SBDd = \{\|BDd_0\|, \|BDd_1\|, ..., \|BDd_{\#B-1}\|\}$ the set of the buckets' sizes of F and Dd respectively. We denote by $GB = \{GB_0, GB_1, ..., GB_{\#B-1}\}$ the set of the buckets' groups, where each group GB_i, $i \in 0..\#B-1$, is a collection of buckets

that have the same join key. The group GB_i is composed of one bucket $BF_i \in BF$ and one bucket $BDd_i \in BDd$ of each dimension $Dd, d \in 1..k$. Formally we can put $GB_i = BF_i \biguplus_{d=1}^{k} BDd_i$. We denote by $N = \{n_1, n_2, ..., n_e\}$ the set of the cluster' DataNodes.

For the two methods, our objective is to build the set BF and BDd, $d \in 1..k$, where the standard deviation of the sets SBF and $SBDd$, $d \in 1..k$, be smaller as possible, then we distribute the groups GB_i, $i \in 0..\#B-1$ evenly over the cluster' DataNodes N, in which that the distributed system can perform the star join operation of an OLAP query in only one Spark stage, without a shuffle phase. Figure 3.1 shows the steps of our approach that we will explain in the following subsections.

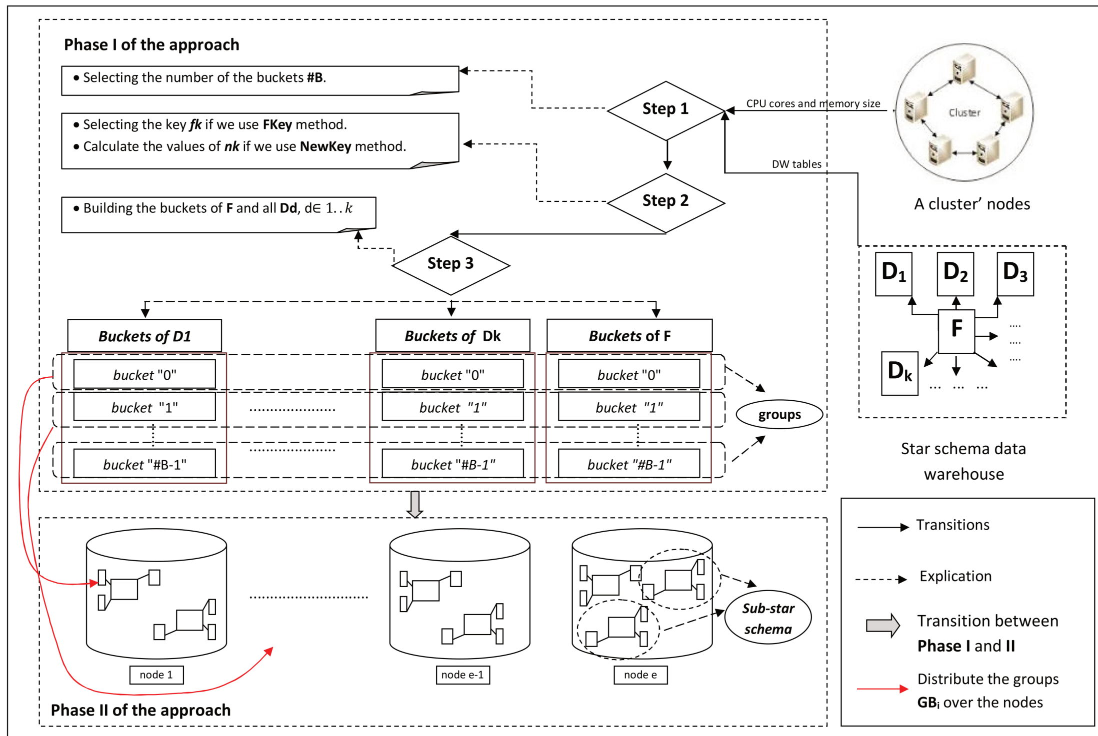


Figure 3.1: The steps of our approach

3.2.2 Construct the buckets

To create the buckets of the sets BF and BDd, $d \in 1..k$, we follow three steps: (1) selecting the near-perfect number of the buckets should be built, i.e. $\#B$; (2) selecting the bucket key: fk_m if we use FKey method and calculating the values of nk if we use NewKey method; and (3) building the buckets of F and all Dd.

3.2.2.1 Selecting #B

Selecting the near-best number of the buckets, namely $\#B$, to guide our partitioning method is critical. To do that, we address some technical challenges as outlined below.
We should select $\#B$ as follows:

$$\#B \in [\#B_{min}, .., \#B_{max}] \tag{3.1}$$

Where $\#B_{min}$ is the minimum value of the number of the buckets and $\#B_{max}$ is the maximum value. To determine these values, we follow these rules:

- **Rule 1.** We should exploit almost all idle CPU cores of the cluster's nodes. So, $\#B_{min}$ should be equal to $\#CT$, the total number of CPU cores assigned to execute an application task. In other words, $\#CT$ is the total number of CPU cores affected to all Spark executors. Our aim is: assign at least one RDD partition to each CPU core (in our case, a Resilient Distributed Dataset (RDD) partition is a group GB_i, $i \in 0..\#B-1$).

- **Rule 2.** Selecting a large number of $\#B$ ($\#B \gg \#CT$) can disrupt the distributed system as a result of increasing the number of HDFS chunk and the I/O operations, and this can incur significant overhead for processing the RDD partition and maintaining partition-level meta-data by the NameNode. Hence, and since our processing is in-memory, using Spark, we determine $\#B_{max}$ as follows:

$$\#B_{max} \leq \lfloor \#B_{min} \times max(1, V_E/V_M) \rfloor \; and \; \#B_{max} \leq |T| \tag{3.2}$$

Where V_E is the size of the DW E, V_M is the sum of all memory sizes of all slave nodes, and T is the smallest dimension in E.
Our argument is based on the following: the first part of Eq. (3.2) means that if the total memory size of the cluster is large, i.e., $V_M \approx V_E$, in that case, we can process a large RDD partition. However, if the memory size is small, namely, $V_M \ll V_E$, then $\#B_{max}$ increases and processing a small RDD partition become preferable. The second part, i.e., $\#B_{max} \leq |T|$, means that we must not get an empty bucket for all BF and BDd.

- **Rule 3.** We have seen in the previous rules that we should select $\#B$ from the interval $[\#CT, .., \lfloor \#CT \times \left(\frac{V_E}{V_M}\right) \rfloor]$. However, and since the values' number of the range $\lfloor \#CT \times \left(\frac{V_E}{V_M}\right) \rfloor - \#CT$ may achieve several hundreds. Hence, it's not realistic to test all values of this interval to find the near-perfect solution, namely, doing hundreds of empirical tests with different values of $\#B$. Thus, we should optimize our method to select $\#B$. So, if we assume that processing all tasks in each Spark wave finish roughly in the same time, i.e., the number of all job tasks is divisible by $\#CT$, therefore, to select the near-best value of $\#B$, we execute the queries with $\#B$=$\#CT$, and each time we increment $\#B$, i.e. $\#B$=$\#B$+$\#CT$, until $\#B$=$\#B_{max}$ or until the execution time of the queries increases.

3.2.2.2 Selecting the bucketed key

As have indicated earlier, we can use two methods (FKey and NewKey) to select the bucketed key. In the following, we detail these methods.

FKey method. In this section, we detail how to select fk_m from the set FK. The selecting of fk_m is based on two important measures, namely the **density** and the **skewness**. The two following rules explain how to use these measures to select the best key $fk_m \in FK$.

- **Rule 1.** The number of distinct values of a foreign key $fk_i \in FK$ has

an impact to the number of tuples in each bucket of BF. Thus to avoid obtaining an empty bucket in BF, the foreign key fk_i must respect the following formula:

$$\frac{1}{Density(fk_i)} \geq \#B \qquad (3.3)$$
$$Density(fk_i) = \frac{1}{number\ of\ distinct\ values\ of\ fk_i}$$

- **Rule 2.** After selecting the set of the foreign keys that respect the Eq. (3.3). To choose the fk_m among these keys, we use the skewness measure (denoted Sk):

$$Sk = \frac{n}{(n-1)(n-2)} \sum (\frac{x_j - \mu}{\sigma})^3 \qquad (3.4)$$

Where n is the cardinality of fk_i, x_j is the element j of fk_i, σ is the standard deviation of fk_i and μ is the average value in fk_i, (For more detail see Wikipedia [54]).

The best key fk_m is the foreign key fk_i that has the minimum value of Sk. The advantage of the Eq. (3.4) is to control the homogeneity of the SBF bucket' sizes. The fact is that there are no criteria in the literature to ensure that a distribution $Dist$ has light or heavy data skew. According to the recommendations of some works [36] and [39]), we can consider that a set $Dist$ has heavy data skew if $|Sk| > 2.0$.

Remark We can use other measures to evaluate the skewness such as the covariance measure:

$$Cov(Dist) = \frac{std}{mn} \times 100$$

Where std is the standard deviation of $Dist$ and mn is the mean value of $Dist$.

NewKey method. We have seen earlier that the FKey method cannot ensure always to obtain a near-zero standard deviation of SBF. To overcome this drawback, we propose a new method, called NewKey, to build BF and BDd, $d \in 1..k$. In NewKey, we add a key nk at the fact table F such that the values of this key allow obtaining buckets roughly equal in size when we bucketed F by this key in

$\#B$ buckets. We can use the range-partitioning technique to achieve our objective (i.e., obtain near-zero standard deviation of SBF); however, there is an essential factor that can influence to the size of the buckets of the new dimensions built when constructing BDd, $d \in 1..k$, as we will show in subsection 3.2.2.3. This factor is the similarity of the tuples in each bucket of BF. To overcome this issue and obtain an approximately optimal solution, we propose the following method.

1. ***From the fact table*** F***, we create the matrix*** MB, such that:

$$MB = \begin{pmatrix} V_{FD11} & V_{FD21} & .. & V_{FDk1} \\ .. & .. & .. & .. \\ .. & .. & V_{FDdj} & .. \\ .. & .. & .. & .. \\ V_{FD1n} & V_{FD2n} & .. & V_{FDkn} \end{pmatrix}$$

   Where V_{FDdj} is the value of the foreign key fk_d at line j of the fact table F, and $n = |F|$. (these values are integer type)

2. ***After building*** MB***, we clustered it in*** $\#B$ ***clusters.*** Thus, we obtain the values of the column nk. Our clustering method should trade-off between the number of tuples in each bucket against the similarity of the tuples in each bucket. Apply traditional K-means to cluster this matrix can give imbalanced clusters size, since, the objective of this algorithm is to lower the intra-cluster distances. In the standard k-means algorithm, the most common criterion to optimize is the Mean-Square-Error (MSE), calculated by the following formula:

$$MSE = \sum_{j=1}^{k} \sum_{X_i \in C_j} \frac{\|X_i - C_j\|^2}{n}$$

   Where X_i denotes data point locations (in our case point is tuple or vector of matrix MB), C_j denotes centroid locations, and $n = |MB|$.
   In our method, we focus to use balance-constrained clustering [23, 83, 131], where the cluster size balance is a mandatory requirement that must be met and minimizing MSE is a secondary criterion. The first reason to choose this

kind of algorithm is to get roughly balanced buckets' sizes of the set BF, and the second one is to minimize the disc space of the new created dimensions.

We have finished using the approach of [83] which used Hungarian algorithm [25]. This algorithm has minimum complexity time $O(n^3)$, where n is the size of MB. The first reason to choose this algorithm is to minimize the standard deviation of SBF and the second one is to decrease the size of the newly built dimensions $D'd$ (see why and how to build $D'd$ in subsection 3.2.2.3). The output of the algorithm is (n mod $\#B$) clusters of size $\lceil n/\#B \rceil$ and $\#B$-(n mod $\#B$) clusters of size $\lfloor n/\#B \rfloor$.

3. ***Finally,*** we affect the cluster values obtained to the nk column, and we can apply our bucketing method, as explained in the next section.

This clustering technique only ensures minimizing the standard deviation of SBF and not all $SBD'd$. However, since the size of $D'd$ is small compared to F, the size of the bucket of $SBD'd$ also remains small compared to the bucket of SBF, and hence the sizes of the GB_i, $i \in 0..\#B-1$, remain roughly equal (see the notations in Sect. 3.2.1). Moreover, in our clustering algorithm, we have not included another factor which can increase the size of the new dimensions: the number of the attributes of a dimension. Some dimensions can have a few attributes while others have many attributes (e.g. hundreds or thousands). However, with the "**Parquet**" and "**ORC**" storage formats, only the attributes solicited by the queries are loaded into memory and not the whole bucket of the new dimension. Furthermore, with the new compression and coding techniques in HDFS (e.g., Gzip and Snappy), the column nk occupies negligible disk space since the number of its distinct values is not large, i.e., $values(nk) \in 0..\#B$-1.

3.2.2.3 Building the buckets

After selecting the near-best buckets' number $\#B$ and the key of the bucketing for both methods FKey and NewKey, in this section, we show how to build the sets BF and BDd, $d \in 1..k$.

FKey method After selecting fk_m, it is trivial to create the buckets of F and those of the dimension Dm (which correspond to the foreign key fk_m), however to build BDi, $i \in 1..k/m$, of the other dimensions, we must make some transformations to these dimensions, in order to can bucket them with the same join key fk_m. For this, first we construct an intermediate table, denoted IDi, correspond to the dimension Di, $i \in 1..k/m$. IDi is composed of two columns: the first column is the foreign key fk_i in F and the second, also denoted fk_m, their values obtained by the formula $fk_m[j]\ mod\ \#B$. Where $fk_m[j]$ is the values of the foreign key fk_m in F and $j \in 1..|F|$. Then we delete all duplicate tuples from IDi (Initially, we have $|IDi|$=$|F|$). After that, we join IDi with Di to obtain a new dimension $D'i$, $i \in 1..k/m$, that contains the join key fk_m. Finally we build $BD'i$, as we have done with BF and BDm. Figure 3.2 shows an example of how to build a new dimension $D'i$ with FKey method.

NewKey method The case of building the buckets in NewKey is straightforward as FKey method. So, after constructing BF using the key nk, we start to build BDd, $d \in 1..k$. So, to have the same join key nk in these dimensions as the fact table F, we make some transformations. For this, first we construct an intermediate table, denoted IDd, correspond to the dimension Dd, $d \in 1..k$. IDd is composed of two columns: the first column is the foreign key fk_d in F, and the second is the newly added key, namely nk. Then we delete the duplicate tuples from IDd (Initially, we have $|IDd|$=$|F|$). The magic of balanced K-means algorithm appears here (see subsection 3.2.2.2). By applying this algorithm to define the values of nk, we increase the probability to obtain duplicate tuples in IDd, since the similarity of the tuples in each bucket is high. As a result, we decrease the size of IDd and the new dimension $D'd$, which obtained by joining IDd and Dd. Finally, we build the buckets of $BD'd$, $d \in 1..k$, which contain the join key nk. Figure 3.3 shows an example of how to build a new dimension $D'1$ and its buckets with NewKey method.

Note that the sizes of the new dimensions $D'd$, $d \in 1..k$, are large compared to the original ones in both methods. In NewKey method, the sizes of the newly built dimensions changed according to (1) the value of $\#B$ and (2) our clustering

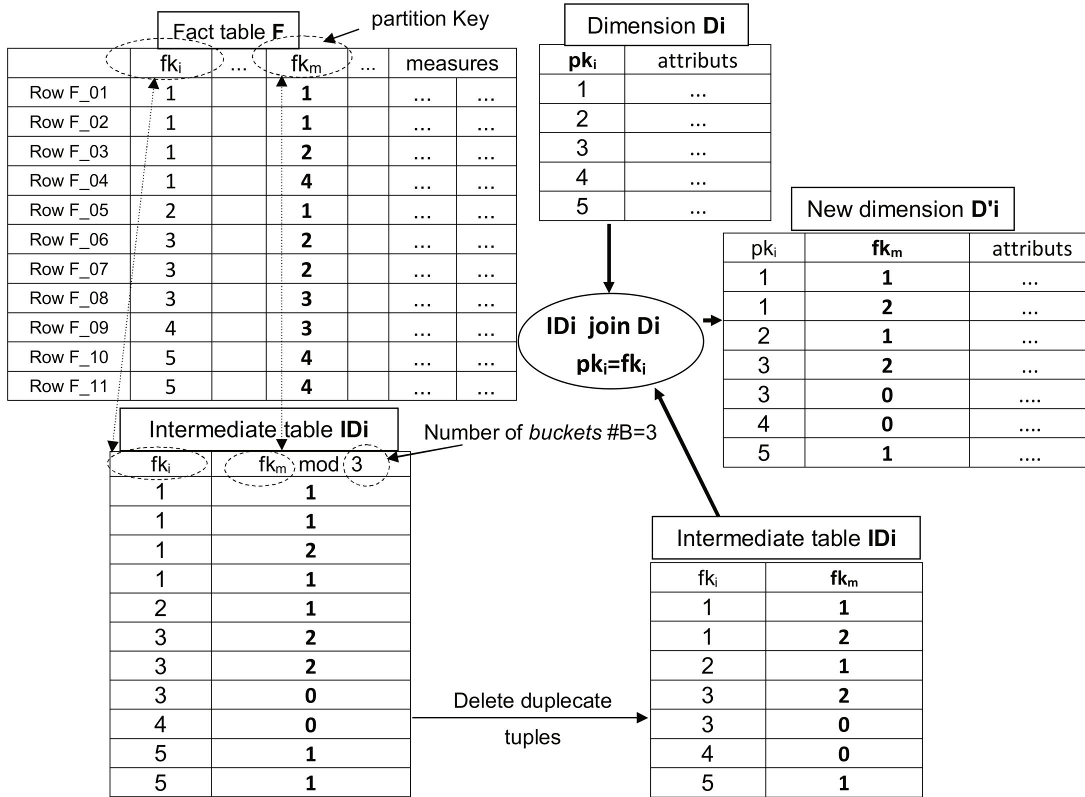


Figure 3.2: Building a new dimension with FKey method

method applied to choose the values of nk and to limit data redundancy due to the duplicated tuples in the newly obtained dimensions. However, the size of the new dimensions, $D'd$ remains small compared to the fact table F whatever the size of the original dimensions Dd. Indeed, in BDW, the size of the fact table is generally immense compared to the size of the dimensions, whatever the latter. In the case of small dimensions, the system can easily broadcast them, and in the case where the dimensions are too large, they can not fit into memory. Moreover, the rate of repeated values will be high with small dimensions, and it may decrease with large dimensions. Furthermore, the number of distinct values of nk is not high since we have limited their values between 0 and $\#B$-1 using balanced k-means algorithm, and the maximum of $\#B$ is fixed by using our heuristic method detailed in subsection 3.2.2.2. Hence, the probability of obtaining duplicated tuples in IDd,

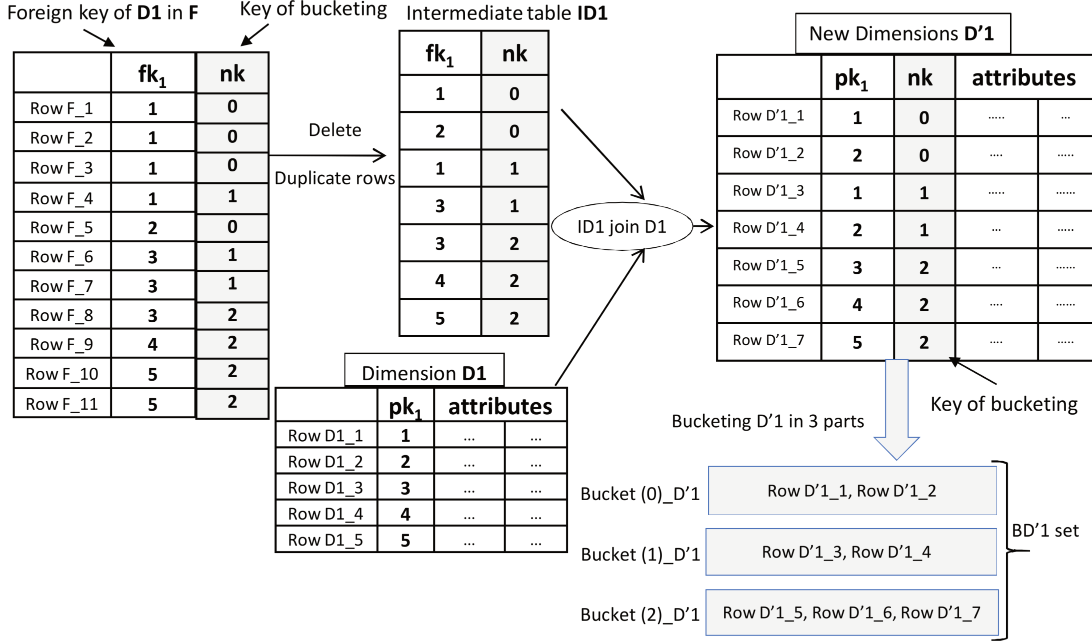


Figure 3.3: Building a new dimension and its buckets with NewKey method

$d \in 1..k$, will be high. As a result, the size of the new dimensions, $D'd$ remains small compared to F. Our experiments in Sect. 3.3 confirm this argument.

3.2.3 Placement of the Buckets

In phase II (see Fig 3.1), we evenly distribute the GB_i, $i \in 0..\#B-1$, of the groups GB over the cluster's nodes. Thus, we start to place the GB_0 in node 1, GB_1 in node 2,...,and the GB_{p-1} in the node e, such that $e{=}p \ mod \ \#B$ and $p <= \#B$. We restarted the operation with **round robin fashion**, we put GB_p in node 1, GB_{p+1} in node 2, and so on, until the $group_{\#B-1}$. This data placement strategy allows performing the star join operation locally and in **a single Spark stage without a shuffle phase.**

3.2.4 Query Transformation

In our approach, we do some changes in the join condition yet such that it still produces the correct join result. Namely, with FKey method we add to the join condition $F.fk_i = D'i.pk_i$, $i \in 1..k/m$, the second condition $F.fk_m \ mod \ \#B = D'_i.fk_m$. That is, in FKey method, the join condition becomes ($F.fk_i = D'i.pk_i$ and $F.fk_m \ mod \ \#B = D'i.fk_m$), where $i \in 1..k/m$. In NewKey method the join condition becomes ($F.fk_d = D'd.pk_d$ and $F.nk = D'd.nk$), where $d \in 1..k$. Note that in the both methods, we must activate the Sort-Merge-Bucket (SMB) join and disable Hash-Broadcast (HB) and Shuffle (SH) join of Spark SQL.

The following example demonstrates the execution plan of an OLAP query with our approach (NewKey method). We consider an OLAP query Q extracted from the TPC-DS benchmark, e.g.

```
select c_first_name, c_last_name, d_year, sum(ss_sales_price)
from customer, date_dim, store_sales where
customer.c_customer_sk = store_sales.ss_customer_sk
and store_sales.ss_sold_date_sk = date_dim.d_date_sk and date_dim.d_year=2000
Group by c_first_name, c_last_name, d_year;
```

After transformation, we obtain the query Q':

```
select c_first_name, c_last_name, d_year, Sum(ss_sales_price)
from customer', date_dim', store_sales' where
(store_sales'.nk = customer'.nk and store_sales'.ss_customer_sk=customer'.c_customer_sk)
AND (store_sales'.nk= date_dim'.nk and store_sales'.ss_sold_date_sk = date_dim'.d_date_sk)
AND date_dim.d_year=2000 Group by c_first_name, c_last_name, d_year;
```

The execution plan of the query Q' in Spark SQL with our approach is as follow: First the system scans each bucket of the table date_dim', executes the filter d_year=2000, and retrieves only the attributes involved in the query Q', i.e., retrieves the fragments (date_dim.nk, date_dim.d_date_sk, date_dim.d_year). As a result, we obtain the fragment f_date_dim' (this fragment is distributed over $\#B$ RDD partitions, where $\#B$ is the number of buckets); the same process is applied to the other tables store_sales' and customer'; we obtain the fragments f_store_sales(store_sales'.nk, store_sales'.ss_sold_date_sk, store_sales'.ss_customer_sk, store_sales'.ss_sales_price) and f_customer'(customer'.nk, customer'.

c_customer_sk, customer'.c_first_name, customer'.c_last_name) which are also distributed over the same number of RDD partitions (the system gathers all the buckets that have the same number into one RDD partition). Then the system executes the join condition between f_store_sales' and f_date_dim'. However, in our approach, we enforce the system to execute the two join condition (store_sales'.nk=date_dim'.nk and store_sales'.ss_sold_date_sk=date_dim'.d_date_sk), locally in each RDD partition, and avoid broadcasting join and shuffle join. As a result, we obtain the new fragment f'(store_sales'.nk, store_sales'.ss_customer_sk, date_dim.d_year), and finally we join f' with f_customer' to obtain f_final (c_first_name, c_last_name,d_year, ss_sales_price). At the end, the system performs Group by and aggregate function Sum(ss_sales_price). Figure 3.4 illustrates how to perform a star join between these fragments.

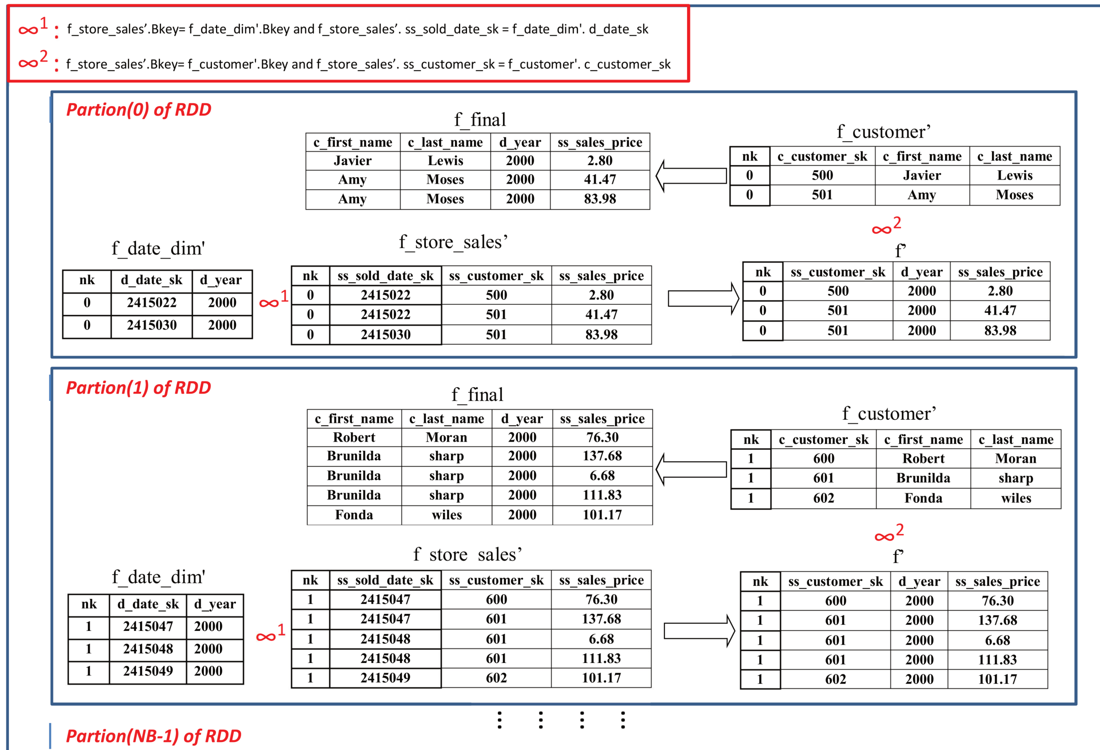


Figure 3.4: Example of how to perform star join with our approach(using NewKey method)

3.2.5 Scalability of our approach using NewKey method

The update of the data warehouses is not as the databases, the changing of dimensions, and the fact table will be done infrequently (e.g., update every six months). There are several methods to update the DW. In our case, to deal with updating the DW, we use our bucketing technique as a "secondary" partitioning scheme, namely, before bucketing the tables with nk, we partition some tables which can change over time, by the *date* attribute (e.g. code in Hive "..*Partitioned By (date)...Bucketed By (nk) into #B buckets...*"). For example, when a new partition date='2019-06-08' is added to the *customer* table we run: *ALTER TABLE customer ADD PARTITION (date='2019-06-08')....* So, with this technique, we can invoke our bucketing technique and apply balanced K-means algorithm on this newly inserted partition without affecting the existing data.

The following scenario explains the DW update process: Suppose that we have loaded and distributed our DW, in first time at the date='09-12-2018' with our method NewKey. After six months we want to update the DW tables, so, only the tuples updated and the new tuples will be inserted into the new partition (date='08-06-2019') using our data placement strategy, and so on. Moreover, to obtain correct results, the optimizer of the system must execute the star join operation of each partition in separate Spark waves. In other words, it cannot gather the buckets located in independent HDFS-partitions in one RDD-partition, although that has the same value of nk. Figure 3.5 shows this mechanism clearly. We can notice in this schema, i.e., Fig. 3.5, that although some buckets of the partition P1 and P2 have the same bucket number (i.e., the value of nk), they are executed in different waves of Spark stage.

Remark Note that although this technique may increase data redundancy, however, it is necessary for a DW analysis purpose.

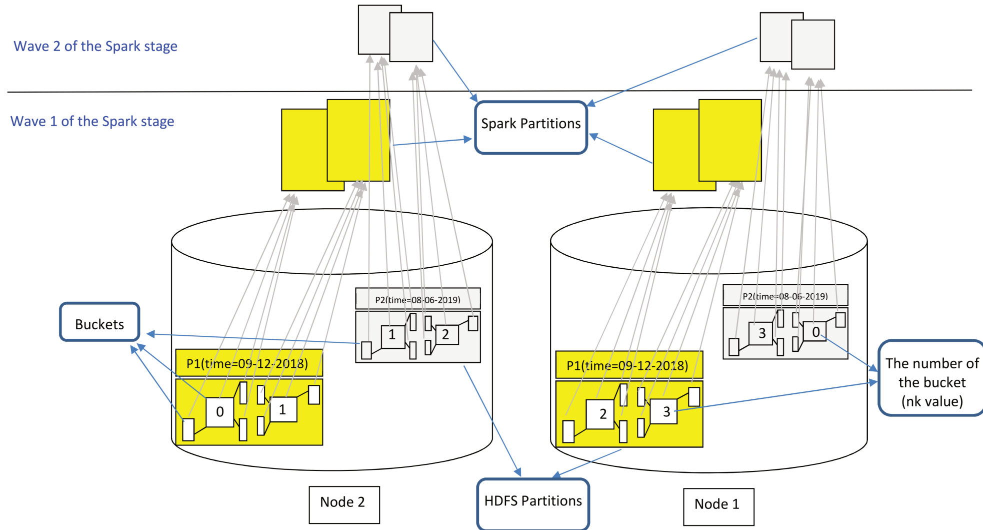


Figure 3.5: Illustrative example to explain how to deal with DW updates with our method NewKey

3.3 Experiments

In this section, we explain how to implement our approach, and we also show its advantages and its limitations. We start to present how to deploy our distributed system and how to generate the data warehouse, then we explain how to implement the different parts our approach, and we finish by evaluated our methods using some baseline approaches.

3.3.1 Experimental Setup

First of all, we generate the DW using the TPC-DS benchmark, where we store the data directly in HDFS using the Parquet format. To show the scalability of our approach, we have tested our methods, namely Fkey and NewKey with two clusters nodes, one with 5 Datanodes and one master node, denoted $C1$, and the second denoted $C2$, with 15 Datanodes and one master node. We use two

data warehouse sizes, the first, denoted $DW1$ with 500 GB in CSV format, and the second, denoted $DW2$, has about 5 TB in CSV format (about 1420 GB in Parquet format). Each node is characterized by CPU Pentium I7 with 8 cores, 16 GB of memory, and 2 TB of a hard disk. We installed in all nodes Hadoop-YARN V-2.9.2, Hive V-2.3.3, Apache Spark V-2.3.3, TPC-DS benchmark, Scala language, and Java. To the master node we added MySQL server to store the Hive meta-data, "Maven tools" and Scala Build Tool "SBT" to build jar packages. For the HadoopDB tests (see Sect.3.3.2), we added PostgreSQL to all the data nodes.

In Apache Spark, there are more than 150 configurable parameters [93] that can affect the job execution time. In our experiments, we focused on configuring the candidate parameters as recommended in [17, 93]. So, we configured some Spark parameters as follows: spark.executor.memory=6 GB and spark.executor.cores=3 CPU cores. In $C1$, we set spark.executor.instances=10 and in $C2$, spark.executor.-instances=30. We kept the default block size 128 MB and kept 3 as the number of replications. For the memory size and CPU cores, we should not exploit all idle resources. Thus, for all slave nodes, we kept 4 GB and 2 CPU cores for "operating system," "executors," and for "Application Master." With this configuration, we can run $3 \times 10 = 30$ tasks in parallel in $C1$ and $3 \times 30 = 90$ tasks in $C2$.

3.3.1.1 Data generation

We adapted the *spark-sql-perf* application [56], using Scala language and Spark. In our experiments and since we focused of optimizing the star join operation, we used one fact table among seven and nine dimensions among seventeen of the TPC-DS [48] benchmark (see Table 3.1). Because of the limitations and physical characteristics of our cluster and to avoid memory overflow during the data generation, we generated the fact table *store_sales* by partition. We chose the foreign key *ss_store_sk* of dimension *store* as "partition key" because the primary key of this table, namely *ss_store_sk* has the minimum number of distinct values compared to the other keys of other dimensions (see Table 3.1).

3.3.1.2 Implementation

To build the buckets of the fact and dimension tables, we used the instruction, e.g.

```
DF.write.bucketedBy(\#B, "nk").sortBy("nk").format("parquet").
mode("overwrite").saveAsTable("DB.tablename")
```

Where DF is a Spark $dataframe$, nk is the partition key with NewKey method, *parquet* is the used storage format, DB is a Hive database, and *tablename* is the name of the bucketed table. By applying the rules of Sect. 3.2.2-FKey method, the bucketed key selected in the FKey method is the foreign key fk_m=ss_item_sk which correspond to the dimension *item*. In NewKey method, the values of nk are calculated using our clustering method detailed in Sect. 3.2.2 (NewKey method). We have implemented a balanced K-means algorithm using SparkR. To deploy the first phase of our approach, we have used three essential components: $Dataframe$, *Dataset* of Spark, and $ArrayBuffer$.

Bucketing with Spark: it creates many files for each bucket. To implement the phase II of our approach (see our works [3, 4]), we do not modify the block policy placement of HDFS as in [35], since the framework API of Hadoop V-2.x or V-3.x would need severe modifications. Our strategy of placement is currently implemented as an external balancer tool, that is to say, we let Hadoop distribute

Table 3.1: Characteristics of DW tables

	Table name	Data Warehouse 1 ($DW1$)		Data Warehouse 2 ($DW2$)	
		Number of records	Parquet Format	Number of Records	Parquet Format
1	*store_sales*	2 879 995 413	142.6 GB	28 799 954 135	1 420 GB
2	*customer*	12 000 000	607.8 MB	65 000 000	3 210 MB
3	*customer_address*	6 000 000	111.4 MB	32 500 000	603.63 MB
4	*customer_demographics*	1 920 800	7.4 MB	1 920 800	7.4 MB
5	*item*	300 000	27.3 MB	402 000	36.52 MB
6	*time_dim*	86 400	1 126 KB	86 400	1 126 KB
7	*date_dim*	73 049	1 740 KB	73 049	1 740 KB
8	*household_demographics*	7 200	30.0 KB	7 200	30.0 KB
9	*promotion*	1 500	76.0 KB	2 000	98.77 KB
10	*store*	1 002	88.0 KB	1 500	128.96 KB

the HDFS blocks of the buckets by the default placement policy (random targeting of the nodes), then with an off-line process, we rebalance the files or the blocks. This has the advantage of keeping safe the code of the HDFS default block placement policy and avoid invoking auto co-locating of the blocks when one or more nodes crash, which can harm the distributed system. Moreover, although may we cannot completely ensure placing all the buckets that have the same value of the join key in the same node, however, in the both methods, i.e. FKey and NewKey, by exploiting SMB join and disabling the HB and SH join (see related works in section 2.4), the execution of the star join operation remains executed in a single Spark stage without a shuffle phase.

3.3.2 Results

We carried out some experiments in the two clusters, $C1$ and $C2$, with two BDW, namely $DW1$ and $DW2$. Table 3.2 shows different approaches used in our experiments. Since we have used a DW part of the TPC-DS benchmark, we can only use the queries that solicited our DW tables (about 32 among 99 queries). So, in our experiments, and for reasons of simplicity (without loss of generality), we used six queries from among the thirty-two (see Table 3.3) with different levels of complexity. We aim to show how Catalyst optimizer of Spark SQL can run the star join operation in only one stage with the two methods of our approach whatever the used OLAP query. The different characteristics of the six queries are detailed in Table 3.4. We executed these queries with five values of $\#B$, which selected according to our recommendations detailed in Sect. 3.2.2.1.

Table 3.2: Notations for the compared approaches

Notation	Description
SHDB	default partitioning and distributing scheme of HadoopDB with PostgreSQL database and Spark as the execution engine.
SSH	default partitioning and distributing scheme of Hadoop and Spark, using default Spark Shuffle Hash join (SH join). (like repartition join [20] in MapReduce.).
SHB	default partitioning and distributing scheme of Hadoop and Spark, using Hash Broadcast Join (HB join).
FKey	our partitioning scheme with FKey method, by exploiting Spark SMB join and disabling HB and SH join, using $\#B$ buckets.
NewKey	our partitioning scheme with NewKey method, by exploiting Spark SMB join and disabling HB and SH join, using $\#B$ buckets.
NewKeyH	our partitioning scheme with default Hadoop distribution policy, by exploiting Spark SMB join and disabling HB and SH join, with NB buckets.
NewKeyR	similar to NewKey but instead of using our balanced K-means algorithm, we just create roughly equal buckets of the fact table randomly by using range partitioning method.

We consider SHDB, SSH, and SHB as the baseline approaches, which we compare with our solutions, namely FKey, NewKeyH, NewKeyR, and NewKey. In SSH and SHB approaches, we set the parameter $spark.sql.shuffle.partitions$ to 180 in $C1$ and in $C2$ with $DW1$, and we set 630 in $C2$ with $DW2$ (these values are selected according to the bucket's number in our approaches). To deactivate the default hash-broadcast (HB) join of Spark, we add the instruction *Session.conf.set("spark.sql.autoBroadcastJoinThreshold",-1)*. In this case, Spark executes Hash-Shuffle (SH) join (see section 2.4.1). In FKey, NewKeyH, NewKeyR, and NewKey, since we bucket all the DW tables with the same attribute (i.e. join key nk), we can run SMB join in the right way. With our configuration, we can run 30 tasks in parallel in the cluster $C1$ and 90 tasks in $C2$. Thus, by following the rules of Sect. 3.2.2.1, in $C1$, we get: $\#B_{min} = 30$ and $\#B_{max} \leq \lfloor 30 \times \frac{500}{16 \times 5} \rfloor = 187$ (we put $\#B_{max} = 180$). remember that the number of CPU cores exploited in $C1$ is 30, our data warehouse size is 500 GB (i.e. in $DW1$), and the size of the memory in each datanode is 16 GB. With the same way, in $C2$: with $DW1$, we get $\#B_{min}$=90 and $\#B_{max}$=180; and with $DW2$, we get $\#B_{min}$=90 and $\#B_{max}$=$\lfloor 90 \times \frac{5120}{16 \times 15}$=1920. However, with $DW2$, since the smallest table *store* has 1500 tuples (see Table 3.1), so by following Eq. 3.2, we get $\#B_{max}$=1440.

Table 3.3: Selected queries

name	query
Q1	select dt.d_year, item.i_brand, item.i_brand_id, sum(ss_sales_price) from date_dim dt, item, store_sales where dt.d_date_sk = store_sales.ss_sold_date_sk and store_sales.ss_item_sk = item.i_item_sk and item.i_manufact_id = 128 and dt.d_moy=11 group by dt.d_year, item.i_brand, item.i_brand_id limit 100;
Q2	select dt.d_year, dt.d_month_seq, item.i_brand, item.i_brand_id, item.i_class, sum(ss_sales_price) from date_dim dt, item, store_sales where dt.d_date_sk = store_sales.ss_sold_date_sk and store_sales.ss_item_sk = item.i_item_sk and item.i_manufact_id = 128 and dt.d_moy=11 group by dt.d_year, dt.d_month_seq, item.i_brand, item.i_brand_id, item.i_class limit 100;
Q3	select c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country, c_login, c_email_address, d_year, d_month_seq, sum(ss_sales_price) from customer, date_dim, store_sales where c_customer_sk=ss_customer_sk and ss_sold_date_sk=d_date_sk group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country, c_login, c_email_address, d_year, d_month_seq limit 100;
Q4	select a.ca_city, d.d_month_seq, i.i_brand, sum (ss_list_price) from customer_address a, date_dim d, item i, store_sales s where a.ca_address_sk = s.ss_addr_sk and s.ss_sold_date_sk = d.d_date_sk and s.ss_item_sk = i.i_item_sk and i.i_manufact_id = 128 and dt.d_moy=11 group by a.ca_city, d.d_month_seq, i.i_brand limit 100;
Q5	select a.ca_city, d.d_month_seq, i.i_brand, sum(ss_list_price) from customer_address a, date_dim d, item i, store_sales s where a.ca_address_sk = s.ss_addr_sk and s.ss_sold_date_sk = d.d_date_sk and s.ss_item_sk = i.i_item_sk group by a.ca_city, d.d_month_seq, i.i_brand limit 100;
Q6	select c.c_customer_id, c.c_first_name, c.c_last_name, c.c_preferred_cust_flag, c.c_birth_country, c.c_login, a.ca_city, a.ca_state, a.ca_country, d.d_month_seq, d.d_date, i.i_brand, i.i_class, i.i_product_name, sum(ss_list_price) from customer_address a, customer c, date_dim d, item i, store_sales s where a.ca_address_sk = s.ss_addr_sk and c.c_customer_sk = s.ss_customer_sk and s.ss_sold_date_sk = d.d_date_sk and s.ss_item_sk = i.i_item_sk group by c.c_customer_id, c.c_first_name, c.c_last_name, c.c_preferred_cust_flag, c.c_birth_country, c.c_login, a.ca_city, a.ca_state, a.ca_country, d.d_month_seq, d.d_date, i.i_brand, i.i_class, i.i_product_name limit 100;

Table 3.4: Characteristics of the six selected queries

query	characteristics
Q1	we join 2 small dimensions with the fact table, we select a few attributes and we use 2 filters.
Q2	is similar to Q1, but we select more attributes.
Q3	we join 2 dimensions whose one is large with the fact table store_sales, and we selected more attributes than Q1 and Q2 without using filters.
Q4	is similar to Q1, but we add the dimension customer_address.
Q5	is similar to Q4, but without using filters.
Q6	we perform star join operation with 4 dimensions of which 2 are large, and we select more attributes than Q5.

Figure 3.6 shows the queries runtime in $C1$ with $DW1$, Figures 3.7 and 3.8 show the queries runtime in $C2$ with $DW1$ and $DW2$ respectively.

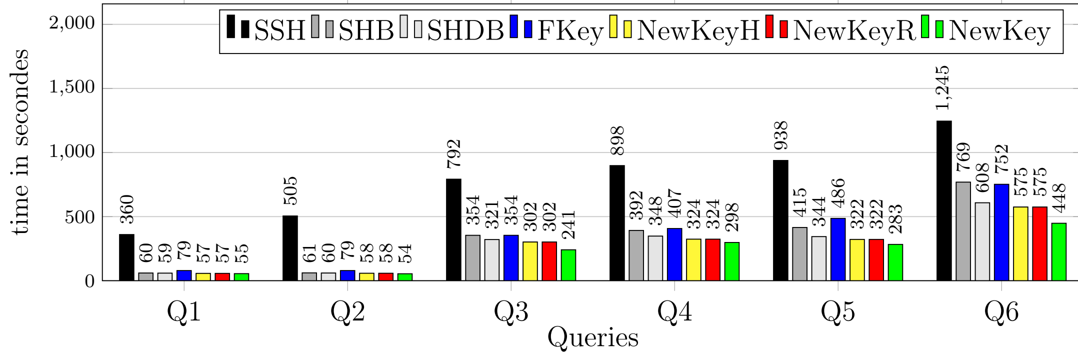


Figure 3.6: Queries runtime in $C1$ with $DW1$ ($\#B$=180).

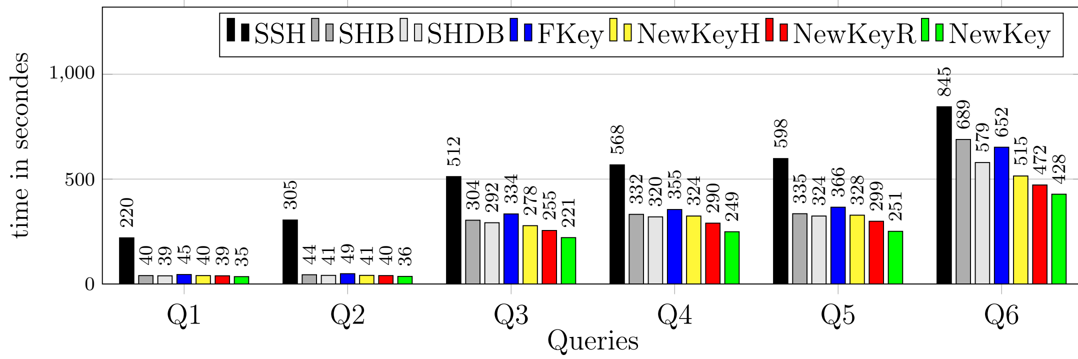


Figure 3.7: Queries runtime in $C2$ with $DW1$ ($\#B$=180).

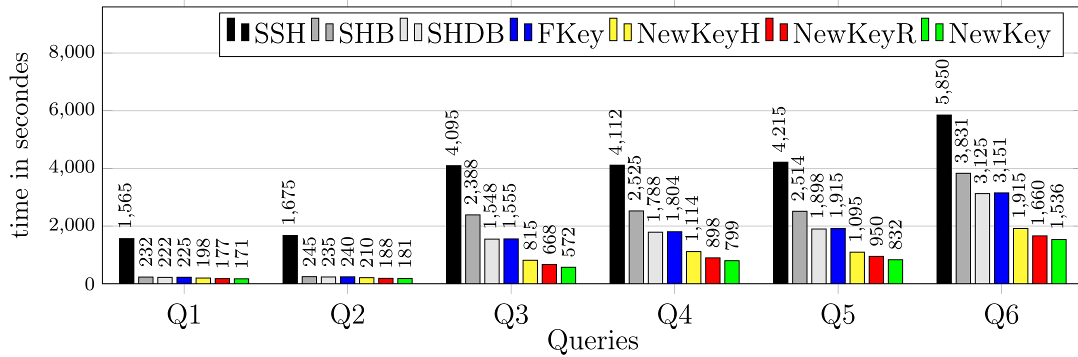


Figure 3.8: Queries runtime in $C2$ with $DW2$ ($\#B$=630).

In Fig. 3.9, we show the execution times of $Q1$, $Q3$, and $Q6$, with five values of $\#B \in \{30, 60, 90, 180, 360\}$, in $C1$ with $DW1$, using NewKey approach. Figures 3.10 and 3.11 show also the impact of the bucket's number on the query runtime in $C2$ with $DW1$ and $DW2$ respectively, using different values of $\#B$.

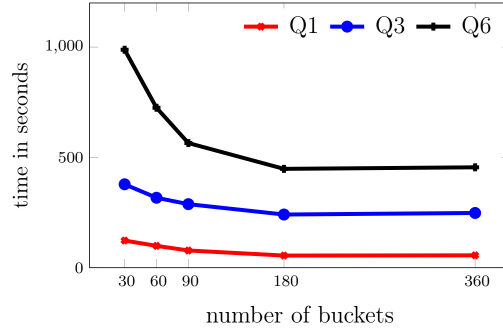


Figure 3.9: Impact of $\#B$ on the query execution time in $C1$ with $DW1$

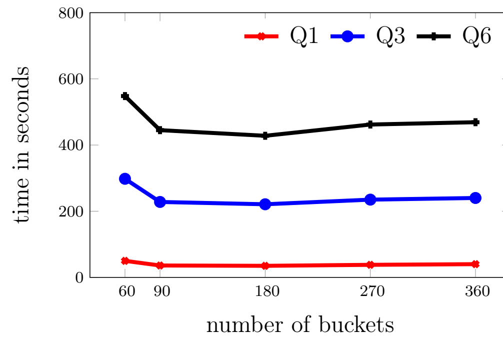


Figure 3.10: Impact of $\#B$ on the query execution time in $C2$ with $DW1$

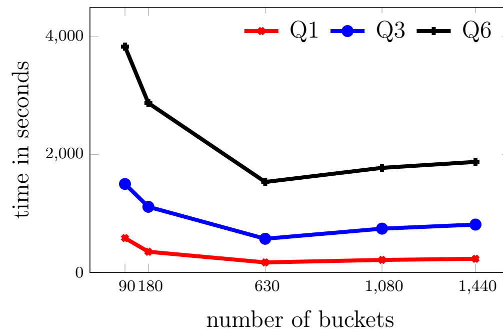


Figure 3.11: Impact of $\#B$ on the query execution time in $C2$ with $DW2$

In Fig. 3.12 and Fig. 3.13, we compare the original dimension sizes (i.e. when using SHB or SSH approaches) with the new built dimension sizes, i.e., when using the methods NewKeyR and NewKey, with $DW1$ and $DW2$ respectively.

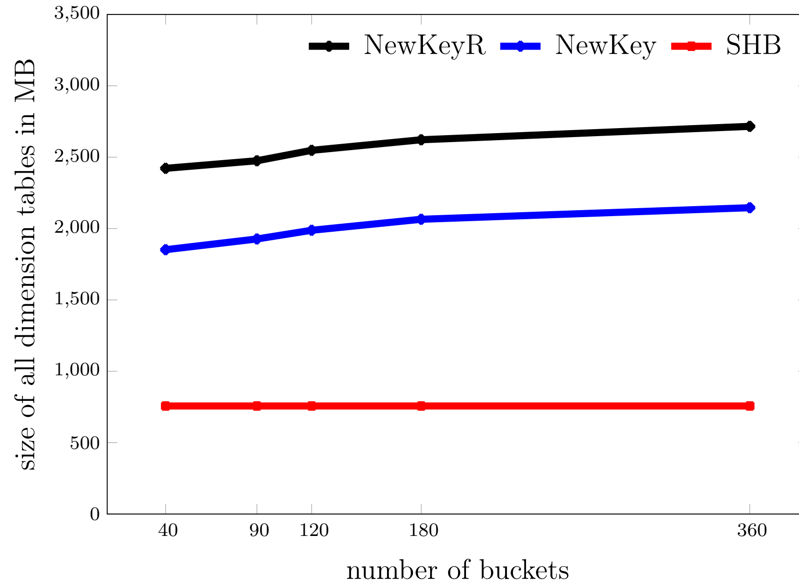


Figure 3.12: Impact of $\#B$ on the size of dimensions with DW1

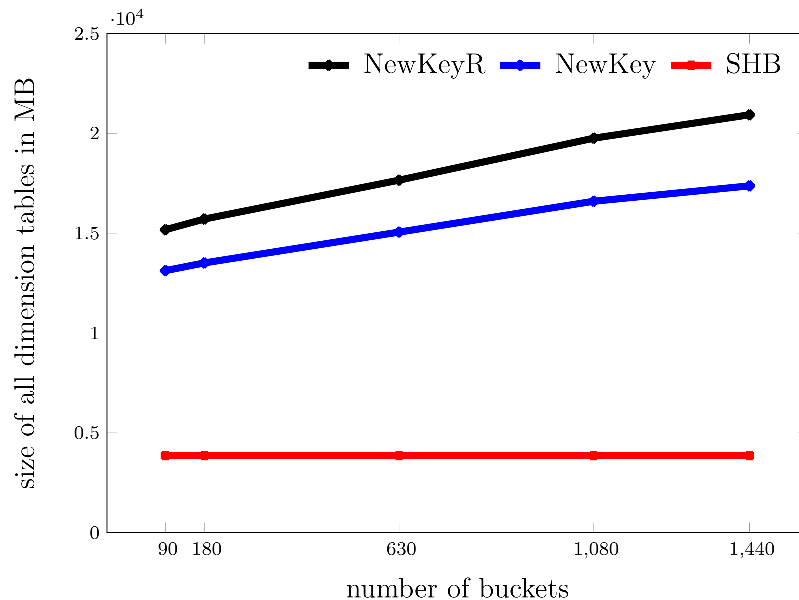


Figure 3.13: Impact of $\#B$ on the size of dimensions with DW2

3.3.3 Discussion

As shown in the bar chart of the Figures 3.6, 3.7, and 3.8 our strategy NewKey has improved the query execution time compared to all other approaches (i.e., baseline methods and our solutions FKey, NewKeyH, and NewKeyR). In all queries, we can see that the worst results obtained with the SSH approach, this is due to the high rate of data shuffling, especially with Q6 with data warehouse $DW2$ (in the cluster $C2$). In Q1 and Q2, since we have selected few attributes of two small dimensions, namely *item* and *date_dim*, the broadcasting of the RDD partitions become fast, and we can see that the execution times of these queries in the SHB approach are roughly the same as our approaches FKey, NewKeyR, and NewKey, particularity with $DW1$ (see Figs. 3.6 and 3.7. However, in Q3, Q4, Q5, and Q6, the performance of SHB suffers, especially with $DW2$ (see Fig. 3.8). The reason is that in the SHB approach when the table is large, the system cannot broadcast it, and must combine with broadcast-join and shuffle-join (i.e., SSH and SHB approaches) to perform these queries.

We can see also that the results in FKey method not promised although the star join operation executed in only one Spark stage, this due: (1) the join condition added $F.fk_m \ mod \ \#B = D'i.fk_m$ (see Sect.3.2.4) may increase the JVM overhead and the CPU cost, since the optimizer must first calculate the modulo operation before checking the equality condition; and (2) the bucket sizes of the set SBF may have high value of skewness since the distribution of the values of the foreign key chosen (i.e., *ss_item_sk*[1]) is not homogeneous, and this can degrade the parallel treatment and create stragglers in the map phase.

We can see that the baseline approach SHDB (when using HadoopDB) is much better than the other baselines (i.e., SSH and SHB) and our method FKey with $DW1$ and $DW2$. Of course, since in SHDB, we duplicated all dimensions over the cluster nodes, the star-join operation can be performed locally and in one MapReduce cycle. However, we can see that our approaches, namely NewKeyH,

[1]Note that according to our rules in subsection 3.2.2.2, we find that the near-best foreign key to bucket the fact table and the dimensions is fk_m=*ss_item_sk*

NewKeyR, and NewKey, perform better than SHDB, especially for Q3 and Q6 (when using the two large dimensions *customer* and *customer_address*). The reason is that although in SHDB all the primary keys and the foreign keys are located in the same node, the system takes a long time to retrieve the relevant tuples from the dimensions stored in the DBMS and join them with the tuples of the fact table *store_sales* which stored in HDFS.

On the other hand, in our methods NewKeyH, NewKeyR, and NewKey, because we have bucketed all the new dimensions and the fact table with the same join key *nk*, and since we have activated SMB join, and disable SH and HB join, the optimizer of Spark SQL can gather all the buckets that have the same value of the join key *nk* into one RDD partition, and performs the star join operation locally[2] and in only one Spark stage. In this case, almost all the work of an OLAP query (except Group by operation and aggregate functions) performed in parallel during the first Spark stage. Moreover, we can notice that the execution time of the queries in NewKey is much better than in NewKeyR, and this demonstrates the efficiency of our algorithm balanced K-means. Also, the random clustering applied in NewKeyR can increase the size of some new dimensions built and degrade the system performance.

Furthermore, we can see from Figs. 3.6, 3.7, and 3.8 that NewKey is up to 1.20 times faster than NewKeyH with $DW1$ and up to 1.35 with $DW2$. Of course, in NewKeyH the buckets of the fact table *store_sales* and the nine dimensions may not be located in the same node and the network communication cost increase due to data transfers. If the nodes located in different racks, the results in NewKey will become 1.5 times better or more than NewKeyH.

Figures 3.9, 3.10 and 3.11 show that $\#B$ has an impact on the query performance. In the queries Q1, Q3, and Q6, the best results are obtained when $\#B \in [180, .., 360]$ in the cluster $C1$ with $DW1$, when $\#B \in [90, .., 180]$, with the cluster $C2$ using data warehouse $DW1$, and when $\#B \in [630, .., 720]$, in $C2$

[2] We should note that in NewKeyH (and in few cases of NewKeyR and NewKey), the buckets that have the same join key are not located in the same node, and the system must exchange some buckets between the nodes to perform star join operation

with $DW2$. These results confirm the reliability and efficiency of our method for selecting the near-best value for the number of buckets (see subsection 3.2.2.1). Furthermore, as shown in Figs. 3.12 and 3.13, we can notice that $\#B$ has a light impact on the sizes of the new dimensions, such that the sizes of the new dimensions built in our data warehouse are only increased by a factor of 2.8 compared to the original ones with $DW1$ and about 4.5 factor with $DW2$. This percentage doesn't influence our approach performances.

Finally, we should note that if we had used the JOUM approach [15] (denormalized model), we would have generated a massive amount of data. More specifically, if we had applied this method to our $DW2$, which contains 179 attributes (sum of all column tables), we would have obtained a big table, that has 15051 GB of size (in Parquet format), this size is about 10 times greater than the size of the original fact table $store_sales$.

3.4 Conclusion

In this chapter, we have presented a novel data placement strategy for distributed big data warehouse over a Hadoop cluster. Our experiments show that our static PLB strategy allows performing a star join operation of an OLAP query locally and in only one Spark stage, without a shuffle phase. Also, the NewKey method outperforms the existing data warehouses partitioning schemes in term of query executing time, whatever the OLAP query used and the data warehouse size. Moreover, we should note that it is easy to deal with the DW update with our data placement strategy. Also, our approach is scalable to large clusters and for huge amounts of data as we have shown in our experiments (see section refsec:3.3). This approach has been acted in the papers [1, 3, 4].

Furthermore, we can adapt our approach with other distributed engine such as Tez [69] and Flink [55]. In addition, we can extend our method to skip loading some unnecessary data blocks if we partition some tables by the attributes frequently used in the query filters (based on a stable OLAP workload), using

the hash-partitioning technique, as we will show in the chapters 4 and 5. On the other hand, we can noticed that although we have roughly balanced the buckets sizes of the DW tables (i.e., balance the split inputs), we have, due to the data skew, gotten unbalanced partition sizes after performing filtering and the star join operation (i.e., unbalanced mapper outputs), this what you call imbalanced in the intermediate results or reducer loads. Curing this issue can improve significantly Group-By operation and aggregate functions, as we will show in chapter 6.

Chapter 4

Skipping Loading Unnecessary Data Blocks and Load balancing

I think you can have a ridiculously enormous and complex data set,
but if you have the right tools and methodology,
then it's not a problem.
Aaron Koblin

4.1 Introduction

Horizontal partitioning techniques are used for many purposes in "shared-nothing systems", such as load balancing, skipping unnecessary data loads [29, 92, 108], and to guide the physical design of databases or data warehouses with workload-driven [11, 79, 108]. Skipping to load some unnecessary blocks of tuples and load balancing can significantly speed up the query processing, especially in the OLAP workload. We can distinguish two types of horizontal partitioning: hash-partitioning and range-partitioning. Range partitioning used for many purposes; however, it may not be ideal for generating fine-grained blocks for skipping. Hash-

partitioning is more suitable in the case where the distribution of the attribute values is homogeneous, and the density value[1] of the attribute is not very low; otherwise, it may incur high unbalanced of blocks size and increasing the size of metadata.

Hadoop becomes the standard platform for big data process. It was built to organize and store massive amounts of data. The main components of Hadoop V-2.x are: (1) HDFS [21, 104], which is designed and optimized for storing very large files, and (2) YARN [116], which supports a more flexible execution engine than MapReduce [30], like Spark [123], Tez [102], and other frameworks. Apache Hive [62] is considered as de facto standard for interactive SQL queries over petabytes of data in Hadoop. In Hive and Spark SQL [13], we can use different partitioning and bucketing techniques to manage the metadata of the data warehouse tables, and we can cure the problem of the density and data skew; however, selecting the appropriate attributes to *Partition By* the DW tables and the attributes to *Bucket by* the DW tables remains questionable. If we partition a table with an attribute that has a low density the number of partitions can increase while each partition is a small file, the distributed system incurs significant overhead to process the partition and maintain the partition-level metadata. Also, if the density of an attribute is high, we risk obtaining a large partition that can overflow the memory during the process. On the other hand, if we partition a large table with an attribute that has unbalanced distributed values (i.e., has high skewness value), some partitions are much bigger than the others, and it generates a straggler in parallel execution.

In this chapter, we use a novel Partitioning, Bucketing, and Sorting techniques (denoted PBS for short) to skip loading some unnecessary HDFS blocks and to improve the parallel treatment of an OLAP query being executed over Hadoop cluster. Our approach [2] has the same objective as the works of Sun et al. [108], however, our method is different to it in two main points: (1) the work of [108] focus on two factors, predicate selectivity, and data skew, and they make some assumptions to the occurrence rate of the same filters used in the queries, in a way

[1]Here, the density of an attribute "A" in a table T is calculated by the formula: Density(A)=$\frac{1}{number\ of\ distinct\ values\ of\ A\ in\ T}$

that their fine-grained partitioning method can achieve maximally level of skipping the blocks, however, in our work, we focused on the density of the attributes and data skew, without make any assumption to the filters used in the workload; (2) though, we cannot achieve maximally level of skipping the blocks, however, instead of using sophisticated clustering algorithm, we used smooth decision method base on physical characteristics of the cluster's nodes and the size of the data warehouse.

In our contribution, we have used the hash-partitioning technique without making any assumptions about the filters used in the OLAP queries. Our method based on the use frequencies of the attributes of the query filters. We have implemented and evaluated our PBS (Partitioning, Bucketing, and Sorting) techniques over a cluster of homogeneous nodes, to partition a big relational data warehouse, using a workload composed of tens of OLAP queries, a Hadoop-YARN platform, and Hive and Spark SQL systems. We take into account: the **density** of the attributes, **data skew**, and **the physical characteristics of the cluster**.

The rest of the chapter is structured as follows. In Section 4.2 we detail our approach, we present our experiments in Section 4.3, before concluding our work in Section 4.4.

4.2 PBS approach

Our PBS approach consists of building horizontal fragments of the fact and dimension tables of a big relational data warehouse, using PBS techniques, based on queries workload. As shown in Fig. 4.1, our solution is composed of 3 steps:

1. **Selecting the frequent attributes.** We select the attributes more frequently used in the OLAP workload, using the frequent items mining technique.
2. **Creating the reference table.** We create the reference table, which allows guiding our PBS techniques.

3. **Partitioning and distributing the data warehouse.** We create physically the new partition schema of the data warehouse by following the reference table created during step 2.

Before detailing these steps, we define our partitioning problem in the following section.

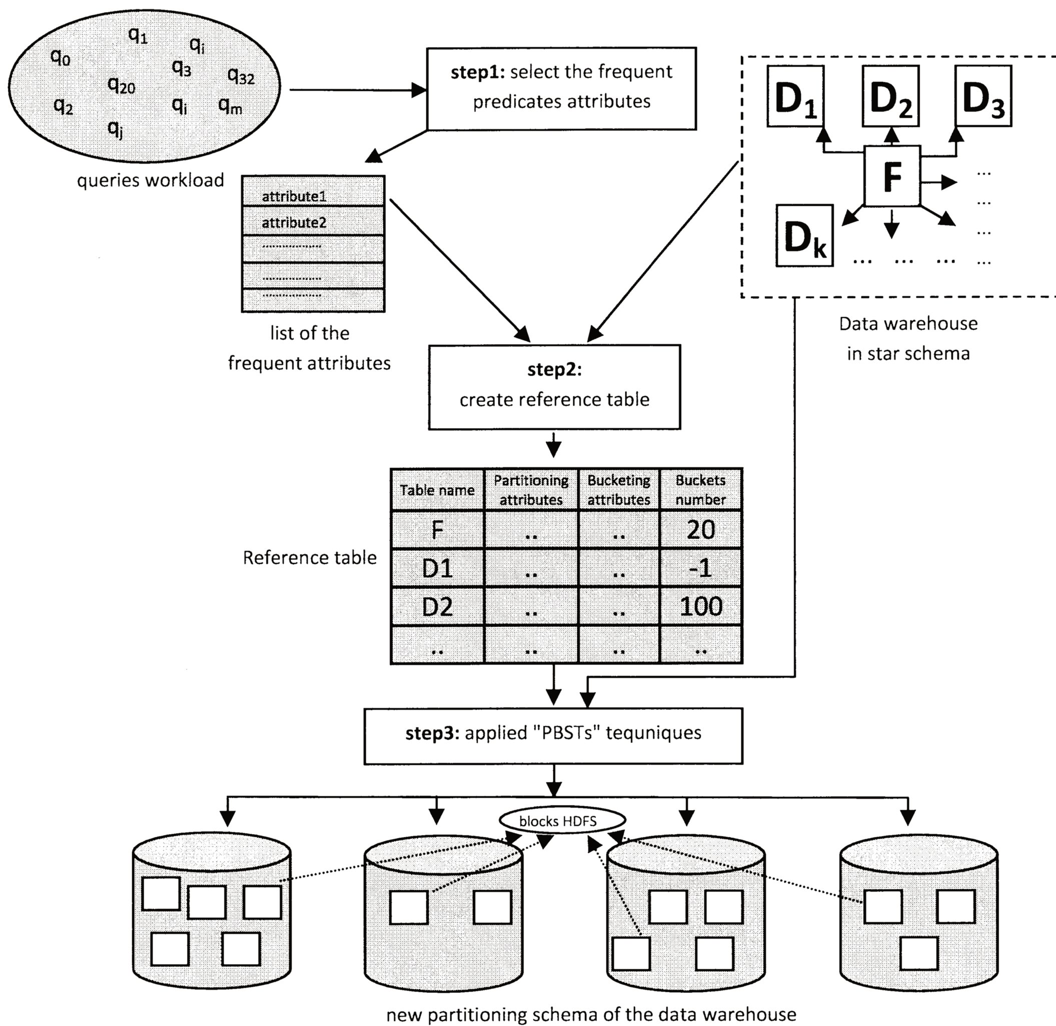


Figure 4.1: The different steps of PBS approach

4.2.1 Problem Formulation

Suppose we have a data warehouse E=$\{F$,D_1, D_2,...,$D_k\}$, in stars schema, such that F is the fact table, and D_i, $i \in 1...k$, are dimension tables. We denote by A=$\{A_0$,A_1,...,$A_k\}$ the set of all attributes of E, such as A_0 is the set of the fact table attributes, and A_i, $i \in 1...k$, is the set of the dimension D_i attributes. We also denote by Q the set of the queries used, such that Q=$\{q_1$,q_2,...,$q_m\}$, and by φ the set of the use frequencies of the queries q_j, $j \in 1...m$, such as φ=$\{f_1$,f_2,...,$f_m\}$. We define the queries workload W by the set of all queries Q, used by its corresponding use frequencies φ, in a time period t (e.g. $t = 24$ hours), such that $|W|$=$\sum_{j=1}^{m} f_j$. We denote by R=$\{R_0$,...,$R_k\}$ the set of all frequent predicate attributes used in the filters of the queries workload W, such that R_0 is the set of the frequent predicate attributes used of the fact table F, and R_i correspond to the dimension tables D_i, $i \in 1...k$. Note that $R_i \subseteq A_i$ and may $|R_i|$=0. We denote by N=$\{n_1$,n_2,...,$n_d\}$, the set of all homogeneous nodes of the cluster. Our issue is, how to partition the tables of the data warehouse E, using the set of the frequent attributes R, upon the cluster N, to obtain the optimal running time of the queries workload W. This is an NP-hard problem as demonstrated by [84, 108]. So, using a heuristic method to find the approximate solution is the right way. In the following, we detail our heuristic solution.

4.2.2 Selecting the Frequent Predicates Attributes

In this step, we used the frequent items mining technique to select the frequent attributes used in the filters of the workload. As formulated in Algorithm 1, we take as an input the workload W, and we obtain the table of the attributes most frequently used, denoted by TFA. In the first step, we prepared the table of the items[2] as an input (line 1 in Algorithm 1), denoted by TIF. For each query $q_j \in Q$ (lines 2-7), we selected the set I_j, $j \in 1...m$, such as we denote by I_j the set of all the attributes used in the query's filters, then we add different distinct items of the collection I_j to the TIF table f_j times. For the queries that composed of

[2] In our case, an item is an attribute selected from the filters of a query.

some subqueries qsl_j, such that $l \in 1...b$ and b is the number of subqueries in the query j, we create the set Isl_j for each subquery, we gather all Isl_j in one set (i.e., I_j), then we add different distinct items of the collection I_j to the TIF table f_j times. The reason of counting only once the attributes of subqueries is that the system scan in the first stages the blocks HDFS from the disc, then for the rest of the query processing stages, it may read from memory cache.

Algorithm 1: Selecting Frequent Predicate Attributes

Input: Q, φ, Th `// see notations in section 4.2.1`

Output: TFA `// the table of frequent attributes used in the filters of` W.

1 $TIF \longleftarrow \emptyset$ `/* set of items (i.e. attributes) */`
2 **for** $(q_j \in Q, j = 1...m)$ **do**
3 $I_j \longleftarrow \emptyset$;
4 **for** (*subquery* $qsl_j \in q_j, l \in 1..b$) **do**
5 $Isl_j \longleftarrow \emptyset$ `/*` b `is the total of subquery in the query` q_j `*/`
6 select items from the "clause Where" of the subquery qsl_j
`/*` $Isl_j \longleftarrow items$ `*/`
7 $I_j \longleftarrow I_j \bigcup Isl_j$;
8 delete duplicate items from the set I_j ;
9 insert items of I_j in the set TIF f_j times ;
10 delete all the items of not integer type from the set TIF ;
11 delete from the table TIF all the items who have the rate of occurrence less than the threshold Th ;
12 $TFA \longleftarrow TIF$;

We assume that an attribute is a frequent item if its rate of occurrence in the table TIF is more or equal than the threshold Th. The parameter Th is defined by the administrator of the system (Note that Th may change from table to another). After that, we delete from the table TIF all the items that have the occurrence rate less than Th and all the attributes of not integer type (i.e. real, boolean, and string type) since we can only partition and bucket a table by an attribute of type integer (lines 10-11 of Algorithm 1). Finally, we create TFA table[3].

[3] You should not partition or bucket a small table, e.g., a table that has a few KB of size.

Table 4.1: Examples of queries

query 1	**query 2**
*select * from T1, T2* ***where*** *"join condition" and T1.attr1=25 and T1.attr2 = 3.5 and T2.attr3="Smith" and T2.attr4=false and T2.attr5=01/02/2018*	select * from T1, (select attr2, attr3 from T2 **where** T2.attr2=12 and T2.attr3="julien"), T3 **where** "join conditions" and T1.attr1>14 and T3.attr4 in (select distict(attr4) from T3 where T3.attr4*2>T3.attr5+20)

To more understand the Algorithm 1, let give you two examples (see Table 4.1). For query 1, first we select the items from "*Where* clause" I_1={attr1, attr2, attr3, attr4, attr5}, then we add all items of I_1 to the table TIF, f_1 times, such as f_1 is the occurrence number of the query 1 in the workload W, query 2 has two subqueries, we select I_{s1}={attr2, attr3} from the first subquery and I_{s2}={attr4, attr5} from the second one. We obtain as a result the set I_2={attr1, attr2, attr3, attr4, attr5}. We add each item of I_2 into TIF, f_2 times. Finally, we delete from the table TIF all the items of not integer type and who have the occurrence rate less than the threshold Th.

4.2.3 Creating the Reference Table

From the table TFA (see precedent section), that contains all the frequent attributes used (result of Algorithm 1), we create the set R, i.e., we assign each attribute of the table TFA to correspond set R_i, $i \in 0...k$ (e.g., R_0 for the fact table, R_1 for dimension D_1, and so on). Then, we build the reference table, denoted by RT, to guide our PBS techniques. The RT table has four columns (see Fig. 4.1), the first column is the "table names" of the set E, the second is the set of "partitioning attributes" of the table, the third column is the set of "bucketing attributes", and the fourth one contains the "buckets' number". If the third column is empty, we put the number of the buckets equal to -1.

Partitioning a big relational data warehouse upon a cluster is a challenging task, many factors can influence to the partitioning scheme, such as the **density** of the attributes, **data skew**, the predicate selectivity, the size of the tables, and the physical characteristics of the cluster (e.g., number of CPU cores, memory size, and so on). In our strategy, we focused on investigating the problem of data skew and the density of the attributes used in the query's filters.

Before detailing how to create the reference table, we need to define the following rules:

• **Rule 1.** We consider a set of values, denoted by $Dist$, has a high skewness value, if the absolute value of the skewness, denoted by Sk, is bigger than the value 2.0 (In fact, there isn't a way in the literature to determine if a set is skewed or not. We choose this number (i.e. 2.0) according to the recommendations of some works [36] and [39]). There are several methods to calculate the skewness. In our case, we choose this formula:

$$Sk = \frac{n}{(n-1)(n-2)} \sum (\frac{x_i - \mu}{\sigma})^3 \tag{4.1}$$

Equation 4.1 is used in chapter 3 (i.e. Eq.3.4), to select the near-best foreign key with FKey method. For more detail how to measure the skewness see [54]).

• **Rule 2.** Since our processing is In-Memory using Spark, we assume that any attribute has a high density if its distinct values not exceed the value S, calculated by the formula:

$$S = \lfloor N_{ct} \times (V_E/V_M) \rfloor \tag{4.2}$$

Here N_{ct} is the total number of CPU cores assigned for processing tasks, V_E is the volume of the data warehouse E, and V_M is the overall size of memory used in the Datanodes of the cluster N (see notations in section 4.2.1). If $V_E \leq V_M$, we put V_E/V_M=1. In other words, N_{ct} is the total number of CPU cores affected to all the Spark executors (an executor is worker nodes' process in charge of running individual tasks in a Spark job. We assign to an executor a couple of resources (α cores, β of memory space) that are required to execute tasks, where α is the number of CPU cores and β is the memory size allocated to perform tasks. By default, one task per CPU core). According to our experiments with training dataset and base on the recommendations in [93] and [105], to select the candidates Spark

parameters, we consider that the maximum size of metadata table, persisted in memory by the NameNode, should not exceed the value S. Our reason in this is: if the total memory size allocated by the cluster is large ($V_M \approx V_E$), the number of partition decrease, therefore, we can process large partitions such as we assign at least for each CPU core one Spark partition (i.e., we can process in parallel N_{ct} partitions in one wave of Spark stage[4]). Each stage can perform in one or many waves. The number of waves calculated by the formula f=(number of partitions/total number of CPU allocated for tasks)). However, if the memory size is small ($V_M \ll V_E$), processing small partition size is preferable.

• **Rule 3.** In the case where the attribute a_j of the set R_i, $i \in 1...k$ and $j \in 1...|R_i|$, has low density (see **Rule 2**), we should bucket and sort table D_i by the attribute a_j in Nb buckets. We denote by Nb the number of buckets. We should select Nb such that:

$$Nb \in [N_{ct}, ..., S],\ Nb\ modulo\ N_{ct} = 0,\ and\ Nb <= |T| \tag{4.3}$$

Here T is the smallest table in the data warehouse E. Our argument for choosing the two bounds of this interval is the same as we have explained in the **Rule 2** and subsection 3.2.2.1 of Chapt. 3, and the reason for choosing Nb value divided by N_{ct}, is to assign in each wave of Spark stage the same number of partitions (i.e. we consider that almost tasks in each wave are roughly balanced[5]). Equations 4.2 and 4.3 is similar to Eqs. 3.1 and 3.2 in chapter 3, when selecting the near-best bucket's number.

Note that the administrator can determine the number of Nb, to bucket dimension D_i according to their sizes.

• **Rule 4.** We assume a table $T \in E$ is small and occupied a negligible space in memory if $|T| \leq 10 \times S$, otherwise it considers a large table. This formula, i.e., $10 \times S$, is obtained according to our experiments, and the administrator can adapt it according to their need, for example, changing the number 10. The size of T also depends on the number of the attributes in this table and not only the number of the tuples. To create the table RT, we follow the Algorithm 2.

[4]in Spark, a stage is a set of consecutive operators that can be grouped and executed together, per partition (i.e., each partition per task)

[5]In fact the tasks may not finish at the same time due to various factors such as partitions data skew, and the heterogeneity of the nodes.

Algorithm 2: Create reference table "RT"

Input: E, TFA, N_{ct}, V_E, V_M `// see notations in sections 4.2.1 and 4.2.3.`

Output: RT table `//` RT `is composed of four columns.`

1 create the set R=$\{R_0$,..,$R_k\}$ from TFA table and the data warehouse E;
2 $i \longleftarrow 0$;
3 partitioning or bucketing the fact table as detailing in **Procedure 1**;
4 add attribute(s) to RT ; `// see detail of Procedure 1 below`
5 $i \longleftarrow i+1$;
6 **while** $(i \leq k)$ **do**
7   partitioning or bucketing dimension D_i as detailing in **Procedure 2**;
8   add attribute(s) to RT ; `// see detail of Procedure 2 below`
9     $i \longleftarrow i+1$;

• **Procedure 1. Partitioning the fact table.** In general, the fact table F is composed of foreign keys of different dimensions and measures. Some measures can be integer type and other real, string, or boolean type. Also may $|R_0| = 0$ (i.e. no frequent predicate attributes) or $|R_0| \neq 0$, and some attributes have low density and other have high density. Whatever the case, if we partition or bucket the fact table by any attribute, we can obtain fort unbalanced block sizes, since the values of an attribute may not uniformly distributed, and this can dramatically disrupt the parallelism of the treatment. Moreover, if we hash-partition the fact table with all the foreign keys, such as the number of these keys may reach the hundreds, the volume of the metadata increases and the Namenode cannot manage the metadata table. Hence, analyze the skewness of the distribution values in these attributes becomes a mandatory requirement to partition the fact table.
To partition the fact table, we can distinguish three cases:

A) If $R_0 = \emptyset$ (no frequent attribute or measure of the fact table found in the predicates). In this case, for each measure of the fact table (we except the foreign keys), of type integer, we calculate the value of skewness (see **Rule 1**), and we applied one of the following steps:

A.1 If all the measures have high skewness value, whatever the density value, we should bucket the fact table by a new attribute added, denoted by *index*, into Nb buckets (see **Rule 3**). The values of *index* are selected from the interval $[0, .., Nb - 1]$ (see Eq. 4.3) and these values must uniformly distributed. We fill in the values of *index* attribute with round-robin fashion, such we start to put value 0, Nb times, then the value 1 Nb times,..., value Nb-1 Nb times, and we restart the operation with the same manner, until the last row of the fact table F. We have chosen Nb as the value of repetition since Nb value is negligible compared to $|F|$. With this method, we can create partitions of F roughly equal in size.

A.2 If there are measures that have low skewness value, but all the measures have low-density value, we bucketed the fact table by a new attribute added as explained in the precedent step.

A.3 If there are measures that have low skewness and have high density, we bucketed the fact table by the measure that has the lowest value of skewness.

B) If $|R_0| = 1$ (i.e., there is one frequent measure of the fact table, of type Integer) and the measure of R_0 has high-density value, we can distinguish two cases:

B.1 If the measure of R_0 has low skewness, we can partition the fact table by this measure (see **Rule 2**).

B.2 If the measure of R_0 has high skewness, we should bucket and sort the fact table by a new attribute added as explained in **Procedure 1, case A.1**.

If $|R_0| = 1$ and the measure of R_0 has low-density value, we can distinguish two cases:

B.3 If the measure of R_0 has low skewness, we can bucket the fact table by this measure (see **Rule 3**).

B.4 If the measure of R_0 has high skewness, we bucket and sort the fact table by a new attribute added as explained in **Procedure 1, case A.1**.

C) If $|R_0| > 1$ (i.e., there is more than one frequent measure), we divided R_0 into two sets, R'_0 and R''_0 set, such as R'_0 contains only the measures that have high density and R''_0 that have low density. Then we sort R'_0 in the ascendant way, where the attribute that has the highest density and the lowest skewness value comes at the beginning. Then we partitioned or bucketed the fact table as explained in **Procedure 1, case B.**

- **Procedure 2. Partitioning the dimensions.** We can distinguish three cases:

A) If $R_i = \emptyset$, we should bucket and sort dimension D_i by its primary key (see **Rule 3**).

B) If the $|R_i|$=1 and the attribute of R_i has high density (see **Rule 2**), we can distinguish four cases:

B.1 If the attribute of R_i has low skewness (see **Rule 1**) and the dimension D_i is large (see **Rule 4**), we should partition dimension D_i by this attribute.

B.2 If the attribute of R_i has low skewness and D_i is small, we should not partition dimension D_i since it occupied negligible memory space.

B.3 If the attribute of R_i has high skewness and D_i is small, we should not bucket dimension D_i since it occupied negligible memory space in Spark executor.

B.4 If the attribute of R_i has high skewness and D_i is large, we should bucket and sort D_i by its primary key.

If the $|R_i|$=1 and the attribute of R_i has low density (see **Rule 2**), we can distinguish four cases:

B.5 If the attribute of R_i has low skewness (see **Rule 1**) and the dimension D_i is large (see **Rule 4**), we should bucket and sort dimension D_i by this attribute into Nb buckets (see Eq. 4.3)

B.6 If the attribute of R_i has low skewness, but D_i is small, we should not bucket dimension D_i since it occupied negligible memory space in Spark executor.

B.7 If the attribute of R_i has high skewness and D_i is small, we should not bucket dimension D_i since it occupied negligible memory space in Spark executor.

B.8 If the attribute of R_i has high skewness and D_i is large, we should bucket and sort dimension D_i by its primary key.

C) If $|R_i| > 1$ we divided R_i into two sets, R'_i and R''_i set, such as R'_i contains only the attributes that have high density and R''_i that have low density. Then we sort R'_i in the ascendant way, where the attribute that has the highest density and lowest skewness value comes at the beginning. Then we partitioned, bucketed, and sorted D_i as explained in **Procedure 2, case B**.

Remark. Please note that selecting too much partitioning attributes (case **C** of the **Procedure 1. and 2.**) can overflow the memory since the system may scan millions of files. To handle this issue, we limit the number of partitioning attributes (i.e., $|R'_i|$, $i \in 0...k$), such that the maximum number of files created, by our partitioning method, not exceed the value $S \times S$. Moreover, although the bucketing operation is not necessary for data skipping purposes, however, it is essential to speed up the parallel processing of data blocks whatever the type of the scan used (full scan table, range scan, etc.).

We have seen that our partitioning approach base on two factors: **density** and **data skew**, using the physical characteristics of the cluster and the size of the data warehouse tables. Also, we have used a **sort** operation to speed up query processing, since we can avoid scanning all the tuples of a block or file, by reading the metadata of the file (e.g., the min-max value of an attribute when we use pushdown filters in Apache Parquet).

4.2.4 Partitioning and Distributing the Data Warehouse

From the reference table RT, we can redistribute the tables of data warehouse E with the new partitioning scheme. For distributing data, since we have used a

cluster of homogeneous nodes, and as we can write out partitions data on HDFS in parallel, through Spark executors, we should keep the default block placement policy of Apache Hadoop.

4.2.5 Scalability of the Approach

In a data warehouse environment, many tables partitioned by time, and most queries have a *time* attribute range filter. For example, when a new partition time='2019-09-01' is added to the table *customer*:
ALTER TABLE *customer* ***ADD PARTITION (time='2019-09-01')...***
We can invoke our PBS techniques on this newly inserted partition without affecting existing data. This method ensures the scalability of our approach, and the tuples added to the new partitions do not influence on existing data. For this, we applied our solution as **secondary** partitioning schema under each such partition, i.e., we add in *RT* table, for some tables of the data warehouse, the *time* attribute as the first "partition key." Our approach can be executed as an offline process at data loading time and maybe re-executed later to account for workload changes or update the data warehouse.

In our approach, if few nodes crash or we add few nodes to our cluster[6], we should not update the parameters S and Nb (see Eqs. 4.2 and 4.3), to keep safe our partitioning scheme (note that, it is hard to redistribute a BDW every time).

4.3 Experiments

To evaluate our PBS approach, we have conducted some experiments with a BDW, using a cluster of homogeneous nodes. The steps of our implementation and evaluation detailed in the next subsections.

[6] In the case when we add too many nodes to our cluster, we should update our partitioning scheme. This issue has not studied in this thesis

4.3.1 Experiment Setup

In this section, we present the implementation steps of our approach. First, we generated the data warehouse using TPC-DS [48] benchmark, then we prepared our queries workload. After that, we applied our partitioning scheme. To achieve this, we used: 15 slave nodes and one master characterized by, CPU Pentium I7 with 8 cores, 16 GB of memory and 2 TB of hard disk. We installed in all nodes the last versions of the platform Apache Hadoop-YARN, Apache Hive system, the processing engine Apache Spark, TPC-DS benchmark, Scala language, Java JDK, and we add in the master node MySQL and the Scala Build Tool (SBT) to build packages. Since we used YARN-client mode, Spark-driver executed in the client machine where we submit the job.We configured the Spark system as follows: spark.executor.instances=30, spark.executor.memory=6 GB, spark.executor.cores=3 CPU cores. For the size of HDFS blocks, we kept the default size 128 MB and 3 as a replication number. For the memory size and CPU cores, we should not exploit all resources idle. Thus, for all the slave nodes, we keep 4 GB and 2 CPU cores for "operating system," "*executors*," and for "Application Master." With this configuration, we can run $3 \times 30 = 90$ tasks in parallel. In the following, we detail the steps of our implementation.

4.3.2 Generation of Data and the Workload

We adapted *spark-sql-perf* [56], using Scala language and Spark, where we generated a part of the data warehouse composed of one fact table and nine dimensions (see Table 4.2). Data stored in HDFS with Parquet format. We generated a big data warehouse whose the fact table *store_sales* has more than 28.7 billion of tuples (we can generate a data warehouse whose the size of the fact table *store_sales* may be more than 287 billion of tuples if we use a cluster of 150 nodes or more). In view of the physical characteristics limitations of our cluster and to avoid memory overflow during data generation, we generated the fact table *store_sales* by partition, where we chose the foreign key *ss_store_sk* as partition key, because the key *ss_store_sk* has the highest density value and a low skewness (see data ware-

house characteristics in Table 4.2. We have used this technique in the precedent approach (see Chapt. 3). Note that by default the partition key of the fact table *store_sales* is the attribute *ss_sold_date_sk* which has 73049 distinct values. This number of partitions can harm our system). So, by using *ss_store_sk* which as only 1500 distinct values, we decrease the size of the metadata table, persisted in memory by the NameNode. The data generator of the TPC-DS benchmark might create several null values of foreign keys in the fact table *store_sales*. By using Spark *dataframes*, we update the fact table *store_sales*, to replace some null values.

For the workload W, we selected 20 queries among the 99 queries of TPC-DS [48] benchmark, namely, we selected only the queries that solicited our DW tables. The set Q of the queries used and their frequencies given in Table 4.3. The values of the use frequency of the queries are chosen according to the works of [99], such as the small queries represent the majority of job submissions. Thus, our distribution of φ values corresponds to 75%, 20%, and 5% for small, medium, and large jobs, respectively. So we get $|W|=\sum_{j=1}^{20} f_j = 140$, such $f_j \in \varphi$.

4.3.3 Implementation

To evaluate our approach, we created three databases. We denoted by DB_DP the database built by *spark-sql-perf* application, we construct from DB_DP a second database, denoted by DB_CSV, using CSV format, and we denote by DB_PBS the third database, built by our PBS scheme. We used Mysql as Hive Metastores, and we activated the Hivecontext in Spark because it is deactivated by default since version 2.0 of Spark, namely we use Spark SQL as a query-processing system and Hive to manage the metadata of the tables. In all databases, we updated the fact table *store_sales*, where we replace some null values of some foreign keys, using the instruction *dataframe.na.fill(..)*. After that, we create *RT* table (see Table 4.4) where we follow Algorithm 2. To create the tables of DB_PBS, we use the following instruction:

Table 4.2: Characteristics of data warehouse tables

Table name	**tuples number**	**Volume**
store_sales	28 799 954 135	1 420 GB
customer	65 000 000	3 210 MB
customer_address	32 500 000	603.63 MB
customer_demographics	1 920 800	7.4 MB
item	402 000	36.52 MB
time_dim	86 400	1 080 KB
date_dim	73 049	1 740 KB
household_demographics	7 200	29.95 KB
promotion	2 000	98.77 KB
store	1 500	128.96 KB

Table 4.3: The queries used in the workload and their use frequency

Query	**The use frequency f_i**	**Query**	**The use frequency f_i**
q_3	9	q_{46}	5
q_6	1	q_{48}	6
q_7	5	q_{52}	7
q_{13}	2	q_{54}	6
q_{19}	8	q_{55}	9
q_{27}	6	q_{60}	8
q_{34}	10	q_{65}	1
q_{42}	7	q_{68}	3
q_{43}	12	q_{79}	6
q_{44}	16	q_{96}	13

```
Session.sql("create\ table  tablename (..put all the attributes of tablename ..)
using parquet partitioned by("att1",..) clustered by (bucketed\_attribute)
 sorted by (bucketed\_attribute) into Nb\_Bucket buckets").
```

Such that we denote by ***tablename*** the name of table of the database DB_PBS, "***parquet***" the data format storage, '***"att1",..,***' the set of attributes retrieved from

the column 2 of the table RT, ***bucketed_attribute*** are the bucketed attribute retrieved from the column 3 of RT, and ***Nb_Bucket*** is the number of buckets retrieved from the column 4.

To load data, we execute the following instruction:

```
Session.sql("insert into table tablename select * from existtable").
```

Such that *existtable* is the table created in DB_DP. In our implementation, we used three main components: $Dataframe$, *DataSet* of Spark, and $ArrayBuffer$. As illustrated in Table 4.4 and explained in section 4.2.5, we add the attribute *time* to the five dimensions *customer*, *customer_address*, *item*, *promotion*, *store*, and the fact table *store_sales*, since their sizes may change with the time.
From the **Rules** of section 4.2.3, we have:
• The total number of CPU cores used is $N_{ct} = (3 \times 30) = 90$ CPU cores, $S = \lfloor N_{ct} \times \left(\frac{V_E}{V_M}\right) \rfloor = \lfloor 90 \times \left(\frac{5000}{16 \times 15}\right) \rfloor = 1875$, and finally we get $Nb \in [90, .., 1440]$, Since the smallest table *store* has 1500 tuples (see Eq. 4.3).
• As shown in Table 4.4, we bucketed the fact table *store_sales* in $90 \times 7 = 630$ buckets. Also, since the sizes of dimensions are negligible compared to the fact table, we chose the value 90 as buckets' number to bucket some dimensions like *customer* and *customer_address*. Note that since the size of dimensions *promotion* and *store* are small, we should not partition or bucket whatever the used workload.

4.3.4 Experiment Results

To evaluate our approach, we executed our queries workload W using the databases DB_CSV, DB_DP, and DB_PBS, with two modes: *full scan* and *range search*. Here we don't confuse with other terms existing in literature, such as *index full scan* or *index-range scan* of Oracle [47]. In our case, *full scan* or *Parquet full scan*, means that the system reads the whole files or HDFS blocks to extract tuples, and

Table 4.4: Reference table *RT*

Table name	**partitioning attributes**	**bucketing attributes**	**buckets' number**
store_sales	*time*	*ss_ticket_number*	630
customer	*time*	*c_customer_sk*	90
customer_address	*time*, *ca_state*	Null	-1
customer_demographics	*cd_gender*, *cd_marital_status*, *cd_education_status*	Null	-1
item	*time*, *i_manager_id*	Null	-1
time_dim	*t_hour*, *t_minute*	Null	-1
date_dim	*d_moy*, *d_year*	Null	-1
household_demographics	Null	Null	-1
promotion	*time*	Null	-1
store	*time*	Null	-1

range search or *optimize search*, means that the system can read the metadata of the file and skipping to scan the whole file or the block.

We launched our workload W as a single Spark framework and we run our job using *spark-submit* in *yarn-client* mode. Also, we flush the operating system file cache before each execution of a query. We randomly select a query from W, then we perform it in one Spark session using the instruction:

```
SparkSession.builder().config(sparkConf).enableHiveSupport().getOrCreate();
```

In this way, we delete at each time all the RDD and the Dataframes, created during the previous query execution.
Please note that in *full scan* and *range search* mode, we exploit the metadata of Hive Metastores in the three databases.

Also, to ensure *Parquet full scan table* mode in Spark SQL (i.e. in DB_DP and DB_PBS), we turn off three essential features: *Parquet push-down*, *statistics filtering*, and *dictionary filtering*. To do this, we apply the following instructions:

```
Session.conf.set("parquet.filter.statistics.enabled", "false");
```

```
Session.conf.set("parquet.filter.dictionary.enabled","false");
Session.conf.set("spark.sql.parquet.filterPushdown","false");
```

Also, we can turn-off schema merging and push-down features, if we execute the instruction:

```
Session.conf.set("spark.sql.parquet.mergeSchema", "false");
Session.conf.set("spark.sql.hive.convertMetastoreParquet.mergeSchema", "false");
```

To use *Parquet range search* mode and exploit the metadata of Parquet file, we turn on the precedents features, see the above instructions[7].

We executed our application five times using the three databases with the two modes described above (i.e., *full scan* and *range search*), and we took the average running time for each mode. Figure 4.2 shows the execution time of the workload W using the three databases with the two modes (i.e. *full scan* and *range search*). Figure 4.3a shows in detail the execution time of the 20 queries of the workload W (see Table 4.3) using the two approaches (i.e., DB_DP and DB_PBS) in *full scan* mode, and Fig. 4.3b shows the amount of data read during the execution of the 20 queries.

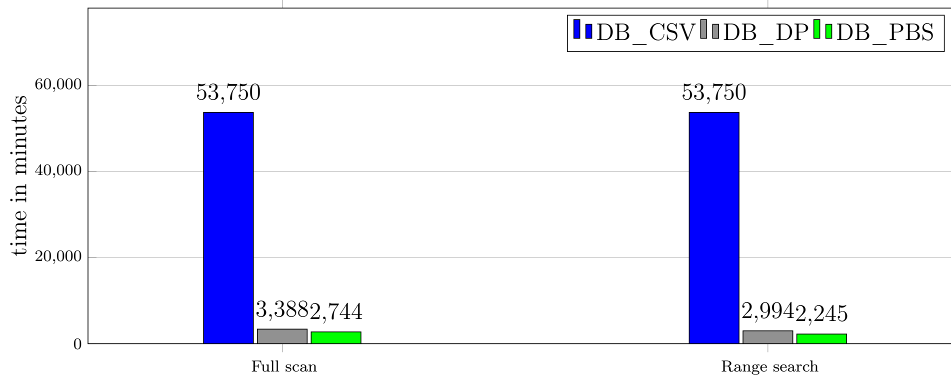
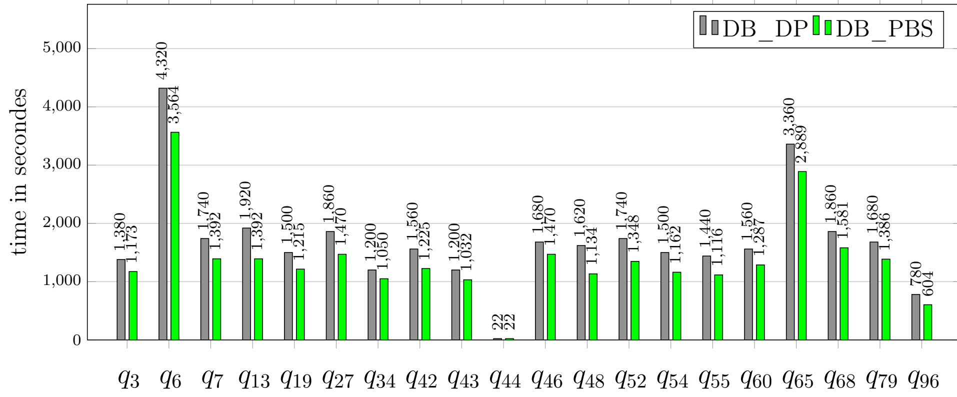
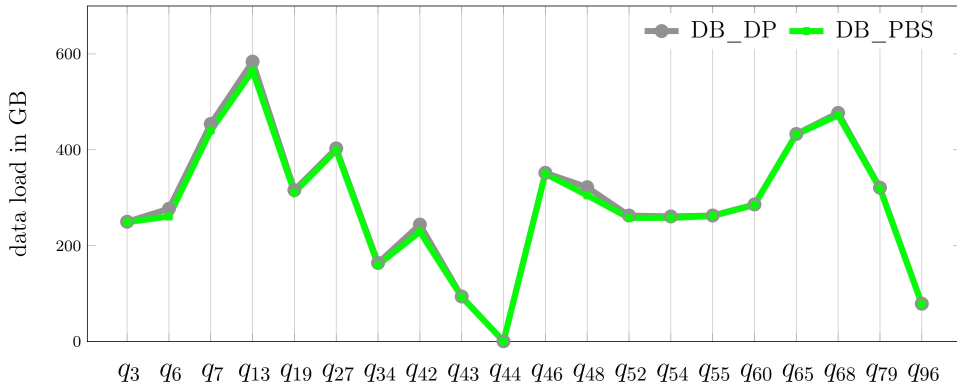


Figure 4.2: Execution time of the workload W using the three databases DB_CSV, DB_DP and DB_PBS with two modes.

[7] Note that by default, these features are enabled.



(a) Response time of the queries with full scan mode



(b) The amount of data loaded in each query with full scan mode

Figure 4.3: Queries performances with *DB_DP* and *DB_PBS* approaches

4.3.5 Discussion

First, we interpret the result of *RT* table (see Table 4.4). The fact table *store_sales* is bucketed and sorted by the attribute *ss_ticket_number* into 630 (Note that $630 \in [90, .., 1440]$, see **Rule 3**, section 4.2.3). Since we haven't any frequent attribute of *store_sales*, has been selected from the filters of the workload W, and because the measure *ss_ticket_number* is of integer type has low-density value and his skewness less than 2.0, we can apply **Procedure 1, case A.3 in section 4.2.3**. For dimension *customer*, since no frequent attributes of this dimension are used in the predicates, we bucketed and sorted the dimension by its

primary key *c_customer_sk* into 90 buckets (see **Procedure 2, case A** in section 4.2.3). For dimension *customer_address*, we note that the attribute *ca_state* is most frequently used in the filters, has a high density, and has low skewness value (i.e. $Sk \leq 2.0$), so we can partitioned this dimension by *ca_state* (see **Procedure 2, case B.1**). For dimensions *customer_demographics*, *time_dim*, and *date_dim*, we applied **Procedure 2, case C**. The case of dimension *item* like *customer_address*, such as we partitioned by the attribute *i_manager_id*. For dimensions *household_demographics*, *promotion*, and *store*, since their sizes are negligible (i.e. less than $10 \times S$, see **Rule 4** in section 4.2.3), we should not partition or bucket them. Moreover, we notice that we have added the attribute *time* as the fist partitioned schema for some tables to ensure the scalability of our approach (see section 4.2.5).

We can note from Fig. 4.2 and according to the *full table scan* and *range search* bar chart, that CSV database, i.e. DB_CSV, shows worst performance compared to the other approaches[8]. Such as the queries of W, executed with DB_DP and DB_PBS, are 20x faster than DB_CSV. The reason is in CSV format, we load the whole file into memory, not as in Parquet format, where we load only the attributes solicited by a query. We also notice that in CSV format the time in *full scan* mode is the same as the *range search* mode.

On the other hand, we notice that our approach has improved the execution time of the workload W up to 38% compared to DB_BP with *full scan* mode and up to 50% with range search mode. Moreover, we notice that the *range search* mode is better than the *full scan* mode for both the approaches. Thus, exploited *pushdown filtering* and *metadata statistics* of Parquet file have an impact on speeding up the scan for some tables (e.g., read min-max or null values of a column from file metadata)[9].

Moreover, it seems that the percentages 38% and 50% not promised rates of improvement, because almost of queries used in the workload W is selected

[8]Note that the execution time of the workload W is obtained as follow: T(W)=$\sum(T(q_i) * f_i)$, such as $T(q_i)$ is executed time of query q_i.

[9]Please note that in general *pushdown filter* is efficient only for the attributes of integer type.

randomly from TPC-DS benchmark, and they haven't any filters that used the attributes of the fact table *store_sales*, which is the biggest table in the data warehouse. So, since the highest rate of a query execution time taken by scanning the fact table, we obtain such results. For example, if we have an attribute of integer type in the table *store_sales* frequently used in the filters of the queries, we can obtain significant improvement in scan operation.

We also notice from Fig. 4.3a, that the best improvement have obtained in the queries $q13$ and $q48$ (up to 60%), the reason is that in the filters of these queries, the attribute frequently used is *ca_state*, which relevant to the large dimension *customer_addresse*. Also, we note that the execution time of the query $q44$ is negligible compared to the other queries, because the system doesn't scan the fact table *store_sales*. In Fig. 4.3b, we noticed that the amount of data skipped is little, although, with our approach PBS, we improved the query executed time between 25% to 60% compared to DB_DP, as illustrated in Fig. 4.3a. If we do some calculus, using SparkMeasure tool [58], we can note that the system in PBS approach has skipped to load 117.01 GB of data (such as 117.01=(5843.72-5726.71)=$\sum$(data load in each query $q_i \in W$ using DB_DP) - $\sum$(data load in each query $q_i \in W$ using our approach, i.e. DB_PBS)).

Finally, we should note that the use of the index has been introduced recently in Spark SQL to speed up the scan of HDFS blocks, such as the work of Cui et al. [28] and the current version of Apache Parquet (parquet-index version 0.2.3 or above). However, there are three essentials lacks when using the index in the current versions of Spark SQL: (1) Indexed columns must be top-level primitive columns with some type such "Integer, Long and String;" (2) Indexed columns cannot be the same as partitioning columns; and (3) Append mode is note yet supported for Parquet table when creating index (see Ref. [57]). Moreover, some NoSQL systems, such Cassandra [46] and HBase [61] use also index (e.g. *secondary index scan*) to improve query processing. In this thesis, we haven't compared our approach with these systems, and we plan to do this in the future works.

4.4 Conclusion

In this chapter, we have developed and implemented a novel strategy for partitioning and bucketing tables of a big relational data warehouse to speed up query processing in Hadoop ecosystems. Our experiments show that our PBS techniques not only allow to skip scanning some unnecessary HDFS blocks, but also improve the load balancing and the parallel treatment. Though our approach may not achieve max skipping level, however, our decision method, which based on density and skewness measures, is smooth and doesn't require to make some assumptions about the filters used in the workload. Moreover, although PBS approach is scalable for a large cluster and a massive amount of data, however, if the workload is significantly changed or when we update the data warehouse, we need to update our partitioning scheme, something that can harm the distributed system. We will address this problem in our future research.

We have a proposed in the precedent chapter, a novel approach based on the data-driven model, which can improve the star join operation and the OLAP query execution time. Moreover, in this chapter, we have proposed a new method to avoid scanning some unnecessary data blocks and to enhance the parallel treatment, base on a stable workload (workload-driven model). In the next chapter, we show how to combine between the two precedent contributions to build a novel physical design strategy for a distributed BDW over a Hadoop cluster.

Chapter 5

SkipSJoin: A New Physical Design for Distributed Big Data Warehouse on Hadoop Cluster

Big data will replace the need for 80% of all doctors
Vinod Khosla

5.1 Introduction

We have proposed in the two precedents contributions (see Chapters 3 and 4) two novel approaches. The first approach published in [1, 3, 4], is a static PLB based on data or schema-driven model of distributed BDW over a cluster of homogeneous nodes, this data placement allows performing star join operation in only Spark stage without a shuffle phase, something that can enhance the OLAP query execution time drastically compared to some partitioning schemes. In the second approach published in [2], we have proposed, some novels rules that allow omitted loading some unnecessary data blocks through a static OLAP workload and en-

hance the parallel treatment of a query being executed through a load balancing strategy.

In this chapter, we propose a new physical design for a distributed BDW over a Hadoop cluster, called **SkipSJoin**, which based on a model that is both data-driven and workload driven, using a static technique for the PLB of the data. We take into account: the size of the DW, the distribution of the foreign and the primary keys of the fact and dimension tables, the used query workload, and the cluster's characteristics. Our strategy allows performing the filtering, projection, and the star join operations of an OLAP query, locally and in a single Spark stage. Moreover, with **SkipSJoin**, we can avoid reading some data blocks that are not relevant to an OLAP query, based on a stable OLAP workload. We have developed and evaluated our approach on the TPC-DS [48] benchmark using Scala language over a cluster of homogeneous nodes, a Hadoop-YARN [60], a Spark [66] engine, and Hive [62].

The key points of our contributions are:

1. We propose a heuristic technique for the PLB of data to balance the split inputs of a distributed *BDW* evenly over a Hadoop cluster, to improve the parallel treatment for executing an OLAP query.
2. We propose a new data placement strategy which allows performing the star join operation in only one Spark stage without a shuffle phase, no matter what OLAP query is used, based on a data-driven model, using the bucketing technique.
3. We propose a new method of partitioning to skip loading unnecessary HDFS blocks through a stable workload.

The rest of this chapter is organized as follows. In Section 5.2 we detail our physical design, We present our experiments in Section 5.3, before concluding our work in Section 5.4.

5.2 SkipSJoin Approach

In the first part of **SkipSJoin**, we build horizontal fragments (buckets) of the fact and dimension tables of the DW, using our hash-partitioning method, i.e. bucketing technique (see section 5.2.2 and section 3.2.2 in Chapt. 3). Then, we distribute these buckets evenly over the cluster's nodes, in which we can execute the star join of an OLAP query locally and in only one Spark stage (i.e. one MapReduce cycle). The second part of **SkipSJoin** allows skipping the scanning of some unnecessary data blocks, by hash-partitioning some DW tables with the most frequent attributes of the queries' filters. That is, we extend the first part using a stable workload [1].

We suppose that we have a priori knowledge of the DW schema and the workload used, as well as the characteristics of the cluster. Our approach is composed of 6 steps (see Fig. 5.1): (1) Selecting the near-best number of buckets; (2) Adding a new partition key to the fact table; (3) Creating the new dimensions that contain the same bucketed key as the fact table; (4) Retrieving the most frequent attributes from the queries' filters; (5) Partitioning and bucketing the tables of the DW; and finally (6) Balancing the buckets over the cluster's nodes. Before giving the details **SkipSJoin**, let's formulate our problem.

5.2.1 Formalization

Suppose, we have a star schema DW E=$\{F, D1, D2,..,Dk\}$, such that F is the fact table and Dd, $d \in 1..k$, are the dimension tables. We denote by FK the set of all foreign keys of the fact table F coming from dimension Dd and by PK the set of primary keys of the dimensions Dd. We denote by Q the set of distinct queries used, such that Q=$\{q_1, q_2,...,q_m\}$, and by φ the set of the use frequencies of the queries q_j, $j \in 1...m$, such that φ=$\{f_1,f_2,...,f_m\}$. We define the workload W the

[1] A stable workload means the set of columns used in *Where* clauses and *Group by* remains relatively stable over time but the *filters* may change. The validity of this assumption has been empirically observed in a variety of real-world production workloads [10, 11].

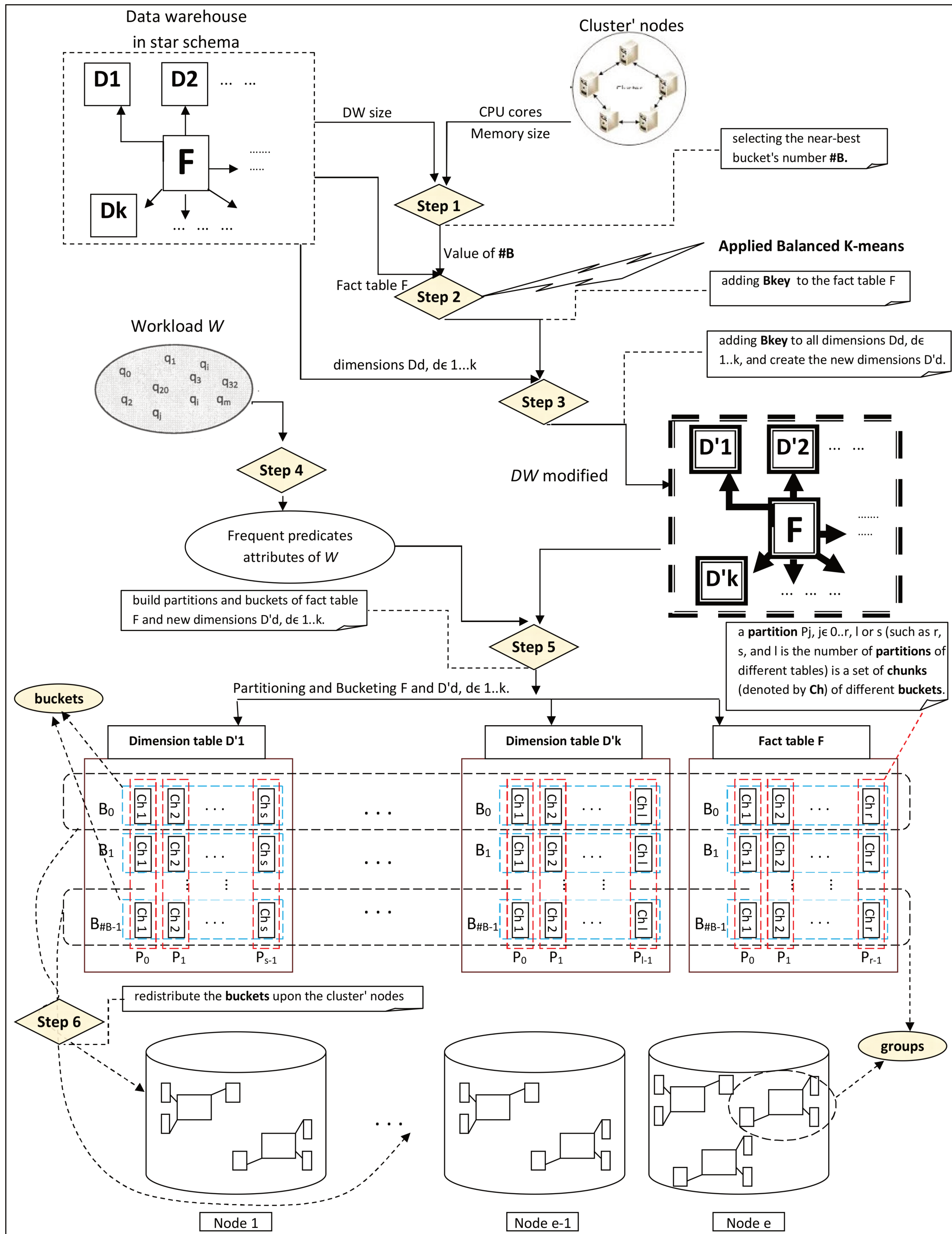


Figure 5.1: The steps of building SkipSJoin

set of all queries Q, used by its corresponding use frequencies φ, in a period of time t (e.g., t is one week), such that $|W|=\sum_{j=1}^{m} f_j$. We denote by $R=\{R0,...,Rk\}$ the set of all frequent attributes used in the queries' filters of the workload W, such that $R0$ is the most frequent attribute selected to hash-partition the fact table F, and Rd, $d \in 1...k$, is the most frequent attribute selected to hash-partition the dimension Dd (Note that at most $|Ri| = 1$, $i \in 0..k$). We denote by $Bkey$ the partition key used to bucket the fact and all dimensions tables in $\#B$ buckets, here $\#B$ is the number of the buckets should be created. We denote by $BF = \{BF_0, BF_1, ..., BF_{\#B-1}\}$ the set of distinct buckets created by bucketing the fact table F with $Bkey$ into $\#B$ buckets, and by $BDd = \{BDd_0, BDd_1, ..., BDd_{\#B-1}\}$, the set of distinct buckets of each dimension Dd, $d \in 1..k$. We denote by $BSF = \{BSF_0, BSF_1, ..., BSF_{\#B-1}\}$ the set of sizes of the buckets of the fact table F, and $BSDd = \{BSDd_0, BSDd_1, ..., BSDd_{\#B-1}\}$, $d \in 1..k$, the set of buckets sizes of each dimension Dd, $d \in 1..k$. Note that each bucket is composed of a set of chunks (i.e., files). We denote by a *group* the set of the buckets that have the same value of $Bkey$, it is composed of one bucket of F and one bucket of each Dd. We denote by $N=\{n_1, n_2,.., n_e\}$ the set of all homogeneous nodes of the cluster.

The first aim is to choose $Bkey$ and $\#B$ for building the buckets of BF and all BDd, $d \in 1..k$, in such a way as to keep them roughly balanced in size, i.e., with the minimum standard deviation of BSF and all $BSDd$, $d \in 1..k$, and how to distribute them over the cluster N to perform filtering, projection, and the star join operation, locally and in only one Spark stage (as we have done in Chapt. 3). The second objective of **SkipSJoin** is to construct and exploit the set R to skip loading unnecessary HDFS blocks of some DW tables through the workload W.

Figure 5.1 summarize the steps of our partitioning strategy. In the following, we detail our heuristic solution step by step.

5.2.2 Selecting #B and Bkey

In this section, we give in detail the first three steps of our approach (i.e., steps 1, 2, and 3 of Fig. **??**). In step 1, we determine $\#B$; In step 2, we show how to add the $Bkey$ column to fact table F, and how to fill this key; In step 3, we show how to add $Bkey$ to all dimensions Dd, $d \in 1..k$.

5.2.2.1 Selecting $\#B$

To know how to select the near-best bucket's number $\#B$, you can refer to the subsection 3.2.2.1 of Chapt 3.

5.2.2.2 Adding Bkey column to the fact table

To create a *group* of the buckets which allow performing the star join operation in a single Spark stage, we can add a new key $Bkey$ of integer type to all the tables of the DW E and co-partition the tables of E by $Bkey$. However, the way to fill the $Bkey$ column remains a challenging task. If the distribution of the values of $Bkey$ is skew and not uniform, we obtain unbalanced bucket sizes in BSF and all $BSDd$, $d \in 1..k$. In this case, our application seems un-parallelizable. So, to balance the split inputs of our DW, we should study how to calculate the values of the $Bkey$ column. Moreover, since the sizes of the dimensions are small compared to the fact table, we can focus on only minimizing the standard deviation of BSF and not all $BSDd$, $d \in 1..k$, by bucketing F with a simple range partitioning method. However, there is an essential factor that can affect the size of the newly constructed dimensions (denoted $D'd$): the similarity of the tuples in each bucket of BF. This can increase the number of tuples in each bucket of $BD'd$, $d \in 1..k$, as we will show in the following. To overcome this issue and obtain an approximately optimal solution, we use balanced K-means algorithm, as explained in subsection 3.2.2.2 (NewKey method).

5.2.2.3 Adding Bkey column to the dimension tables

After adding $Bkey$ to F, and in order to construct the *groups*, we must also add $Bkey$ to all the Dd, $d \in 1..k$, and obtaining new dimesions $D'd$, $d \in 1..k$. To do this, we carry out the following steps. First, we create an intermediate table IDd corresponding to the dimension Dd. The IDd table is composed of two columns, fk_d and $Bkey$, such that: (1) fk_d is the foreign key of dimension Dd in fact table F and (2) $Bkey$ is the partition key added in F. The IDd table initially has the same number of tuples as the fact table F. So, before joining IDd with Dd to obtain $D'd$, we delete all duplicate tuples in IDd. This method is similar to the NewKey method in Chapt.3. By creating $D'd$ we can build $BD'd$ and $BD'd$, $d \in 1..k$, (see section 3.4). Note that the sizes of the new dimensions $D'd$, $d \in 1..k$, are large compared to the original ones, their sizes are changed according to (1) the value of $\#B$ and (2) our clustering method applied to choose the values of $Bkey$ and to limit the data redundancy, however, the size of $D'd$ remains small compared to the fact table F whatever the size of the original dimensions Dd.

5.2.3 Selecting the Frequent Attributes

In this section (step 4 in Fig. 5.1), we show how to select the most frequent attribute used in the queries' filters, i.e., how to create the set R. The main idea in the creation of R is to hash-partition some tables through the corresponding attributes of R so that the system can skip the loading of unnecessary data blocks. You should note here that instead of selecting several frequent attributes for each DW table (as we have done in the approach of Chapt. 4), we choose only the most frequent attribute for each DW table if exist. The objective of this is to handle the metadata table size, persisted in memory by the Namenode. To construct R, but without using a sophisticated clustering method like the max skipping algorithm [108], we finish by using a smooth decision strategy which based on three essentials rules (see section 4.2.3). Note that this step is independent of the previous ones. Before explaining how to create R, we recall these rules.

• **Rule 1.** We consider that a distribution $Dist$ has a heavy data skew if the value of the skewness, denoted by Sk, is more than 2.0. There are numerous methods to calculate Sk of a given set. In our case, we use the equation 4.1 of chapter 4.
• **Rule 2.** Since our processing is in-memory, using the Spark engine, we assume that an attribute A of a table T has a high **density** D if:

$$\#B_{min} \leq 1/D(A) \leq \#B_{max}$$

where $D(A) = \frac{1}{number\ of\ distinct\ values\ of\ A\ in\ T}$, $\#B_{min}$, and the $\#B_{max}$ are selected as we have explained in the rules 2 and 3 of section 4.2.3.

To create the set R, we follow these steps: (1) We retrieve from the queries' filters all the attributes of integer type and we keep only the frequent ones, as we have done in section 4.2.2; (2) we keep only one attribute, i.e., the most frequent attribute, for each table of the DW E that has a high density D and the lowest value of the skewness Sk; Finally (3) we create the set R_i, $i \in 0..k$. For more details about these techniques, see chapter 4. The main reason for choosing these rules is to strike a balance between the number of HDFS files created and the sizes of these files.

5.2.4 Building the Partitions and the Buckets

After adding $Bkey$ to F, building the new dimensions $D'd$, $d \in 1..k$, and creating the set R, we can construct BF and $BD'd$, $d \in 1..k$. Thus, for each table T of the DW E, we hash-partition T by the corresponding attribute $a \in Ri$, $i \in 0..k$ if one exists, then we bucket T by the $Bkey$ column into $\#B$ buckets. For example, to build the buckets of the set BF, we assign each tuple of F that has the same value of $Bkey$ to the corresponding BF_i, $i \in 0..\#B-1$. Hence, for each table, T, we obtain a set of chunks, as shown in step 5 of Fig. 5.1.

5.2.5 Placement of the Buckets

In this section, see step 6 of Fig. **??**, we redistribute the *groups* created evenly over the cluster nodes, using round robin fashion. Formally, we can denote by $group_i = BF'_i \uplus_{d=1}^{k} BD'd_i,\ i \in 0..\#B-1$. Thus, we start to place the $group_0$ in node 1, $group_1$ in node 2,..., and the $group_{p-1}$ in the node e, such $e{=}p\ modulo\ \#B$ and $p <= \#B$. We restarted the operation with same way, we put $group_p$ in node 1, $group_{p+1}$ in node 2,..., until the last $group_{\#B-1}$. This physical design allows performing star join operation in a single spark stage whatever the OLAP query used, and allows to skip loading some unnecessary HDFS chunks of some DW tables.

Remark Note that the way to update the DW tables with our method is a trivial task as we have explained in section 3.2.5.

5.3 Experiments

To evaluate our physical design SkipSJoin, we have done some experiments as outlined in the next section.

5.3.1 Experimental Setup

In this section, we present the steps that implement our approach. First of all, we generate the DW using the TPC-DS benchmark; After that, we implement the different phases of our approach. We used a cluster of 15 slave data nodes and one master node characterized by CPU Pentium I7 with 8 cores, 16 GB of memory, and 2 TB of the hard disk. We installed in all nodes Hadoop-YARN V-2.9.2, Hive V-2.3.3, Apache Spark V-2.3.2, TPC-DS benchmark, Scala language, and Java. As we have done in sections 3.3.1 and 4.3.1.

5.3.1.1 Generation of the data and the workload

To generate the data, we adopted the *spark-sql-perf* [56] application, using the Scala language and Spark; we created a part of the DW composed of one fact table from the seven fact tables of the TPC-DS [48] benchmark and nine dimensions from among its seventeen, see the DW characteristic in Table 4.2. For the workload W, we selected 20 queries from among the 99 queries of the benchmark. Namely, we selected only the queries that solicited our DW tables. The set Q of the queries used and their frequencies φ are given in Table 4.3.

5.3.1.2 Implementation of SkipSJoin

To build the partitions and the buckets of our DW tables, we use the instruction:

```
DF.write.PartitionBy("a").bucketedBy(#B,"Bkey").sortBy("Bkey").format("parquet").
mode("overwrite").saveAsTable("DB.tablename")
```

Where *DF* is a $dataframe$, the column a is the partition key, retrieved from the set R (see Section 3.3), $\#B$ is the best number of buckets, *Bkey* is the bucketed key, *parquet* is the storage format used, *DB* is the database created in Hive, and *tablename* is the name of the partition table. If a does not exist, we only bucket *DF*. Before creating the buckets of the fact table $store_sales$, we add *Bkey* column. The values of this attribute are calculated using balanced k-means algorithm (see Sect. 3.2). To implement the steps 1 to 5 (see Fig. 5.1) of our approach, we have used two essential components: $Dataframe$ and $Dataset$.

For distributing the chunks, we use the same re-balancing technique used in section 3.3.1.2. Note that, although may we cannot ultimately ensure placing all buckets that have the equal value of join key *Bkey* in the same node; however the star join operation always performed in only one spark stage, without a shuffle phase.

5.3.2 Results

To evaluate our approach, we carried out some experiments with a BDW, denoted DW, which has about 5 TB in the CSV format (about 1420 GB in the Parquet format, see Table 4.2). We divided our experiments into two parts.

In the first part, we carried out some experiments without using the workload W, as NewKey method in the experiments of section 3.3.2. The aim of this is to show how to perform filtering, projection, and the star join operation in a single Spark stage whatever the used OLAP query. We have compared our approach with different baseline approaches in detail in Table 5.1. In this case, we have selected 6 queries from the TPC-DS benchmark with different levels of complexity (see Tables 3.3 and 3.4).

We executed the six queries with five values of $\#B$, $\#B \in \{90, 180, 630, 1080, 1440\}$. These values are selected according to our recommendations (detailed in Sect. 3.2.2.1). In SSH and SHB we set the parameter $spark.sql.shuffle.partitions$ to 630. In our approach, since we bucketed all the tables with the same join key $Bkey$ and since we disable the broadcast and shuffle join of Spark SQL, we can exploit SMB join correctly, which allows performing star join operation in only one Spark stage without a shuffle phase. With our configuration, we can run 90 tasks in parallel.

In the second part of the experiments, we included our hash-partitioning technique based on the W (we combine between the data-driven and the workload-driven strategy). So, in $SkipSJoin$, we use our bucketing technique as in the $SSMBO$ approach (NewKey method in Chapt.3) and our skipping method as given in detail in Section 5.2.3.

Figure 5.2 shows, for the different approaches, the execution time of the six selected queries. Note that in this experiment, we applied only our bucketing technique based on a data-driven model.

In Figure 5.3, we compare the runtimes of W in the approaches SHB, SSMBO,

Table 5.1: Notations for the approaches

Notation	Description
SSH	default partitioning and distributing schema of Hadoop/Spark, using default Spark Shuffle Hash join (*SH* join). (like repartition join [20] in MapReduce.). We deactivated Hash Broadcast Join by setting spark.sql.autoBroadcastJoinThreshold to "-1".
SHB	default partitioning and distributing schema of Hadoop/Spark, using Hash Broadcast Join.
SSMBO	our partitioning schema (i.e., our bucketing technique only without based on queries workload), using balanced k-means, and exploiting Spark *SMB* join optimization.
SSMBO'	like SSMBO but without using balanced k-means algorithm. In this case, we just create roughly equal buckets' size of the fact table, using range-partitioning method.

Table 5.2: Reference table *RT*

Table name	P. attributes	B. key
store_sales	–	*Bkey*
customer	–	*Bkey*
customer_address	*ca_state*	*Bkey*
customer_demographics	*cd_gender*	*Bkey*
item	*i_manager_id*	*Bkey*
time_dim	*t_hour*	*Bkey*
date_dim	*d_moy*	*Bkey*
household_demographics	–	*Bkey*
promotion	–	*Bkey*
store	–	*Bkey*

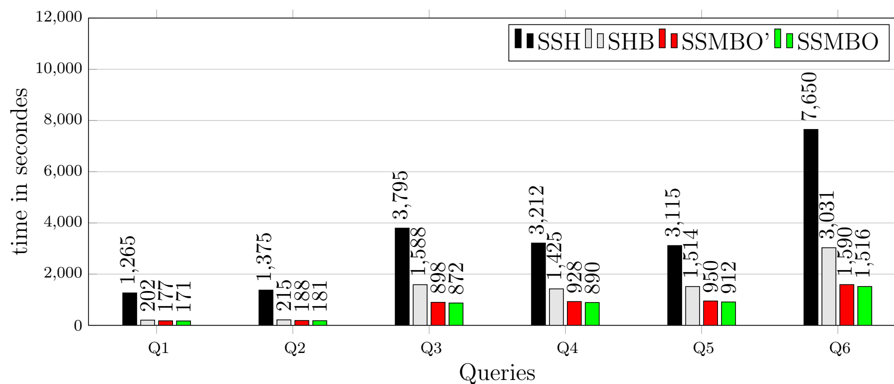


Figure 5.2: Runtime of the queries

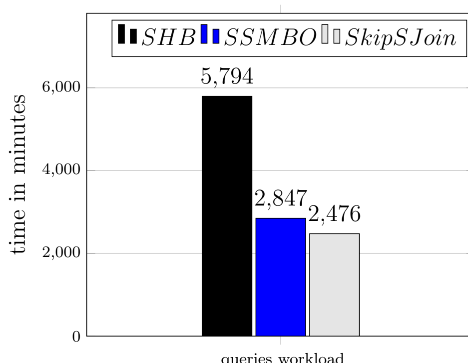


Figure 5.3: Execution time of W with different approaches

and the hybrid approach SkipSJoin, using Table 5.2 to guide our partitioning and bucketing strategy. Note, that we get Table 5.2 from the set R and the bucketed key $Bkey$. So, to perform W, we randomly selected a query from W w.r.t φ, then we executed it in one Spark session. In this way, we delete at each time all the RDD partitions created during the previous query execution, and we can ensure that the system can't read data from the HDFS cache. Note that in our experiments, we turn-on three essential features: Parquet push-down, statistics filtering, and dictionary filtering.

5.3.3 Discussion

As shown in the bar chart of Fig. 5.2, the query execution time with SSMBO approach is up 2 times better over the SHB approach. We can see that the worst results obtained with the SSH approach; this is due to the high rate of data shuffling and JVM overheads. In $Q1$ and $Q2$, since we have selected few attributes of only two small dimensions *item* and *date_dim*, the broadcasting of the RDD partitions become fast and as a result, the execution time of these queries in the SHB approach is roughly the same as with our strategies SSMBO' and SSMBO. However, in Q3, Q4, Q5, and Q6, the performance of SHB suffers. The reason is that in the SHB approach when the table is large, the system cannot broadcast it, and must combine with broadcast join and the shuffle join (i.e., SSH approach) to

perform these queries.

We noticed that the runtime of the queries with SSMBO approach is much better than the SSMBO' approach, and this demonstrates the efficiency of our balanced k-means algorithm. The random clustering applied in SSMBO' can increase the size of some new dimensions and degrade the system performances, as we have explained in section 3.3.3. Moreover, we can see from Fig. 5.3 that our hybrid approach SkipSJoin is much better than SSMBO. The reason is that since we hash-partition some tables by the most frequent selected attributes from the queries' filters of W, we can omit loading some chunks into memory.

We have seen that our physical design SkipSJoin is reliable and has benefits for OLAP queries, especially with large dimensions. We have also seen that our approach is much better than the other state-of-the-art schemes. We have also dealt with the problem of data availability and balancing the split inputs. Also, by considering a given workload, we have omitted loading unnecessary data blocks.

5.4 Conclusion

In this chapter, we have presented a new strategy for partitioning and distributing a big data warehouse over a Hadoop cluster. Our approach SkipSJoin [6] allows performing the star join operation of an OLAP query locally and in only one Spark stage, without a shuffle phase. Moreover, by taking into consideration the given workload, SkipSJoin can skip loading some unnecessary data blocks. Furthermore, We can adapt SkipSJoin smoothly to other processing engines, such as Tez [69] or Flink [55]. We have seen that although we have roughly balanced the split inputs, we get unbalanced intermediate results due to the selectivity of some of the filters.

In the next chapter, we will propose a dynamic approach to balance on the fly the reducer loads or the intermediate results (i.e., the mapper outputs, after performing filtering, projection and the star join operation).

Chapter 6

SGMAS: Optimize Group-By Operation Using Multi-agent System

Every company has big data in its future, and every company will eventually be in the data business
Thomas H. Davenport

6.1 Introduction

We have seen in the previous chapters (i.e., Chapt. 3 to 5), different static approaches of partitioning and load balancing of a distributed big data warehouse over Hadoop cluster. These approaches enhance star join operation and skip loading some unnecessary data blocks when scanning the DW tables. Our experiments show that these approaches improve the OLAP query execution time drastically.

OLAP queries are typically expensive queries that take a long time to be performed over a distributed Big Data Warehouses (BDW). In distributed BDW, im-

proving OLAP query processing is a challenging task. An OLAP query is composed of several clauses, and it usually takes the form of "*Select... Function()...From... Where...Join-Predicates...Filters...Group-By....*" Each clause can be performed in the map phase or in the reduce phase, and each operation executed in one or several MapReduce iterations or Spark stages, with a considerable amount of data shuffled among the Datanodes. For example, in some case, the star joins operation will need $n-1$ or $2(n-1)$ [96] Spark stages with many shuffle phases, where n is the number of tables used by the OLAP query. A star join is not the only expansive operation for an OLAP query, *Group-By* also may involve considerable communication cost during the shuffle phase and may incur the stragglers (a straggler is a task that performs more poorly than similar ones due to insufficient assigned resources) in the reduce phase, especially with Rollup and Cube operators.

In our previous works [6] (see also chapters 3 to 5 for more details), a new physical design for a distributed BDW has been proposed, which allows executing projection, filtering, and the star joins operation of an OLAP query in only one Spark stage, without a shuffle phase, and helps the system to skip loading some irrelevant data blocks through a stable workload. This chapter is a complement to our project [6], where we propose a smart method, called SGMAS (Smart Grouping using Multi-Agent System), to balance the reducer loads, something that can improve *Group-By* operation and aggregate functions execution time.

The standard method to enhance *Group-By* task for an OLAP query is to balance the Intermediate Results (IRs) and avoid the need to obtain heavy data skew in the Reducer Inputs (RIs). Note that while the balancing of data split inputs (e.g., chunks of HDFS blocks) can be handled by using a static balancing technique [1, 3, 4, 6], since the load balancing decisions may be known in advance, however, to make an efficient partition scheme of the RIs, the system needs to pick up some knowledge on the fly, by using a dynamic technique of Partitioning and Load Balancing (PLB) of the data. Numerous studies have tackled the issue of the load imbalance between reducers for the MapReduce paradigm. While we cannot list them all in this chapter, we can divide them into two categories. The first category [27, 37, 73, 98, 110, 120] attempt to balance the RIs in advance, by changing the default hash-partitioning technique used by some distributed systems

such as Hadoop and Spark. The second category [16, 80] uses heuristic algorithms to balance reducer loads during reducer process, namely, after the shuffle phase. Whatever the technique used to balance the reducer loads, the algorithms used for the PLB technique used to balance the reducer tasks must be executed swiftly and in real-time, and should not affect the job runtime itself.

Some works of the first category [98, 110, 120] have used sample technique with approximative calculus to obtain near-optimal solutions. Although these methods can give an efficient partition scheme of the RIs, by using a sampling technique over a considerable amount of data [81] can provide wrong results, because some of the mappers' keys may not consider in the distribution. Moreover, some other works such as [40, 73] have changed the default mechanism of MapReduce, such that they require waiting until all or majority of map tasks are completed to gather partition size information before shuffle phase can begin.

Note that changing the default synchronizing method between the mappers and the reducers can: (1) overload the memory buffer of the IRs and the system may crash; (2) if the mapper loads are heavily skewed, then we obtain stragglers in the map phase and the job runtime increases. On the other hand, the solutions of the second category (i.e., [16, 80]) may also degrade the system performances and slow down the query processing, because the system must halt the reducer tasks before it triggers the re-balance algorithm. Our approach SGMAS is similar to Gufler et al. [40] solution, such we used Multi-Agent System (MAS) to balance on the fly the reducer loads, however, the main key of our contribution is that we keep the default mechanism of the MapReduce paradigm.

In many practical scenarios, the OLAP workload is stable during a period and may slightly be updated if the need analysis changes; that is, the set of columns used in *Where* clauses and *Group-By* remain relatively stable over time[1]. This assumption has been empirically observed in a variety of real-world production workloads [10, 11]. This means that we have a high probability to obtain the same *fragments* (here a data *fragment* is the subset of all tuples with the same key and list(values)[43]) produced by the mappers when performing these OLAP

[1]Obviously, the variation in filters also impacts the *fragments* produced by the mappers

queries on a cluster nodes. Hence, the occurrence frequencies of these *fragments* also remain stable[2] during a period of time and may change when new workload comes or when we update the Data Warehouse (DW).

Meanwhile, with some cluster Resource Manager (RM) such as YARN of Hadoop, Mesos, and Spark, the execution of a set of OLAP queries over cluster nodes can be done concurrently or sequentially. In other words, the RM assigns the resources to the users that launch a set of queries, where these queries have a different level of complexity, and the RM can affect dynamically or statically, and with a smart way, a specific set of resources (e.g., executors or workers) to perform each query, this means that the RM can allocate the appropriate idle resources for each query (i.e., each query has a minimum and maximum of resources needed to be performed). Namely, the *fragments* produced by a query before *Group-By* operation would be the same every time (except when we update the DW or when the workload change); however, for each execution of a query, the number of the mappers and the reducers (i.e., CPU cores and the number of partitions (or buckets) in the reducer phase) assigned by the RM to execute a query varies from time to time.

Based on the OLAP workload specifications and some real RMs features, we can propose a smart method to balance the RIs on the fly and improve *Group-By* operation runtime. SGMAS approach can benefit from the previous executions of the queries such that the system can react with an online fashion to make an efficient partition scheme of the RIs.

Our idea is as follows: we add to the query optimizer of such distributed system, like Spark SQL or Hive-QL, a MAS which can spread the *fragments* smartly over the reducers, such that a smart agent learns through interaction with the environment and exploit the execution history of the queries, to adapt on the fly a good partition scheme of the RIs, which can help the system to parallelize the treatment when performing the aggregate functions.

[2]However the distribution of the *fragments* over the reducers changes from an execution to another following data locality and the number of reducing buckets (or partitions).

The rest of this chapter is structured as follows. In Section 6.2, we explain our issue through an example. We detail our approach in Section 6.3. We present our experiments in Section 6.4, and we conclude in Section 6.5.

6.2 Problem Explanation and Motivation

The following example explains our issue. We assume an OLAP query Q from TPC-DS benchmark, such as:

```
SELECT d_year, i_brand_id, SUM(ss_sales_price)
FROM  date_dim, item, store_sales
WHERE date_dim.d_date_sk = store_sales.ss_sold_date_sk
AND store_sales.ss_item_sk = item.i_item_sk
AND i_manufact_id = 128
GROUP BY d_year, i_brand_id;
```

The execution plan of this query with Spark SQL system is as follows: for each table, the system scans HDFS blocks, retrieves the attributes involved by the query, executes filters (e.g., *i_manufact_id = 128*), then it performs the star join operation (e.g., using SMB join as in [4, 6]), and finally it executes *Group-By* and *SUM*(ss_sales_price). Figure 6.1 shows a part of the execution steps of this query after running star join operation. In Fig. 6.1, we have 6 map tasks executed by 3 mappers (e.g. 3 CPU cores) and 3 reduce tasks executed by 3 reducers. The mappers generate 7 keys. Note that each *fragment* (i.e., shape) has a *key*, and the number inside the shape represents a list(*values*) of this *fragment*. For example, the shape triangle of map task M_1 has the *key* number 1 and the list(*values*) contains 20 *tuples*, such that each *tuple* contains the *value* of (d_year, i_brand_id, ss_sales_price). Note that in this example, we do not consider combine phase[3] (i.e., we calculate SUM function only in the reduce phase). By using the blind hash-partitioning method, the reducers R_1 receives 7 *fragments* with 141 *tuples*,

[3]Some functions, like standard-deviation and variance, could be only performed in the reduce phase.

reducer R_2 receives 6 *fragments* with 74 *tuples*, and R_3 receives 6 *fragments* with 40 *tuples*. We notice that we have a problem of load imbalance between the reducers. Obviously, the runtime of the job is taken by the slower reducer R_1 that contains 7 *fragments* with 141 *tuples*.

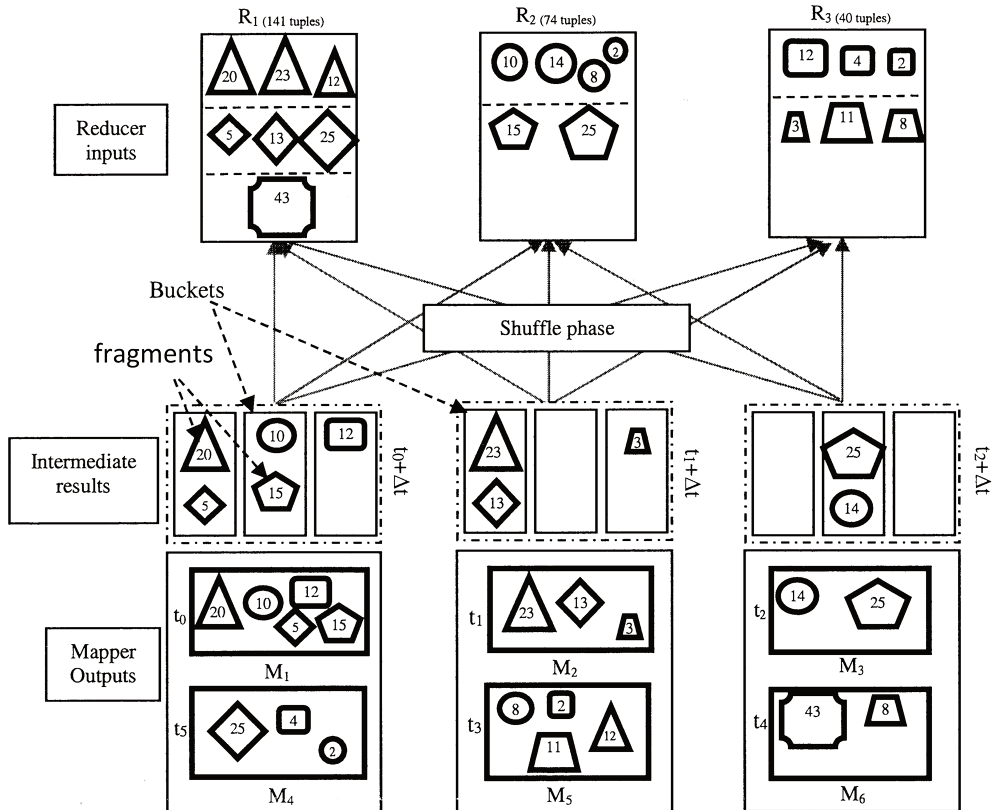


Figure 6.1: Distribution of the fragments (key, list(values)), before and after shuffle phase with default hash-partitioning method

With the default mechanism of Hadoop MapReduce, when the mapper executes the first map task M_1 and write-out the result in the buckets of the IRs, at time $t_0 + \triangle t$ (note that in Fig. 6.1 $t_0 < t_1 <...< t_5$), the shuffle phase can start and the system transfers the *fragments* over the reducers. Also, the reducers R_1, R_2, and R_3 cannot begin performing the reducer function until all of the mappers finish their work. This mechanism invokes that each mapper is unaware of the actions taken by the other mappers for each map task. In addition, each mapper

cannot know the numbers of the *fragments* produced by the other mappers, nor their sizes (number of the tuples of the *fragment*). Therefore, we need to develop a method that able to define or predict the unknown parameters cited above (i.e. the *fragments* produced by each mapper and the appropriate place of these *fragments* in the bucket of the IRs), in order to balance the reducer loads.

In several real systems, each query retrieved from the OLAP workload, has a use frequency, i.e., many users use it, So, if the system use the blind hash-partitioning technique to distribute the *fragments* over reducing buckets (in *Group-By* operation), then a considerable time is lost. If we assume that we have thousands of OLAP queries that are used by thousands of users, then the time lose will be exponential when the system uses the blind technique. Hence, proposing a smart distribution of these *fragments* is a mandatory requirement to overcome this issue. In the following, we detail our SGMAS approach.

6.3 Our Proposal SGMAS

As we have indicated earlier, we can resolve our issue using a smart method based on multi-agent system, such that each mapper become an Agent-Mapper (AM), and each reducer becomes an Agent-Reducer (AR). We can consider that the number of the mappers and the reducers are equal to the number of CPU cores assigned by the RM of the system. So, the RM allocates appropriate idle resources to perform each query launched by a user (Note that each query executed over a subset of Datanodes called small-cluster). In this small-cluster, the system (driver or Application-Master) create a MAS which composed of a Supervisor Agent (AS) (i.e., handle-agent or actor), a set of AMs (as workers or actors in Spark [66] or Ray [59] system), a set of ARs. This MAS exchange information throws a Knowledge Base (KB) (which can consider as a black-board component). This KB shared between all of the existing small-clusters. In other words, with the KB, the AS and the other agents can coordinate with each other to build an efficient partition scheme of the RIs. This KB contains the meta-data of the execution history of the OLAP queries which can help the AMs to assign the *fragments* to

the appropriate reducer. Namely, through the KB, an AM can place each *fragment* in the appropriate reducing bucket.

We can consider that the problem of distributing the *fragments* over the reducers is similar to a stochastic games or repeated games, which can be modeled by Markov Decision Process (MDP), where the components of the game are the *fragments* and the actions of this MDP is to place the *fragment* produced by the AMs in the appropriate reducing bucket. This MDP has two unknown parameters, the *fragment* provided by each AM (i.e., the state), and where the AM places this *fragment* in the appropriate bucket of IRs (i.e., the transaction). The standard method to define or predict these parameters (i.e., expect the payoff values (or rewards) of each action and the probabilities of the transactions) is to explore and exploit the history executions of the queries, by using, e.g. Reinforcement Learning method, such as the Bayesian Multi-Agents Reinforcement Learning (BMRL) technique of [26]. However, since the space of the states (i.e., number of *fragments* produced at each stage) and the area of the actions (i.e., the number of possible places to assign the *fragments* over reducers) are considerable in big DW environment, this involves that using the previous solutions is an infeasible task. Meanwhile, since, in our case, we can know the *fragments* produced by each query through the KB, then the AS can find the best distribution of the *fragments* over the reducers that can achieve the near-best equilibrium [74], by using a smart method as we will show in the next sections. Note that making the best equilibrium means obtaining light data skew in the reducer loads.

Before detailing the architecture of SGMAS, we will first define some concepts and notations used in this chapter.

6.3.1 Preliminary Concepts and Notations

As we have seen earlier, we can consider that distributing the *fragments*, produced by the AMs, over the ARs, for a *Group-By* operation, as stochastic games G with n-player cooperative (have common interest). Each player i is an AM_i. From

standard game-theoretic terminology [86], we assume a collection α of n AM[4] and n AR, each AM_i $i \in \alpha$ having available to it a finite set of individual actions A_i. An action $a_i \in A_i$ taken by an AM_i is placing a *fragment*(*key*,list(*values*)) in the appropriate bucket in the IRs (see Fig. 6.1). Note that the number of the buckets in the IRs is the same as the number of the partitions of the reducers, denoted by p. We denote by a *fragment* key $F.key$, and its size (sum of tuples) $F.size$. The chosen actions by all of the AMs of the set α, at any state s, constitute a joint action, denoted by ă$=\times_{i\in\alpha} a_i$. At each execution of ă, the system passes from stage l to stage $l+1$. An AM_i can reach a state s' after executing an action a_i from the state s. An AM_i executes some available actions $a_i \in A_i$ to achieve its objective.

A randomize strategy for an AM_i is a distribution $\pi_i \in \Delta(A_i)$ (where $\Delta(A_i)$ is the set of all possible distributions of the *fragments* over the AM's action set A_i). We denote by a strategy profile $\Pi = \{\pi_i : i \in \alpha\}$, a collection of all possible strategies for all the AMs. Note that at each time $t \in T = \{t_0, t_1, ..\}$, an AM_i may follow any strategy π_i^t. It is possible that we have $\pi_i^{t_0}=\pi_i^{t_1}$, where $(t_0, t_1) \in T$. We denote by the distribution profile $\Pi^e \in \Pi$, the profile where the system can achieve the equilibrium [74]. In other words, the distribution $\Pi^e \in \Pi$ allows to balance the partitions of the reducers. We denote by $D = \{v_1, ..., v_p\}$ a distribution values of the *fragments*' sizes over p partitions[5] at the final state, such as $v_j = \sum_{k=1}^{f_j} Fk.size$, and f_j is the number of the *fragment* Fk in each partition j ($j \in 1..p$). For example, in Fig. 6.1, at the final stage, we have $D = \{141, 74, 40\}$.

In our approach SGMAS, since we can pick up the final *fragments* distribution, for each query, through the KB, we do not need to define the probabilities transaction for each action, nor the received rewards as used in some prediction mechanism (e.g. BMRL method of [26]).

[4]You should note that n is the number of CPU cores used to perform a query and is not the number of chunks of HDFS blocks, this means that each AM_i treats many map tasks.

[5]each partition treated in one reducer task (like the parameter: *spark.sql.shuffle.partitions* in Spark.)

6.3.2 Architecture of our system

In this section, we explain the architecture of our system (i.e., the whole cluster features). To make on the fly an efficient partition scheme, for each query executed in small-cluster, the driver of the cluster or the Application-Master creates a MAS to perform smartly a query launched by a user. In this chapter, we focus on showing the features of this MAS when performing the Group-By operation of an OLAP query (see the architecture of small-cluster in Fig. 6.2). The driver of the system keeps track of a sufficient history of the queries execution. This history maintained in the Knowledge Base (KB) of our platform.

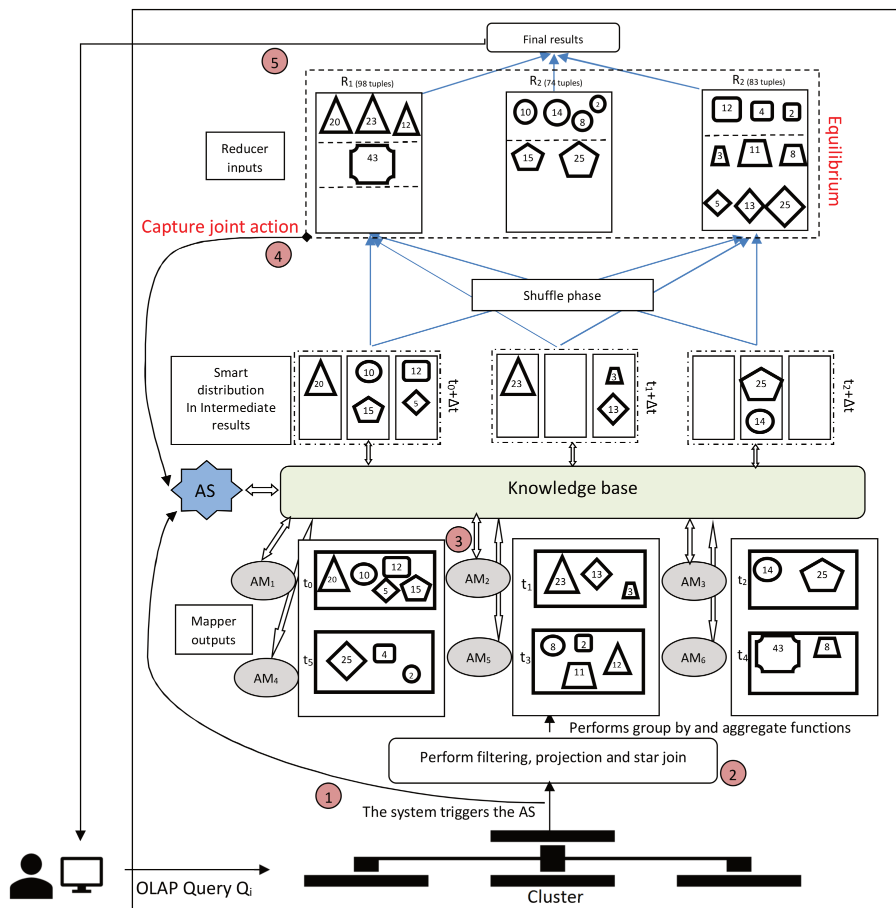


Figure 6.2: Smart distribution of the *fragments* over the reducers in a small-cluster of our system

After that, the Application-Master creates a MAS, the AS of this MAS monitors appropriate observable history from the previous execution of this query to create the best profile distribution scheme. In other words, each AS of a MAS can create the best profile distribution scheme throw the *fragments*' meta-data maintained in the KB and helps the AMs to achieve the near-best equilibrium. Note that the KB shared between all these small-clusters. In Fig. 6.3, we show an example of some possible transactions observed in the RIs by an AS in a small-cluster, using the blind hash-partitioning method denoted $\Pi^b \in \Pi$ (Π^b is the example detailed in Section 2). The observations in Fig. 6.3 are captured at the moment of the shuffle phase, from the partitions of the reducers.

In SGMAS system, the trade-off between long-term benefit and short-term cost should be addressed. That is: (1) the size of the meta-data stored in the KB must be the smaller as possible (else the distributed system prefers to maintain the results of the queries or cached them rather to store this meta-data) and (2) the execution time of our mechanism to create the near-best profile must be rational and must not affect to the query runtime itself. So, before detailing the execution process of our architecture, we start to define some constraints and rules.

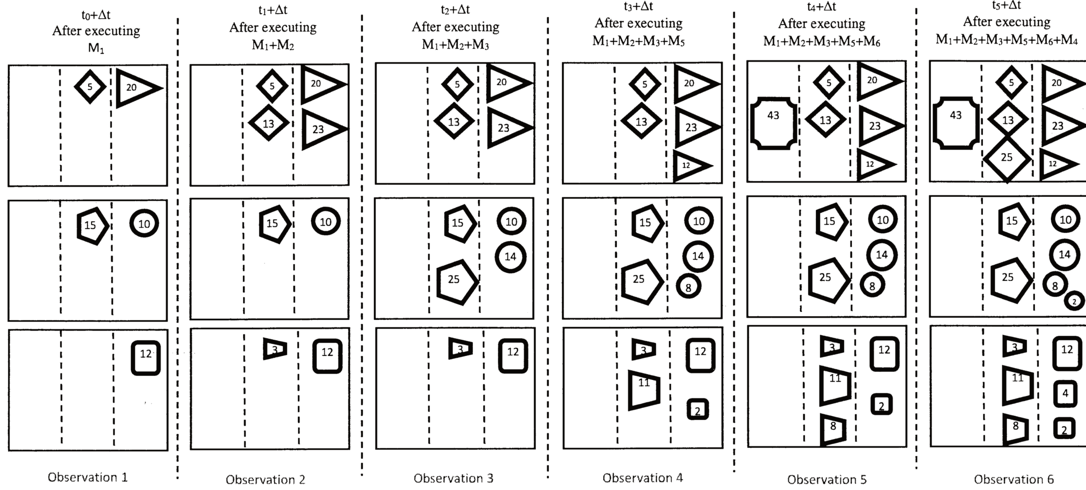


Figure 6.3: Different observations captured in sequence times of reducer inputs, using blind hash-partitioning method.

6.3.2.1 Knowledge Base (KB) Size

We should note that using a Reinforcement Learning technique such as the work of [26], need to pick up and maintain a considerable amount of data in the belief state (i.e., in KB), to make a right prediction. In our case, keeping all of this information (i.e., different probabilities transactions and different possibles received rewards for each action executed) is an unnecessary and infeasible task.

In SGMAS, we can cure the above problem as follows: for the first execution of a query q, launched by any user u, at a time $t \in T$, The driver creates a MAS, the AS maintains only in KB, the critical information that can help it to define the best profile distribution in the future execution. Therefore, the AS maintains a part of the strategy profile Π_q information. This part denoted Π_F_q, is composed of the different *fragments* produced by each AM_i at the final stage (i.e., after executing all joint actions). Consequently, from this profile, the AS can define the strategy profile Π_q^e that can help the AMs to achieve the best equilibrium (see an example of the best distribution in a small-cluster, in Fig. 6.2). To measure the equilibrium, see subsection 6.3.2.2.

In the KB, the structure of Π_F_q is a dictionary with multiple keys, as shown in Fig. 6.4. Such that the key is a list of the query names, e.g., q_1 and q_5 in Fig. 6.4, and the value is a list of *fragments* meta-data. For each new execution of a query q at time t, the driver of the system creates a MAS running in the small-cluster. The AS creates, if q not exist in KB, a new *list* (i.e. new key) for this query and a list of values that contain some meta-data of all the *fragments* produced by the AMs (see lists values $L1$,..,Ls in Fig. 6.4). For the sake of clarity, we call *fragment-merged* the *fragment* maintained in the KB. Note that we have many keys (query names) for each list of the *fragments-merged*. That is, it is possible to find some queries that produce the same *fragments*. As shown in Fig. 6.4, we maintain only some meta-data of the *fragments-merged*, namely, the *fragment* key, and the sum of all the *fragments* sizes that have the same key. Note that the size of Π_F_q is negligible. The driver can create many MASs to perform the queries concurrently and also controls how the ASs of these MASs read and write the lists in the KB.

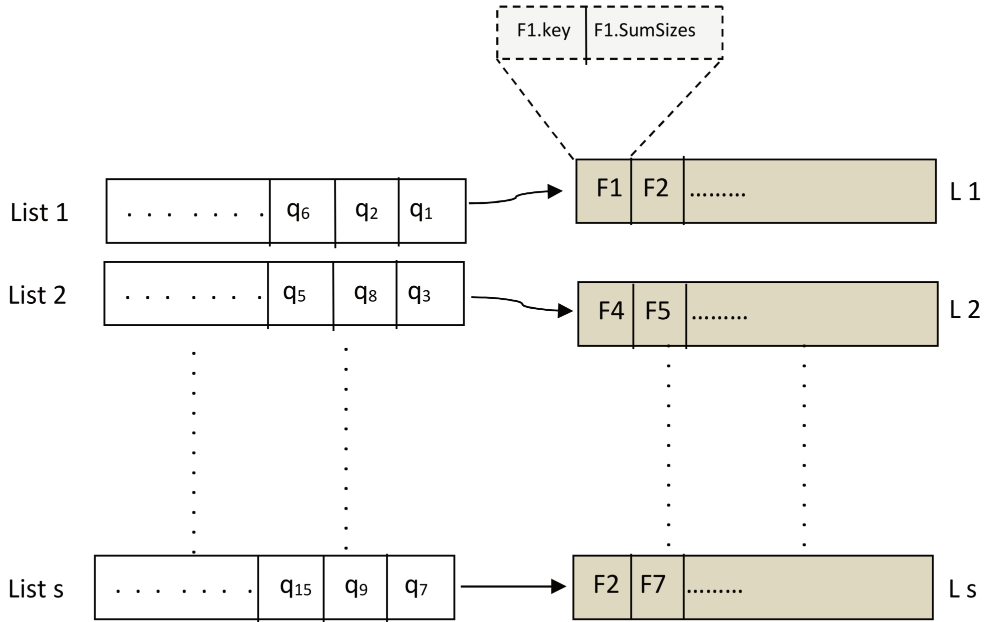


Figure 6.4: How the system store the previous joint actions of the history queries execution in KB

6.3.2.2 Equilibrium Measurement

In our approach, the equilibrium is evaluated using the following covariance measure:

$$Cov = \frac{std}{mn} \times 100 \tag{6.1}$$

Where std is the standard deviation of a distribution D (see notation in subsection 6.3.1), and mn is the mean value of D. For example, in Fig. 6.2, after performing the last joint action, we obtain $D1 = \{98, 74, 83\}$ and $Cov1 = \frac{12.12}{85} \times 100 = 14.25$. For the example of section 6.2, we have $D2 = \{141, 74, 40\}$ and $Cov2 = \frac{51.39}{85} \times 100 = 60.45$. We can see that $Cov1 < Cov2$, so, the best equilibrium is that it has a minimum value of Cov. In the case when we have more than one equilibrium (i.e., $Cov1{=}Cov2{=}\cdots$), the system randomly chooses any best profile Π_q^e. In our platform and according to some recommendations works [36, 39], to determine the light and heavy data skew, we can distinguish three types of equi-

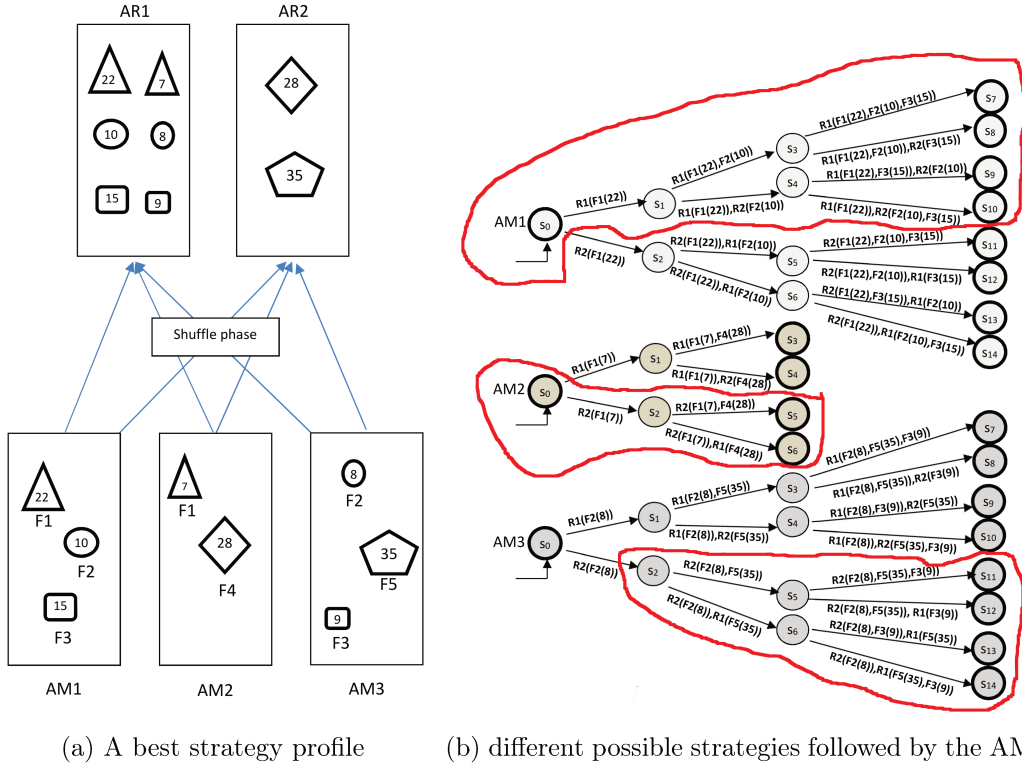
librium: the best equilibrium is when $Cov(D) < 20.0$; if $20.0 \leq Cov(D) < 40.0$, the system achieves suboptimal equilibrium; the non-equilibrium or the worst case is when $40.0 \leq Cov(D)$. Briefly, the objective of our approach is not reaching the perfect equilibrium, but we look for the best or at least suboptimal equilibrium.

6.3.2.3 Avoid Infeasible Calculus

As we have seen earlier, our aim is to achieve the best equilibrium, or at least obtain suboptimal ones, and avoid drastic distribution of the *fragments* over the reducers. Figure. 6.5 shows another example of distributing 5 *fragments* by 3 AMs over 2 reducers $R1$ and $R2$ (i.e. two partitions). Fig. 6.5 (a) shows a strategy profile Π^e, where the system achieves best equilibrium ($Cov(D) = \frac{51.39}{85} \times 100 = 8.44$), and Fig. 6.5 (b) shows different possible strategies profiles π_i^t, that can be followed by each AM_i at time t (here we have 3 AMs). Note that some strategies are not reachable. For example, the two strategies surrounded by the red color cannot be realized together, because if AM_1 decides to place the *fragment* $F1(22)$ in $R1$, AM_2 cannot place $F1(7)$ in $R2$ because we have the same *fragment key*. The same case between AM_1 and AM_3 with $F2$. Also, we can notice that some actions should not be taken, because they are not needed anyway (i.e., repeated transactions). Moreover, some computations are not necessary at all and may be infeasible, especially with a high number of the *fragments* and the reducers (note that the possible transactions for an AM_i that distributes c *fragments* ($c > 1$) over r reducers is given by the equation $\delta = \frac{1-r^{c+1}}{1-r} - 1$).

Given these reasons, developing a smart technique that can define the best Π^e for each query or at least obtaining suboptimal equilibrium, with simple calculus, is a mandatory requirement to achieve our objective and enhancing *Group-By* operation.

After defining some constraints and rules of our system, in the following section, we explain in detail our smart method SGMAS.



(a) A best strategy profile (b) different possible strategies followed by the AMs

Figure 6.5: Example of distributing 5 fragments by 3 AMs upon 2 reducers

6.3.2.4 Architecture Detail

The architecture of our small-cluster is composed of 5 steps, as shown in Fig. 6.2. In Fig. 6.2, the number inside the small circle shape represent the number of the step. We summarize these steps in Algorithm 3. In this Algorithm, we have as inputs a query q, launched by a user u, and the meta-data of the *fragments-merged* in KB, if exist (see notations in section 6.3.1). The output of the Algorithm 3 is the table Π_q^e that can help the AMs to make an efficient scheme of the reducer loads (i.e., balance the partition sizes) and to speed up *Group-By* operation and aggregate functions. As mentioned in line 3 of Algorithm 3, the resource manager of the cluster assigns the idle resources, located in different cluster nodes (i.e., small-cluster), which are sufficient to perform the query q; that is, the driver or the Application-Master of the system define n, and p parameters for each MAS created.

Algorithm 3: The steps our approach

Input: $q \in W = \{q_1, q_2, ..., q_f\}$, KB

`/*` q `is an OLAP query to be launched by a user` u `and KB is the knowledge base of the system */`

Output: OLAP query out put.

1 **while** *true* **do**
2 user u launch a query q;
3 define the parameters n and p by the RM of the system ; `/*` n `is the number of the AMs and ARs (CPU cores) and` p `is the number of partitions. see Fig. 6.2 */`
4 the driver triggers the AS of the MAS;
5 $exist$:=**CheckingExisting**(q) ; `//` *exist* `is boolean variable`
6 the driver or application-master starts to perform filtering, projection and star join operation;
7 **if** $exist$==*true* **then**
8 **DefineBestProfile**() ; `/* the AS define the best profile if exist, i.e. create` Π_q^e`. */`
9 **else**
10 the AMs use the default blind hash-partitioning method to distribute the *fragments* ;
11 the AMs continue the execution of the query through the default mechanism of the MapReduce paradigme, using Π_q^e or the default blind hash-partitioning method ;
12 **if** $exist$==*false* **then**
13 the AS captures from the reducers the last joint action taken by the AMs and retrieves all meta-data of the *fragments* produced ; `// before to start executing the function in the reduce phase.`
14 The AMs perform the function of the OLAP query q ; `// e.g. Sum, Average, and so on`
15 the AS return the result to the driver and this one send it to the user u ;

The driver assigns one worker to the AS of the MAS and n workers for the AMs and ARs. After that the system (e.g., Application-Master or driver in Spark)

triggers the AS in line 4 (i.e., step 1 of our approach), the AS checks the existing of q in KB through the function **CheckingExisting(q)** (see line 5). In this function, we have used a simple lexical and syntactic analysis method to check the similarity of q in KB. In step 2 (i.e., line 6 in Algorithm 3), the Application-Master continues to perform filtering, projection and star join operation. Note that the step 1 and 2 done in parallel. When the AS ensures the existing of q in the KB, it launches the **DefineBestProfile()** procedure in line 8. The result of Algorithm 4 is Π_q^e table, which contains the different actions must be taken by the AMs to achieve the equilibrium; that is, the best place of each *fragment* in the bucket of the IRs. This table is maintained in the KB and broadcasted over the nodes, such that all the AMs can read it locally. Table 6.1 shows the structure of Π_q^e table. Before detail the Algorithm 4, we continue to explain Algorithm 3.

In the step, 3 of our architecture the AS decide to use either the default blind hash-partitioning technique or the scheme of our SGMAS approach. So, if q does not exist in KB, the AMs use the default hash partitioning technique to distribute the *fragments* (see lines 7 to 10 in Algorithm 3). Moreover, in our case, the AMs follow the default mechanism of the MapReduce paradigm (see line 11); that is, after the first AM_i finishes distributing its *fragments* over the buckets of the IRs, the shuffle phase can start transferring the *fragments*.

In the line 13 (i.e., step 4 in Fig. 6.2), the AS captures the different *fragments* in the RIs before performing the merge operation (i.e., before gathering the *fragments* that have the same key in one partition) and executes the aggregate functions. Finally (i.e., step 5 in Fig. 6.2), the AS outputs the results to the driver, and this one send it to the final user.

The inputs of Algorithm 4 are the variable *list* of the *fragments-merged* produced by a query q, retrieved from the KB and the variable P which contains the list of the partitions. In the line 1 of Algorithm 4, we create the list *stack* to sort the *fragments-merged list* in a descended way; that is, the *fragments-merged* that has the most significant size goes in the first position, and the second-highest goes next, and so on. In line 2, we calculate the average size of *stack*. This parameter (i.e., *range*) helps to balance loads of the reducers (i.e., the partitions). The idea of our approach is trying to distribute the *fragments-merged* evenly over

Algorithm 4: Define best profile

```
Input:  listF = {F1, F2, ..., Fk}, P={P_1, P_2,..,P_p}
/* listF is a list of fragments-merged of KB and P is a set of the
   partitions treated by the ARs.                                          */
Output: Π_q^e ;              // table of fragments placement (i.e., actions)
1  stack:=listF.sort ;                         // sorted list in descended way
2  range:=⌈(Σ_{j=1}^{k} Fj.size) / p⌉ ;              // average size of reducers
3  nP:=1, round:=1 ;                           // nP number of the partition
4  for j:=1 to p do
5      P[j].size=0;                                  // initialize partition
6  while (stack!=null) or (round<=p) do
7      pos:=0; decho:=stack.size;
8      while (P[nP − 1].size ≤ range) or (pos<stack.size) do
9          frag=stack[pos] ;                  // get the first fragment-merged
10         P[nP − 1].size:=P[nP − 1].size+frag.size;
11         if (P[nP − 1].size ≤ range) or (frag.size>=range) then
12             P[nP − 1].add(frag);
13             stack.deleteElement(frag);
14             round:=1;
15             pos:=pos+⌈decho/2⌉, decho:=⌈decho/2⌉;
16         else
17             P[nP − 1].size:=P[nP − 1].size-frag.size;
18             pos:=pos+⌈decho/2⌉, decho:=⌈decho/2⌉;
19             if pos>=stack.size then
20                 round++;
21     nP:=nP mod p+1;
22 DitributeRemindFragments(stack, P);
23 CreateBestProfile();
```

Table 6.1: Π_q^e structure

	Query q			
Fragments	F1	F2	...	Fk
Actions	Pos_1	Pos_5	...	Pos_2

the reducers of the set P, which can minimize the covariance Cov (i.e., obtain best or at least sub-optimal equilibrium). We can distinguish two significant problems that can obstruct to achieve our objective: (1) may some *fragments-merged* size are bigger than the average size *range*, and (2) the distribution values of the *fragments-merged* size of *stack* list is fort imbalanced. The two above problems increases the value of Cov and slow down the *Group-By* operation.

To cure the problem (1), we have decided to distribute these *fragments-merged* earlier (i.e. in the first round of the loop *While*, see condition ($frag$.size$>=$*range*) in line 11), because, if the system split this big *fragments-merged* to the small chunks (as some state-of-the-art approaches done [40, 110]), it needs to create more MapReduce cycles to complete evaluating the aggregate functions. To resolve the issue (2), we have adapted the round-robin fashion to distribute evenly the *fragments-merged* over the reducers. Moreover, we have used the dichotomy search to scan all the elements of *stack* list. The advantage of this is to fill in each reducer with variant size values of the *fragments-merged* (see lines 15 and 18). This technique allows for minimizing the degree of data skew between the reducers.

At the end of Algorithm 4, we add the procedure **DitributeRemindFragments (*Stack*,*R*)** to distribute the reminders of the *stack* elements, and the procedure **CreateBestProfile()** (see Algorithm 6) to create Π_q^e table. The idea of Algorithm 5 is simple, such that we put the higher *fragment-merged* size in the lower partition size. **Note that our heuristic method, used in Algorithm 4 ensures achieving best or suboptimal equilibrium in several cases, as we will show in our experiments**[6].

[6]Note that if the distribution of the *fragments* sizes is heavy skewed, then we obtain the stragglers in the reduce phase.

Algorithm 5: DitributeRemindFragments($Stack$, P)

```
1 while (stack!=null) do
2     frag:=stack.getElement() ;                  // get the first element
3     part:=min(P) ;                              // select the lower reducer in size
4     part.add(frag) ;                            // add the fragment to the reducer
5     stack.delete(frag) ;                        // delete the fragment from stack list
```

Algorithm 6: CreateBestProfile()

```
1 indice=1;
2 for (j=1 to P.size) do
3     part=P[j];
4     for k=1 to part.listFragments.size do
5         Π_q^e[indice].key=part.listFragments[k].key ;    // get a fragment key
6         Π_q^e[indice].position=j;                         // placement of the fragment in the
          partition.
7         indice++;
```

6.4 Experiments

To evaluate our approach SGMAS, we have done some experiments, as outlined in the next sections.

6.4.1 Experiment Setting

We have evaluated our approach SGMAS on a practical test cluster where the experimental environment based on the Ray [85] system, which allows simulating of the MapReduce paradigm. Ray [59] is a flexible, high-performance distributed execution framework designed for reinforcement learning applications. Ray is a

hybridization between Bulk-synchronous parallel systems such as MapReduce and Apache Spark, and the actor model for asynchronous tasks such as akka [50]. Note that we have not directly implement SGMAS on Spark or Hadoop for two reasons: (1) it is not trivial to change the default synchronization mechanism of these systems, especially with the current API of Spark SQL (we plan to do this in the future works); (2) we can smoothly simulate some baseline approaches on Ray cluster such as Hash-partitioning partitioning approach (denoted HashP for short), or Gufler et al. [40] approach (denoted Closer), and compared them with our method SGMAS.

6.4.1.1 Implementation

We have evaluated SGMAS on a cluster composed of 6 nodes, each node has 8 CPU cores and 16 GB of memory size (note that these nodes haven't GPU). We have installed in all nodes Redis [45] server, Ray library version 0.7.0.dev3, Anaconda [65] system with python version 3.7.3, and other dependencies libraries.

In Ray cluster and unlike Hadoop or Spark cluster, each node can be the master node at any time by executing the instruction *ray start - - head - -redis-port*=6379, such that 6379 is the port of Redis server. A node can joint the cluster after performing the instruction "*ray start - - redis-address=IP-address:*6379", such that *IP-address* is the address of the head node previously defined. Ray system cannot use all resources of the nodes, so we must keep some resources for the operating system and other applications. Namely, we use in each node, 6 CPU cores, and 12 GB for Ray system and Redis server.

To implement the agent mappers (AMs), agent reducers (ARs), and the agent supervisor (AS), we have used the actor model of Ray. The actor works autonomously in the node. When we instantiate an actor, a new worker created, and the methods of this actor are scheduled on that specific worker and can access and mutate the state of that worker[7]. The below example shows a part of the program

[7] A worker is a process that executes tasks invoked by a driver or another worker [85]

code in Python to implement the AS:

```
import ray
import numpy as np
....
@ray.remote
class AgentSupervisor(object):
  numPart=1
  step=1
  ....
  def buildBestProfile(self):
      if self.checkExistOfQuery():
      listClusters.sort(key=lambda C: C.sumsize,
      reverse=True)
      ....
  .....
```

In the previous code, we defined the actor as a class with some attributes and *methods*. To execute these *methods* remotely, we add the instruction *@ray.remote* before to define the class. Tha AS defined in the Application-Master (i.e., Driver or main program) as a Handle-actor[8].

The structure of the knowledge base KB is a dictionary *dict()*. The other agents, namely, AMs and ARs, have the same structure as the AS but not handle-actor. The result of the AS is a dictionary represents the best profile Π_q^e, such that the key of this *dict()* is the *fragment* key and their value is the number of the bucket of the IRs of each AM. With the Ray system, it is not hard to simulate the MapReduce paradigm with asynchronous *methods*. So, when the first AM finishes its works (i.e., write out all the *fragments* in the buckets), the AS can trigger the ARs to perform the shuffle phase (i.e., transform data from each bucket of the AM to the partition).

[8]Note that for the sack of clarity, our Application-Master create one MAS to show how our SGMAS method work.

To estimate the advantages and the defects of SGMAS, we have compared it with five baseline approaches as detailed in Table 6.2. We have tested these approaches in a dataset that has a different level of data skew (i.e., different queries level complexity), as we will show in the next subsection.

Table 6.2: Approach notations

Notation	**Description**
HashP	HashP is the default mechanism in the Spark computation environment and Hadoop MapReduce. HashP can obtain a good performance only when keys equally appear with a uniform distribution of the cluster sizes.
RangeP	The range partition algorithm is widely used in partitioning distribution. In this method, the intermediate (key, value) tuples are sorted by key first, and then the tuples are assigned to reduce tasks according to this key range sequentially. RangeP can enhance the data balance among reduce tasks if data (i.e., size of *fragment*) is uniformly distributed [67].
SCID	SCID algorithm [110](i.e. Splitting and Combination algorithm for skew Intermediate Data blocks). SCID uses a sampling method to predict the frequent keys produced. Note that although this algorithm uses sampling technique such as [120] and [98] which can affect to the load balancing decision, it is efficient even with Big Data, since it distributes smartly the remainder mapper keys which haven't considered in the sampling algorithm.
Closer	This dynamic method [40] distributes on the fly the *fragment* by changing the default synchronization mechanism of MapReduce paradigm.
LEEN	Ibrahim, S., et al. [73] have developed LEEN approach to minimize data transfer during shuffle phase by ensuring data locality in reducer task. To do this the authors have changed the default synchronization mechanism of MapReduce to make good partition scheme of reducer loads.

6.4.1.2 Preparation the data and OLAP workload

In the current version of Ray, there is no API as Spark SQL to execute SQL queries. So, given that our aim is to improve *Group-By* operation, we do not need to run the filtering and join process in our experiments. To well show the skew in the mappers and the reducers we have used the TPC-DS benchmark which vouched for DBMS to implement data skew [97].

We have generated data with *spark-sql-perf* tool [56], using Scala language and Spark, where we store data directly in HDFS. We set the HDFS replication factor to 1 only, and we keep the default parameters of Spark. Moreover, since we use star DW schema, we have used a part of this benchmark composed of one fact table among seven and nine dimensions among 17 of the TPC-DS benchmarks (see Table 6.3). We have generated two DWs, the first denoted DW1, has 10 GB of size, and the second indicated DW2, has 100 GB (in parquet format). Furthermore, we have selected and adapted 18 queries among 99 ones of the benchmark; namely, we choose only those queries that solicited our DWs tables, and we eliminate some nested queries and those that use ROLLUP and CUBE operator[9].

We cannot run these queries directly in Ray; we must transform as shown in Table 6.4. So first we delete *Group-By* operation and aggregate functions, and we keep the other operations. After that, we performed the queries transformed with Spark, and we write-out the result on HDFS with Parquet format. We can assume that each result represents a materialized view stored as files (chunks) in HDFS. Finally, we can load these files with Ray cluster, and performing *Group-By* operation and aggregate functions of the original queries. Note that each chunk (i.e., Parquet file of this materialize view), loading by an AM, will be transformed to a Panda DataFrame[10] to facilitate its treatment.

[9]Note that in this chapter we do not consider the operators *CUBE* and *ROLLUP* which require to generate other keys and values in the intermediate results

[10]Note that Pandas on Ray become a component of Modin. Moreover, unlike Spark DataFrame, the in-memory format for Pandas on Ray is a Pandas DataFrame on each partition.

Table 6.3: Characteristics of the DW tables

	Table name
1	*store_sales*
2	*customer*
3	*customer_address*
4	*customer_demographics*
5	*item*
6	*time_dim*
7	*date_dim*
8	*household_demographics*
9	*promotion*
10	*store*

6.4.2 Performance Evaluation

Since our aim is enhancing *Group-By* operation and aggregate function for an OLAP query, and optimize the parallel treatment, we have evaluated and compared SGMAS with the baseline approaches cited in Table 6.2, in two aspects: (1) execution time and (2) load balancing.

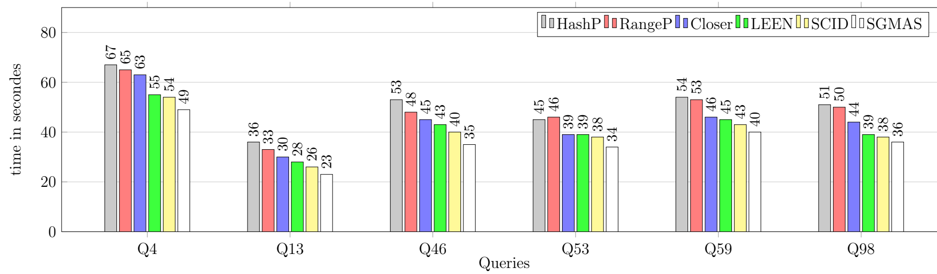
(1) Execution time. We have divided the 18 queries selected in two categories, those that have light and medium data skew of the mapper keys, and those that have heavy skew distribution. Figure 6.6 shows the execution of the queries with DW1 and Fig. 6.7 with DW2 in different approaches. Note that for the sake of clarity, we have shown the execution time of 9 queries among the 18 queries.

With light and medium data skew distribution of the mapper keys (see Fig. 6.6 (a) and Fig. 6.7 (a)), SGMAS outperforms almost the baseline approaches. We can notice that the worst results obtained with the blind approach HashP and RangeP, this due to the unaware distribution with these techniques. In Fig. 6.6 (a), SCID

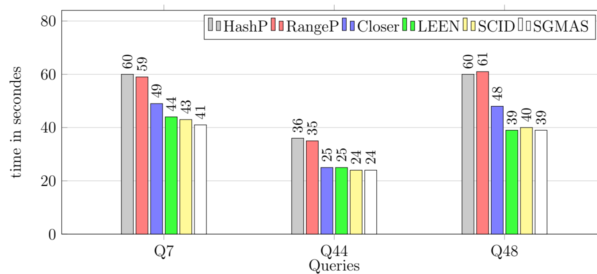
Table 6.4: Query tranformation

Query original	**Query after transformations**

`SELECT d_year, i_brand_id, SUM(ss_sales_price)`
`FROM  date_dim, item, store_sales`
`WHERE date_dim.d_date_sk=store_sales.`
`ss_sold_date_sk`
`AND store_sales.ss_item_sk=item.i_item_sk`
`AND i_manufact_id=128`
`GROUP BY d_year, i_brand_id;`
	`SELECT d_year, i_brand_id, ss_sales_price`
	`FROM  date_dim, item, store_sales`
	`WHERE date_dim.d_date_sk=store_sales.`
	`ss_sold_date_sk`
	`AND store_sales.ss_item_sk=item.i_item_sk`
	`AND i_manufact_id = 128;`

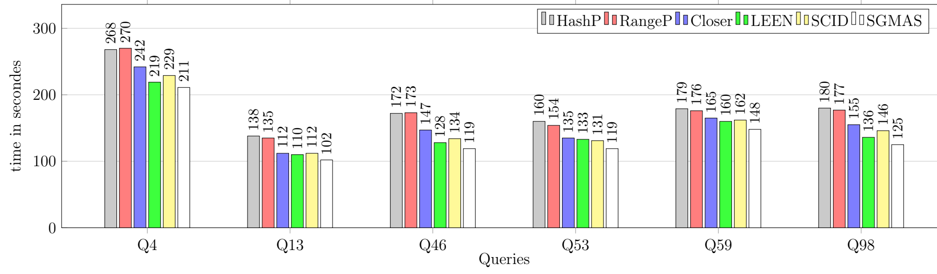


(a) Light and medium data skew of the mapper keys

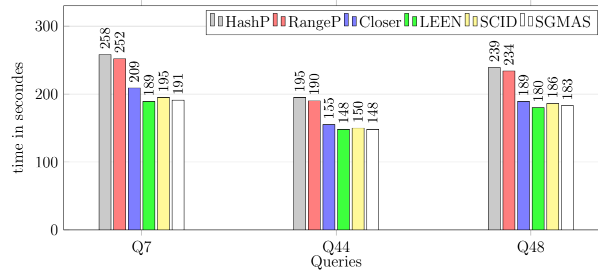


(b) Heavy data skew of the mapper keys

Figure 6.6: Queries runtime in different approaches with DW1



(a) Light and medium data skew of the mapper keys



(b) Heavy data skew of the mapper keys

Figure 6.7: Queries runtime in different approaches with DW2

has slightly had the same performance as SGMAS; the reason is since the number of keys is not too large, the time of sampling in SCID becomes negligible. In Fig. 6.7 (a), LEEN provides roughly the same result as SGMAS with the queries Q4, Q13, and Q46, this because LEEN focuses on data locality optimization and this decrease the amount of data shuffled between the nodes, compared to the other approaches. Moreover, We can see that although the Closer approach can provide better load balancing than all the approaches, as we will show in the next experiments, however, it takes a considerable time to gather some relevant information that helps to make right load balancing decision. Hence, the results in Closer still not promised.

With heavy data skew in DW1, i.e., Fig. 6.6 (b), the results of SGMAS, LEEN, and SCID are roughly the same. The reason is that in SGMAS, we don't look for achieving optimal load balancing (i.e., getting optimal equilibrium), which involve some stragglers in the reducers and the query runtime increases. In Fig. 6.6 (b)), due to the rise of data load (the amount of data shuffled growth), LEEN is slightly better than the other approaches. We can see that Closer remains suffer because

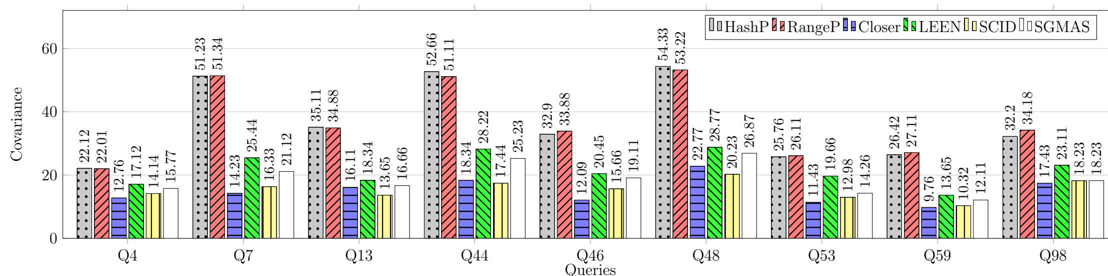


Figure 6.8: Load balancing of the reducers with DW1

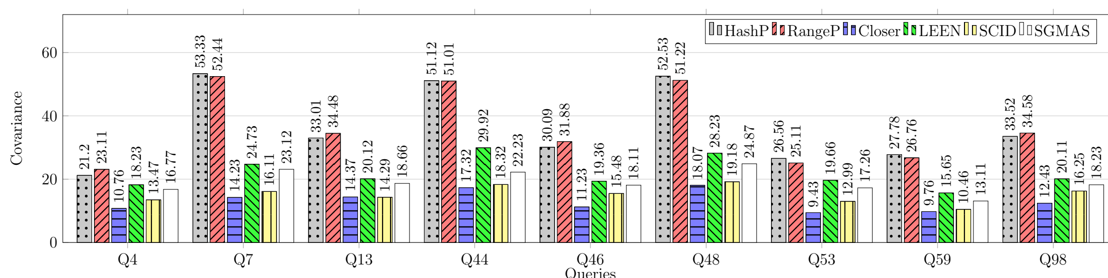


Figure 6.9: Load balancing of the reducers with DW2

of the unbalance in the mapper tasks. Further, we can see that the long-time took with Q4, this because in this query, we select many attributes of the large dimension *customer* without using any filters.

(2) Load balancing. Figures 6.8 and 6.9 show the load balancing of the reducers. We can see that although the execution times of the queries with SGMAS are better than the baseline approaches; however, in the load balancing, SGMAS is not the best, this because Closer and SCID look to achieve optimal load balancing, while in our approach, we look for getting best or suboptimal load balancing, and we focus on decreasing the *Group-By* execution time. We can notice that the blind methods HashP and RangeP give the worst load balance results, this due to the unaware distribution of the keys over the reducers. We can see also that since LEEN focus on data locality optimization, it is not well in the load balancing aspect, compared to Closer and SCID approaches.

6.5 Conclusion

In this chapter, we have presented a new method to enhance *Group-By* operation for OLAP Workload. Our contribution is a complement to the previous static approaches (see Chapts 3-5). In the SGMAS approach, we have used a novel dynamic partitioning technique, based on multi-agent system, which can improve *Group-By* and aggregate functions without change the default synchronization mechanism of the MapReduce paradigm. Our experiments show that SGMAS outperforms the baseline approaches in the situation of light or medium data skew distribution of the mappers' keys and provide promise results with heavy data skew. We have also seen how to model the MapReduce paradigm using asynchronous tasks through Ray system. Moreover, although SGMAS doesn't achieve optimal load balancing of the reducer loads, such in the SCID and the Closer approaches, however, our method is smooth and efficient in several kinds of OLAP workload.

In the future research: (1) we consider more complicate operators such as CUBE and ROLLUP operators (2) we improve our approach through data locality technique such as LEEN approach and (3) we integrate our algorithm in Spark SQL or Hive-QL system.

Chapter 7

General Conclusion and Perspectives

The data fabric is the next middleware.
Todd Papaioannou.

7.1 Conclusion

We have seen in this thesis different static and dynamic partitioning and load balancing techniques for distributed Big data warehouse over Hadoop cluster. We have seen in the state-of-the-art chapter the different partitioning and distributing techniques that can improve the performances of databases and data warehouses in both the parallel DBMS and in distributed systems (i.e., MapReduce technologies) such as Hadoop and Spark. Although in this thesis, we have focused on improving the OLAP query executed time on distributed big data warehouse over Hadoop cluster, however, and to clarify more the different partitioning and load balancing techniques applied in the shared-nothing systems, we have tackled in the chapter 2 different approaches that enhance OLTP and OLAP workload for both parallel

DBMS and in the distributed systems that use MapReduce paradigm.

We have proposed a novel static partitioning and load balancing technique on distributed big data warehouse over Hadoop cluster to enhance the star join operation for an OLAP query. This operation increases the disk spill when scanning large dimensions and the fact table. In addition, this operation rises the network communication cost drastically during the shuffle phase, especially with large dimension tables.

To overcome the above issues, we have proposed a new data placement strategy, which allow to perform the star join operation in only **one Spark stage** (i.e., **one MapReduce cycle**), without a shuffle phase, whatever the OLAP query used (i.e., approach based on data-driven model). With our static partitioning scheme, the distributed system can become ovoid the memory overhead, since we control in advance the size of the partitions (or the buckets), based on a heuristic technique. Moreover, our approach is scalable for the considerable amount of data and for the large cluster as our experiments demonstrate, and is stable for the DW updates. Although our static PLB technique has some limitations, however, we can cure them in the future works and extend it without making a complex changed to our DW scheme.

To enhance scan operation and the parallel treatment for an OLAP query being executed on a distributed BDW over a Hadoop cluster, we have proposed a novel approach which able to avoid scanning some unnecessary data blocks through the filters of the OLAP workload. Our static PLB technique also improves the parallel treatment and prevent the memory overflow, which can happen following the increase of the metadata table size, persisted in memory by the Namenode. These have cured in our PBS approach through some Data Mining methods. Moreover, although our static PLB techniques cannot achieve max-skipping level, however, our process is smooth and reliable for any stable OLAP workload (i.e., without making any assumptions on the filters). Furthermore, although our PBS approach has some limitations, since, is evaluated on a stable workload, however, we can deal with the workload changed and DW updates based on some techniques. We will plan in the future works (see our perspectives in the next section).

We have also proposed a hybrid approach, when we have combineed between our data-driven approach presented in Chapt. 3 and our heuristic strategy described in Chapt.4, which based on a workload-driven model. Our new physical design for a distributed big data warehouse, use both partitioning and bucketing techniques and ability to perform the star join operation in a single Spark stage, without a shuffle, and to avoid loading some HDFS chunks that are not relevant to an OLAP query. As the previous approaches, this scheme has also some drawbacks that will plan to resolve them in our future research.

The approaches presented in the chapters 3, 4, and 5 are based on static PLB of the data, using data-driven and workload-driven models, and have tackled the problems of the star join and data scan operations for an OLAP query. As we have discussed before, OLAP query is typically expensive query which composed of some cost operations, such as filtering, projection, star join, group-by, and aggregate functions. While optimize filtering and star join operation could be handled through static partitioning and load balancing technique, since we can collect in advance some knowledge that can help the distributed system to elaborate good DW scheme (i.e., balance the data split inputs), however, to improve *Group-By* operation execution time and load balancing the reducing buckets, we need to use a dynamic technique of PLB of data, where the system collects on the fly some pertinent information that can help it to balance the intermediate results or the reducer inputs.

Although the SGMAS approach has tested on a simulated environment with Ray system, however, our experiments reveal that SGMAS is suitable and reliable and outperforms some baseline approaches in term of execution time. Note that it is not hard to adapt SGMAS in real query processing systems such as Spark SQL and Hive-QL. Moreover, we can extend our solution by considering other operators such as *Cube* and *Roll-up*, as we have planned in the perspectives.

7.2 Perspectives

Nevertheless, no work is perfect, and all work must be subject to constructive criticism. We have seen in the conclusion section and the different chapters of our thesis, the limitations of our proposed approaches , so in the future research we plan to:

1. Attempt to extend the first approach with a cost model (such as the cost model of [17, 105]), to demonstrate (not only with experiments evaluation as we have done in section 3.3) that our data placement strategy gives better performances than some partitioning scheme for star join operation.
2. Improve our technique method to select the best bucket' number in the approaches of chapters 3, 4, and 5.
3. Compare the performances of our physical design of a distributed big data warehouse, deployed on a Hadoop cluster (see chapter 5 and our work [6]), with No-SQL data warehouses schemes, such as Cassandra and HBase system.
4. Extend our heuristic technique (NewKey method used in chapters 3 and 5) to support other DW schema (not only star data warehouse schema), such as constellation schema, e.g., joining two fact tables.
5. Study how to deal with the workload changed in the static approaches proposed in the chapters 4 and 5, such we extend these approaches in a way that the distributed system can re-balance the data warehouse automatically over the cluster node if the workload changed, without interrupting the user transactions.
6. Plan to implement our SGMAS approach (presented in chapter 6) in Spark SQL query processing system.
7. Plan to take into account other sophisticated operators such as as *Cube* and *Roll-up* to balance the reducer loads with our SGMAS approach. Note

that using these operators in Group-By operation need to produce new keys in intermediate results, and obviously the load balancing decision of our SGMAS approach must be adapted.

8. Attempt to combine data locality technique (such as LEEN approach [73]) and SGMAS approach to decrease the amount of data shuffled when *Group-By* operation executed.

Glossary

AM	Agent-Mapper.
AR	Agent-Reducer.
AS	Agent-Supervisor.
PLB	Partitioning and Load Balancing.
DW	Data Warehouse.
BDW	Big Data Warehouse.
MAS	Multi-Agent System.
HB	Hash-Broadcast join.
HDFS	Hadoop Data File System.
IRs	Intermediate Results are the outputs of the Map phase.
KB	The Knowledge Base is a distributed memory space over the cluster's nodes to maintain the fragments metadata.
MP	MapReduce paradigm.
OLAP	On-Line Analytical Processing.
PBS	Partitioning, Bucketing, and Sorting techniques.
RDD	Resilient Distributed Dataset.
RIs	Reducer Inputs are the partitions of the reduce phase, after performing the shuffle phase and before starting to calculate the reduce function.
RM	Resources Manager of the distributed system.
RT	References Table.
SGMAS	Smart Grouping of the fragments using Multi-Agent System.
SH	Hash-Shuffle join.
SkipSJoin	Skip to scan unnecessary data blocks and optimize Star Join operation.
W	Workload.
SMB	Sort-Merge-Bucket join.
YARN	Yet-Another Resource Negotiator.

Our contributions

[1] Ramdane, Y., Boussaid, O., Kabachi, N., and Bentayeb, F. Optimize Star Join Operation for OLAP Queries in Distributed Data Warehouses. In ASD, pp. 405-418, May 2018.

[2] Ramdane, Y., Boussaid, O., Kabachi, N., and Bentayeb, F. Partitioning and Bucketing Techniques to Speed up Query Processing in Spark-SQL. In 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), pp. 142-151, December 2018.

[3] Ramdane, Y., Boussaid, O., Kabachi, N. et Bentayeb, F. Conception physique d'un entrepôt de données distribuées basée sur K-means équilibré. In Extraction et Gestion des Connaissances, EGC'2019, Vol. 79, January 2019.

[4] Ramdane, Y., Kabachi, N., Boussaid, O., and Bentayeb, F. SDWP: A New Data Placement Strategy for Distributed Big Data Warehouses in Hadoop. In International Conference on Big Data Analytics and Knowledge Discovery. Springer, Cham., pp. 189-205, August 2019.

[5] Ramdane, Y., Kabachi, N., Boussaid, O. et Bentayeb, F. SGIA : Stratégie de Regroupement Intelligente pour Améliorer le Traitement des Requêtes OLAP en MapReduce. In Entrepôts de Données et Analyse en ligne, EDA, vol. RNTI-B-15, pp.93-108, Octobre 2019.

[6] Ramdane, Y., Kabachi, N., Boussaid, O., and Bentayeb, F. SkipSJoin: A New Physical Design for Distributed Big Data Warehouses in Hadoop." In 38th International Conference on Conceptual Modeling - ER-2019, Springer, November 2019. **(to appear)**

Bibliography

[7] Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Rasin, A. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proceedings of the VLDB Endowment, vol. 2, no. 1, pp. 922-933, 2009.

[8] Abuelyaman, E. S. An optimized scheme for vertical partitioning of a distributed database. IJCSNS International Journal of Computer Science and Network Security, vol. 8, no. 1, pp. 310-316, 2008.

[9] Afrati, F. N., and Ullman, J. D. Optimizing multiway joins in a map-reduce environment. IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 9, pp. 1282-1298, 2011.

[10] Agarwal, S., Kandula, S., Bruno, N., Wu, M. C., Stoica, I., and Zhou, J. Re-optimizing data parallel computing. In Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pp. 281-294, 2012.

[11] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., and Stoica, I. BlinkDB: queries with bounded errors and bounded response times on very large data. In Proceedings of the 8th ACM European Conference on Computer Systems. ACM, pp. 29-42, April 2013.

[12] Ahmed, S., Ali, M. U., Ferzund, J., Sarwar, M. A., Rehman, A., and Mehmood, A. Modern data formats for big bioinformatics data analytics. arXiv preprint arXiv:1707.05364, 2017.

[13] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K.,... and Zaharia, M. Spark sql: Relational data processing in spark. In Proceedings of the 2015 ACM SIGMOD international conference on management of data. ACM, pp. 1383-1394, May 2015.

[14] Arres, B., Kabachi, N., and Boussaid, O. . A data pre-partitioning and distribution optimization approach for distributed data warehouses. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), pp. 454, 2015.

[15] Azez, H. S. A., Khafagy, M. H., and Omara, F. A. JOUM: an indexing methodology for improving join in hive star schema. Int. J. Sci. Eng. Res, vol. 6, pp. 111-119, 2015.

[16] Baert, Q., Caron, A. C., Morge, M., and Routier, J. C. Fair multi-agent task allocation for large data sets analysis. In International Conference on Practical Applications of Agents and Multi-Agent Systems. Springer, pp. 24-35, June 2016.

[17] Baldacci, L., and Golfarelli, M. A Cost Model for SPARK SQL. IEEE Transactions on Knowledge and Data Engineering, vol. 31, no. 5, pp. 819-832, 2018.

[18] Benkrid, S., Bellatreche, L., and Cuzzocrea, A. Designing parallel relational data warehouses: A global, comprehensive approach. In New Trends in Databases and Information Systems. Springer, pp. 141-150, 2014.

[19] Bissiriou, C. A., and Chaoui, H. Big data analysis and query optimization improve HadoopDB performance. In Proceedings of the 10th International Conference on Semantic Systems. ACM, pp. 1-4, September 2014.

[20] Blanas, S., Patel, J. M., Ercegovac, V., Rao, J., Shekita, E. J., and Tian, Y. A comparison of join algorithms for log processing in mapreduce. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, pp. 975-986, June 2010.

[21] Borthakur, D. HDFS architecture guide. Hadoop Apache Project, vol. 53, no. 2, pp. 1-13, 2008.

[22] Boussahoua, M., Boussaid, O., and Bentayeb, F. Logical schema for data warehouse on column-oriented NoSQL databases. In International Conference on Database and Expert Systems Applications. Springer, pp. 247-256, August 2017.

[23] Bradley, P. S., Bennett, K. P., and Demiriz, A. Constrained k-means clustering. Microsoft Research, Redmond, vol. 20, no. 0, pp. 0, 2000.

[24] Brito, J. J., Mosqueiro, T., Ciferri, R. R., and de Aguiar Ciferri, C. D. Faster cloud star joins with reduced disk spill and network communication. Procedia Computer Science, vol. 80, pp. 74-85, 2016.

[25] Burkard, R., Dell'Amico, M., and Martello, S. Assignment Problems Revised Ed. 2012.

[26] Chalkiadakis, G., and Boutilier, C. Coordination in multiagent reinforcement learning: a Bayesian approach. In Proceedings of the second international joint conference on Autonomous agents and multiagent systems. ACM, pp. 709-716, July 2003.

[27] Chen, Y., Liu, Z., Wang, T., and Wang, L. Load balancing in MapReduce based on data locality. In International Conference on Algorithms and Architectures for Parallel Processing. Springer, pp. 229-241, August 2014.

[28] Cui, Y., Li, G., Cheng, H., and Wang, D. Indexing for large scale data querying based on Spark SQL. In 2017 IEEE 14th International Conference on e-Business Engineering (ICEBE), IEEE, pp. 103-108, November 2017.

[29] Curino, C., Jones, E., Zhang, Y., and Madden, S. Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment, vol. 3, no. 1-2, pp. 48-57, 2010.

[30] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Proceedings of the 6th symposium on Operating Systems Design and Implementation, OSDI 04, USENIX Association, pp. 137-150, 2004.

[31] Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. Communications of the ACM, vol. 51, no. 1, pp. 107-113, 2008.

[32] DeWitt, D. J., and Gray, J. Parallel database systems: The future of high performance database processing. University of Wisconsin-Madison Department of Computer Sciences. 1992

[33] Dittrich, J., Quiané-Ruiz, J. A., Jindal, A., Kargin, Y., Setty, V., and Schad, J. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). Proceedings of the VLDB Endowment, vol. 3, no. 1-2, pp 515-529, 2010.

[34] Dittrich, J., and Quiané-Ruiz, J. A. Efficient big data processing in Hadoop MapReduce. Proceedings of the VLDB Endowment, vol. 5, no. 12, pp. 2014-2015, 2012.

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

[36] Field, A. Discovering statistics using IBM SPSS statistics. sage, 2013.

[37] Gao, Y., Zhou, Y., Zhou, B., Shi, L., and Zhang, J. Handling data skew in MapReduce cluster by using partition tuning. Journal of healthcare engineering, 2017.

[38] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system, SOSP'03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. 2003.

[39] Gravetter, F. J., and Wallnau, L. B. Statistics for the behavioral sciences. Cengage Learning. 2016.

[40] Gufler, B., Augsten, N., Reiser, A., and Kemper, A. The partition cost model for load balancing in MapReduce. In International Conference on Cloud Computing and Services Science. Springer, New York, NY, pp. 371-387, May 2011.

[41] Han, H., Jung, H., Eom, H., and Yeom, H. Y. Scatter-Gather-Merge: An efficient star-join query processing algorithm for data-parallel frameworks. Cluster Computing, vol. 14, no. 2, pp. 183-197, 2011.

[42] Hefny, H. A., Khafagy, M. H., and Ahmed, M. W. Comparative study load balance algorithms for map reduce environment. International Journal of Computer Applications, vol. 106, no. 18, pp. 41-50, 2014.

[43] Heintz, B., Chandra, A., Sitaraman, R. K., and Weissman, J. End-to-end optimization for geo-distributed mapreduce. IEEE Transactions on Cloud Computing, vol.4, no. 3, pp. 293-306, 2014.

[44] Hsiao, H. I., and DeWitt, D. J. Chained declustering: A new availability strategy for multiprocessor database machines. University of Wisconsin-Madison Department of Computer Sciences. 1989.

[45] http://download.redis.io/redis-stable.tar.gz

[46] http://cassandra.apache.org/

[47] http://www.dbaref.com/home/dba-routine-tasks/whatisfastfullindexscanvsindexrangescan

[48] http://www.tpc.org/

[49] http://www.vertica.com/

[50] https://akka.io/

[51] https://avro.apache.org/

[52] https://data-flair.training/blogs/bucket-map-join/

[53] https://data-flair.training/blogs/hive-sort-merge-bucket-join/

[54] https://en.wikipedia.org/wiki/Skewness

[55] https://flink.apache.org/

[56] https://github.com/databricks/spark-sql-perf

[57] https://github.com/lightcopy/parquet-index

[58] https://github.com/LucaCanali/sparkMeasure

[59] https://github.com/ray-project/ray

[60] http://hadoop.apache.org/

[61] https://hbase.apache.org/

[62] https://hive.apache.org/

[63] https://orc.apache.org/

[64] https://parquet.apache.org/

[65] https://repo.continuum.io/archive/Anaconda3-5.0.1-Linux-x86_64

[66] https://spark.apache.org/

[67] http://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/RangePartitioner.html.

[68] https://techvidvan.com/tutorials/spark-cluster-manager-yarn-mesos-and-standalone/

[69] https://tez.apache.org/

[70] https://wikibon.com/forecasting-sparks-adoption-in-the-context-of-systems-of-intelligence/

[71] https://www.bbva.com/en/five-vs-big-data/

[72] https://www.slideshare.net/AmazonWebServices/apache-spark-and-the-hadoop-ecosystem-on-aws

[73] Ibrahim, S., Jin, H., Lu, L., He, B., Antoniu, G., and Wu, S. Handling partitioning skew in mapreduce using leen. Peer-to-Peer Networking and Applications, vol. 6, no 4, pp. 409-424, 2013.

[74] Kalai, E., and Lehrer, E. Rational learning leads to Nash equilibrium. Econometrica: Journal of the Econometric Society, pp. 1019-1045. 1993.

[75] Kalinsky, O., Etsion, Y., and Kimelfeld, B. Flexible caching in trie joins. arXiv preprint arXiv:1602.08721. 2016.

[76] Karloff, H., Suri, S., and Vassilvitskii, S. A model of computation for MapReduce. In Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms . Society for Industrial and Applied Mathematics, pp. 938-948, January 2010.

[77] Kreps, J., Narkhede, N., and Rao, J. Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB, pp. 1-7, June 2011.

[78] Kimball, R. and Ross, M. The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. Wiley Computer Publishing, 2 edition, 2002.

[79] Kumar, K. A., Quamar, A., Deshpande, A., and Khuller, S. SWORD: workload-aware data placement and replica selection for cloud data management systems. The VLDB Journal?The International Journal on Very Large Data Bases, vol. 23, no. 6, pp. 845-870, 2014.

[80] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. Skewtune: mitigating skew in mapreduce applications. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, pp. 25-36, May 2012.

[81] Lee, D., Kim, J. S., and Maeng, S. Large-scale incremental processing with MapReduce. Future Generation Computer Systems, vol. 36, pp. 66-79, 2014.

[82] Lu, Y., Shanbhag, A., Jindal, A., and Madden, S. AdaptDB: adaptive partitioning for distributed joins. Proceedings of the VLDB Endowment, vol. 10, no. 5, pp. 589-600, 2017.

[83] Malinen, M. I., and Fränti, P. Balanced k-means for clustering. In Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR). Springer, Berlin, Heidelberg, pp. 32-41, August 2014.

[84] Miettinen, P., Mielikäinen, T., Gionis, A., Das, G., and Mannila, H. The discrete basis problem. IEEE transactions on knowledge and data engineering, vol. 20, no. 10, pp. 1348-1362, 2008.

[85] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., ... and Stoica, I. Ray: A distributed framework for emerging AI applications. In

13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 561-577, 2018.

[86] Myerson, R. Game Theory: Analysis of Conflict Harvard Univ. Press, Cambridge. 1991.

[87] Myung, J., Shim, J., Yeon, J., and Lee, S. G. Handling data skew in join algorithms using MapReduce. Expert Systems with Applications, vol. 51, pp. 286-299, 2016.

[88] Nishanth, S., Radhikaa, B., Ragavendar, T. J., Babu, C., and Prabavathy, B. CoHadoop++: A load balanced data co-location in Hadoop Distributed File System. In 2013 Fifth International Conference on Advanced Computing (ICoAC). IEEE, pp. 100-105, December 2013.

[89] Nonava, P. Hdfs blocks placement strategy (Doctoral dissertation, Master?s thesis, University of Fribourg), 2014.

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

[91] O'Neil P, O'Neil E, Chen X. The star schema benchmark 2009 Available at: http://www.cs.umb.edu/ poneil/StarSchemaB.PDF [3 January 2014]

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

[93] Petridis, P., Gounaris, A., and Torres, J. Spark parameter tuning via trial-and-error. In INNS Conference on Big Data. Springer, Cham, pp. 226-237, October 2016.

[94] Plase, D., Niedrite, L., and Taranovs, R. A comparison of HDFS compact data formats: Avro versus Parquet/HDFS glaustųjų duomenų formatų palyginimas: Avro prieš Parquet. Mokslas?Lietuvos ateitis/Science?Future of Lithuania, vol. 9, no. 3, pp. 267-276, 2017.

[95] Bradley, P. S., Bennett, K. P., and Demiriz, A. Constrained k-means clustering. Microsoft Research, Redmond, vol 20, no 0, pp. 0, 2000.

[96] Purdilă, V., and Pentiuc, Ş. G. Single-scan: a fast star-join query processing algorithm. Software: Practice and Experience, vol. 46, no. 3, pp. 319-339, 2016.

[97] Rabl, T., Poess, M., Jacobsen, H. A., O'Neil, P., and O'Neil, E. Variations of the star schema benchmark to test the effects of data skew on query performance. In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ACM, pp. 361-372, April 2013.

[98] Ramakrishnan, S. R., Swart, G., and Urmanov, A. Balancing reducer skew in MapReduce workloads using progressive sampling. In Proceedings of the Third ACM Symposium on Cloud Computing, ACM, pp. 16, October 2012.

[99] Ren, Z., Xu, X., Wan, J., Shi, W., and Zhou, M. Workload characterization on a production Hadoop cluster: A case study on Taobao. In 2012 IEEE International Symposium on Workload Characterization (IISWC). IEEE, pp. 3-13, November 2012.

[100] Rodríguez, L., and Li, X. A dynamic vertical partitioning approach for distributed database system. In 2011 IEEE International Conference on Systems, Man, and Cybernetics. IEEE, pp. 1853-1858, October 2011.

[101] Rosà, A., Chen, L. Y., and Binder, W. Profiling actor utilization and communication in Akka. In Proceedings of the 15th International Workshop on Erlang. ACM, pp. 24-32, September 2016.

[102] Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A., and Curino, C. Apache tez: A unifying framework for modeling and building data processing applications. In Proceedings of the 2015 ACM SIGMOD international conference on Management of Data. ACM, pp. 1357-1369, May 2015.

[103] Shanbhag, A., Jindal, A., Lu, Y., and Madden, S. A moeba: a shape changing storage system for big data. Proceedings of the VLDB Endowment, vol. 9, no. 13, pp. 1569-1572, 2016.

[104] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In MSST, Vol. 10, pp. 1-10, May 2010.

[105] Singhal, R., and Singh, P. Performance Assurance Model for Applications on SPARK Platform. In Technology Conference on Performance Evaluation and Benchmarking. Springer, Cham, pp. 131-146, August 2017.

[106] Song, J., Guo, C., Wang, Z., Zhang, Y., Yu, G., and Pierson, J. M. HaoLap: a Hadoop based OLAP system for big data. Journal of Systems and Software, vol. 102, pp. 167-181, 2015.

[107] Song, J., Li, T., Liu, X., and Zhu, Z. Comparing and analyzing the energy efficiency of cloud database and parallel database. In Advances in Computer Science, Engineering & Applications . Springer, Berlin, Heidelberg, pp. 989-997, 2012.

[108] Sun, L., Franklin, M. J., Krishnan, S., and Xin, R. S. Fine-grained partitioning for aggressive data skipping. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, pp. 1115-1126, June 2014.

[109] Tao, Y., Zhou, M., Shi, L., Wei, L., and Cao, Y. Optimizing multi-join in cloud environment. In 2013 IEEE 10th International Conference on High Performance Computing and Communications and 2013 IEEE International Conference on Embedded and Ubiquitous Computing, IEEE, pp. 956-963, November 2013.

[110] Tang, Z., Zhang, X., Li, K., and Li, K. An intermediate data placement algorithm for load balancing in Spark computing environment. Future Generation Computer Systems, vol. 78, pp. 287-301, 2018.

[111] Tarkoma, S., Rothenberg, C. E., and Lagerspetz, E. Theory and practice of bloom filters for distributed systems. IEEE Communications Surveys and Tutorials, vol. 14, no. 1, pp. 131-155, 2011.

[112] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S.,... and Murthy, R. Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, vol. 2, no. 2, pp. 1626-1629, 2009.

[113] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., ... and Murthy, R. Hive-a petabyte scale data warehouse using hadoop. In 2010 IEEE 26th international conference on data engineering (ICDE 2010). IEEE, pp. 996-1005, March 2010.

[114] Trivedi, A., Stuedi, P., Pfefferle, J., Schuepbach, A., and Metzler, B. Albis: High-performance file format for big data systems. In 2018 USENIX Annual Technical Conference (USENIXATC 18), pp. 615-630, 2018.

[115] Valvåg, S. V., Johansen, D., and Kvalnes, Å. Cogset: a high performance MapReduce engine. Concurrency and Computation: Practice and Experience, vol. 25, no. 1, pp. 2-23, 2013.

[116] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., ... and Saha, B. (,). Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing. ACM, pp. 5, October 2013.

[117] Vernica, R., Balmin, A., Beyer, K. S., and Ercegovac, V. Adaptive MapReduce using situation-aware mappers. In Proceedings of the 15th International Conference on Extending Database Technology. ACM, pp. 420-431, March 2012.

[118] Vohra, D. Apache parquet. In Practical Hadoop Ecosystem. Apress, Berkeley, CA, pp. 325-335, 2016.

[119] Wang, K., and Khan, M. M. H. Performance prediction for apache spark platform. In 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems. IEEE, pp. 166-173, August 2015.

[120] Xu, Y., Zou, P., Qu, W., Li, Z., Li, K., and Cui, X. Sampling-based partitioning in MapReduce for skewed data. In 2012 Seventh ChinaGrid Annual Conference. IEEE, pp. 1-8, September 2012.

[121] Yang, H. C., Dasdan, A., Hsiao, R. L., and Parker, D. S. Map-reduce-merge: simplified relational data processing on large clusters. In Proceedings of the

2007 ACM SIGMOD international conference on Management of data. ACM, pp. 1029-1040, June 2007.

[122] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. Improving MapReduce performance in heterogeneous environments. In Osdi, vol. 8, no. 4, pp. 7, December 2008.

[123] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. HotCloud, vol. 10, no. 10-10, pp. 95, 2010.

[124] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., ... and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, pp. 2-2, April 2012.

[125] Zamanian, E., Binnig, C., and Salama, A. Locality-aware partitioning in parallel database systems. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 17-30, May 2015.

[126] Zhang, C., Wu, L., and Li, J. Efficient processing distributed joins with bloomfilter using mapreduce. International Journal of Grid and Distributed Computing, vol. 6, no. 3, pp. 43-58, 2013.

[127] Zhang, X., Chen, L., and Wang, M. Efficient multi-way theta-join processing using mapreduce. Proceedings of the VLDB Endowment, vol. 5, no. 11, pp. 1184-1195, 2012.

[128] Zhang, Y., Cao, T., Li, S., Tian, X., Yuan, L., Jia, H., and Vasilakos, A. V. Parallel processing systems for big data: a survey. Proceedings of the IEEE, vol. 104, no. 11, pp. 2114-2136, 2016.

[129] Zhou, G., Zhu, Y., and Wang, G. Cache conscious star-join in MapReduce environments. In Proceedings of the 2nd International Workshop on Cloud Intelligence. ACM, pp. 1, August 2013.

[130] Zhu, H., Zhou, M., Xia, F., and Zhou, A. Efficient star join for column-oriented data store in the mapreduce environment. In 2011 Eighth Web Information Systems and Applications Conference, IEEE. pp. 13-18. October 2011.

[131] Zhu, S., Wang, D., and Li, T. Data clustering with size constraints. Knowledge-Based Systems, Vol. 23, no. 8, pp. 883-889, 2010.