



HAL
open science

Workload- and Data-based Automated Design for a Hybrid Row-Column Storage Model and Bloom Filter-Based Query Processing for Large-Scale DICOM Data Management

Cong-Danh Nguyen

► **To cite this version:**

Cong-Danh Nguyen. Workload- and Data-based Automated Design for a Hybrid Row-Column Storage Model and Bloom Filter-Based Query Processing for Large-Scale DICOM Data Management. Databases [cs.DB]. Université Clermont Auvergne [2017-2020], 2018. English. NNT : 2018CLFAC019 . tel-01961254

HAL Id: tel-01961254

<https://theses.hal.science/tel-01961254v1>

Submitted on 19 Dec 2018

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.

Order N°: XXX

EDSPIC : XXX



CLERMONT AUVERGNE UNIVERSITY

Doctoral School of Engineering Sciences

PHD THESIS

To obtain the degree of

DOCTOR OF PHILOSOPHY

Discipline: Computer Science

Defended by

Cong-Danh NGUYEN

**Workload- and Data-Based Automated Design for
a Hybrid Row-Column Storage Model and Bloom Filter-Based
Query Processing for Large-Scale DICOM Data Management**

Publicly defended on May 4th, 2018

Committee:

Reviewers:

Prof. Christine COLLET Institut polytechnique de Grenoble, France

Prof. Abdelkader HAMEURLAIN University of Paul Sabatier, Toulouse, France

Examiners:

Prof. Farouk TOUMANI University of Clermont Auvergne, France

Supervisors:

Prof. Laurent D'ORAZIO University of Rennes 1, France

Prof. Mohand-Said HACID University of Lyon, France

Msc. Nga TRAN Micro Focus - Vertica, Cambridge, Massachusetts, USA

Author



Cong-Danh NGUYEN - Ph.D student.

Email: nguyenda@isima.frr

ncdanh@cit.ctu.edu.vn

Cong-Danh NGUYEN was born in Vinh Long, Vietnam in 1977. He received his B.S degree in Computer Science from Can Tho University, Vietnam in 2000. In 2006, he received his M.S. degree in Information Technology from King Mongkut's University of Technology North Bangkok, Thailand. In 2009, he worked as a Researcher at the Software Engineering Research Group (AGSE), the Technical University of Kaiserslautern, Germany. In September of 2014, he started working towards his PhD degree in Computer Science at CNRS, LIMOS UMR 6158, Clermont Auvergne University, France. His research interests include Database System, Big Data, Software Process, Software Cost Estimation and Software Quality Assurance.

Declaration

This thesis has been completed by Cong-Danh NGUYEN under the supervision of Professor Laurent D’ORAZIO, Professor Mohand-Said HACID and Nga TRAN. It has not been submitted for any other degree or professional qualification. I declare that the work presented in this thesis is entirely my own except where indicated by full references.

SIGNATURE

Acknowledgements

First and foremost I would like to thank the Ministry of Education and Training of Vietnam for offering me an International Postgraduate Research Scholarship (Project 911) to make my dream of pursuing a doctoral program come true. I would also like to express my deep gratitude to the Can Tho University for allowing me to receive and retain this scholarship.

I am extremely grateful to three PhD supervisors, Prof. Laurent d’Orazio, Prof. Mohand-Said Hacid and Nga Tran for their guidance and help. Prof. Dr. Laurent d’Orazio has been a wonderful professor who has brought me to the amazing world of database research. He showed me the way to deal with a specific research problem effectively and how to bring it up to a new level. He also provided me necessary resources needed for performing my research. Many thanks to Prof. Mohand-Said Hacid for his supervising the thesis and discussions. He supported the research and provided many valuable advices on the work during its phases. Nga Tran has been a very dedicated supervisor who has had significant experience in the development and the application of modern database systems in industrial contexts. I am also thankful for her support of my study and sharing her valuable experience and knowledge. Once again, I would like to thank all of my supervisors who supported, guided and reviewed the work through all of its phases.

Thanks especially to Prof. Christine Collet and Prof. Abdelkader Hameurlain for your very careful review, and for the comments, corrections and suggestions made on the thesis. Many thanks to Prof. Farouk Toumani who worked as an examiner and gave the suggestions to the thesis. Thanks to all of you who also served as the members of my defense committee.

Also, I would like to thank Frédéric Gaudet for technical support and help in providing necessary machines and building a distributed query processing environment.

It has never been easy for me to live and study in a foreign country. The various challenges I faced throughout the duration of this study have been passed. Many thanks to the good friends from Blaise Pascal University - Clermont II: To Baraa Mohamad for providing valuable information related to the DICOM standard and for giving me the initial idea of depending on expert opinion to organize DICOM data; To Thuong-Cang Phan and Jean Connier for guiding many administrative procedures during the study period.

Finally, I would like to thank my family for always being by my side in difficult moments, motivating me.

Clermont-Ferrand, Mar 22, 2018



Cong-Danh NGUYEN

Abstract

In the health care industry, the ever-increasing medical image data, the development of imaging technologies, the long-term retention of medical data and the increase of image resolution are causing a tremendous growth in data volume. In addition, the variety of acquisition devices and the difference in preferences of physicians or other health-care professionals have led to a high variety in data. Although today DICOM (Digital Imaging and Communication in Medicine) standard has been widely adopted to store and transfer the medical data, DICOM data still has the 3Vs characteristics of Big Data: high volume, high variety and high velocity. Besides, there is a variety of workloads including Online Transaction Processing (OLTP), Online Analytical Processing (OLAP) and mixed workloads. Existing systems have limitations dealing with these characteristics of data and workloads. In this thesis, we propose new efficient methods for storing and querying DICOM data.

We propose a hybrid storage model of row and column stores, called HYTORMO, together with data storage and query processing strategies. First, HYTORMO is designed and implemented to be deployed on large-scale environment to make it possible to manage big medical data. Second, the data storage strategy combines the use of vertical partitioning and a hybrid store to create data storage configurations that can reduce storage space demand and increase workload performance. To achieve such a data storage configuration, one of two data storage design approaches can be applied: (1) expert-based design and (2) automated design. In the former approach, experts manually create data storage configurations by grouping attributes and selecting a suitable data layout for each column group. In the latter approach, we propose a hybrid automated design framework, called HADF. HADF depends on similarity measures (between attributes) that can take into consideration the combined impact of both workload- and data-specific information to generate data storage configurations: Hybrid Similarity (a weighted combination of Attribute Access and Density Similarity measures) is used to group the attributes into column groups; Inter-Cluster Access Similarity is used to determine whether two column groups will be merged together or not (to reduce the number of joins); and Intra-Cluster Access Similarity is applied to decide whether a column group will be stored in a row or a column store. Finally, we propose a suitable and efficient query processing strategy built on top of HYTORMO. It considers the use of both inner joins and left-outer joins. Furthermore, an Intersection Bloom filter (IBF) is applied to reduce network I/O cost.

We provide experimental evaluations to validate the benefits of the proposed methods over real DICOM datasets. Experimental results show that the mixed use of both row and column stores outperforms a pure row store and a pure column store. The combined impact of both workload-and data-specific information is helpful for HADF to be able to produce good data storage configurations. Moreover, the query processing strategy with the use of the IBF can improve the execution time of an experimental query up to 50% when compared to the case where no IBF is applied.

Key words: DICOM, big data, sparse datasets, HYTORMO, hybrid storage model, row store, column store, hybrid similarity, Bloom filter, Intersection Bloom filter, join.

Résumé

Dans le secteur des soins de santé, les données d'images médicales toujours croissantes, le développement de technologies d'imagerie, la conservation à long terme des données médicales et l'augmentation de la résolution des images entraînent une croissance considérable du volume de données. En outre, la variété des dispositifs d'acquisition et la différence de préférences des médecins ou d'autres professionnels de la santé ont conduit à une grande variété de données. Bien que la norme DICOM (Digital Imaging et Communication in Medicine) soit aujourd'hui largement adoptée pour stocker et transférer les données médicales, les données DICOM ont toujours les caractéristiques 3V du Big Data: volume élevé, grande variété et grande vélocité. En outre, il existe une variété de charges de travail, notamment le traitement transactionnel en ligne (en anglais Online Transaction Processing, abrégé en OLTP), le traitement analytique en ligne (anglais Online Analytical Processing, abrégé en OLAP) et les charges de travail mixtes. Les systèmes existants ont des limites concernant ces caractéristiques des données et des charges de travail. Dans cette thèse, nous proposons de nouvelles méthodes efficaces pour stocker et interroger des données DICOM.

Nous proposons un modèle de stockage hybride des magasins de lignes et de colonnes, appelé HYTORMO, ainsi que des stratégies de stockage de données et de traitement des requêtes. Tout d'abord, HYTORMO est conçu et mis en œuvre pour être déployé sur un environnement à grande échelle afin de permettre la gestion de grandes données médicales. Deuxièmement, la stratégie de stockage de données combine l'utilisation du partitionnement vertical et un stockage hybride pour créer des configurations de stockage de données qui peuvent réduire la demande d'espace de stockage et augmenter les performances de la charge de travail. Pour réaliser une telle configuration de stockage de données, l'une des deux approches de conception de stockage de données peut être appliquée: (1) conception basée sur des experts et (2) conception automatisée. Dans la première approche, les experts créent manuellement des configurations de stockage de données en regroupant les attributs des données DICOM et en sélectionnant une disposition de stockage de données appropriée pour chaque groupe de colonnes. Dans la dernière approche, nous proposons un cadre de conception automatisé hybride, appelé HADF. HADF dépend des mesures de similarité (entre attributs) qui prennent en compte les impacts des informations spécifiques à la charge de travail et aux données pour générer automatiquement les configurations de stockage de données: Hybrid Similarity (combinaison pondérée de similarité d'accès d'attribut et de similarité de densité d'attribut) les attributs dans les groupes de colonnes; Inter-Cluster Access Similarity est utilisé pour déterminer si deux groupes de colonnes seront fusionnés ou non (pour réduire le nombre de jointures supplémentaires); et Intra-Cluster Access La similarité est appliquée pour décider si un groupe de colonnes sera stocké dans une ligne ou un magasin de colonnes. Enfin, nous proposons une stratégie de traitement des requêtes adaptée et efficace construite sur HYTORMO. Il considère l'utilisation des jointures internes et des jointures externes gauche pour empêcher la perte de données si vous utilisez uniquement des jointures internes entre des tables partitionnées verticalement. De plus, une intersection de filtres Bloom (intersection of Bloom filters, abrégé en IBF) est appliqué pour

supprimer les données non pertinentes des tables d'entrée des opérations de jointure; cela permet de réduire les coûts d'E / S réseau.

Nous fournissons des évaluations expérimentales pour valider les avantages des méthodes proposées par rapport aux jeux de données DICOM réels. Les résultats expérimentaux montrent que l'utilisation mixte des magasins de lignes et de colonnes surpasse le magasin de lignes pur et le magasin de colonnes pur. L'impact combiné des informations spécifiques à la charge de travail et aux données permet à HADF de produire de bonnes configurations de stockage de données. En utilisant l'IBF, la stratégie de traitement des requêtes peut améliorer le temps d'exécution d'une requête expérimentale jusqu'à 50% par rapport au cas où aucun IBF n'est appliqué.

Mots clés : DICOM, données volumineuses, données clairsemées, HYTORMO, modèle de stockage hybride, stockage en lignes, stockage en colonnes, similarité hybride, filtre Bloom, intersection de filtres Bloom, joindre.

Contents

Author	i
Declaration	ii
Acknowledgements	iii
Abstract	iv
Résumé	v
Contents	vii
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Overview.....	1
1.2 Research Context	1
1.3 Motivation.....	2
1.4 Research Scope and Approach.....	5
1.5 Problem Statement.....	7
1.6 Dissertation Goals.....	8
1.7 Research Hypotheses	8
1.8 Research Contributions.....	10
1.9 Thesis Structure	10
2 DICOM Data Management Systems and Requirements	13
2.1 Overview.....	13
2.2 DICOM Standard and Data	13
2.2.1 DICOM Standard	13
2.2.2 Characteristics of DICOM Data and Workloads.....	18
2.3 DICOM Data Management Systems	20
2.3.1 Expected Requirements.....	20
2.3.2 Existing Systems	22
2.3.3 Conclusion.....	30
2.4 Summary and Conclusion.....	31
3 Databases and Related Techniques	33
3.1 Overview.....	33
3.2 Classifications.....	34
3.2.1 OLTP and OLAP Workloads	34
3.2.2 Relational Databases	34
3.2.3 NoSQL Databases	35
3.2.4 NewSQL Databases.....	36
3.3 Cluster Computing Frameworks.....	37

3.3.1	MapReduce	37
3.3.2	Spark	39
3.4	Data Layouts	40
3.4.1	Row-oriented Storage Model	40
3.4.2	Column-oriented Storage Model	41
3.4.3	Hybrid Storage Models	42
3.5	Vertical Partitioning and Bloom Filter Techniques	47
3.5.1	Vertical Partitioning	47
3.5.2	Bloom Filter and Intersection Bloom Filter	49
3.6	Key Components of the New System	51
3.6.1	Data Model	52
3.6.2	Data Storage Model	52
3.6.3	Data Schema	52
3.6.4	Query Processing	52
3.7	Summary and Conclusion	53
4	HYTORMO and HADF	57
4.1	Overview	57
4.2	HYTORMO and Strategies	58
4.2.1	HYTORMO Architecture	58
4.2.2	Data Storage Strategy	58
4.2.3	Query Processing Strategy	62
4.3	Automated Design Approach for DICOM Data	64
4.3.1	Observations	64
4.3.2	Formal Representation	66
4.3.3	Configuration Cost Estimation	68
4.4	Hybrid Automated Design Framework	75
4.4.1	Overview of the Framework	75
4.4.2	Similarity Measures	78
4.4.3	Implementation of the Framework	81
4.4.4	Examples	89
4.5	Summary and Conclusion	93
5	Query Processing for HYTORMO	95
5.1	Overview	95
5.2	Query Rewriting	96
5.2.1	Examples	96
5.2.2	Query Execution Plan	100
5.2.3	Determining Left-Outer Joins	101
5.2.4	Reducing the Number of Left-Outer Joins	102
5.3	Intersection Bloom Filter	104

5.3.1 Query Execution Plan with the IBF.....	104
5.3.2 Cost-effectiveness Analysis.....	106
5.3.3 Incremental Intersection Bloom Filter	116
5.4 Summary and Conclusion.....	118
6 Performance Evaluation	119
6.1 Overview.....	119
6.2 Experimental Environment.....	120
6.2.1 Spark Cluster.....	120
6.2.2 Datasets	120
6.2.3 Workloads.....	123
6.3 Experiment Execution	126
6.3.1 Experiment 1: Evaluating the Effectiveness of HYTORMO and the Usefulness of HADF	126
6.3.2 Experiment 2: Evaluating HYTORMO and HADF using More Data and Multiple-table Joins	135
6.3.3 Experiment 3: Comparison between HADF and HoVer	137
6.3.4 Experiment 4: Evaluating the Effectiveness of the IBF	140
6.4 Analysis and Interpretation.....	146
6.4.1 H1 - Effectiveness of HYTORMO.....	146
6.4.2 H2 - Usefulness of HADF.....	147
6.4.3 H3 - Effectiveness of the Query Processing Strategy	148
6.5 Summary and Conclusion.....	148
7 Conclusion and Future Works.....	149
7.1 Overview.....	149
7.2 Summary and Conclusion.....	150
7.2.1 Existing DICOM Data Management Systems	150
7.2.2 Current Databases and Related Techniques	151
7.2.3 HYTORMO and DICOM Data Storage Strategy	151
7.2.4 HADF.....	152
7.2.5 Query Processing Strategy with the Use of an IBF.....	152
7.2.6 Validations of Proposed Methods.....	152
7.3 Future Works.....	153
7.3.1 Hybrid Storage Model.....	153
7.3.2 HADF.....	153
7.3.3 Query Processing Strategy	154
7.3.4 Non-precomputed and Precomputed BFs.....	154
Bibliography	155

List of Figures

Figure 1.1: Example of metadata and image data in a DICOM file	2
Figure 1.2: Research focus.....	5
Figure 1.3: Research approach.....	6
Figure 1.4: Causal relationship between problems, goals and research hypotheses....	9
Figure 2.1: Mapping real-world examinations to the information model [28]	14
Figure 2.2: Transforming an object in real world into an IOD object	15
Figure 2.3: Detailed DICOM information model [28]	15
Figure 2.4: Structure of a DICOM attribute (data element)	16
Figure 2.5: Some attributes used in a DICOM file	17
Figure 2.6: Different attributes used for Patient IE of CT and CR images.....	18
Figure 2.7: Typical PACS-based workflow	22
Figure 2.8: eDiaMoND architecture [42]	23
Figure 2.9: Architecture of Grid Data Service [42]	24
Figure 2.10: Sample DICOM Image Database using Oracle.....	25
Figure 2.11: Database tables in the DCMDSM model [54].....	26
Figure 2.12: Example of DICOM data stored in CouchDB [40].....	28
Figure 2.13: DICOM attributes stored over row- and column-oriented layers [57]..	29
Figure 2.14: Distributed Mediator [57].....	30
Figure 3.1: Relation instance of the relation Patient.....	34
Figure 3.2: Examples of NoSQL databases	35
Figure 3.3: A job that counts the number of patients by sex using MapReduce.....	38
Figure 3.4: Comparison between Hadoop MapReduce and Spark.....	40
Figure 3.5: NSM layout of the relation Patient.....	40
Figure 3.6: DSM layout of the relation Patient.....	41
Figure 3.7: Physical representation of the DSM layout of the relation Patient	41
Figure 3.8: A disk page of PAX layout of the relation Patient.....	43
Figure 3.9: A disk page of Data Morphing layout of the relation Patient.....	44
Figure 3.10: Mirrors and fractured mirrors [86]	45
Figure 3.11: Copy-on-update mechanism [93]	46
Figure 3.12: Example of the application of a Bloom filter	50
Figure 4.1: Architecture of HYTORMO.....	58
Figure 4.2: Process of extracting, organizing and storing DICOM data	59
Figure 4.3: Row and column tables of the entity Patient.....	61
Figure 4.4: General form of a user query.....	63
Figure 4.5: Combined use of vertical partitioning and a hybrid store	64
Figure 4.6: Example of Attribute Usage Matrix and query frequencies.....	66

Figure 4.7: Example of the horizontal table T	67
Figure 4.8: Four difference configurations of the horizontal table T	69
Figure 4.9: Reading effectiveness in (a) a column store and (b) a row store	72
Figure 4.10: Overview of HADF	76
Figure 4.11: Venn diagram.....	79
Figure 4.12: Attribute Access Correlation Matrix	83
Figure 4.13: Attribute Density Correlation Matrix	85
Figure 4.14: Example of cluster usage of a workload	87
Figure 4.15: Workload- and data-specific information of the horizontal table T	90
Figure 4.16: Table created for Configuration 1	91
Figure 4.17: Two tables created for Configuration 2	91
Figure 4.18: Two tables created for Configuration 3	92
Figure 5.1: Representation of (a) the query Q1 and (b) its execution plan tree.....	97
Figure 5.2: Transformation of the query Q2a using a left-outer join.....	99
Figure 5.3: Transformation of the query Q2a using an inner join	99
Figure 5.4: Execution plan transformation for the query Q.....	100
Figure 5.5: Transformation of the query Q2b to two equivalent execution plans ...	103
Figure 5.6: Transformation of the execution plan after applying Rule 3.....	104
Figure 5.7: Query execution plan with the IBF.....	105
Figure 5.8: Left-deep sequential execution plan with the application of the IBF....	107
Figure 5.9: Phases of the IBF with component BFi's and hash functions.....	107
Figure 5.10: Left-deep processing tree of the query Q with the use of the IBF.....	114
Figure 5.11: Query execution plan with the incremental IBF.....	117
Figure 6.1: AUM of the entity table GeneralInfoTable in Workload W1	127
Figure 6.2: AUM of the entity table SequenceAttributes in Workload W2.....	129
Figure 6.3: AUM of the entity table Patient in Workload W3	130
Figure 6.4: AUM of the entity table Study in Workload sW4	133
Figure 6.5: Execution plan for the query Q4,3	141
Figure 6.6: Execution plan for the query Q4,3 with the IBF	142
Figure 6.7: Execution plan for the query Q4,3 with the incremental IBF	145

List of Tables

Table 1.1: Overview over Chapter 1	1
Table 1.2: Problem statements	8
Table 1.3: Thesis goals	8
Table 2.1: Overview over Chapter 2	13
Table 2.2: A subset of the attributes in the Study IOD	17
Table 2.3: Example of DICOM file sizes	19
Table 2.4: Example of statements to manipulate DICOM data in Oracle	26
Table 2.5: Comparison of the existing systems	31
Table 3.1: Overview over Chapter 3	33
Table 3.2: Input and output formats of the phases in MapReduce	37
Table 4.1: Overview over Chapter 4	57
Table 4.2: Examples of user queries	63
Table 5.1: Overview over Chapter 5	95
Table 5.2: Row and column tables used to store the entity tables	96
Table 5.3: Sample data of the table RowPatient	98
Table 5.4: Sample data of the table RowPregnancy	98
Table 5.5: Result of the query Q2a when using a left-outer join	99
Table 5.6: The wrong result of the query Q2a when using an inner join	100
Table 5.7: Correct result of the query Q2b	103
Table 5.8: Notations	108
Table 5.9: Example of table sizes and selectivity factors of join operations	113
Table 6.1: Overview over Chapter 6	119
Table 6.2: Mixed DICOM datasets used in the experiments	120
Table 6.3: Sizes of the entity tables of the dataset MDB1	121
Table 6.4: Sizes of the entity tables of the dataset MDB2	122
Table 6.5: Queries and their occurrence frequency in Workload W1	123
Table 6.6: Queries and their occurrence frequency in Workload W2	124
Table 6.7: Queries and their occurrence frequency in Workload W3	124
Table 6.8: Queries and their occurrence frequency in Workload W4	125
Table 6.9: Major steps of Experiment 1	126
Table 6.10: Typical candidate configurations for GeneralInfoTable	127
Table 6.11: Typical candidate configurations for SequenceAttributes	129
Table 6.12: Typical candidate configurations for Patient	131
Table 6.13: Workload sW4 for the entity table Study	133
Table 6.14: Typical candidate configurations for Study	134
Table 6.15: Major steps of Experiment 2	135

Table 6.16: Configuration G^* of Experiment 2	135
Table 6.17: Configuration G_1 of Experiment 2.....	136
Table 6.18: Configuration G_2 of Experiment 2.....	136
Table 6.19: Execution time of Workload W4 over 3 configurations using MDB1 .	137
Table 6.20: Execution time of Workload W4 over 3 configurations using MDB2 .	137
Table 6.21: Major steps of Experiment 3	137
Table 6.22: Good HADF-generated configuration for GeneralInfoTable	138
Table 6.23: HoVer-generated configurations for GeneralInfoTable	138
Table 6.24: Good HADF-generated configurations for Sequenceattributes.....	139
Table 6.25: HoVer-generated configurations for Sequenceattributes	140
Table 6.26: Major steps of Experiment 4	140
Table 6.27: Sets of predicates on the attributes in the input tables.....	143
Table 6.28: Comparison of the execution time of using and not using the IBF	143
Table 6.29: Comparison of the sizes of the input tables before and after using IBF	144
Table 6.30: Comparison between the IBF and incremental IBF	146
Table 7.1: Overview over Chapter 7.....	149

Introduction

1.1 Overview

This chapter describes how thesis goals are connected to challenges in DICOM (Digital Imaging and Communication in Medicine) data management. The overview of the chapter is given in Table 1.1.

Table 1.1: Overview over Chapter 1

1.2 Research Context
1.3 Motivation
1.4 Research Scope and Approach
1.5 Problem Statement
1.6 Dissertation Goals
1.7 Research Hypotheses
1.8 Research Contributions
1.9 Thesis Structure

First of all, the chapter introduces the research context. Next, it presents the motivation to propose a new DICOM data management system. Then, the research scope and approach are described. After that, the chapter depicts the problem statement, the dissertation goals and hypotheses. It also points out the research contributions. Finally, a description of the thesis structure is given.

1.2 Research Context

In health-care industry, the development of imaging technologies, long-term retention, and increase of image resolution are causing a tremendous growth in data volume. Besides, the variety of acquisition devices and the differences in preferences of physicians or other health-care professionals have led to a high variety in data. Although DICOM standard [1] has been popularly used for storing the medical image data, DICOM data still has characteristics of Big Data such as high complexity, high variety, high and ever-increasing volume, and high velocity [2]. In addition, types of queries/retrieval operations on this data may be Online Transaction Processing (OLTP), an Online Analytical Processing (OLAP) or a mixture of both OLTP and OLAP. As a consequence, all of these have caused many issues in data management.


Metadata consists of : <ul style="list-style-type: none"> - <i>patient's name</i> - <i>patient's ID</i> - <i>type of media in imaging (CT, MRI, medical reports, etc.)</i> - ... 	<table border="1"> <thead> <tr> <th colspan="2">Image</th> <th colspan="4">Header</th> </tr> <tr> <th>Tag</th> <th>Tag Description</th> <th>VR</th> <th>Length</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>0002, 0000</td> <td>Group Length</td> <td>UL</td> <td>4</td> <td>192</td> </tr> <tr> <td>0002, 0001</td> <td>File Meta Information Version</td> <td>OB</td> <td>2</td> <td>0</td> </tr> <tr> <td>0002, 0002</td> <td>Media Storage SOP Class UID</td> <td>UI</td> <td>26</td> <td>1.2.840.10008.5.1.4.1.1.7</td> </tr> <tr> <td>0002, 0003</td> <td>Media Storage SOP Instance UID</td> <td>UI</td> <td>50</td> <td>1.2.840.113619.2.1.2411.1031152...</td> </tr> <tr> <td>0002, 0010</td> <td>Transfer Syntax UID</td> <td>UI</td> <td>18</td> <td>1.2.840.10008.1.2</td> </tr> <tr> <td>0002, 0012</td> <td>Implementation Class UID</td> <td>UI</td> <td>18</td> <td>1.2.840.113619.6.5</td> </tr> <tr> <td>0002, 0013</td> <td>Implementation Version Name</td> <td>SH</td> <td>6</td> <td>1_2_5</td> </tr> <tr> <td>0002, 0016</td> <td>Source Application Entity Title</td> <td>AE</td> <td>12</td> <td>CTN_STORAGE</td> </tr> <tr> <td>0008, 0000</td> <td>Group Length</td> <td>UL</td> <td>4</td> <td>414</td> </tr> <tr> <td>0008, 0008</td> <td>Image Type</td> <td>CS</td> <td>20</td> <td>DERIVED\SECONDARY\3D</td> </tr> </tbody> </table>	Image		Header				Tag	Tag Description	VR	Length	Value	0002, 0000	Group Length	UL	4	192	0002, 0001	File Meta Information Version	OB	2	0	0002, 0002	Media Storage SOP Class UID	UI	26	1.2.840.10008.5.1.4.1.1.7	0002, 0003	Media Storage SOP Instance UID	UI	50	1.2.840.113619.2.1.2411.1031152...	0002, 0010	Transfer Syntax UID	UI	18	1.2.840.10008.1.2	0002, 0012	Implementation Class UID	UI	18	1.2.840.113619.6.5	0002, 0013	Implementation Version Name	SH	6	1_2_5	0002, 0016	Source Application Entity Title	AE	12	CTN_STORAGE	0008, 0000	Group Length	UL	4	414	0008, 0008	Image Type	CS	20	DERIVED\SECONDARY\3D
	Image		Header																																																											
Tag	Tag Description	VR	Length	Value																																																										
0002, 0000	Group Length	UL	4	192																																																										
0002, 0001	File Meta Information Version	OB	2	0																																																										
0002, 0002	Media Storage SOP Class UID	UI	26	1.2.840.10008.5.1.4.1.1.7																																																										
0002, 0003	Media Storage SOP Instance UID	UI	50	1.2.840.113619.2.1.2411.1031152...																																																										
0002, 0010	Transfer Syntax UID	UI	18	1.2.840.10008.1.2																																																										
0002, 0012	Implementation Class UID	UI	18	1.2.840.113619.6.5																																																										
0002, 0013	Implementation Version Name	SH	6	1_2_5																																																										
0002, 0016	Source Application Entity Title	AE	12	CTN_STORAGE																																																										
0008, 0000	Group Length	UL	4	414																																																										
0008, 0008	Image Type	CS	20	DERIVED\SECONDARY\3D																																																										
Image data consists of image pixels																																																														

Figure 1.1: Example of metadata and image data in a DICOM file

The DICOM standard was released the first time in 1985 as ACR/NEMA standard. It includes a set of non-propriety specifications regarding structure, format, and exchange protocols for digital-based medical images. Each DICOM file contains a *header*, *metadata* and *pixel data*: the header is used to recognize if a file is a DICOM file; the metadata contains attributes storing information about real-world entities (such as *Patient*, *Study*, etc.) related to the corresponding image; and the pixel data represents actual image pixels. Figure 1.1 illustrates the data of a DICOM file.

The wide use of the DICOM standard has led to the development of DICOM data management systems. In general, after a DICOM file is acquired using a specific medical equipment (e.g., a CT scanner, a MRI scanner, etc.), metadata and pixel data will be extracted, organized and stored according to a particular data storage strategy. Full-content images are usually stored in a file system from which they can be used for content-based image retrieval or for image parsing at pixel level. Furthermore, the attributes of the metadata can stored and indexed in metadata catalogs and/or databases in a way to provide more flexibility for users to perform query/retrieval operations [3]. However, due to the above-mentioned characteristics of DICOM data and workloads, existing systems still exist limitations in performance, efficiency, scalability, elasticity or supported query language. In fact, the manner in which DICOM data is stored has a strong impact on storage space demand and workload execution time.

In this thesis, we analyze existing practices in DICOM data management and propose efficient methods to store and query DICOM data.

1.3 Motivation

Nowadays, the DICOM standard is used in most hospitals in America, Europe and Asia [4]. There is a real need to propose a new data storage model together with efficient methods to store and query DICOM data. Our study is motivated by our analysis on the characteristics of DICOM data and workloads. Additionally, we are motivated by recent researches in the field of database system: (1) optimizing query

performance for mixed OLTP and OLAP workloads; (2) reducing storage space demand for sparse datasets; (3) filtering the redundant input data of queries; and (4) applying cloud-based solutions.

First, our analysis on the characteristics of DICOM data show that the following characteristics of DICOM data can cause challenges in data management: (1) *High Complexity*: the information model (provided by the DICOM standard) consisting of many entities and relationships among the entities. Each entity may include a large number of attributes. (2) *High Variety*: data consists of images and metadata. Metadata schemas are heterogeneous and evolutive. The number of attributes is very large (more than 3500), but some of them are mandatory while others are optional. The number of attributes used in a DICOM file varies considerably based on a particular examination modality (e.g., CT and MRI). The used attributes can also be modified if an image acquisition device (e.g., CT scanner) is modified. (3) *High Volume*: data size is terabytes or petabytes. For instance, in France, information and test results of a patient should be stored for up to 30 years [5]. (4) *High Velocity*: some applications need real-time processing of high-volume data streams, e.g., in-coming streams of images containing relevant information required for diagnosis.

Our observations on real DICOM datasets revealed that as a result of the high complexity and the high variety, entities usually contain a large number of attributes, many of which have null values (e.g., optional attributes) while others seldom get null values (e.g., mandatory attributes). Thus, if storing such entities in single wide-tables, the presence of the null values may cause a waste of storage space. For example, the entity *Patient* consists of the following attributes: *PatientName*, *PatientID*, *PatientBirthDate*, *PatientSex*, *EthnicGroup*, *IssuerOfPatientID*, *PatientBirthTime*, *PatientInsurancePlanCodeSequence*, *PatientPrimaryLanguageCodeSequence*, *PatientPrimaryLanguageModifierCodeSequence*, *OtherPatientIDs*, *OtherPatientNames*, *PatientBirthName*, *PatientTelephoneNumbers*, *SmokingStatus*, *Pregnancy*, *LastMenstrualDate*, *PatientReligiousPreference*, *PatientComments*, *PatientAddress*, *PatientMotherBirthName*, and *InsurancePlan Identification*. Only the first three attributes, i.e., *PatientName*, *PatientID*, and *PatientSex*, have low values of null ratio (e.g., 0.00 - 1.48%), whereas the remaining attributes are very sparse (their null ratios are 83.55 - 100.00%). Obviously, if storing the entire entity *Patient* in a single wide table, the null values will cause a big storage overhead. Therefore, there is a need for a storage design approach to remove the null values.

Besides the above characteristics of DICOM data, we analyzed several workloads and found that there is a variety of attribute usage and queries often consist of multi-table join operations with highly selective predicates. Some attributes are accessed more frequently than others; some are frequently accessed together in the same queries. For instance, the first four attributes of the entity *Patient*, i.e., *PatientName*, *PatientID*, *PatientBirthDate*, and *PatientSex*, are frequently accessed together in the same queries, whereas the others are seldom accessed. Especially, two attributes *Pregnancy* and *LastMenstrualDate* are not frequently accessed, but once used, they often appear together. Therefore, depending on the given workloads, the attributes can be grouped and stored together so that the queries can mainly access the relevant attributes, thereby reducing the number of redundant data accesses. To achieve this, we need a data storage design approach to deal with various workloads.

Second, in database research community, vertical partitioning algorithms has been proposed to create efficient physical database designs. They can be classified into two approaches: workload-based and data-based. The former approach tries to group and store frequently-accessed-together attributes into the same tables [6-13] in a way to decrease the number of irrelevant data accesses (reducing I/O cost) and thus improve the workload performance. On the other hand, the latter approach attempts to group and store co-occurrence attributes (having non-null values) together [14-16] in order to avoid storing null values, thereby reducing the storage space demand. The vertical partitioning, therefore, would be a potential solution to the problems of DICOM data management (reducing storage space demand and query performance).

In the last decades, several different data layouts have been applied deal with the different types of workloads of applications. Row stores (such as Oracle, DB2, and SQL Server) store all data associated with a row together. Each row contains attribute values for a single tuple/record and is stored sequentially on disk. This organization helps a system easily to add/modify a row and efficiently read all (many) columns of a single row at the same time. Therefore, the row stores are suitable for write-intensive (OLTP) workloads. However, they wastes I/O costs if only few attributes are needed to answer a query because all the attributes of a table have to be read into memory from disk, no matter how many attributes that query requires [17]. In contrast, column stores (such as MonetDB [18] and C-Store [19]) organize data by column. Each column contains data for a single attribute of a tuple and stored sequentially on disk. Using this organization, a system can read only relevant attributes and efficiently aggregates over many rows but only for a few attributes. Hence, the column stores are suitable for read-intensive (OLAP) workloads, but their tuple reconstruction cost in OLTP workloads is higher than that of the row stores. To overcome the gap between the row and column stores, some hybrid stores (e.g., HYRISE [12], SAP HANA [20]) have been proposed to optimize the performance for both types of the workloads. Using a hybrid store thus may be an efficient solution to store DICOM data as well.

Third, with regards to the problem of high volume of data, cloud-based systems have provided solutions for high performance computing together with availability, reliability, scalability, elasticity and so on. For instance, Spark [21], an in-memory cluster computing system which can run on Hadoop, has been introduced to cope with the high latency problem and provide high performance for interactive queries. Therefore, if used for large-scale DICOM data management, such a cloud-based system can supply an opportunity to speed up interactive queries as well as a scalable data storage for the high volume of DICOM data.

Fourth, besides storing data, due to the common use of highly selective predicates, the multi-table join queries usually involve a large amount of irrelevant data, not required in final results. There is an opportunity to improve performance of the queries by applying a query processing strategy that can reduce irrelevant data from input data. Bloom filters and Intersection Bloom filters [22-25] have shown their ability to filter redundant input data out of queries and thus can be applied to the context of DICOM data processing.

In a nutshell, there is a real need to propose a new data management system together with efficient methods to store and query DICOM data. The vertical partitioning approaches show that they can reduce the redundant data accesses and the

storage space demand. The hybrid stores show their potential to improve the performance of queries in mixed OLTP and OLAP workloads. Bloom filters and Intersection Bloom filters can improve query performance by removing irrelevant input data. Besides, the cloud-based systems also introduce possible solutions to deal with the problem of high volume of data. We believe it would be beneficial if combining these approaches together to build a new data storage model and efficient methods for storing and querying DICOM data.

1.4 Research Scope and Approach

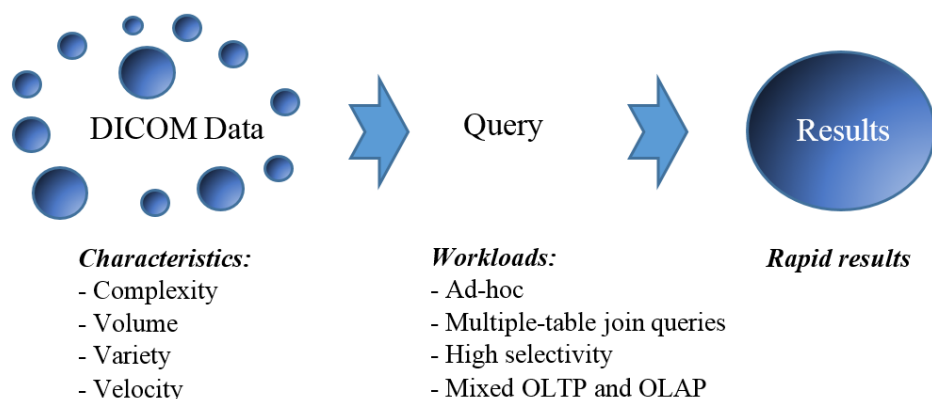


Figure 1.2: Research focus

This section describes the research scope and research approach. A typical application case of DICOM data involves extracting metadata and pixel data from DICOM files, storing them into a data storage(s), processing queries and presenting results to users. Figure 1.2 illustrates our research focus. Data storage and query performance for such an application have been challenged from the perspective of Big Data characteristics (i.e., complexity, variety, volume and velocity) as well as the variety of workloads (i.e., ad-hoc, high selectivity, mixed OLTP and OLAP workloads).

Our research focuses on efficient methods for storing and querying DICOM data. We attempt to provide a data storage strategy and a query processing strategy to reduce storage space and improve query performance.

- **Data storage strategy** refers to the way in which the data is organized and stored in the data storage system.
- **Query processing strategy** refers to strategies intended to improve efficiency of query processing.

Both the above strategies are challenged by the characteristics of DICOM data and workloads. However, the scope of our research is limited to the problems raised by the first three characteristics of DICOM data (i.e., complexity, volume and variety) and the various workloads. This is because the velocity is usually involved in stream processing-based applications [26] rather than a business analytics application (i.e., interactive ad-hoc query and analysis) as our focus.

Research approach

The research is carried out in six steps as given in Figure 1.3.

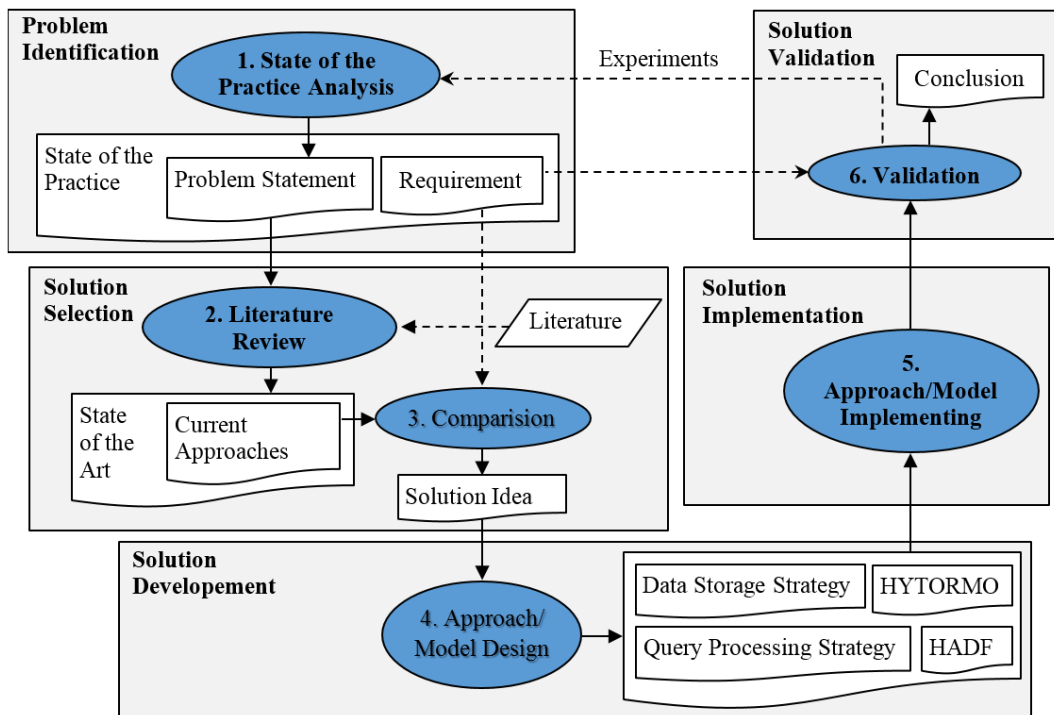


Figure 1.3: Research approach

1. State of the Practice Analysis: Practices and challenges of storing and querying DICOM data in current systems are analyzed. Problems and expected requirements are formulated. The results of this step are described in Chapter 2.

2. Literature review: Papers in the field of database system, including relational, NoSQL and NewSQL databases, hybrid storage systems, cluster computing frameworks, data layouts, vertical partitioning and Bloom filter techniques are searched and reviewed to identify current approaches and potential solutions to the given problems. The results of this step are presented in Chapter 3.

3. Comparison: The current approaches are compared with respect to the expected requirements to find out their limitations and to select suitable approaches for a new DICOM data management system. The results of this step are described in Chapter 3.

4. Approach/Model Design: A hybrid row-column storage model called HYTORMO, a hybrid automated design framework called HADF, and a query processing strategies with the integration of an Intersection Bloom filter (IBF) are proposed to satisfied the expected requirements. The results of this step are described in Chapters 4 and 5.

5. Approach/Model Implementing: HYTORMO together with the proposed methods are implemented. The results of this step are partially presented in Chapter 6.

6. Validation: To validate the proposed methods, real DICOM datasets are collected. Experiments are performed. The results of this step are described in Chapter 6.

1.5 Problem Statement

In Section 1.3, we showed our observations on the characteristics of DICOM data and workloads and potential approaches to deal with the problems of storing and querying DICOM data, including vertical partitioning, hybrid stores, Bloom filters and cloud-based solutions. With the research focus on storage space usage and query performance, we believe that it would be beneficial to combine those approaches together. However, this will introduce new research challenges: *How to combine the current methods including vertical partitioning, row- and column-stores, Bloom filters and cloud-based solutions in a right way to obtain efficient methods for storing and querying DICOM data? How can we improve workload execution time while still decreasing storage space? What are knowledge gaps that need to be filled in order to achieve the efficient methods for storing and querying DICOM data?*

First, although some researches have proposed hybrid storage models such as HYRISE [12] and SAP HANA [20], they have not been designed for storing DICOM data. For instance, they need additional storage space to store duplicate data across different data layouts and have not dealt with the problem of high volume and sparse data. Obviously, there is a need for a new DICOM data management system that is able to provide performance, efficiency, huge storage capacity, scalability, elasticity, normalized data, and declarative query language support, and to cope with the characteristics of DICOM data and workloads. The problem is how to provide a new hybrid storage model that can satisfy such requirements.

Second, in Section 1.3 we showed that, based on workload- or data-specific information, several vertical partitioning algorithms have been proposed to improve query performance (by eliminating redundant data accesses) or to reduce storage space size (by removing null values). However, there is a lack of an algorithm or a data design advisory tool that is able to capture the combined impact of both workload- and data-specific information. Moreover, the existing algorithms are implicitly assumed that vertical partitioning results will be stored by using a single data layout (e.g., a row store), instead of a hybrid store. Therefore, a problem is how to propose a new data storage design approach that is able to provide sufficient decision-support for the decision makers in determining the combined impact of workload- and data-specific information and a hybrid store on the quality of a data storage configuration (including schemas and data layouts) that can reduce both query performance and data storage demand.

Finally, another import problem is to provide a suitable and efficient query processing strategy built on top of the hybrid storage model. There is a need to propose a suitable query processing strategy that can correctly construct query results from vertically partitioned tables, e.g., inner and left-outer joins should be used. Besides, because queries usually consist of multi-table join operations with highly selective predicates, they may involve a large amount of irrelevant input data. As a result, when these queries are executed in a distributed query processing environment, the irrelevant input data may causes high network I/O cost and results in poor performance of the queries. In [25, 27], the authors proposed to apply an IBF computed from pre-computed BFs to improve the performance of MapReduce queries. However, an existing problem is how to apply the IBF built from non pre-computed BFs.

Table 1.2 summarizes three problems P1 – P3 addressed by this thesis:

Table 1.2: Problem statements

P1	Inefficient data storage model for storing and querying DICOM data.
P2	Insufficient decision-support for decision makers in data design for DICOM data to create good data storage configurations (including schemas and their corresponding data layouts) in terms of storage space demand query performance.
P3	Lack of a suitable and efficient query processing, especially when high network I/O cost is caused by irrelevant data.

1.6 Dissertation Goals

Based on the problem statement presented in the previous section, this section highlights our research goals. As earlier mentioned, a single data storage technique may not provide the best performance for different types of workloads; instead, it is expected that a hybrid storage model will yield a better performance. We also need efficient methods to reduce storage space size, tuple reconstruction cost and disk and network I/Os. Besides, a cloud-based systems can provide high performance, efficiency, scalability, elasticity and so on. In Table 1.3, we list three goals O1 – O3 of the thesis.

Table 1.3: Thesis goals

O1	Provide a new hybrid storage model, called HYTORMO, together with an efficient data storage strategy to improve query performance and decrease storage space size with respect to the characteristics of DICOM data and workloads. HYTORMO is able to provide high performance, efficiency, scalability, elasticity, normalized data and declarative query language.
O2	Provide a hybrid automated design framework, called HADF, to support decision making in database design for DICOM data. HADF is able to: <ul style="list-style-type: none"> • Take into account the combined impact of both workload-specific and data-specific information as well as the use of a hybrid store on the quality of a data storage configuration in terms of storage space size and query performance. • Generate a data storage configuration that can improve workload performance while still decreasing storage space demand.
O3	Provide a query processing strategy built on top the hybrid storage model with the use of inner joins, left-outer joins to create correct answers for queries and an IBF to remove irrelevant tuples from input tables of join operations.

1.7 Research Hypotheses

In order to evaluate the benefits of HYTORMO, data storage strategy, HADF and query processing strategy, three hypotheses are formulated:

- **H1 - Effectiveness of HYTORMO with respect to workload execution time:**
The hybrid storage model, i.e., HYTORMO, together with the proposed data

storage strategy, gives a faster workload execution time than a pure row store and a pure column store. This hypothesis is to assess Goal O1.

- **H2 - Usefulness of HADF for decision making in database design for DICOM data:** The hybrid automated design framework, i.e., HADF, can support decision making in database design for DICOM data. To be useful as a decision-support model, two following aspects are evaluated:
 - a) Taking into account the combined impact of both workload- and data-specific information can help HADF to produce better data storage configurations than using pure workload-specific information or pure data-specific information.
 - b) HADF is able to generate a data storage configuration that can decrease storage space demand and workload execution time at the same time.

This hypothesis is to assess Goal O2.

- **H3 – Effectiveness of the query processing with the integration of an IBF with respect to query execution time:** The query processing strategy with the integration of an IBF runs faster than without the IBF. This hypothesis is to assess Goal O3.

In Figure 1.4, we describe the causal relationship between the problem statements, thesis goals and research hypotheses. The goals are referred to as the proposed solutions to the research problems, and the hypotheses show what will be validated to evaluate the benefits of such solutions.

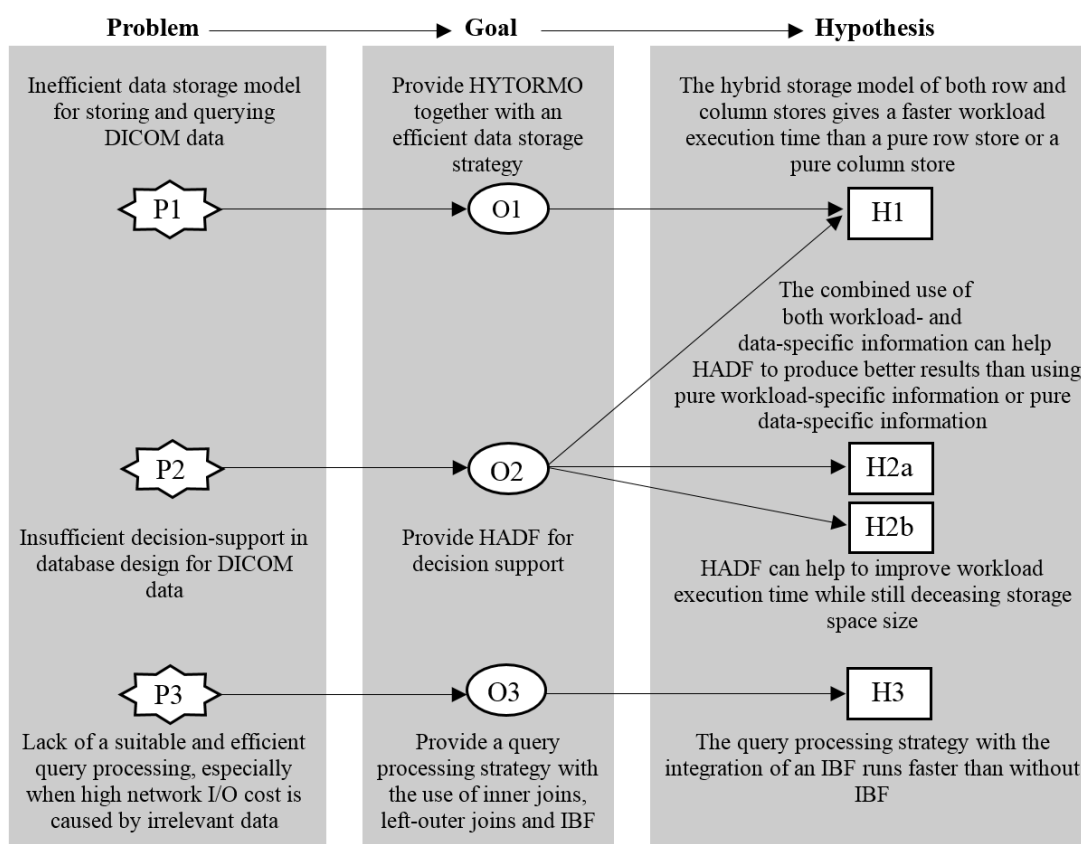


Figure 1.4: Causal relationship between problems, goals and research hypotheses

1.8 Research Contributions

The thesis has the following contributions:

- **Comprehensive evaluation of the existing DICOM data management systems:** The existing systems are evaluated against expected requirements in DICOM data management. The evaluation is described in Chapter 2.
- **State of the art review of the current databases:** This state of the art review presents a comprehensive background of the most prevalent databases (relational, NoSQL and NewSQL databases). It highlights advantages and disadvantages of these databases with respect to their suitability when used for various workloads (i.e., OTLP and OLAP) and data structures (structured and semi/unstructured data). This state of the art review is described in Chapter 3.
- **HYTORMO together with a data storage strategy for DICOM data:** HYTORMO provides high performance for mixed workloads. It is designed based on the relational data model to provide facilitates for users (e.g., to use DICOM entity tables and SQL language). It is implemented on top of an in-memory cluster computing framework, called Spark [21], to supply high performance for interactive workloads, huge storage capacity, scalability and elasticity. The data storage strategy aims to reduce storage space and query execution time. HYTORMO and the data storage strategy are shown in Chapter 4.
- **HADF - a hybrid automated design framework:** HADF is proposed to provide decision-support for decision makers in selecting good data storage configurations. It is able to take into account the combined impact of both workload- and data-specific information as well as the mixed use of both row and column stores to generate a data storage configuration. HADF is described in Chapter 4.
- **Query processing strategy with the integration of an IBF:** The query processing strategy built on top of HYTORMO with the use of inner joins, left-outer joins and an IBF. This query processing strategy is given in Chapter 5.
- **Validations of the proposed methods:** HYTORMO, the data storage strategy, HADF and the query processing strategy are validated using real DICOM datasets and different workloads. The validation results are presented in Chapter 6.

1.9 Thesis Structure

The remainder of this thesis is organized as follows: Chapter 2 gives general background on the DICOM standard, existing DICOM data management systems, problems and expected requirements for a new system. Chapter 3 presents the state of the art review of workload types, the most prevalent databases, cluster computing framework, data layouts, vertical partitioning and Bloom filter techniques, and key components of the new system. Chapter 4 presents HYTORMO and HADF. The query processing with the integration of an IBF is described in Chapter 5. The evaluation of the proposed methods is reported in Chapter 6. Chapter 7 concludes the thesis and introduces future works.

PART I

BACKGROUND AND RELATED WORKS

DICOM Data Management Systems and Requirements

2.1 Overview

This chapter presents background on existing DICOM data management systems and requirements for a new system. An overview of the chapter is shown in Table 2.1.

Table 2.1: Overview over Chapter 2

2.2 DICOM Standard and Data		
2.2.1 DICOM Standard	2.2.2 Characteristics of DICOM Data and Workloads	
2.3 DICOM Data Management Systems		
2.3.1 Expected Requirements	2.3.2 Existing Systems	2.3.3 Conclusion
2.6 Summary and Conclusion		

First, the chapter gives an overview of background information on the DICOM standard. Next, we determine the major characteristics of DICOM data and workloads that may cause challenges in data management. Then, we present the expected requirements for a new DICOM data management system. After that, we give an overview of the existing DICOM data management systems as well as discuss their strengths and weaknesses. We make a comparison among these systems and conclude with their limitations in satisfying the expected requirements. We finally present summary and conclusion of the chapter.

2.2 DICOM Standard and Data

This section provides an overview of background information on the DICOM standard and then presents characteristics of DICOM data and workloads.

2.2.1 DICOM Standard

Information Model

The DICOM standard defines an *information model* based on an object-oriented abstract data model to specify information and relationships among real world objects. The information model is built according to the way images created by different modalities managed in a department, e.g., radiology departments. Figure 2.1 illustrates the mapping of real-world examinations to the information model. There are four

levels of information: *Patient*, *Study*, *Series* and *Image*. The *Patient* level is the highest level where all information related to a single patient who has one or more studies. The *Study* level is the most important level because it keeps the result of a required examination for the patient. Most works in the department, where the modalities are managed, mainly concern on handling of the studies. All information related to the same study is maintained. A single patient may have multiple studies, each of which may require several examinations performed on different modalities. This creates different series of one or more images. The *Series* level keeps information about date/time when the series are created, type of the used modality, used equipment and so on. The *Image* level is referred to as DICOM files that are stored for later use.

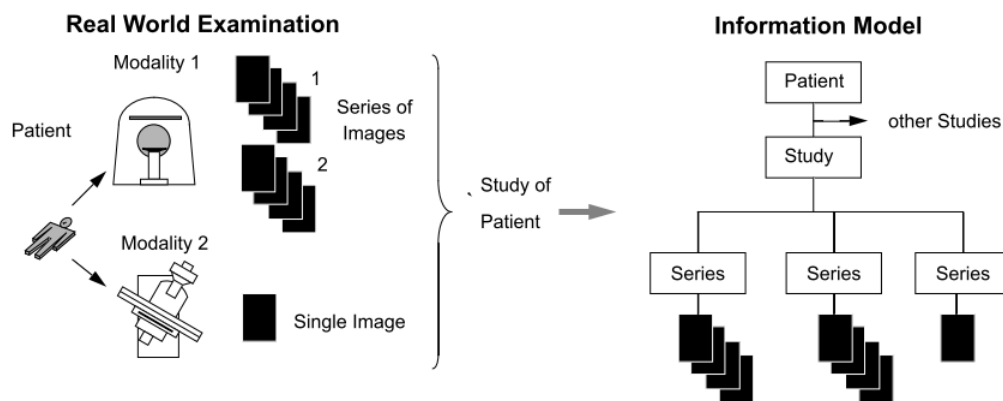


Figure 2.1: Mapping real-world examinations to the information model [28]

Service Classes and SOP Classes

The data exchange between two systems (or partners) in a distributed processing environment is performed using the *Service Class*. This class describes the roles of each partners (a Service Class User or a Service Class Provider) and the context of the defined services. It also defines information and operations [28]. For these works, the DICOM standard uses an object oriented class definition, called *Service Object Class* (*SOP Class*), to integrate information and operations together. The SOP Class definition combines a single *Information Object Definition (IOD)* with several *services*. Before any data exchange occurs, two partners must agree to use a SOP Class and must verify their role as described with regarding to the context. The type of the data exchange may be network or media. For example, a SOP Class, called Media Storage Service Class, stores information in a file on a media. This class defines services permitting to use the media type of data exchange. The processes on both partners must agree on what information will be exchanged using the media type.

Information Object Definitions (IODs)

IODs are used to define the information part of a SOP Class. An IOD is regarded as a set of interrelated parts of information, kept in *information entities*. Each *information entity* (IE) represents information about a single real world object such as *Patient*, *Study*, *Series*, *Equipment* and *Image* [1]. Each IE in turn consists of a list of *attributes* describing the corresponding object. It is worthy to note that an IOD does not represent an instance of a real-world object; instead, it describes an object or a class of objects.

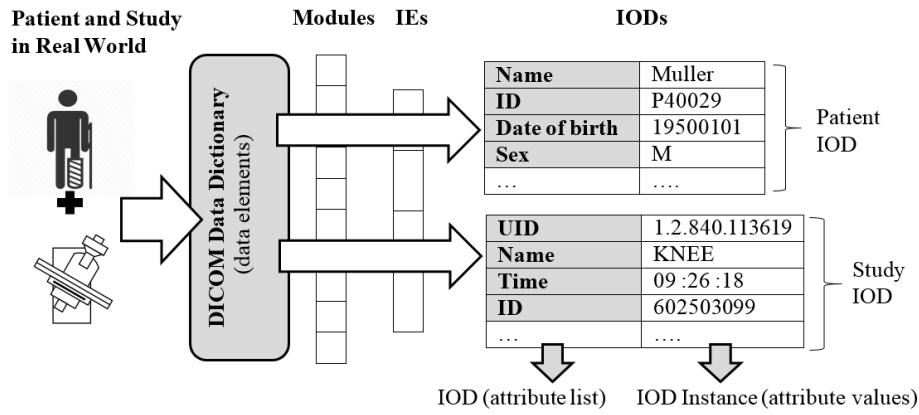


Figure 2.2: Transforming an object in real world into an IOD object

Depending on requirement of the Service Class of the SOP Class, an IOD can contain a single information entity (called a *normalized IOD* or NIOD) or a mixture of several information entities (called *composite IOD* or CIOD). A NIOD represents a single real-world entity whose attributes inherently describe the corresponding real-world entity. For example, a *Patient* NIOD only consists of attributes that inherently describe a patient such as *Patient Name*, *Patient Identifier*, *Patient Date of Birth*, *Patient Sex* and so on. Similarly, a *Study* NIOD only contains inherent attributes of a study such as *Study Unique Identifier*, *Study Name*, *Study Time*, *Study ID*, *Referring Physician* and so on, but it would not include any attribute of a patient such as *Patient Name*. The DICOM standard uses a data dictionary to maintain a list of all attributes. Each attribute belongs to one of value representations (VRs) types or data types, e.g., *Person Name (PN)*, *Unique Identifier (UI)*, *Date (DA)* and so on [1]. The process of transforming an object in real world into an IOD object is illustrated in Figure 2.2. In contrast to a NIOD, a CIOD contains inherent attributes as well as non-inherent attributes; it mixes several real-world entities or their parts.

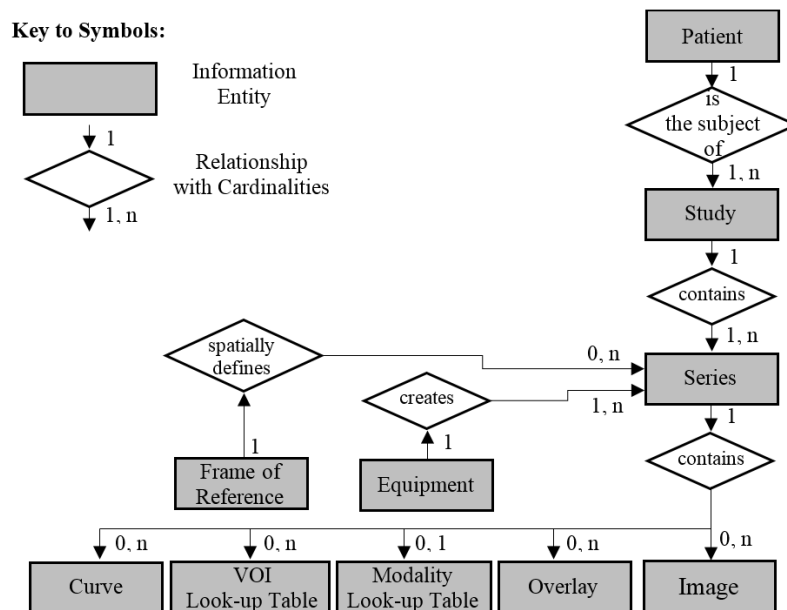


Figure 2.3: Detailed DICOM information model [28]

The semantically related attributes can be grouped together to create *information object modules* (IOMs) so that these IOMs can be used in one or more IODs. By this way, the DICOM standard can define an object-oriented abstract data model that represents the relationships among different IEs: *Patient*, *Study*, *Series* and *Instance* (e.g., *Image*). The information model shown in Figure 2.3 is a detail version of the one presented in Figure 2.1. Each rectangle block represents an information entity (IE) of a composite Information Object Definition (composite IOD) that is used in a SOP Instance. A relationship with cardinalities describes a relationship between IEs. For instance, the information model indicates that the *Patient* IE may have relationship with many *Study* IEs, each of which in turn may have multiple *Series* IEs, and so on.

Modules

Each type of image (e.g., CT, MRI) has a standard set of mandatory (M), conditional (C) and user optional (U) modules specified. A module is an abstract information entity that may contain an individual attribute or a set of attributes that are grouped together for describing a certain aspect of the context of the image. For instance, the Image Pixel module includes the attributes that describe the encoding and the format of the pixel matrix of the image such as Bits Allocated, Bits Stored, Pixel Data, etc. There is a list of modules defined in the DICOM standard [28]. Therefore, when building an IOD, the module can be selected from this list.

Attributes (Data Elements)

Each attribute or data element has a meaning and is listed in the DICOM data dictionary. Each attribute is composed of *tag*, *value representation*, *value length* and *value*, as shown in Figure 2.4.

Tag	VR	Value Length	Value
-----	----	--------------	-------

Figure 2.4: Structure of a DICOM attribute (data element)

These components are described as follows [29]:

- A *Tag* identifies an attribute or an element. It is composed of two identifiers: (*Group identifier*, *Element identifier*), represented by hexadecimal numbers. The attributes are organized into groups corresponding to real-world entities, e.g., *Patient* (0010), *Study* (0008), etc. We can identify an attribute via its tag, e.g., *Patient Name*: (0010, 0010), *Patient ID*: (0010, 0020), *Study Date*: (0008, 0020), *Study Time*: (0008, 0030).
- A *Value Representation (VR)* defines data type and format of an attribute value. Figure 2.5 lists a subset of attributes in a DICOM file. This file uses the following VRs: PN: Person Name; LO: Long String; DA: Date; CS: Code String; AS: Age String; DS: Decimal String; LT: Long Text; SH: Short String; IS: Integer String.
- A *Value Length* specifies the length of the value (in bytes).
- A *Value* contains the data of an attribute.

Group	Element	Tag Description	VR	Length	Value
0010	0010	Patient's Name	PN	6	P40029
0010	0020	Patient ID	LO	6	P40029
0010	0030	Patient's Birth Date	DA	8	19000101
0010	0040	Patient's Sex	CS	2	M
0010	1010	Patient's Age	AS	4	065Y
0010	1030	Patient's Weight	DS	6	77.111
0010	21B0	Additional Patient History	LT	4	
0018	0020	Scanning Sequence	CS	2	RM
0018	0021	Sequence Variant	CS	4	NONE
0018	0022	Scan Options	CS	12	SEQ_GEMS\PFF
0018	0023	MR Acquisition Type	CS	2	2D
0018	0025	Angio Flag	CS	2	N
0018	0050	Slice Thickness	DS	2	5
0018	0080	Repetition Time	DS	2	5
0018	0081	Echo Time	DS	6	1.424
0018	0082	Inversion Time	DS	2	0
0018	0083	Number of Averages	DS	2	1
0018	0084	Imaging Frequency	DS	10	63.851238
0018	0085	Imaged Nucleus	SH	2	1H
0018	0086	Echo Number(s)	IS	2	1
0018	0087	Magnetic Field Strength	DS	4	1.5
0018	0088	Spacing Between Slices	DS	4	6.5
0018	0091	Echo Train Length	IS	2	1

Figure 2.5: Some attributes used in a DICOM file

As mentioned in Section 2.2.3, an OID object can contain a single information entity or a mixture of several information entities. In order to achieve this, attributes that have semantic relationship are organized in the same IOD. Table 2.2 gives an example of a subset of the attributes of the *Study* IOD.

Table 2.2: A subset of the attributes in the *Study* IOD

Tag	Unique Attribute Name	Attribute Description	Type
(0020,000D)	Study Instance UID	Unique identifier for the study	1
(0008,0020)	Study Date	Date the study started	2
(0008,0030)	Study Time	Time the study started	2
(0020,0010)	Study ID	User or equipment generated study identifier	2
(0008,0090)	Referring Physician's Name	Name of the patient's referring physician	2
(0008,0050)	Accession Number	A generated number to identify order of Study	2

The type of an attribute in an IOD specifies not only whether the corresponding attribute is a mandatory or optional attribute, but also whether that attribute is required to represent with or without a value if it is a mandatory attribute. In particular, a type is 1 for mandatory with an actual value, 2 for mandatory that is allowed to get a null value or 3 for optional. Furthermore, types 1 and 2 can add a 'C' (i.e., 1C and 2C, respectively) to make an attribute mandatory if certain conditions are met.

It is possible to add new attributes that have not been defined in the DICOM standard. By this way, a vendor can define attributes specific to their own equipment. These attributes may not be used by other vendors.

2.2.2 Characteristics of DICOM Data and Workloads

This section concentrates on the characteristics of DICOM data and its workloads that may cause challenges in data management. The characteristics of DICOM data include complexity, high variety, high and ever-increasing volume, and high velocity. DICOM data thus has the characteristics of *Big Data* (characterized by *three V's* (or 3V's): *volume*, *variety* and *velocity* [30, 31]. Additionally, there is a variety in workloads accessing this data such as OLAP, OLTP and mixed workloads.

High Complexity

The DICOM information model, introduced in Section 2.2.1, represents multiple IEs and the relationships among these IEs. The information of the IEs are interrelated to each other in some ways (directly or indirectly). All of this gives us an example of complex data.

High Variety

Group	Element	Tag Description	VR	Length	Value
0008	0050	Accession Number	SH	16	2069440034810711
0010	0010	Patient's Name	PN	6	P40028
0010	0020	Patient ID	LO	6	P40028
0010	0030	Patient's Birth Date	DA	8	19000101
0010	0040	Patient's Sex	CS	2	M
0010	1000	Other Patient IDs	LO	0	
0010	1010	Patient's Age	AS	4	067Y
0010	21B0	Additional Patient History	LT	8	

Buttons: Hide zero length fields, Hide private fields, OK, Cancel, Export to text file..., Help...

(a) CT image

Group	Element	Tag Description	VR	Length	Value
0008	0050	Accession Number	SH	8	G-63275
0010	0010	Patient's Name	PN	14	Chest~Portable
0010	0020	Patient ID	LO	4	234
0010	0030	Patient's Birth Date	DA	8	19441217
0010	0040	Patient's Sex	CS	2	M

Buttons: Hide zero length fields, Hide private fields, OK, Cancel, Export to text file..., Help...

(b) CR image

Figure 2.6: Different attributes used for *Patient* IE of CT and CR images

Variety of data usually refers to the fact that data can be represented in different data types and data structures [30, 31]. There is a high variety in the data type and the data structure of DICOM data: the data consists of image data and metadata; moreover, the metadata can be represented in the form of structured and semi/unstructured data. The schemas of metadata are heterogeneous and evolutive.

Heterogeneous Schemas: The number of attributes stored in a DICOM file is very large, with more than 3,500 attributes (a full list of all standard DICOM attributes in *Digital Imaging and Communications in Medicine (DICOM) - Part 6: Data Dictionary* [32]). However, the attributes that are actually used in a particular context are often known just at the time when the DICOM files are created (for an examination modality such as CT, CR, MRI and so on). Some attributes are mandatory, while others are optional. Moreover, a vendor (such as Philip, Siemen or others) can have its own private attributes for its image acquisition equipment. Besides these reasons, different health-care professionals (e.g., physician, doctors) can make various decisions about what attributes are necessary for a particular case. Figure 2.6 illustrates a heterogeneous schema in which the used attributes may vary from one DICOM file to another: Figure 2.6(a) presents a CT image while Figure 2.6(b) presents a CR image. Here, we only focus on the attributes used for the *Patient IE*. In the former file, three attributes *Other Patient IDs*, *Patient's Age* and *Additional Patient History* are used, but they are not used in the latter file.

Evolutive Schemas: Schema evolution refers to changes in schemas of the metadata through time as attributes are modified. The schema evolution can occur in several ways: (1) a modality is newly added or modified, thus several private attributes for its equipment may need to be added or modified to the existing schemas; and (2) domain experts require to newly add or to modify some attributes with respect to their needs.

High and Ever-increasing Volume

Table 2.3: Example of DICOM file sizes

Modality	Typical Images/study	Average size of images (MB)	Typical study size (MB)
Magnetic Resonance (MR)/Computed Tomography (CT)	64	0.36	22
Cardiac CT	2051	0.5	1031
Visible Light (VL)	16	1.6	26
Mammography (MG)	4	26.4	106
Ultrasound (US)	1	27.5	28
Pathology	4	1319	5276

Volume refers to size of data. Sizes of DICOM files are usually large and vary considerably according to the following factors: digital imaging modality, vendor of the used equipment, resolution, image size, bit depth (number of bits per pixel) and color space (such as grayscale, RGB or CMYK). Increasing pixel bit depth will improve image quality, but will cause an increment of the file sizes. For instance, a computed radiography (CR) image that comprises of a 2,500 x 2,500 matrix with a grayscale bit depth of 12 bits will have a size of $2,500 \times 2,500 \times (12/8) = 9.375$ MB.

Similarly, the size of a computed tomography (CT) examination containing 2,500 images, each of which is made up of a 512 x 512 matrix at a grayscale bit depth of 16 bits, is determined by $2,500 \times 512 \times 512 \times (16/8) = 1.31072$ GB [33].

Table 2.3 gives another example of the DICOM file sizes in a benchmark dataset, presented in the white paper by Oracle [34]. This dataset contains the DICOM files created by six different digital imaging modalities. Its total size is about 2 terabytes, including 2.4 million images of 20,080 studies.

Besides the high volume, the DICOM files collection is ever-growing because more and more DICOM files are produced and stored for a long periods of time.

High Velocity

Velocity is regarded as the speed of the coming data streams that need to be processed as fast as possible to satisfy requirements of applications [30, 31]. For example, in the context of DICOM data, the in-coming streams of images containing relevant information required for diagnosis applications usually have a high velocity.

Various Workloads

Besides the characteristics of data, queries over DICOM data often consist of multi-table join operations with highly selective predicates on attributes of the entity tables that are used to store the IEs according to the DICOM information model. Additionally, there is a variety in attribute access patterns: Some attributes are frequently accessed together in the same queries while other attributes are seldom used. Some groups of attributes are used more frequently than others. These characteristics of queries imply that mixed OLTP and OLAP workloads may be applied to DICOM data.

2.3 DICOM Data Management Systems

In this section, we first lists expected requirements based on which new efficient methods will be proposed for storing and querying DICOM data to achieve the dissertation goals as given in Section 1.6 in Chapter 1. Next, we present the existing systems. Finally, we conclude the section based on an evaluation of the existing systems.

2.3.1 Expected Requirements

The Big Data characteristics of DICOM data have caused many challenges in data management. First, for the complexity of data, queries may require to integrate information from multiple IEs and thus may need a high computational cost for joining multiple tables of these IEs. Second, in order to handle the schema heterogeneity, suitable solutions should be proposed [35, 36]. If using a wide table of a relational database to store a large number of attributes, queries with different attribute access patterns in workloads generally make redundant attribute accesses which drastically decrease the system performance. Using such a wide table can also results in the waste of storage space as missing data values that are usually represented by sentinel values, e.g., “null”. Third, the schema evolution introduces other challenges. It is hard to efficiently manage the schema evolution in Relational Database Systems (RDBMSs)

since their relational data model is based on tables that do not supply flexible schema. Handling mutable schemas should be applied in a manner so that the current system still can continue to operate normally in the presence of “new version” schemas. Besides, the system should be easy to use; it can provide transparency to users so that it can use the last version of a schema without knowledge about how the corresponding data is stored on underlying storage.

The high and ever-increasing volume of data has presented challenges to modern data management. Although there is no specific threshold to determine how much data is “high” or “big”, in order to manipulate and analyze the high volume of data, database systems, infrastructures, strategies for long-term storage and data processing should have the capability to deal with large-scale datasets [30, 31, 37]. A common solution is to add more computer resources (CPU, memory, storage space, and so on) to the existing system to guarantee the speed of processing [38]. However, this solution is expensive, but the system performance might not be significantly improved if the existing system infrastructure and database are not suitable for storing and processing such massive data.

The high velocity has posed several issues in handling streams of large datasets because data processing operations (e.g., to retrieve and display a large set of images containing relevant information at the time when diagnostic decisions are being made) are relatively time-consuming and thus can cause considerable time delays. Therefore, the speed of data processing operations needs to be considered [39]. However, in our research we focus the attention on improving the speed of queries in mixed OLTP and OLAP workloads instead of data streams.

The mixed OLTP and OLAP workloads may cause a negative impact on the performance of queries because of irrelevant attribute accesses, high tuple reconstruction cost, cache utilization inefficiency and so on. Thus, this characteristic of the workloads needs to be taken into account when proposing a suitable data management system for DICOM data.

To tackle the above problems, we specify the expected requirements for a new DICOM data management system as the followings:

R1) Flexible data: The system is able to deal with complexity of DICOM data by allowing users to easily represent the entity tables and their relationships in the DICOM information model. Normalized data needs to be created. Additionally, the system is able to deal with the variety of DICOM data by supporting flexible and schema-less design to handle heterogeneous and evolutive schemas.

R2) Flexible querying: The system enables users to write SQL ad-hoc queries with join operations.

R3) Efficiency of storage and CPU: First, data needs to be organized based on both workload and data-specific information to reduce storage space demand and execution time of queries in mixed OLTP and OLAP workloads. More particularly, data needs to be organized and stored in a suitable way to reduce redundancy in storing data (e.g., avoiding to store null values), tuple reconstruction cost, and I/O costs. Second, the system is able to provide solutions for efficient query processing over large-scale DICOM datasets. Lastly, it is able to provide huge storage capacity, scalability and elasticity by supporting horizontal scaling.

2.3.2 Existing Systems

PACSs

PACSs (Picture Archiving and Communication Systems) refer to computer systems (comprised of both hardware and software) used for automatically acquiring, storing, distributing and displaying medical images. All the PACSs must follow the DICOM standard. A typical PACS includes the following components: (1) modality scanners such as X-Ray, MRI and CT scanners; (2) a secure network for transmitting images and patients' information; (3) display workstations for displaying and interpreting patients' images; and (4) long- and short-term storages for archiving images, patients' information, and reports. A typical PACS-based workflow in a hospital can be described in Figure 2.7 with the following steps.

1. A patient is prescribed by a doctor to undertake an examination using a particular modality such as a X-Ray, a MRI or a CT scanner. This requirement is sent to a Hospital Information System (HIS) or a Radiology Information System (RIS), and then to the corresponding modality via a DICOM Modality Worklist.
2. A practitioner (e.g., physican) uses information from the DICOM Modality Worklist to scan the patient using a specified modality scanner.
3. Patient's images are sent to the modality console.
4. Some processes are done on the modality console to create DICOM files.
5. The DICOM files on the modality console are stored in a central storage of the PACS server. Then, they can be stored in a long-term archive.
6. Professionals (doctor, physican, radiologist, health care worker, etc.) can use the display workstations that have PACS application software to display and perform image-manipulation techniques for interpreting patient's images.

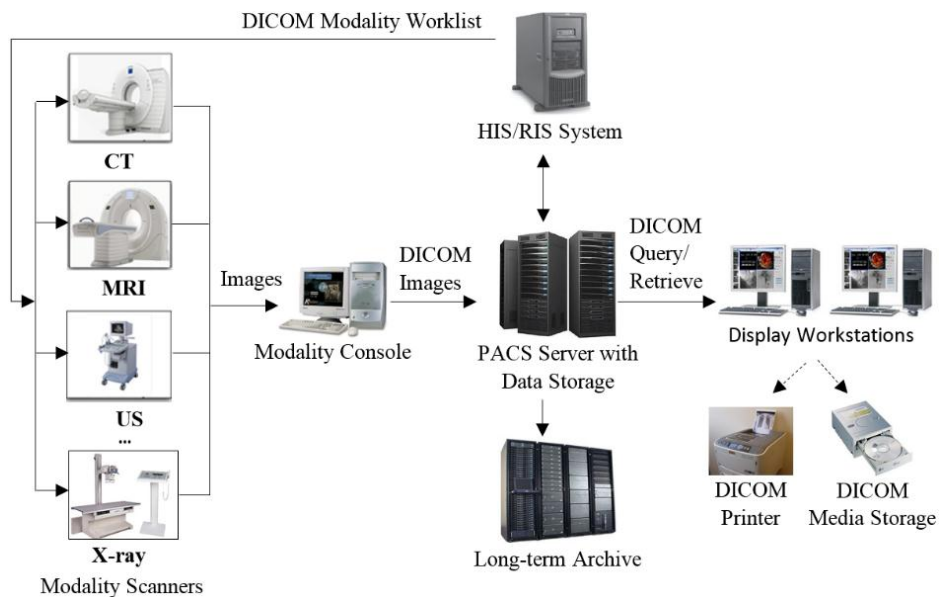


Figure 2.7: Typical PACS-based workflow

The PACSs continue to evolve over the time to adapt to developments in medical image imaging. Their data storages have been able to cope with high and ever-increasing volume of DICOM data [40]. Additionally, they have provided functionalities to help the professionals concurrently display images on different distributed display workstations (e.g., teleradiology network system on cloud) [41].

However, there are several limitations to a PACS. First, the entire PACS depends on its modality devices that are often produced by one or more particular vendors, thus the DICOM files used in different PACSs may consist of different attributes. As a result, integrating data from different PACSs can cause challenges. Second, the data storage of a PACS is generally based on a row-oriented RDBMS such as Oracle, MySQL, SQL Server or PostgreSQL. Therefore, when the DICOM files are sent to the PACS server, the most important attributes are extracted to be archived in columns of database tables while the rest of the attributes are kept in the database as Objects such as BLOB (Oracle, MySQL), IMAGE (SQL server) or BYTEA (PostgreSQL). Although the RDBMSs have provided the tabular form to represent data, normalized data, SQL for easy-to-use and the robust index techniques for speeding of data retrieval operations, the PACSs mainly allow to use queries with predefined parameters (non-ad-hoc queries). Their RDBMSs have not well supported for flexible and schema-less design and are also hard to scale up for high and ever-increasing volume of DICOM data.

eDiaMoND

eDiaMoND (Grid-enabled Medical Imaging Database) project was aimed to develop a prototype for a national medical imaging database of digital mammograms to support the United Kingdom's breast cancer screening [42]. eDiaMoND database is a Grid-enabled medical imaging database. It was designed to store DICOM files and it was intended to be used with two main applications: (1) teaching and training in clinical radiology; and (2) computer-aided diagnosis [43].

In order to develop the eDiamond database, the object-relational approach and Grid technology (OGSA-DAI Grid Data Service [44]) were used. The former is applied to easily manage DICOM information entities. The latter provides a solution to the problem of data federation as well as effective collaboration between healthcare professionals; its aims are to provide inter-operability, scalability and elasticity.

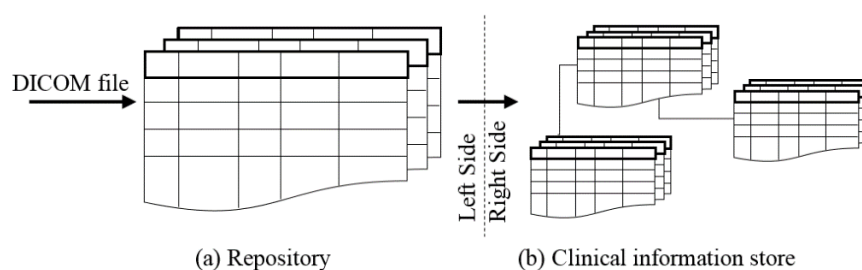


Figure 2.8: eDiaMoND architecture [42]

To be able to handle all types of DICOM data, the architecture of the eDiaMoND database is logically separated into two parts: (1) *repository* and (2) *clinical information store*, as presented in Figures 2.8. When a DICOM file is inserted into the

eDiaMoND database, its metadata is extracted: all attributes (i.e., optional and private attributes) are stored in the repository while only currently-used data is stored in the clinical information store. The data is stored in the repository in an unnormalized fashion to prevent data loss, whereas the data is stored in the clinical information store in a normalized fashion to guarantee the data integrity. Because the clinical information store is keeping the data that also exists in the repository, the eDiaMoND provides an update mechanism to guarantee that the data between two parts is consistent.

Figure 2.9 presents the architecture of the Grid Data Service. Users are not allowed to directly submit a SQL query to the eDiaMoND database. Instead, they will send it to the Query Service in a pre-determined format such as a XML document. After executed, the query result is returned in form of a XML document as well.

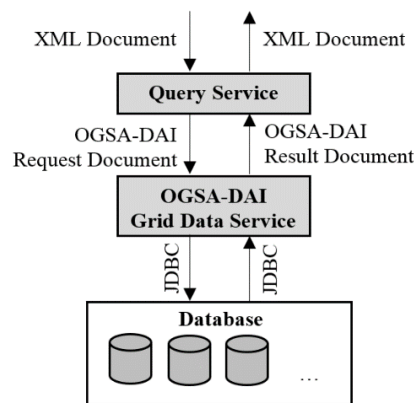


Figure 2.9: Architecture of Grid Data Service [42]

The eDiaMoND database can store all the attributes of the DICOM files, thus it can prevent loss of data and deal with the variety and evolution of DICOM data. Storing data in the tabular form make it easy to represent the entity tables and their relationships in the DICOM information model. However, eDiaMoND needs more disk space because, as mentioned above, some piece of data is stored twice (in the repository store and the clinical information store). Additionally, this database is based on a row-oriented RDBMS (DB2) [45], thus its query performance on very large datasets is limited (e.g., query processing cannot eliminate redundant read accesses if only a few attributes are required by a query). eDiaMoND only provides users with pre-determined queries. Moreover, although horizontal scaling is provided by using a Grid infrastructure, this scaling out is costly and technically complex in terms of Big Data.

Oracle

DICOM feature was first available to developers in Oracle Database 10g Release 2 (10.2) [46]. In this release, ORDImage object type was supported to permit Oracle Multimedia to recognize DICOM content and to extract a subset of embedded DICOM attributes associating to the entities *Patient*, *Study*, *Series*, etc. Oracle Database 11g Release 1 (11.1) [47] continues not only to supply what an ORDImage had in the previous release, but also offers more complete DICOM supports by providing a new type, called ORDDicom object type. Oracle Database 12c supports what have been

provided in Oracle Database 11 and provides improvements. It has DICOM protocol to allow DICOM applications and devices to easily access DICOM data stored in Oracle Database. The DICOM content now can be stored or managed as a part of a clinical workflow. It can also be stored in and accessed from Oracle WebCenter Content to simplify the development and management of applications [48].

Oracle Real Application Clusters (Oracle RAC) 10g [49], 11g [50] and 12c [51] enable to store and manage DICOM data in a cluster environment. Oracle RAC is a cluster database implemented in Oracle Database File System to allow data to be distributed and replicated across a pool of databases that do not share hardware and software. It can provide the following features to OLTP applications: (1) high availability of data in case of failure (because of a replication mechanism applied across nodes); (2) high performance (due to using a distributed and parallel data processing environment); and (3) scalability and elasticity, i.e., a database (or a node) can be added to an existing cluster database to increase overall system capacity. These features enable to build a large-scale storage system of DICOM images.

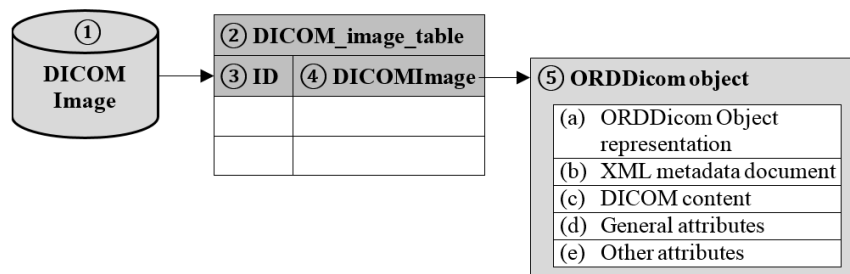


Figure 2.10: Sample DICOM Image Database using Oracle

Figure 2.10 depicts a sample DICOM image database containing a simple table used to store the DICOM content. Items in the figure includes: (1) *DICOM image database*; (2) *DICOM image table* created in the database; (3) *ID*; (4) *DICOMImage*; and (5) *DICOM content* stored in the ORDDicom object. At the high level, an ORDDicom object consists of five components [47]: (a) *ORDDicom Object representation*: an instance of the ORDDicom object contains attributes and methods, such as *makeAnonymous()*, *setProperties()*, *extractValue()* and so on, which are used to perform tasks on the ORDDicom object. (b) *XML metadata document*: the attributes are extracted from the DICOM content and stored in a XML metadata document. (c) *DICOM content*: the original DICOM content is stored within the database as a BLOB (binary large object) or stored in a local file system as a file accessed by using a pointer from the database. (d) *General attributes*: the attributes are frequently accessed such as *SOP Instance UID*, *SOP Class UID*, *Study Instance UID*, *Series Instance UID* and so on. (e) *Other attributes*: the attributes are used internally by Oracle.

Using the Oracle Multimedia feature for a medical image management system gives several advantages. First, it provides mechanisms to handle unstructured data along with structured data inside a relational database. Second, it overcomes the shortcomings of the PACSs because it can provide a modality-independent and vendor-neutral data storage. Finally, it allows users to write their own SQL queries (i.e., ad-hoc queries) to obtain information related to the entities *Patient*, *Study*, *Series*, etc.

Table 2.4: Example of statements to manipulate DICOM data in Oracle

SQL Statements	User Code in Oracle
CREATE TABLE DICOM_image_table	CREATE TABLE DICOM_image_table (ID integer primary key, DICOMImage ordsys.ORDDicom)
SELECT ID, PATIENT_NAME, PATIENT_ID, MODALITY FROM DICOM_image_table	SELECT ID, extractValue(t.dicom.metadata, '/DICOM_OBJECT/*[@name="Patient"s Name"]/VALUE', 'xmlns=http://xmlns.oracle.com/ord/dicom/metadata_1_0') as "PATIENT_NAME", extractValue(t.dicom.metadata, '/DICOM_OBJECT/*[@name="Patient ID"]', 'xmlns=http://xmlns.oracle.com/ord/dicom/metadata_1_0') as "PATIENT_ID", extractValue(t.dicom.metadata, '/DICOM_OBJECT/*[@name="Modality"]', 'xmlns=http://xmlns.oracle.com/ord/dicom/metadata_1_0') as "MODALITY" FROM DICOM_image_table

However, there exist some limitations when using Oracle Multimedia feature. First, Oracle supports standard ANSI SQL, but users have to write queries in a quite complex and unnatural way. Table 2.4 shows sample statements used to create a table and to select attributes from that table. Second, Oracle is a row-oriented RDBMS, thus it can offer high throughput for write-intensive (OLTP) workloads but is not optimized for read-oriented (OLAP) workloads. Third, although Oracle RAC aims to provide availability and performance, it still has limitations in dealing with the characteristics of DICOM data and workloads; due to having to satisfy the ACID properties (Atomicity, Consistency, Isolation and Durability), it does not provide sufficient solutions to increase high availability and query performance. Finally, it is also less scalable and elastic when compared with other databases that aim at handling Big Data (such as Cassandra [52] and MongoDB [53]). Implementing and scaling up a distributed and parallel data processing environment, e.g., by adding a new database to an existing Oracle cluster database, are costly and technically complex.

DCMDSM

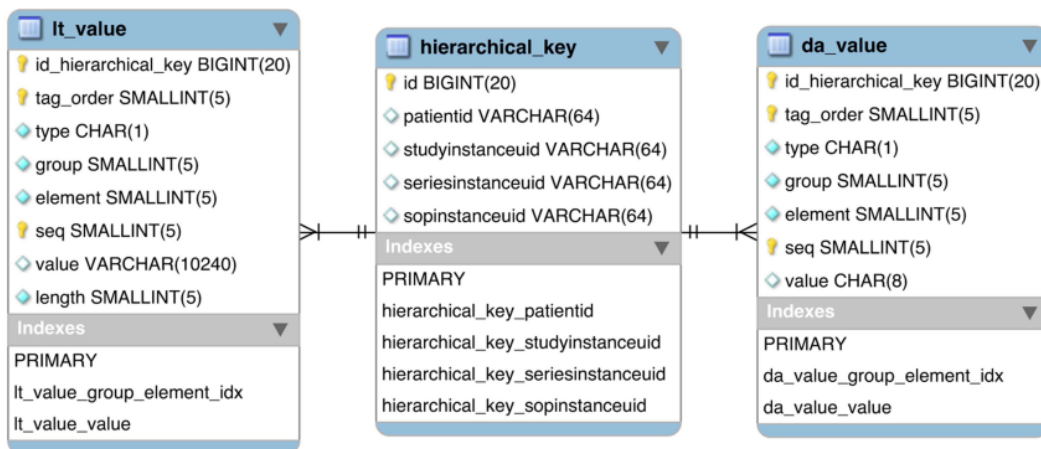


Figure 2.11: Database tables in the DCMDSM model [54]

DCMDSM (DICOM Decomposed Storage Model) [54] was aimed to handle the heterogeneity of DICOM data and to reduce the time required to answer queries/retrieval operations. To achieve these, the DCMDSM was designed based on the original decomposed storage model (DSM) [55]. In the DSM model, values of the same attribute have to be archived in exclusive tables, clustered by key and value; however, in the DCMDSM model, the attributes of the DICOM files are stored in different tables, according to their data types/domains. In particular, the DCMDSM model applies a vertical partitioning strategy that is based on Value Representations (VRs). The attributes from the DICOM files are extracted, parsed and stored in different tables according to their VRs. By this way, the DCMDSM model will create a single table per VR.

Another difference between the DCMDSM and DSM models is the number of attributes per table: while the DSM model uses binary tables, i.e., each table contains a surrogate key (surrogate attribute) and an attribute value, the DCMDSM model uses n-ary tables, with n varying according to each VR. Figure 2.11 presents database tables in the DCMDSM model. There is *hierarchical_key* table and multiple *VR-specific* tables, e.g., *lt_value* for *Long Text VR*, *da_value* for *Date VR* and so on. The *hierarchical_key* table is used to build a relationship between values of the attributes that belong to the same DICOM file. The surrogate key of each VR-specific table is used as a foreign key in the *hierarchical_key* table. Each record of the VR-specific table consists of attributes extracted from a DICOM file such as *tag order*, *type*, *group*, *element*, *value*, *length*, etc. Additionally, indexes can be created on one or a combination of the attributes.

The DCMDSM model brings some advantages. First, it can deal with the complex structure of DICOM data by using tables to represent DICOM data. Second, it can cope with heterogeneous/evolutive schemas of DICOM data. It allows new attributes to be added without significant modifications in the current database schemas: a new single database table is created per VR. Third, it can reduce storage space requirement since null values are removed from vertically partitioned tables. Finally, storing each attribute in a separable table makes it possible to reduce I/O bandwidth when a query accesses only a few DICOM attributes.

However, there are several disadvantages that should be considered before applying the DCMDSM model. First, the existing system has not validated for different workloads. In the cases of unpredictable workloads, the model may cause high CPU consumption for joining multiple small tables together. Second, the current model has not provided huge storage capacity, scalability and elasticity because it has been implemented on the top of a standard RDBMS using a single machine. Besides, the use of the RDBMS may suffer limitations in terms of query performance, storage capacity, scalability and elasticity as mentioned in the cases of PACSs, Oracle and eDiaMoND. Thus, the proposed model has not dealt with high and ever-increasing volume of DICOM data.

Document-based Database

To optimize size and performance of database, the authors in [40] proposed to use CouchDB, an open-source document-based database, for storing and querying DICOM data. In CouchDB database, every document is represented as a list of key-

value pairs without predefined schemas. The document format is self-describing and encoded using standard formats such as XML and JSON. Additionally, attributes stored in a document can be changed from one document to another. Therefore, such a document can be used to hold semi-structured data.

Figure 2.12 presents an example where DICOM data is extracted and stored in the CouchDB database. DICOM metadata is stored in a document; the values of attributes or data elements are stored in the document as key-value pairs. On the other hand, the relevant pixel data, i.e., binary data, can be stored as stand-alone or embedded attachments, i.e., the same way as attachments associated with e-mail. These attachments can be saved in different formats, e.g., DICOM and JPEG.

Queries in the CouchDB database (retrieval, aggregation, etc.) are performed in parallel on multiple machines by batch-oriented processing that is implemented by using the MapReduce programming model. JavaScript is used to implement the MapReduce to compute data represented as a collection of key-value pairs [40].

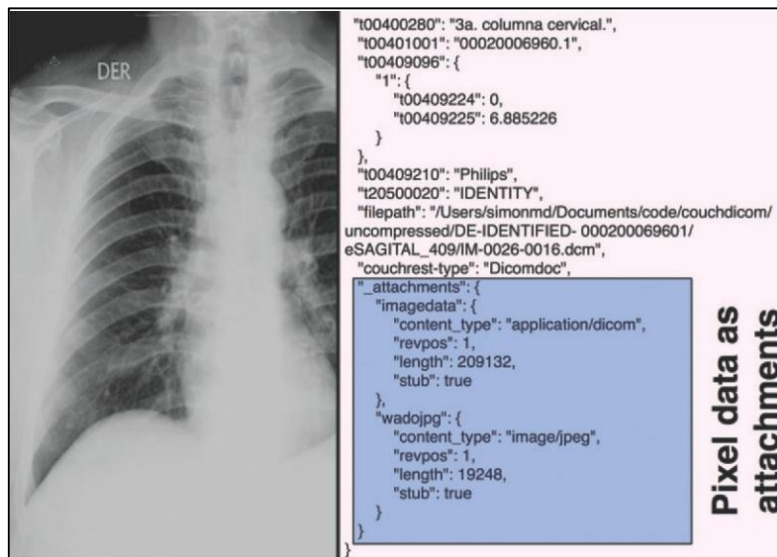


Figure 2.12: Example of DICOM data stored in CouchDB [40]

There are several advantages of using a document-based database such as the CouchDB database to manage DICOM data. Unlike RDBMSs, a document-based database can provide schema-free design, it thus is especially suited to archive the DICOM files which are subject to heterogeneous and evolutive schemas. All metadata is extracted from the DICOM files can be stored in the document-based database without loss of information. This database also reduces storage space demand because it can remove the need for storing null values. Besides, it offers features such as high performance, high availability, high reliability, high scalability and elasticity: the performance comes from easily scaling out the existing system while the availability and the reliability are obtained by replicating data across distributed machines. With these features, the document-based databases can solve issues of performance degradation caused from the rapidly growing volumes of DICOM data.

However, some challenges have come up when using a document-based database to manage DICOM data. It is generally based on the key-value store model, thus it

does not provide SQL support. Developers will find it hard to implement a common query language like SQL. It is not efficient to build and maintain structured-table database formats as well as relationships between tables according to the DICOM information model. Additionally, data denormalization (using a merged tables to reduce the number of join operations across multiple small tables) are usually applied in the document-based database to improve the query performance; however, this results in data redundancy and data inconsistency (updates may not performed to all related data stored in different locations), i.e., the ACID properties are not guaranteed.

Hybrid Cloud-enabled Storage System

B. Mohamad, L. d'Orazio and L. Gruenwald [56, 57] proposed a hybrid (row-column) cloud-enabled storage system for DICOM data management. To store DICOM data into the system, first of all, the authors proposed to classify DICOM attributes into three categories: (1) *Mandatory attributes*; (2) *Frequently-accessed-together attributes*; and (3) *Optional/private/seldom-accessed attributes*. Next, the attributes are manually grouped together into column groups according to these categories. Finally, a suitable data layout is chosen to store each column group. For simplicity, we use terms “row table” and “column table” to refer to a table being stored in a row and a column store, respectively. The selection of data layouts for the column groups is described as follows:

- Attributes that belong to the first two categories (i.e., mandatory and frequently-accessed-together) are grouped together and stored in row tables. This strategy aims at reducing tuple reconstruction cost.
- Attributes that belong to the last category (i.e., optional/private/seldom-accessed) are stored in column tables. The aim of this strategy is to save I/O cost if only few attributes are required by a query.

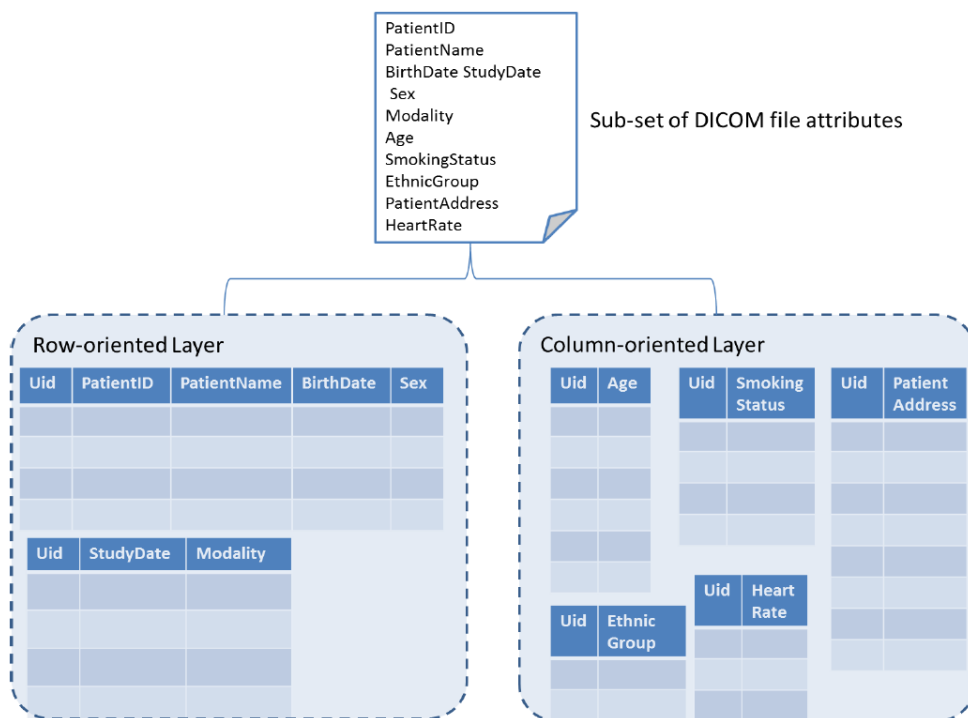


Figure 2.13: DICOM attributes stored over row- and column-oriented layers [57]

Figure 2.13 illustrates the storing of a subset of attributes in row- and column-oriented layers of row and column stores. The mandatory attributes including *PatientID*, *PatientNameBirthDate*, and *Sex* and the frequently-accessed-together attributes consisting of *StudyDate* and *Modality* are stored in row tables. On the other hands, the optional/private/seldom-accessed attributes including *Age*, *SmokingStatus*, *PatientAddress*, *EthnicGroup* and *HeartRate* are stored in a column table.

The above grouping is non-overlapping; that is, each attribute belongs to only one column group (or vertical partition). However, the attribute *UID* is needed in every column group because it will be used to join the corresponding tables storing these column groups together. Besides, null-rows will be deleted from the vertical partitions to save storage space.

Furthermore, the hybrid (row-column) cloud-enabled storage system was implemented using a distributed mediator, as shown in Figure 2.14. The Oracle and MonetDB were used as row and column stores, respectively. The mediator will control the query processing across the storage engines; it routes a SQL user query to be executed on the most suitable storage engine.

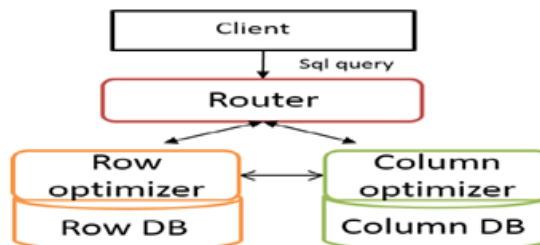


Figure 2.14: Distributed Mediator [57]

The hybrid cloud-enabled storage system can deal with the high complexity and the high variety (heterogeneous/evolutive schemas) of DICOM data. The system provides normalized data, SQL and (ad-hoc) multiple-table join queries. I/Os and tuple reconstruction cost are decreased. However, the proposed system exists some disadvantages. First, grouping of the attributes into the column groups and selecting of suitable data layouts for them are manually performed. In this thesis, we call this approach *expert-based*. Hence, it is difficult and tedious for experts (e.g., database designers) to do this work in such a way, especially when the number of attributes is very large. Second, the query performance is limited because the distributed mediator has to decide the most suitable storage engine to perform a query, and data needs to be moved between storage engines for the query execution. Finally, the system is hard to scale-up (i.e., adding a new node to the current system), thus it is not well suited for the high and ever-increasing volume of DICOM data.

2.3.3 Conclusion

The expected requirements listed in Section 2.3.1 are used as criteria of comparison of the existing DICOM data management systems. Table 2.5 shows the comparison result. In general, the systems using relational databases, including PACSs, eDiaMoND and Oracle/Oracle RAC, can deal with high complexity of data, create normalized data, and provide SQL and join supports. However, they lacks solutions

to: (1) organize data based on both workload and data-specific information to reduce storage space demand (sparseness) and to improve performance of queries in OLTP and OLAP workloads; (2) provide an efficient query processing strategy; and (3) provide huge storage capacity, scalability and elasticity.

Table 2.5: Comparison of the existing systems

Existing DICOM data management systems	Expected Requirements		
	R1	R2	R3
PACSs	0	-	-
eDiaMoND	+	-	-
Oracle/Oracle RAC	+	0	-
DCMDSM	+	0	-
Document-based Database	+	-	0
Hybrid Cloud-enabled Storage System	+	+	0

+ Featured supported, 0 partial, - not supported

The DCMDSM model can help to improve OLAP queries and reduce storage space demand due to depending on the DSM model. Nevertheless, execution cost of OLTP queries may be high because of multi-table joins. Moreover, the existing system was designed and validated using a single machine, thus may has limitations at query performance, storage capacity, scalability and elasticity.

The document-based database and hybrid cloud-enabled storage system have many features that can cope with the characteristics of DICOM data and workloads. The document-based database is a NoSQL database designed to handle Big Data, thus it can deal with the high variety of DICOM data and provide high query performance, huge storage capacity, scalability and elasticity in nature. On the other hand, hybrid cloud-enabled storage system provided solutions depending on both workload and data-specific information to organize and store DICOM data in a manner to improve workload performance and to reduce storage space demand. However, both these systems lacks the following features:

- An automated design approach that uses both workload and data-specific information to design and store DICOM data in a way to reduce both workload execution time and storage space demand.
- Efficient solutions for query processing over large-scale datasets, especially, to reduce network I/Os in a distributed query processing environment.

2.4 Summary and Conclusion

DICOM data has caused challenges in data management due to the characteristics of DICOM data and workloads. Several data management systems have been proposed for storing and querying this data. With regards to the data storage model, the main classifications of databases used in the existing systems include row-oriented database, vertically-decomposed row-oriented database, NoSQL document-based database and hybrid cloud-enabled storage system. They have their own strengths and weaknesses.

Therefore, the main goals of our study are to propose efficient methods for storing and querying DICOM data that will be applied to build a new DICOM data management system. To fill the gaps in the existing systems, the new DICOM data management system needs to meet the following expected requirements: (R1) *Flexible data* (dealing with high complexity, variety and high and ever-growing volume of data and providing normalized data); (R2) *Flexible querying* (supporting SQL ad-hoc queries with joins); and (R3) *Efficiency of storage and CPU* (based on both workload and data-specific information to organize and store data in a manner that reduces both storage space demand and execution time of queries in mixed OLTP and OLAP workloads; providing solutions for efficient query processing over on large-scale datasets; providing huge storage capacity, scalability and elasticity).

The document-based database and hybrid cloud-enabled storage system have showed many features that are able to satisfy the above requirements. However, they still lack an automated design approach that is able to use both workload- and data-specific information to organize and store DICOM data in a manner to reduce both storage space demand and workload execution time. In addition, they lack efficient solutions for query processing over large-scale datasets, especially in a distributed query processing environment.

Key Points
<ul style="list-style-type: none">• We gave an overview of background information on DICOM standard.• We determined the characteristics of DICOM data and workloads that may cause challenges in DICOM data management.• We reviewed the existing DICOM data management systems and discuss their strengths and weaknesses.• We conclude with the limitations of the existing systems with respect to the expected requirements for a new DICOM data management system.

Databases and Related Techniques

3.1 Overview

An overview of this chapter is shown in Table 3.1.

Table 3.1: Overview over Chapter 3

3.2 Classifications	
3.2.1 OLTP and OLAP Workloads	3.2.2 Relational Databases
3.2.3 NoSQL Databases	3.2.4 NewSQL Databases
3.3 Cluster Computing Frameworks	
3.3.1 MapReduce	3.3.2 Spark
3.4 Data Layouts	
3.4.1 Row-oriented Storage Model	3.4.2 Column-oriented Storage Model
3.4.3 Hybrid Storage Models	
3.5 Vertical Partitioning and Bloom Filter Techniques	
3.5.1 Vertical Partitioning	3.5.2 Bloom Filter and Intersection Bloom Filter
3.6 Key Components of the New System	
3.6.1 Data Model	3.6.2 Data Storage Model
3.6.3 Data Schema	3.6.4 Query Processing
3.7 Summary and Conclusion	

We first present backgrounds of different workload types including OLTP and OLAP. Next, we provide comprehensive backgrounds of the most prevalent databases used for Big Data, including relational, NoSQL and NewSQL databases. We elucidate about their advantages and disadvantages. Then, we review common cluster computing frameworks including MapReduce and Spark. The former is based on batch-oriented processing while the latter is regarded as a low-latency version of the MapReduce and popularly used for interactive ad-hoc query and analysis. After that, we present backgrounds on data layouts. Following that, we concentrate on the vertical partitioning techniques that are applied to reduce storage space for the relational databases (especially for sparse datasets). We present Bloom filter (BF) and Intersection Bloom filter (IBF) techniques that can be applied to improve query performance in distributed query processing environments. Next, we discuss about key components of a new DICOM data management system. Finally, we summarize and conclude the chapter by selecting solutions for these key components.

3.2 Classifications

3.2.1 OLTP and OLAP Workloads

OLTP Workloads

OLTP (Online Transaction Processing) is a computer technology term referring to systems that facilitate and efficiently support transaction-oriented applications where the most frequently-used operations are to insert, delete, update or retrieve all (or most) of columns of a table, e.g., to return all information about a specific patient.

The OLTP systems require very fast query processing and maintain data integrity in multi-access environments. Their databases should optimize write operations; besides, they need to support data normalization that minimizes data redundancy and thus improves performance of the write operations. Row-oriented databases are primarily designed for OLTP applications.

OLAP Workloads

OLAP (Online Analytical Processing) is a computer technology term referring to systems that support for analytical applications, which typically focus on analyzing data in their database. In these systems, data is seldom updated, but it is frequently read and aggregated. In other words, OLAP workloads consist of read-intensive queries that need to access or aggregate over many rows but only for a few columns.

Databases should optimize read and aggregation operations. Column-oriented databases are read-optimized, and are thus usually used for the OLAP applications.

3.2.2 Relational Databases

Nowadays, the most popular databases are *Relational Databases* which have implemented the *relational data model* proposed by E. F. Codd in 1970 [58]. This data model was originally designed for structured data and predefined schemas. A *schema* is a logical database design. A relation is used to hold information about entities in the real world. A relation and a relationship among relations are represented as a table made up of *rows* and *column*. Each row represents a *tuple* (record) which describes a single element of the entity while each column represents an *attribute* (field) of that entity. A *relation instance* is a set of rows, each of which conforms to the schema of the corresponding relation. Figure 3.1 illustrates a table storing a relation instance of the relation *Patient* with the following attributes: *PatientID*, *PatientName*, *PatientBirthDate*, *PatientSex* and *EthnicGroup*.

PatientID	PatientName	PatientBirthDate	PatientSex	EthnicGroup
P40028	Smith	19610712	F	Whites
P40029	Muller	19500101	M	Whites
P40030	Young	19700509	M	Asians
P40031	Carol	19900122		
P40032	Garcia	19990515		Blacks

Figure 3.1: Relation instance of the relation *Patient*

3.2.3 NoSQL Databases

However, with the explosion of Big Data, the relational data model finds it difficult to handle semi/unstructured data. There is a trend moving towards NoSQL databases (Not Only SQL database). A NoSQL database is any database whose organization is not based on the relational data model. The NoSQL databases are not a replacement for the RDBMSs, but they are able to fill the gaps of the RDBMSs because they have been built to handle unstructured data and to provide horizontal scalability and high availability with low administrative cost.

Key-value store:

```
"key": "1" value {"PatientName": "Marry" "PatientAge": "41" "PregnancyStatus": "4" "ReferringPhysicianName": "Anthony"}
"key": "2" value {"PatientName": "Paul" "PatientAge": "47" "ReferringPhysicianName": "Lisa" "PerformingPhysicianName": "Carol"}
```

Column-family store:

```
"row": "1", "PatientName" "PatientAge" "PregnancyStatus" "PerformingPhysicianName"
      "Marry"      "41"      "4"      "Anthony"
"row": "2", "PatientName" "PatientAge" "ReferringPhysicianName" "PerformingPhysicianName"
      "Paul"      "47"      "Lisa"      "Carol"
```

Document store:

```
"id": "1" "PatientName": "Marry" "PatientAge": "41" "PregnancyStatus": "4" "ReferringPhysicianName": "Anthony"
"id": "2" "PatientName": "Paul" "PatientAge": "47" "ReferringPhysicianName": "Lisa" "PerformingPhysicianName": "Carol"
```

Graph database:

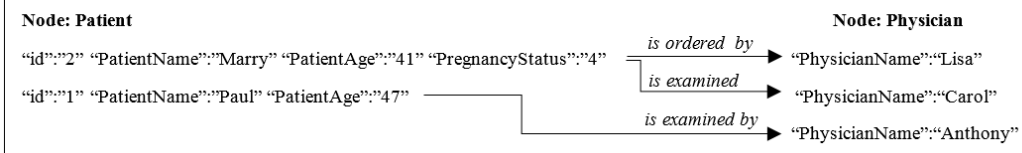


Figure 3.2: Examples of NoSQL databases

NoSQL databases are usually classified into four categories: *key-value stores*, *column-family stores*, *document stores* and *graph databases* [59]. Figure 3.2 gives examples of the NoSQL databases storing the same set of frequently-accessed-together attributes that provide information about patients and physicians involving in the patients' studies. These attributes include *PatientName*, *PatientAge*, *PregnancyStatus*, *Referring-PhysicianName* and *PerformingPhysicianName*. With the use of self-describing structures, these databases can represent only non-null values of tuples.

- **Key-value stores:** Key-value stores represent data as a set of key-value pairs such that values are indexed by keys. The key-value model is the most flexible NoSQL model for modeling data, rapidly changing data structure because it does not enforce any structure on data (e.g., tables). It is also very efficient for storing distributed data and retrieving information by keys, and facilitates for decomposition and replication of data to provide high scalability and scalability. However, a key-value store is not a good choice for applications that require fixed-structured data or multiple-key transactions cross-document operations. Amazon's Dynamo [60] and LinkedIn's Voldemort [61] are using this data model.

- **Column-family stores:** Column-family stores extend the key-value model by re-representing data in forms of table-like data structures. However, unlike strictly structured tables in relational databases, to be able to deal with sparse columns and no fixed schema, the column-family stores are based on a flexible data model: each row consists of a set of columns, where each single column contains a key-value pair; the key is the column name; the value may have an arbitrary data type such as an integer, a string, JSON document or a binary image; additionally, each row may have a different number of columns. Some column-family stores have been commonly used such as Google Bigtable [62], HBase [63] and Cassandra [52].
- **Document stores:** Document stores are regarded as a variation of the key-value stores. They store a set of documents, typically encoded using a standard format such as XML, JSON, BSON or others. For instance, MongoDB [53] uses BSON format while CouchDB [64] applies JSON format. Documents may have different formats. By this way, each document can have a complex format, e.g., containing nested objects inside it, in order to be able to support for efficiently storing semi-structured documents (e.g., email messages). Besides, the document stores allow to create primary indexes on keys and secondary indexes on contents (instead of only on keys as in the case of key-value stores) such that they can provide fully searching either by keys or values. However, similarly to the key-value stores, they are not efficiently used for cross-document transactions.
- **Graph databases:** Graph databases use graph structures as their data model where nodes represent entities, and edges represent relationships among the entities. These entities and relationships are described by key-value pairs. The graph databases are efficiently used for handling the interconnections among different entities because they can apply well-studied graph algorithms to explore relationships among their data [59]. Some graph databases have been commonly known such as Neo4J, DEX Infinite Graph, Infogrid, HyperGraphDB, Trinity, Titan and Allegro Graph [65-67].

3.2.4 NewSQL Databases

NewSQL databases are regarded as modern relational database management systems. They are based on the relational data model, but are able to provide horizontal scalability and high performance as NoSQL databases while still ensuring the traditional ACID guarantees of relational databases. A noticeable characteristic of the NewSQL databases is that although they can use different physical storage layouts (e.g., key-value stores and column-oriented stores), they still provide users with the relational schemas (i.e., tables or relations) and SQL as main mechanisms to interact with any application. They also allow the users to create relationships between tables [68]. Additionally, they can apply shared-nothing architectures of cloud computing to offer horizontal scalability. Some NewSQL databases have commonly known such as VoltDB [69], Clustrix [70], NuoDB [71], Google Spanner [68].

3.3 Cluster Computing Frameworks

There are several large-scale data processing techniques in order to deal with a variety of workloads: (1) *Batch-oriented processing*: processing recurring tasks such data mining or aggregation over very large datasets; (2) *Stream processing*: processing data streams arriving continuously at real time; (3) *OLTP*: processing transactions using NoSQL databases; (4) *Interactive ad-hoc query and analysis*: processing ad-hoc queries and analyses with user interaction; and (5) *Search over semi-structured data items and documents*: retrieving information that satisfies users' need from a high volume of structured and semi/unstructured data [26, 72]. Because our study is scoped to focus on the interactive, ad hoc query and analysis technique (mentioned in Section 1.4 in Chapter 1), we concentrate an in-memory cluster computing framework called on Spark that is able to provide high performance for interactive workloads. Besides, Spark has been developed in order to avoid high latency of MapReduce, a successful batch-oriented programming model, thus we also present backgrounds of MapReduce.

3.3.1 MapReduce

The batch-oriented processing technique processes a high volume of data by splitting a job into multiple tasks which are performed in parallel on multiple nodes (machines). The typical stages of a batch job include split, sort and merge.

MapReduce, originally introduced by Google, has been a successful batch-oriented programming model for recurring tasks such data mining or aggregation over very large datasets on large clusters of commodity nodes [73]. In order to facilitate the development of programs, the MapReduce operates on the top of a distributed file system (DFS) such as Google File System (GFS) or Apache Hadoop Distributed File System (HDFS). MapReduce run-time environments (e.g., Hadoop) are responsible for tasks, including data partitioning, replication, job scheduling and communication between nodes in the cluster such that developers do not have to care about these tasks. In this programming model, a MapReduce job execution plan is divided into two main phases, namely *Map* and *Reduce*, whose computation is expressed employing two user-defined functions: *Map()* and *Reduce()*. Besides, there is a hidden phase between these two phases, called *Shuffle and Sort*, which is also regarded as the first step of the Reduce phase. The input and output formats of these phases are depicted in Table 3.2.

Table 3.2: Input and output formats of the phases in MapReduce

Phase/Step	Input	Output
Map	$(k1, v1)$	List($k2, v2$)
Shuffle and Sort	List($k2, v2$)	$(k2, \text{List}(v2))$
Reduce	$(k2, \text{List}(v2))$	List($k3, v3$)

Map: When a MapReduce job is sent to the MapReduce run-time environment, *Mappers* (also known as *Map tasks*) are started in parallel on nodes in the cluster. Each Mapper reads key-value pairs, $(k1, v1)$, from DFS and applies the Map function to transform them into a list of intermediate key-value pairs, List($k2, v2$), where each key may have multiple values. Intermediate results are stored in the local file system, where the Mappers are running.

Shuffle and Sort: Each Reducer task will transparently start the Shuffle and Sort step as its first step. All the intermediate results from all the Mappers are grouped by key and are split among the Reducers; each Reducer takes all the values associated with the same key. After all the data of the Mappers are sent (shuffled) to the nodes of the Reducers (in their local file system), the key-value pairs are merged and sorted into a larger list of key-value pairs. Next, this list is grouped by key to generate a new list of key-value pairs, $(k2, List(v2))$, where all the key-value pairs sharing a common key are grouped into a single key-value pair. In addition, the resulting key-value pairs are buffered as r local files, where r is the number of Reducers.

Reduce: When all actions in the Shuffle and Sort step complete, the Reducers load the key-value pairs from the local output files in parallel. Each Reducer applies the computation defined in the Reduce function to the values having the same key and generates a new list of key-value pairs, $(List(k3, v3))$. Finally, the results of all the Reduce tasks are written back to DFS and used as the job result.

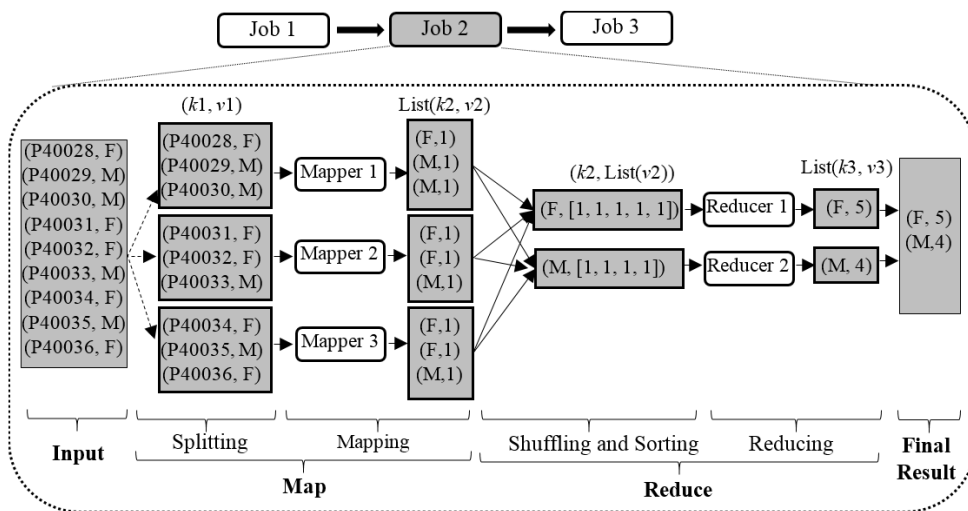


Figure 3.3: A job that counts the number of patients by sex using MapReduce

When a SQL query execution engine is built on top of the MapReduce (e.g., Pig [74] and Hive [75]), to execute a SQL query, its query optimizer generates a query execution plan consisting of a set of one or more MapReduce jobs. The parallelism only occurs within each job. Figure 3.3 describes an example of the computation of a SQL query: the query is transformed into a sequence of three sub-queries executed by three MapReduce jobs 1, 2 and 3. Assume that job 2 is used to compute the sub-query `SELECT COUNT(*) FROM Patient GROUP BY Sex` (to count the number of patients by sex). We also assume that MapReduce environment is using 3 Mappers and 2 Reducers. Here, the input table *Patient* contains only two columns, *PatientID* and *Sex*, with 9 tuples. It is split into 3 splits, each of which contains 3 tuples.

- In the Map phase, each Mapper receives each line in the split (assigned for it) as a key-value pair (*Patient ID*, *Sex*), such as (P40028, F), (P40029, M), etc., and respectively outputs a corresponding intermediate key-value pair (*Sex*, 1), such as (F, 1), (M, 1), etc., where each occurrence of either F or M will be counted as 1. By this way, each Mapper will output a list of intermediate key-value pairs `List(Sex, 1)` for its input data, e.g., Mapper 1 produces (F, 1), (M, 1) and (M, 1).

- In the Reduce phase, first of all, the Shuffle and Sort step is performed as follows: key-value pairs having the same value of the key *Sex* will be sent to the same Reducer; then they are merged, sorted and grouped by this key. For instance, after this step, Reducer 1 obtains one key-value pair (F, [1, 1, 1, 1, 1]). As soon as the Shuffle and Sort step finishes, each Reducer loads the key-value pairs from its local output file, computes the sum of the values of the same key *Sex*, and generates a new list of key-value pairs $List(Sex, Sum)$, e.g., the Reducer 1 loads the key-value pair (F, [1, 1, 1, 1, 1]), computes the sum and generates the pair (F, 5). Finally, the results of all the Reducers are written back to DFS.

MapReduce provides a suitable solution for parallel processing of large-scale data because it increases the locality of data and processing at the nodes where the data is kept. Besides, this programming model is simple since its parallel data processing approach is mainly based on two phases Map and Reduce. However, the execution of each MapReduce job needs to replicate data for local computation at the nodes and has to perform a lot of reads and writes for sharing data across the phases. As a consequence, data replication, disk I/Os and network latency will cause a lot of delays in the architecture of the MapReduce.

Pig [74] and Hive [75] are two software frameworks that facilitate querying and managing Big Data. Both of them provide SQL-like languages, i.e., Pig Latin and Hive QL, respectively. Pig's engine excels at processing complex data flows in parallel, whereas Hive's engine is more suited for Big Data analytics applications, e.g., data summarization and analysis. Their compilers will produce sequences of Map-Reduce programs running in parallel on Hadoop clusters. However, both Pig and Hive are dependent on the Hadoop and MapReduce executions; thus, their queries may have delay time in data processing in HDFS. This implies that they may not be suitable for Big Data analytics applications that need rapid response times.

3.3.2 Spark

The interactive ad-hoc query and analysis technique refers to processes designed to use current data for answering single specific questions or domain specific analyses whose results are analytic reports, statistical models, or other forms of data summarization. These processes are often done through interactions between humans and computer systems. Therefore, they need low-latency so that users can directly perform ad-hoc queries and analyses and can react to current circumstances. Although traditional OLAP systems have supported for these requirements, how to provide fast query response times on any huge business data is still a big challenge.

Batch-oriented processing model of MapReduce is not well suited for the interactive ad-hoc queries and analyses due to its high latency. In recent years, some innovation systems have been proposed for performing interactive ad-hoc analyses at scale such as Apache Drill [26], Hive on LLAP (Live Long and Process) [76], BigQuery [77], CitusDB [78] Hadapt [79], HAWQ [80], Impala [81], Phoenix [82] and Spark [83]. These systems provide low-latency queries, user queries written in a human-readable syntax (e.g. SQL), NoSQL stores (e.g. HDFS) and data presented in tabular or nested form. Below, we give more information about Spark.

Spark [21, 83] is an in-memory cluster computing system which can run on Hadoop and is usually referred to as a low-latency version of the MapReduce. To reduce latency which is caused by data replication and disk I/O operations performed across steps in MapReduce phases, Spark tries to keep the intermediate data in memory as much as physically possible to reduce the need to write the data to disks.

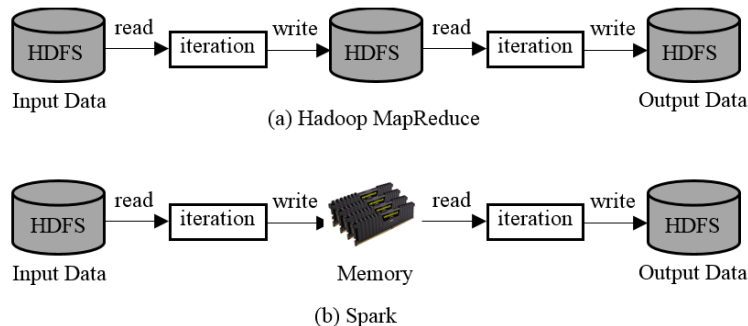


Figure 3.4: Comparison between Hadoop MapReduce and Spark

Figure 3.4 presents a comparison between Hadoop MapReduce and Spark: while the Hadoop MapReduce (Figure 3.3(a)) incurs a high I/O cost (reads and writes) during the query execution, Spark (Figure 3.3(b)) reduces this cost by trying to keep the data in memory. To achieve this, Spark provides a new storage primitive called resilient distributed datasets (RDDs) so that developers can use them to store the data that needs to be processed. Spark then attempts to keep these RDDs in the memory and controls their partitioning to optimize data placement across all nodes in the cluster. Another benefit of the RDDs is its ability to fault tolerance without requiring replication through a notion of lineage: if a partition of an RDD is lost, the RDD will track how to re-compute just that partition from base data on disk. This helps Spark runs faster than other typical distributed systems such as the MapReduce. Besides, DataFrames in Spark allow users to represent data in form of tables. Spark enables querying the data using a SQL-like language integrated with MapReduce-based computations [21].

3.4 Data Layouts

3.4.1 Row-oriented Storage Model

PAGE HEADER	P40028	Smith	19610712	F	Whites
P40029	Muller	19500101	F	Whites	P40030
19700509	M	Asians	P40031	Carol	19900122
P40032	Garcia	19990515	Blacks		

Figure 3.5: NSM layout of the relation *Patient*

Row-oriented storage model (N-ary Storage Model or NSM) has been used in traditional RDBMSs (such as Oracle, DB2, SQL Server, etc.). In this model, all attributes of the same tuple are stored consecutively on disk. Figure 3.5 presents the row-oriented storage model corresponding to the relation *Patient* given in Figure 3.1.

This model is write-optimized, thus it is efficiently used for OLTP workloads. This advantage is achieved because, co-locating the attributes of the same tuple leads to better cache locality. The entire tuple can be read or written with a single disk seek. Besides, the tuple reconstruction cost is also low.

The row-oriented storage model has been widely applied to represent data due to its simplicity to implement horizontal schemas of relational tables. This solution is known as *horizontal representation* [14, 84]. However, the horizontal representation is not well-suited for handling the variety of data. For instance, if data is sparse, storing a large number of null values in a table will cause waste of storage space. The row-oriented storage model is also inefficient when used for OLAP workloads because if only a few attributes of a table are required by a query, the entire table still needs to be read into memory from disk before any projections are performed. This causes a lot of redundant attribute accesses and thus degrades query performance.

3.4.2 Column-oriented Storage Model

In contrast to the row-oriented storage model, column-oriented storage model has been applied in column-oriented RDBMs such as MonetDB [18] and C-Store [19]. This model is built based the Decomposed Storage Model (DSM) [55] where a n-ary table of an horizontal representation (i.e., horizontal table) is vertically decomposed into *n* binary tables, each of which has two columns: *surrogate (sur)* and *attribute*.

Sur	PatientID	Sur	PatientName	Sur	PatientBirthDate	Sur	PatientSex	Sur	EthnicGroup
1	P40028	1	Smith	1	19610712	1	F	1	Whites
2	P40029	2	Muller	2	19500101	2	M	2	Whites
3	P40030	3	Young	3	19700509	3	M	3	Asians
4	P40031	4	Carol	4	19900122			5	Blacks
5	P40032	5	Garcia	5	19990515				

Figure 3.6: DSM layout of the relation *Patient*

Figure 3.6 gives an example of the DSM model corresponding to the relation *Patient* given in Figure 3.1. Here, the horizontal table of the relation *Patient* is divided into five separate binary tables, where only non-null values of the attributes are stored. The use of the surrogate enables values of different attributes (having the same surrogate value) to be tied together to reconstruct the original tuple. Figure 3.7 presents the physical stores corresponding to the above the DSM layout of the relation *Patient*.

PAGE HEADER					PAGE HEADER										
1	P40028	2	P40029	3	P40030	4	P40031	1	Smith	2	Muller	3	Young	4	Carol
5	P40031							5	Garcia						

PAGE HEADER			PAGE HEADER			PAGE HEADER											
1	19610712	2	19500101	3	19700509	1	F	2	M	3	M	1	Whites	2	Whites	3	Asians
4	19900122	5	19990515					5	Blacks								

Figure 3.7: Physical representation of the DSM layout of the relation *Patient*

In general, the column-oriented storage model is read-optimized because it enables to read only the required columns while the rest of columns are ignored. This reduces disk I/Os during the query execution. Thus, it is well-suited for analytic applications (OLAP workloads). However, this model has high cost for writing or reading a complete tuple: writing a new tuple requires updating each of the columns of that tuple; reading a complete tuple requires locating the correct value from each column of that tuple in order to reconstruct the original tuple format. As a result, this model performs full tuple operations more slowly than the row-oriented storage model [85].

3.4.3 Hybrid Storage Models

The data storage models presented in the previous sections are optimized for either an OLTP or an OLAP workload, but not both. Therefore, if an application is involving to a mixed OLTP and OLAP workload, system performance requirement is hard to be satisfied. To overcome this limitation, several hybrid storage models have been introduced. In this section, we present the following models: column-group storage models, Mirror and Fractured Mirrors [86], HyPer [87], Trojan Columns [13] and SAP HANA database [20].

Column-Group Storage Models

Column-group storage models are regarded as hybrid storage models because they are built by organizing column groups in a row-oriented storage layout, a column-oriented storage layout or both of them in order to efficiently process mixed workloads. In this section, we present such storage models including Partition Attributes Across (PAX) [88], Data Morphing [10], and HYRISE [12].

Some researches [88-90] have shown that the performance of modern database systems are impacted not only by the number of disk I/O operations but also by the delays related to their processors (CPUs). Since cost of main memory is decreasing, there has been a trend that the modern database systems attempt to keep a large amount of data in main memory to reduce I/Os between disk and main memory [91]. However, in this way, the performance bottleneck is transferred to the access latency between the processor and the main memory [92]. To reduce this bottleneck, the modern database systems have used a cache memory (that is small, fast but expensive) between the processor and the main memory to supplement for the workings of the processor [90]. If required data is already cached, the overall speed of processing data will increase, otherwise the cache misses will cause the processor to request the required data from the slower main memory. Besides, loading useless data into the cache causes waste of bandwidth and leads to the need of replacing the current data with the relevant data in the future. Therefore, to speed up the data processing, the frequently-used data should be stored in cache to reduce the cache misses.

Partition Attributes Across: *Partition Attributes Across* (PAX) [88] was introduced as a new storage model to overcome the problem of low cache utilization in the DSM model. To achieve this, the PAX model modifies the data organization within each disk page of the NSM model. Similarly to the NSM model, the PAX model also proposed that all attribute values of the same tuple will be stored in the same disk page (i.e., logical block) as in a normal row store. However, unlike the NSM model,

now the PAX model decomposes a disk page into multiple mini-pages, and then groups all values of a particular attribute together on the same mini-page. Figure 3.8 depicts a disk page of the PAX layout used to store the relation *Patient* given in Figure 3.1. Here, the PAX model divides a disk page into five mini-pages, each of which contains only the values of a particular attribute.

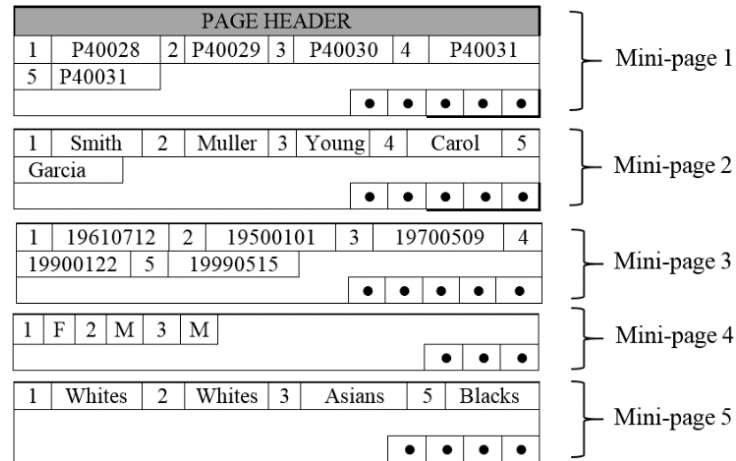


Figure 3.8: A disk page of PAX layout of the relation *Patient*

The PAX model has several advantages and disadvantages. It can fully utilize the cache because only the values of required attributes are loaded into cache from main memory. In addition, tuple reconstruction cost of the PAX model is negligible because only tuples within a disk page need to be reconstructed; this cost is expensive in the case of the DSM model. Therefore, the PAX model combines the advantages of both the NSM and DSM models. Unfortunately, this advantage will be lost, if many columns from a table need to be accessed together to answer a query. Scanning many columns from a table will cause more cache misses because the PAX model has to jump from one column to another in memory. Hence, a decision to use the PAX model should be based on attribute usage.

Data Morphing: *Data Morphing* [10] is considered as the first approach that was proposed to group the frequently-accessed-together attributes and then keep them together in the same place in a data storage. Similarly to the PAX model, the main focus of the Data Morphing model is to increase the CPU cache performance. Furthermore, it extended the PAX model to achieve a more flexible storage model. A disk page of the Data Morphing model is decomposed into zones instead of mini-pages. Each zone stores the values of the same attribute group of the relation. For example, Figure 3.9 depicts a disk page with four zones of Data Morphing layout corresponding to the relation *Patient* given in Figure 3.1. Here, we assume that two attributes *Patient Name* and *Patient Birth Date* are frequently accessed together, so their values are kept in the same zone 2.

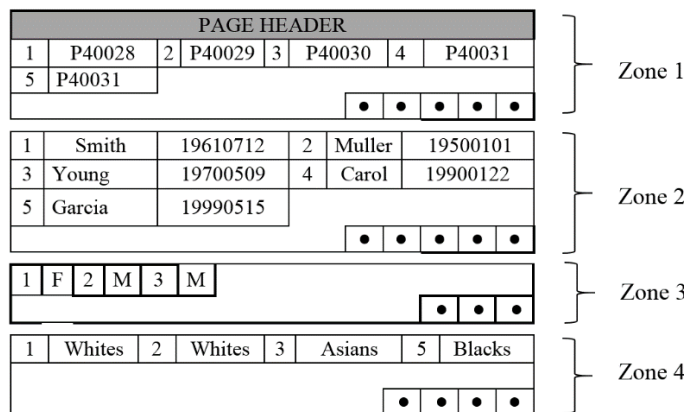


Figure 3.9: A disk page of Data Morphing layout of the relation *Patient*

Besides the above data organization, the Data Morphing model proposed a hill-climbing algorithm to determine the optimal attribute groups depending on a given workload. The Data Morphing model is able to reduce cache misses when performing queries accessing to only a few or multiple attributes. However, the proposed hill-climbing algorithm has an exponential time complexity, with respect to the number of attributes, thus it does not scale to large relations, for example, with hundreds of attributes.

HYRISE: *HYRISE* [12] is a main-memory hybrid database system. To achieve high performance in a mixed workload environment, it provides an automated database design tool to automatically partition a table into multiple vertical partitions (or column groups) with varying widths depending on attribute access patterns. In particular, for OLAP queries, the tool prefers to suggest narrow partitions because such queries frequently access just a few columns of a table. In contrast, for OLTP queries, wide vertical partitions are more efficient to reduce cache misses than narrow ones because these queries usually access all (or most) columns of a table. The *HYRISE* is referred to as an in-memory column-oriented database system since it creates vertical partitions, each of which is composed of frequently-accessed-together attributes and represented by a data structure, called container that is allocated in main memory.

The main improvements of the *HYRISE* over the Data Morphing model are as follows: The cache-miss model of the *HYRISE* is able to capture several additional key concepts such as partial projections, data alignment and query plans [12] (which were missed in the Data Morphing model), thus it can accurately estimate the number of cache misses incurred in a particular partitioning with respect to the attribute access patterns. This helps the *HYRISE* achieve significantly better query performance than the Data Morphing model. Besides, the grouping and pruning algorithms proposed in the *HYRISE* are able to scale to tables with hundreds of columns (the Data Morphing model cannot scale to wide tables). Nevertheless, the disadvantage of the *HYRISE* is that it is a main-memory database system, so it may be suitable for storing small databases, whose size should be smaller than the amount of the physical available memory. This limitation may make the system have performance problems and less efficient when used to handle high and ever-growing volume of data (as DICOM data).

Mirrors and Fractured Mirrors

The *mirrors* and *fractured mirrors* approaches [86] try to store data using both the NSM and the DSM layouts to retain their own advantages for mixed workloads.

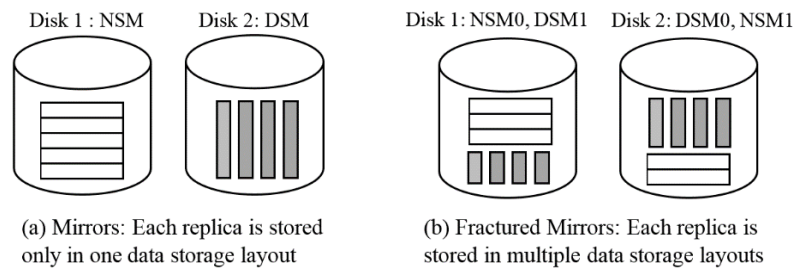


Figure 3.10: Mirrors and fractured mirrors [86]

In the *mirrors* approach, two disks are used. The (original) data is completely replicated into two copies (replicas): one uses the NSM layout while the other uses the DSM layout; each copy is put on one disk, as illustrated in Figure 3.10(a). The query optimizer will redirect each query to its preferred data layout. This approach is simple but exists two limitations: First, if the workload is skewed towards one of the two data layouts, the majority of queries will be executed using data stored on one disk. The workload thus cannot be distributed uniformly across both disks. Second, random seeks cannot be distributed equally across replicas because different methods can be used to retrieve data on each disk: the NSM layout can retrieve a full tuple via a single access while the DSM layout has to perform index lookups on a surrogate on a table.

To overcome the above problems, the *fractured mirrors* approach extended the mirrors approach so that each disk has a complete copy of data stored in multiple data layouts, as illustrated in Figure 3.10(b). This approach can be described as follows: First, like the mirrors approach, the (original) data is completely replicated into two copies using different data layouts: NSM and DSM. Next, the NSM copy is declustered into two horizontal partitions: NSM0 is put on disk 1 while NSM1 is put disk 2. Similarly, the DSM copy is also declustered into two horizontal partitions: DSM0 is put on disk 2 while DSM1 is put on disk 1. By this way, the workload can be spread evenly across both disks even if it is skewed; additionally, the random seeks can be divided equally between disks as well.

The advantage of mirrors and fractured mirrors approaches is that the query optimizer can choose an appropriate data layout (NSM or DSM) to achieve a better query performance. In addition, the approaches ensure against data loss in the event of a hard drive failure. However, there exist several disadvantages of these approaches. First, they need more storage space to store multiple copies. Second, they require complicate data management to ensure data integrity in two copies. Third, the approaches have been implemented in software, instead of hardware, thus they will be inefficient. Last, the current approaches simply create round-robin based schemas, thus they cannot produce efficient schemas for various workloads.

Hyper

HyPer [87] is a hybrid OLTP & OLAP main-memory database system that can handle both OLTP and OLAP workloads simultaneously. The HyPer separates two types of

workloads and controls concurrency transactions by creating transaction-consistent snapshot of the database via hardware-assisted virtual memory management of the operating system. OLTP queries are executed serially by the original process, using original physical memory segments. When many OLTP queries concurrently updates the same memory segments, the operating system creates a physical copy of the data to preserve the snapshot consistent. To avoid any interaction with OLTP, when the HyPer needs to execute an OLAP processing, it performs a fork operation to create a virtual memory snapshot. The forked child process (OLAP) gets an exact copy of the address space of the parent process (OLTP), as illustrated in Figure 3.11. However, because the HyPer uses the virtual memory snapshot functionality, it does not physically copy the memory segments. Instead, it applies a lazy copy-on-update mechanism. At beginning, the parent process (OLTP) and the child process (OLAP) use the same physical memory segments. Then the operating system reroutes (translates) the virtual memory accesses, e.g., to a data item a , to the original physical memory segments. At this time, the virtual memory page has not yet created. Once the data item, e.g., a , is updated, the copy-on-update mechanism is activated to replicate the virtual memory page storing the data item a . Afterwards, the OLTP process can accessed to a new state of the data item, i.e., a' , while the OLAP process can still access the old state of the data item, i.e., a .

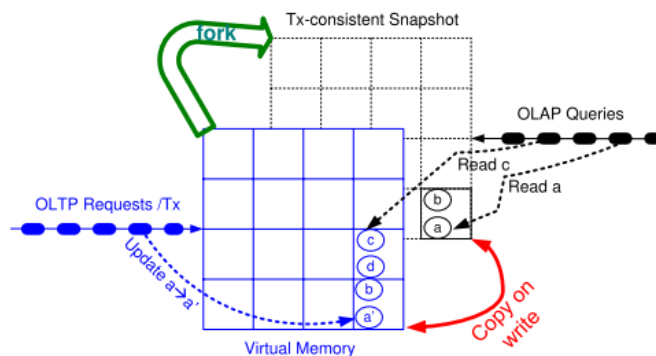


Figure 3.11: Copy-on-update mechanism [93]

The advantage of the HyPer is that it provides an efficient concurrency control mechanism as it deals with simultaneous mixed OLTP and OLAP workloads. The HyPer is regarded as an in-memory column-oriented database system since it transforms the database tables into the column-oriented data layout in vector-based virtual memory. To better utilize the main memory, the column-oriented data is horizontally partitioned and each resulting partition is classified by its access frequency. The seldom-accessed data will be compressed before stored in the main memory. The disadvantage of the HyPer is that it is a main-memory database system, so the problem of main memory limitation will lead to the performance problem.

SAP HANA Database

SAP HANA database (or SAP HANA DB) [20, 94, 95] is an in-memory column-oriented database system. It can handle both OLTP and OLAP workloads and deal with different characteristics of data, such as structured, semi- and unstructured data within the same system. To achieve these features, it uses multiple data processing engines:

Relational engine is responsible for managing relational data (structured data) that can be represented in either row-oriented or column-oriented tables. This engine can process the relational tables represented in both row- and column-oriented storage layouts. Additionally, row- and column-oriented tables can be used together in the same SQL statement. The *graph* and *text engine* are respectively responsible for processing semi- and unstructured data: graph and text data.

To obtain the best performance, the SAP HANA DB pays more attention to organizing data in main memory, instead of disk, for better cache utilization. The data processing engines try to store as much data as possible in main memory. The column-oriented data can be compressed to reduce the size of the data before migrating them into main memory. In addition, the data will be classified as hot or cold data depending on access frequency [95]: the hot data will be cached in the main memory, otherwise stored on disk. The hot data for OLTP workloads usually includes the most recently accessed tuples, while the hot data for OLAP workloads typically consists of the most recently scanned or aggregated columns.

The limitation of the SAP HANA DB is that a system administrator needs to manually determine at definition time whether a new table will be stored in a row- or a column-oriented data layout, and then modify the application to query suitable tables. That is, there is a lack of tool support for automating these works.

3.5 Vertical Partitioning and Bloom Filter Techniques

This section provides background information of common techniques used for schema design of relational databases and query performance improvement. We pay attention on *vertical partitioning* techniques that are able to reduce workload execution time and storage space size for sparse datasets. Besides, *BF* and *IBF* techniques can reduce network and disk I/O costs in distributed query processing environments.

3.5.1 Vertical Partitioning

The vertical partitioning is a technique to divide a table into a number of sub-tables. It aims at reducing I/O costs. Existing vertical partitioning algorithms are usually classified into different approaches based on some dimensions: (1) *measure*: cost-based or affinity-based; (2) *search strategy*: top-down or bottom-up. However, because we are looking for an algorithm that can take into account the combined impact of both the characteristics of data (e.g., sparseness) and workloads (e.g., mixed OLTP and OLAP workloads) on the quality of vertical partitioning results, in order to easily find out the gaps in existing studies, we add a new dimension called *input information*. By this way, we can classify the existing algorithms into *workload-based* or *data-based*. The classifications of the current approaches corresponding to these dimensions are discussed below.

Cost-based vs. Affinity-based Approaches

Cost-based algorithms [96-98] need an objective function (a cost function) to minimize the total workload execution cost of a current system. Such an objective function usually represents a combination of several cost components such as CPU,

I/O and communication costs. The traditional optimization-based techniques such as hill climbing, simulated annealing and genetic algorithm [99-101] can be applied to find out a set of vertical partitions to minimize the objective function. However, a problem is that it is usually hard to build cost functions that accurately express complex execution mechanisms of query optimizers/engines of the current systems [11].

On the other hand, *affinity-based algorithms* [6, 102-104] are based on attribute affinity (which shows how often attributes are simultaneously accessed by the same queries in a given workload) to cluster the attributes into clusters. A limitation of the affinity-based algorithms is that affinity measures are usually independent from the execution of the corresponding query optimizers or query engines of current systems. Thus, the resulting clusters should be further validated on the targeted systems [11].

Top-down vs. Bottom-up Approaches

Top-down algorithms [6-8] usually begin with a schema containing all attributes; and for each step, they decompose that schema into two smaller schemas. This procedure is repeated similarly for each resulting schema until the given objective function (a cost model to compute the total workload execution cost in a given workload) cannot be further improved.

In opposite to the top-down algorithms, *bottom-down algorithms* [9-13] begin with a set of minimally small vertical partitions (i.e., small schemas), each of which may contain either a single attribute or a subset of attributes; and for each step, a pair of vertical partitions are merged together into a larger vertical partition. This procedure is repeated similarly until the objective function cannot be further improved.

Workload-based vs. Data-based Approaches

Workload-based algorithms are the ones depending on workload-specific information (e.g., attribute usage of queries) in order to generate vertical partitions. As such, the above-mentioned approaches (i.e., cost-based, affinity-based, top-down or bottom-up) can be also classified as workload-based approaches if they are using workload-specific information as their input. For instance, we can refer to the vertical partitioning algorithm applying a bottom-down strategy in [13] as a workload-based algorithm. The advantage of the workload-based algorithms is that they can improve the workload performance corresponding to given attribute access patterns. However, they do not take data-specific information (e.g., data sparseness) into consideration, thus they do not mainly focus on reducing storage space size.

In contrast, *data-based algorithms* usually have no knowledge about the workload; instead, they depend on the data-specific information to perform vertical partitioning. Most studies proposed the *data-based algorithms* to design schemas for sparse datasets. Generally, these algorithms used the data-specific information as their input in order to cluster a set of attributes into a number of subspaces (i.e., column groups) in a manner to minimize the sparseness of data. For instance, B. Cui et al. [14] proposed an approach called HoVer that clusters a sparse data space into multiple subspaces. To achieve this, they defined a correlation measure and used it in a heuristic clustering algorithm to group highly correlated attributes (which are frequently co-active) into subspaces. On the other hand, Levandoski and Mokbel [15] proposed data-centric approach that uses a two-phase algorithm to create tables from RDF triples:

first, the clustering phase uses a support threshold to cluster a set of attributes into a number of column groups in order to reduce the number of joins; then, the partitioning phase tries to optimize storage space by reducing the number of null values. However, this approach do not make any assumption about the query workload statistics. Besides, E. Chu et al. [16] proposed wide-table approach to extract hidden schemas from a sparse dataset. To achieve this goal, they applied the Jaccard's coefficient to measure the similarity between any two attributes in terms of the co-occurrence (i.e., simultaneously having non-null values), and implemented a k-NN clustering algorithm, given in CLUTO [105], to group co-occurring attributes together into the same subspace. By this way, hidden schemas can be explored from the sparse datasets.

There is lack of studies that are able to take into consideration the combined impact of both *workload-* and *data-specific information* on the quality of vertical partitioning results. Although studies in [14-16] provided solutions to find out schemas from sparse datasets in a way to reduce storage space demand, from which the query performance can be improved, they replies only on the data-specific information. In fact, the data-centric approach did not assume a particular query workload. Alternatively, the HoVer and wide-table approaches regarded the data-specific information as the workload-specific information: They implicitly assumed that the attributes concurrently having non-null values (or active values) on the same rows in a horizontal table are frequently accessed together by the same queries. However, this assumption does not always hold in the context of DICOM data because the non-null attributes may not be frequently accessed together by the same queries and vice versa. Therefore, the combined impact of both the workload- and data-specific information on both storage space size and query performance has not been explored clearly. Moreover, most studies have not taken into consideration the use of different data layouts to store the vertical partitioning results.

3.5.2 Bloom Filter and Intersection Bloom Filter

Definitions

Bloom filter [22] is a space-efficient probabilistic data structure used for membership test with little error allowable. Let $S = \{x_1, x_2, \dots, x_n\}$ be a finite set of n elements from a universe set U . A Bloom filter (BF) for representing S is described by an array of m bits and a set of k uniform and independent hash functions $h_1(x)$, $h_2(x)$, ..., $h_k(x)$. Initially, all m bits of BF are set to 0 (empty BF). Then, when an element x is inserted into BF , all positions $h_i(x)$ ($1 \leq i \leq k$) of the bit array are set to 1.

Bloom filters allow to answer membership queries like “*Is x in S?*” without the need of the original set S . To check whether an element $x \in S$, we need to check whether all the positions $h_i(x)$ ($1 \leq i \leq k$) of the bit array are set to 1. We then can conclude that x is not presenting in the original set S if at least one of the bit positions $h_i(x)$ is set to 0; otherwise, we conclude that x is probably is a member of S .

Due to hash collisions, there exists an error, also known as a “*false positive*”, such that an element $x \notin S$ has all the positions $h_i(x)$ set to 1. However, there does not exist a false negative when using the BF . The probability of any random bit of the BF to be set to 1 is $P_{SET1} = 1 - (1 - m^{-1})^{kn}$. Thus, the probability where all k bits for a

random element are set to 1 is $P_{BF} = (P_{SET1})^k$; this probability is regarded as the probability of a false positive for an element not in the original set. The false positive probability of the BF can be computed by Formula (3.6.1) [24].

$$P_{BF} = \left(\left(1 - \left(1 - \frac{1}{m} \right)^{kn} \right) \right)^k \approx (1 - e^{-kn/m})^k \quad (3.6.1)$$

To minimize the false positive probability, the information density of the BF has to be optimized. This density is determined by a ratio between the number of true bits (1s) and the length of the BF . The minimum value of the false positive probability occurs when this density is 0.5 [24]; this is achieved when setting the number of hash functions to $k \approx \frac{m}{n} \times \ln(2)$.

Bloom Filter Based Joins

The BF s have been applied in order to filter unnecessary data out of input data of join operations before these operations are performed. Especially in distributed query processing environments, where data is distributed across multiple nodes, the BF s can help to reduce network communication cost for join operations [23].

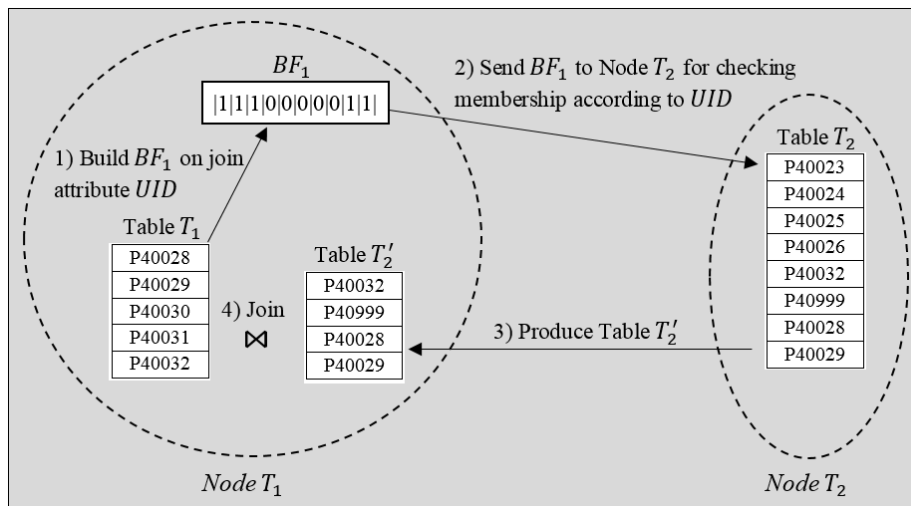


Figure 3.12: Example of the application of a Bloom filter

Figure 3.12 illustrates an example of the application of a Bloom filter for a distributed join operation. Assume that we need to compute a join $T_1 \bowtie_{UID} T_2$ where T_1 and T_2 are two input tables stored at two different nodes $Node T_1$ and $Node T_2$, respectively, and UID is the common join attribute. We also suppose that the join operation will be performed on the $Node T_1$. In order to reduce the amount of data transferred between two nodes, a Bloom filter will be built and applied to remove irrelevant data out of the join inputs. In particular, a join algorithm can be performed in the following steps: (1) The Bloom filter BF_1 is built on values of the join attribute UID of the table T_1 ; here, BF_1 is an array of 10 bits; it uses one hash function $h(x) = \text{LastFiveNumbers}(UID) \% 10$ that returns the remainder after dividing the last five numbers of UID by 10. (2) BF_1 is sent to the $Node T_2$ and applied for checking whether values of the join attribute UID of the table T_2 are contained in BF_1 . (3) The rest of

tuples of the table T_2 (whose $UIDs$ is contained in BF_1) produce a new table T'_2 and this table is transferred to the *Node* T_1 . (4) Finally, T'_2 is joined with T_1 ; during this join, all false positives will be removed from the join result (e.g., in T'_2 , P40999 is a false positive and it will be removed).

As such, the Bloom filters can be applied to remove irrelevant tuples out of input data of the join operations. This helps to avoid transferring unnecessary data over the network as well as to reduce computation cost, due to less input data for processing.

Intersection Bloom Filter

An IBF is used to represent an approximate intersection set of sets [23]. The IBF can be simply computed by performing bitwise AND operations on all the BFs built from input tables. In a distributed query processing environment, the IBF has been proved that they can help join operations to reduce the amount of data transmission on network with a false positive probability less than that of individual component Bloom filters [23, 25]. T.-C. Phan, L. d'Orazio, and P. Rigaux in [25, 27] proposed to use an IBF that is computed from precomputed BFs to filter irrelevant tuples out of input tables of join operations in MapReduce environment. Their experimental results show that amount of intermediate results is reduced and the query performance is increased. On other hands, J. J. Brito et al. in [106] proposed the Spark Bloom-Filtered Cascade Join that applies the BFs to reduce disk spill and network communication by removing irrelevant tuples from input tables of a sequence of joins of the star joins in Spark, in-memory cluster computing framework [21], thereby minimizing the query execution time.

In the context of DICOM data management, to provide the high performance for interactive workloads, Spark is a suitable choice to implement a DICOM data management system. Besides, users' multiple-table join queries may involve a large amount or redundant input data due to high selectivity of predicates. Therefore, a query processing strategy with the integration of the IBF is a potential solution is to improve the performance of the queries. However, there is a lack of studies that apply the IBF that is computed from non-precomputed BFs in a distributed query processing environment, e.g., Spark. Moreover, we need to determine how to integrate an IBF in a particular execution plan and conduct a cost-benefit analysis for this application.

3.6 Key Components of the New System

The main goals of our study are to propose efficient methods to store and to query DICOM data. These methods will be applied to build a new DICOM data management system that satisfies the expected requirements: (R1) *Flexible data*; (R2) *Flexible querying*; and (R3) *Efficiency of storage and CPU*, as introduced in Section 2.3.1.

In order to meet the above requirements, we further specify requirements for key components of the new DICOM data management system: *data model*, *data storage model*, *data schema* and *query processing*.

3.6.1 Data Model

A data model such as the relational data model or the NoSQL data models (key-value, column-family, document and graph) specifies the way data is represented to users. In our study, we need to choose a data model so that it can satisfy the following requirements: First, it is able to easily represent the information entities (*Patient*, *Study*, *Series*, etc.) and their relationships in the DICOM information model. Second, it is able to provide normalized. Third, it is able to provide users with a SQL interface for writing their own queries. Fourth, it is able to provide a huge storage capacity, scalability and elasticity. Lastly, it is able to offer high query performance over high and ever-growing volume of DICOM data.

3.6.2 Data Storage Model

Data storage model is also regarded as data layout (e.g., row-oriented, column-oriented and hybrid-oriented layouts) that defines how data in a database is physically organized on hard disk(s). In our study, a suitable data storage model is proposed in order to improve the performance of queries in mixed OLTP and OLAP workloads. More particularly, we will focus on reducing the following costs: disk I/O cost (caused by redundant data accesses) and tuple reconstruction cost (caused by join operations).

3.6.3 Data Schema

The requirements for schemas are given as follows: First, the schemas need to be designed not only to easily represent entities and their relationships in the DICOM information model, but also to increase the efficiency in storing and querying DICOM data. Besides, to provide ease of use, names of entity tables should be directly used in users' queries instead of other complex forms, e.g., vertically partitioned tables. Second, the variety of DICOM data usually results in sparseness, thus null values need to be removed to save storage space. Third, the schemas need to be designed to increase the performance of queries in mixed OLTP and OLAP workloads. Lastly, an automated design approach need to be proposed to generate data storage configurations that can reduce both storage space demand and workload execution time.

3.6.4 Query Processing

The requirements for query processing can be listed as follows: First, the new DICOM data management system is able to process SQL ad-hoc queries with joins to obtain information from DICOM entities (e.g., *Patient*, *Study*, etc.). Second, the query processing strategy is well-suited to automatically access tables created as results of schema designing (as mentioned above). Lastly, to deal with high and ever-growing volume of DICOM data, the query processing strategy needs to be designed for distributed query processing environment. Furthermore, because the users' queries are usually contain multiple joins with single- and multi-criteria predicates [54], the query processing strategy is needed to remove unnecessary data.

3.7 Summary and Conclusion

In this chapter, we concentrated on reviewing the state of the art of workload types and prevalent databases used for Big Data, cluster computing frameworks, data layouts, vertical partitioning and Bloom filter techniques. We also specify the requirements for key components of the new DICOM data management system so that it is able to efficiently store and query DICOM data:

- **Data model:** The relational data model should be applied for DICOM database. There are a number of reasons for this. Although relational and NoSQL databases have their own benefits, a relational data model excels at providing the following features: First, it is well-suited for representing entities and relationships among these entities in the DICOM information model. This helps to manage the complexity of DICOM data. Second, the relational data model can provide users with SQL ad-hoc queries with joins. Third, using the relational data model, DICOM data can be stored in a normalized way in order to reduce data redundancy and storage space. However, compared to NoSQL databases, relational databases have limitations to provide high query performance, huge data storage and horizontal scalability to deal with the high and ever-growing DICOM data. Thus, it is clear that a pure relational database or a pure NoSQL database alone does not provide all required features. We thus move towards a NoSQL database but need to support to use SQL effectively and to represent data in form of tables.
- **Data Storage model:** A new hybrid storage model should be proposed to store DICOM data. The reason is that a pure row store or a pure column store is optimized for either an OLTP or an OLAP workload, but not both. Moreover, the existing hybrid storage models, such as PAX, Data Morphing, HYRISE, Fractured Mirrors, Trojan Columns and SAP HANA, have some limitations to handle the high and ever-growing volume of data. As a result, we need to design and implement a new hybrid storage model that has a cluster-based storage, e.g., HDFS, to offer huge data storage space, scalability and elasticity.
- **Data schema:** Schemas need to be capable of easily and efficiently representing entities and their relationships in the DICOM information model. The DICOM information model has not optimized in terms of storage space demand and query performance. For instance, wide entity tables can cause data sparseness and redundant data accesses. Existing vertical partitioning algorithms showed their usefulness in schema design, but there is a lack a solution that can take into consideration the combined use of workload- and data-specific information and a hybrid store to automatically create schemas that can reduce both workload execution time and storage space demand. Therefore, there is a need for a novel vertical partitioning approach to overcome this limitation.
- **Query processing:** The query processing needs to provide high performance for interactive workloads. The batch-oriented processing model of MapReduce is not well suited for the interactive workloads due to its high latency. In contrast, the interactive ad-hoc query and analysis technique is good fit to this context; Spark should be chosen because of its ability to offer low latency, high performance, scalability and elasticity. Furthermore, to create the correct answers for join

operations between vertically partitioned tables, inner and left-outer joins need to be applied. Additionally, the BF and IBF should be applied to reduce network I/Os in distributed query processing environment for DICOM data as well.

Key Points

- We present backgrounds of workload types and prevalent databases.
- We review the cluster computing frameworks: MapReduce and Spark.
- We present different types of data layouts.
- We presents related works about the vertical partitioning, BF and IBF.
- We presents the requirements for key components (data model, data storage model, data schema and query processing) of the new system.

PART II

CONTRIBUTIONS

HYTORMO and HADF

4.1 Overview

In Chapter 2, we specified the expected requirements for a new DICOM data management system. In Chapter 3, we presented the solution ideas to efficiently store and query DICOM data. This chapter describes HYTORMO together with data storage and query processing strategies. An overview of the chapter is given in Table 4.1.

Table 4.1: Overview over Chapter 4

4.2 HYTORMO and Strategies	
4.2.1 HYTORMO Architecture	4.2.2 Data Storage Strategy
4.2.3 Query Processing Strategy	
4.3 Automated Design Approach for DICOM Data	
4.3.1 Observations	4.3.2 Formal Representation
4.3.3 Configuration Cost Estimation	
4.4 Hybrid Automated Design Framework	
4.4.1 Overview of the Framework	4.4.2 Similarity Measures
4.4.3 Implementation of the Framework	4.4.4 Examples
4.5 Summary and Conclusion	

First of all, HYTORMO and the data storage and query processing strategies are briefly described in a nutshell. HYTORMO provides high performance for interactive and mixed workloads, huge storage capability, scalability and elasticity. The storage strategy aims at improving workload performance and reducing data storage demand; it combines the use of both vertical partitioning and a hybrid storage model. A high-level query processing strategy is also introduced for HYTORMO.

In order to achieve a data storage configuration according to the above data storage strategy, one of two design approaches, *expert-based* and *automated*, can be applied. The former approach was proposed by B. Mohamad, L. d'Orazio and L. Gruenwald [56, 57], whereas we propose the latter approach is able to automatically generate data storage configurations for DICOM data. We describe our observations from which the formal representation of the automated design problem and cost models are built. However, the solution search space for an optimal data storage configuration is very large, thus we further propose a heuristic approach, a hybrid automated design framework, to rapidly generate good data storage configurations. We describe the framework, similarity measures, implementation of this framework and examples.

4.2 HYTORMO and Strategies

In this section, we present an overview and key components of HYTORMO. First, we describe its architecture. Next, we describe the proposed data storage strategy: what needs to be done in a systematic way to extract, organize and store DICOM data in the hybrid store of row and column stores. Finally, we introduce an overview of the proposed query processing strategy and query form.

4.2.1 HYTORMO Architecture

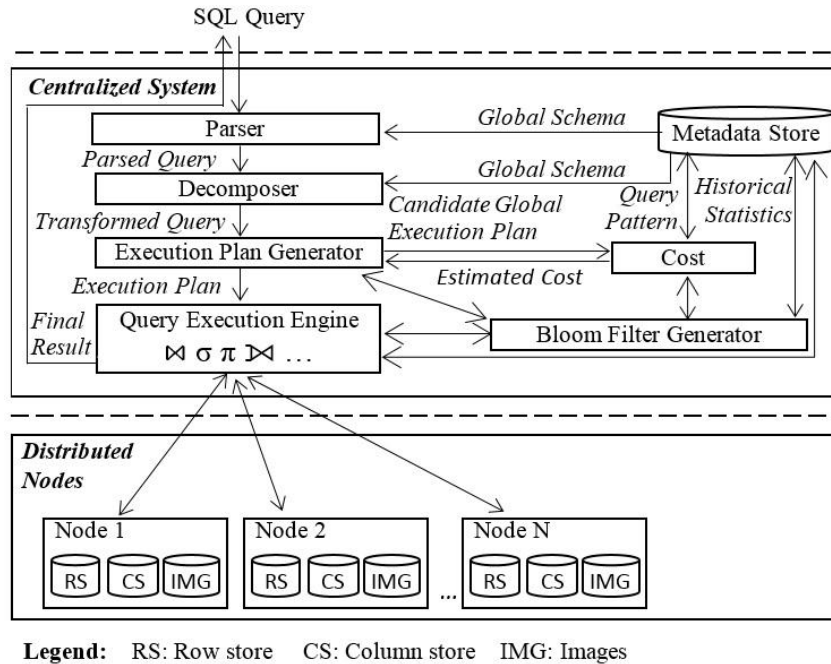


Figure 4.1: Architecture of HYTORMO

Figure 4.1 describes the architecture of HYTORMO. There are two key components: *Centralized System* and *Distributed Nodes*. The query processing is tightly integrated in both Centralized System (a master node) and Distributed Nodes (slave nodes). Query processing tasks are distributed among multiple nodes. DICOM data (metadata and pixel data) are stored across the Distributed Nodes using a distributed file system, e.g., HDFS, which can support for storing DICOM data in both row- and column-oriented storage layouts. HYTORMO is implemented on top of an in-memory cluster computing framework, Spark [21, 83], in order to provide high performance for interactive workloads.

In the following section, the proposed data storage strategy is presented in detail.

4.2.2 Data Storage Strategy

The goals of the data storage strategy are to optimize query performance and storage space over a mixed OLTP and OLAP workload. To achieve these goals, metadata and image data of DICOM files are extracted, organized and stored in a manner to reduce storage space, tuple construction cost and I/O costs.

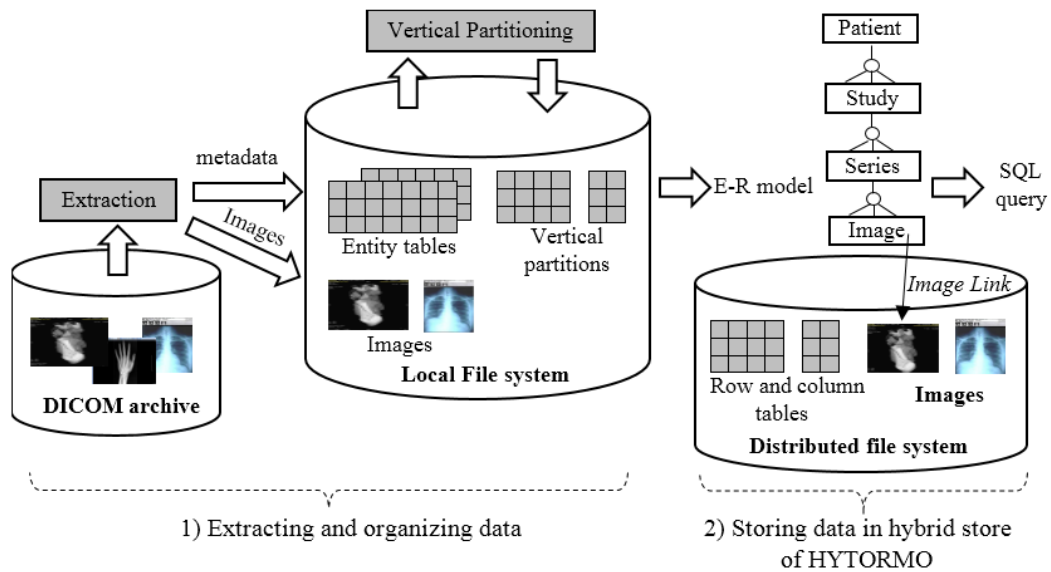


Figure 4.2: Process of extracting, organizing and storing DICOM data

The process of extracting, organizing and storing DICOM data is shown in Figure 4.2. Assume that DICOM files have been produced by specific modalities, and now we need to extract DICOM data from these files, organize and store it in the hybrid store of HYTORMO. First of all, metadata and image data are extracted and stored in a local file system (i.e., DICOM archive). Because we intend to maintain entities and their relationships according to the DICOM information model (presented in Chapter 2), the metadata will be organized into entity tables (relational tables) such as *Patient*, *Study*, *Series*, *Image*, etc. For instance, the entity *Patient* consists of the following attributes: *PatientName*, *PatientID*, *PatientDateofBirth*, *PatientSex*, etc., while the entity *Image* contains *ImageNumber*, *ImageSize*, *ImageType*, *HighBit*, *PatientID*, ..., and *ImageLink* (the *ImageLink* attribute stores the path name of the corresponding image file stored in disk). In our study, we refer to a relational table as a horizontal table (which has not been vertically partitioned yet). The entity-relationship model can be used to visually describe the entities and their relationships.

In order to achieve optimization of storage space and query performance, the proposed data storage strategy is performed as follows: *First of all, the entity tables need to be decomposed into multiple sub-tables (i.e., vertically partitioned tables). Next, these sub-tables will be stored in row and column stores of the hybrid store of HYTORMO (in a distributed file system).*

For simplicity, we refer to (sub-) tables stored in a row store as *row tables*, and tables stored in a column store as *column tables*. After all DICOM data is transferred from the local file system to the hybrid store of HYTORMO, it can be removed from the local file system to save storage space. It is worthy to note that the complexity of the vertical partitioning of the entity tables is transparent to users so that they only need to concentrate on writing interactive and ad-hoc queries by using names of the entity tables in their SQL queries.

In order to achieve a data storage configuration according to the above data storage strategy, one of two design approaches can be applied: *expert-based* and *automated*. In this chapter, before the new automated design approach is introduced, we apply the

expert-based design approach to create data storage configurations, as presented in [107]. In the expert-based design approach, first of all, DICOM attributes are classified into three categories: *mandatory*; *frequently-accessed-together*; and *optional/private/seldom-accessed* (for short, we sometimes call this “*optional*”). Next, the attributes of the first two groups will be stored in a row store while the attributes of the last group will be stored in a column store. In addition to this application, our contribution to this approach is to provide clearly-defined classification of attribute groups in terms of characteristics of both data and workload as follows:

1. *Mandatory attributes* are not allowed to get null values.
2. *Frequently-accessed-together attributes* are allowed to get null values and are frequently accessed together.
3. *Optional attributes* are allowed to get null values but are not frequently accessed together.

As such, the above classification has taken into consideration the similarity relationship among the attributes based on both workload-specific information (i.e., regular attribute access patterns) and data-specific information (i.e., data sparseness) at the same time in order to group the attributes into clusters (i.e., column groups).

In addition to the above definitions, unlike the expert-based design approach in [56, 57] in which a subset of attributes of DICOM files are classified and stored into row and column stores, in this thesis, we use the entity tables (e.g., *Patient*, *Study*, etc.) as a starting point, from which these entity tables will be decomposed into sub-tables. For example, given the entity *Patient* with the following attributes: *PatientName*, *PatientID*, *PatientBirthDate*, *PatientSex*, *EthnicGroup*, *IssuerOfPatientID*, *PatientBirthTime*, *PatientInsurancePlanCodeSequence*, *PatientPrimaryLanguageCodeSequence*, *PatientPrimaryLanguageModifierCodeSequence*, *OtherPatientIDs*, *OtherPatientNames*, *PatientBirthName*, *PatientTelephoneNumbers*, *SmokingStatus*, *Pregnancy*, *LastMenstrualDate*, *PatientReligiousPreference*, *PatientComments*, *PatientAddress*, *PatientMotherBirthName*, and *InsurancePlan Identification*, we will store this entity table as shown in Figure 4.3:

- *PatientName*, *PatientID*, *PatientBirthDate*, *PatientSex*, and *EthnicGroup* are classified as mandatory attributes and stored in a row table, namely *RowPatient*. On the other hand, *PregnancyStatus* and *LastMenstrualDate* are classified as frequently-accessed-together attributes and also stored in a row table, namely *RowPregnancy*.
- *IssuerOfPatientID*, *PatientBirthTime*, *PatientInsurancePlanCodeSequence*, *PatientPrimaryLanguageCodeSequence*, *PatientPrimaryLanguageModifierCodeSequence*, *OtherPatientIDs*, *OtherPatientNames*, *PatientBirthName*, *PatientTelephoneNumbers*, *SmokingStatus*, *PatientReligiousPreference*, *PatientComments*, *PatientAddress*, *PatientMotherBirthName* and *InsurancePlanIdentification* are classified as optional attributes and stored in a column store.

The above grouping of the attributes is non-overlapping; each attribute belongs to only one column group except the attribute *UID* that is used to join the tables together. The null rows will be removed from the vertically partitioned tables. The image data is stored in separate files whose path names are stored in an attribute in relevant tables.

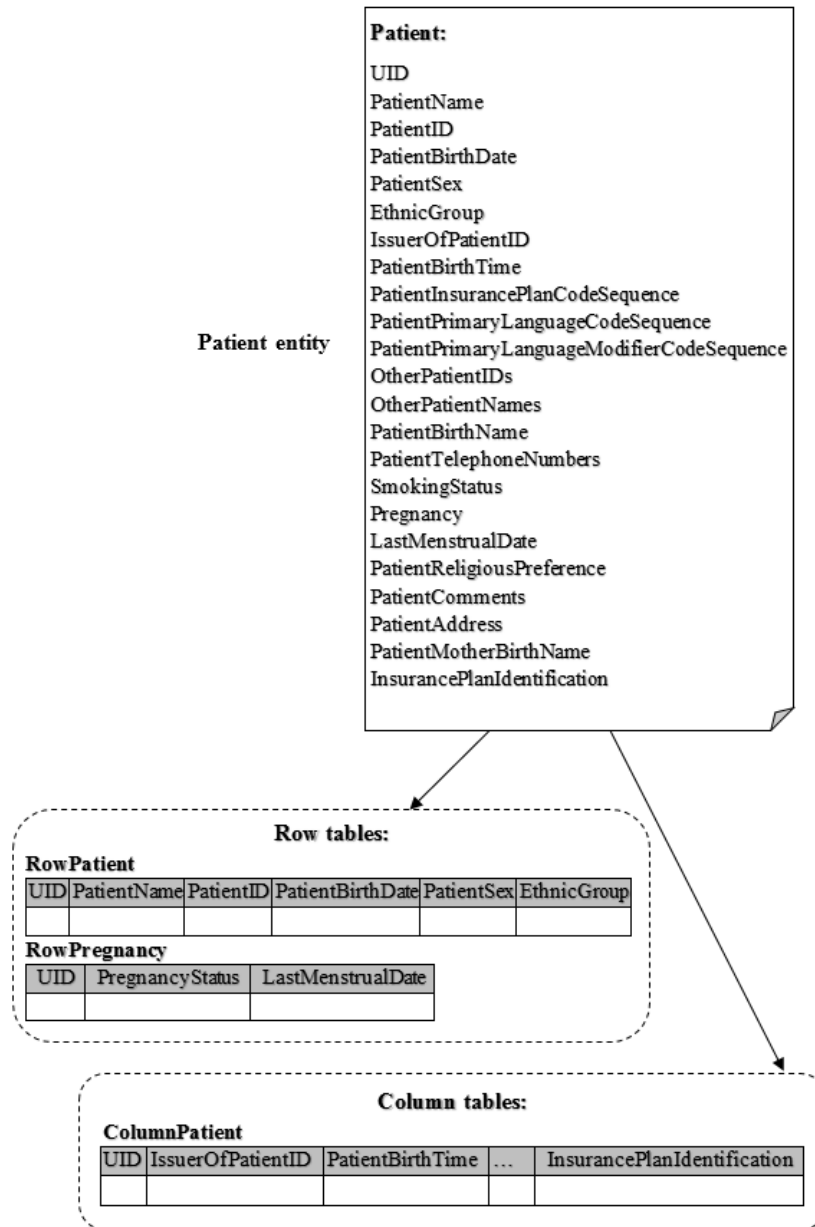


Figure 4.3: Row and column tables of the entity *Patient*

As mentioned earlier, the decomposition of the entity tables is transparent to users. For this purpose, the information about data storage configurations of the entity tables needs to be registered with the Metadata Store of HYTORMO (shown in Figure 4.1): (1) schemas of the entity tables; (2) schemas of the corresponding sub-tables; and (3) data layouts applied to these sub-tables. For instance, assume that the entity *Patient* has been vertically partitioned into two tables *RowPatient* and *ColumnPatient*. However, users may need no knowledge about how *Patient* is vertically partitioned and what data layouts are applied to the corresponding sub-tables. Instead, they simply use name “*Patient*” in their queries.

From the above data storage strategy, we can see clearly that if a query requires to collect information from one or more tables/sub-tables, HYTORMO needs a suitable and efficient query processing strategy to perform join operations across these tables to reconstruct result tuples.

4.2.3 Query Processing Strategy

This section describes the high-level query processing strategy proposed for HYTORMO. Its details will be provided in Chapter 5. The goal of the query processing strategy can be described as follows: *Given DICOM data stored in row and column tables in the distributed file system of HYTORMO, find a well-suited and efficient query processing strategy. In general, both inner joins and left-outer joins are applied. Furthermore, in order to improve query performance, the number of left-outer joins and irrelevant tuples in the input tables of join operations need to be reduced.*

An Overview of the Query Processing Strategy

The query processing includes the following phases: query parsing, query decomposition, query optimization and query execution, as shown in Figure 4.1. The Parser parses a user query (in SQL form). It accesses the Metadata Store to get information about data storage configurations of the entity tables used by the query. The Decomposer splits the query into a set of sub-queries in a way so that the sub-queries access only the relevant row and column tables (containing the attributes required by the query). This helps HYTORMO not only to reduce the size of input data of the query but also to utilize strengths of both row- and column-oriented storage layouts. After the Decomposer completes its works, the Execution Plan Generator generates candidate global execution plans. It consults historical statistics (e.g., cardinality of tables) in the Metadata Store to estimate the execution cost of each plan; after that, it will choose the cheapest one. Since the given query could have a large number of candidate global execution plans due to different join ordering possibilities, an exhaustive search for an optimal execution plan is too expensive. We thus adopt to use a *left-deep sequential tree plan* introduced by M. Steinbrunn et al. [108]. After achieving an execution plan, the Query Execution Engine will execute the query using this execution plan. The sub-queries are executed one after another (across nodes of the Distributed Nodes) according to the join order given in the execution plan. Besides, during the execution of the plan, (Intersection) Bloom Filters, created by the Bloom Filter Generator, are applied to filter irrelevant tuples out of input tables of join operations. Finally, the intermediate results are retrieved and integrated to produce the final query result. It is worthy to remind that when a user submits a query, names of entity tables are used in the query. The query will be automatically rewritten in an equivalent form using a set of sub-queries accessing relevant row and column tables.

Query Form

Our study mainly focuses on user queries consisting of *select*, *project*, *join* and *aggregate operations*. In order to avoid loss of generality, we present a user query Q in a general form given in Figure 4.4. Q is typically a multiple-table join query (or multi-way join query). It can have selection predicates (e.g., comparison predicates consisting of $<$, \leq , $=$, $>$, \geq , etc.) in WHERE clause, aggregate predicates in HAVING clause, a set of attributes in GROUP BY clause and join operations. The entity tables T_i , T_j and T_k are joined together on the attribute UID . Because the attributes of these entity tables may being physically stored in row or column tables, we use the superscripts R_m , R_f , and C to indicate that the corresponding attribute is being stored

in a row table of mandatory attributes, a row table of frequently-accessed-together attributes or a column table of optional attributes, respectively, and RC to denote that the corresponding attribute is being stored in both row and column tables. It is worthy to note that these superscripts are invisible to users.

Q: **SELECT** $T_I.UID^{RC}$, $T_I.att_a^{Rm}$, $T_I.att_b^C$, $T_J.att_x^{Rm}$, $T_J.att_y^{Rf}$, $T_K.att_z^C$
FROM $\{T_I, T_J, T_K\}$
WHERE $\{T_I.UID^{RC} = T_J.UID^{RC}\}$ **AND** $\{T_J.UID^{RC} = T_K.UID^{RC}\}$
 $\{T_I.att_a^{Rm} \theta value_a^{Rm}\}$ **AND** $\{T_I.att_b^C \theta value_b^C\}$ **AND**
 $\{T_J.att_x^{Rm} \theta value_x^{Rm}\}$ **OR** $\{T_K.att_z^C \theta value_z^C\}$
GROUP BY $T_I.att_*$, $T_J.att_*$, or $T_K.att_*$
HAVING $aggregation_operator(T_I.att_*, T_J.att_*$ or $T_K.att_*)$;

where:

- T_I, T_J, T_K : entity tables;
- $T_I(UID^{RC}, att_{-}^{Rm}, \dots, att_{-}^{Rf}, \dots, att_{-}^C, \dots)$: schema of T_I ;
- $T_J(UID^{RC}, att_{-}^{Rm}, \dots, att_{-}^{Rf}, \dots, att_{-}^C, \dots)$: schema of T_J ;
- $T_K(UID^{RC}, att_{-}^{Rm}, \dots, att_{-}^{Rf}, \dots, att_{-}^C, \dots)$: schema of T_K ;
- att_{-}^{Rm} : a mandatory attribute is stored in a row table;
- att_{-}^{Rf} : a frequently-accessed-together attribute is stored in a row table;
- att_{-}^C : an optional/private/seldom-accessed attribute is stored in a column table;
- $value_{-}^{Rm}$, $value_{-}^{Rf}$, $value_{-}^C$: constant values;
- att_{-}^* : a certain attribute of an entity table, such as att_{-}^{Rm} , att_{-}^{Rf} or att_{-}^C ;
- θ : one of $\{<, \leq, =, >, \geq, \text{etc.}\}$;
- $aggregation_operator$: **MIN**, **MAX**, **SUM**, **COUNT**, etc.

Figure 4.4: General form of a user query

Table 4.2 gives examples of user queries: $Q_1 - Q_3$.

Table 4.2: Examples of user queries

Query	SQL Statement	Explanation
Q ₁	SELECT count(*) FROM Patient	Count the number of tuples in the entity <i>Patient</i> .
Q ₂	SELECT UID, PatientName, PatientID, PatientBirthDate, EthnicGroup FROM Patient WHERE PatientSex = 'M' AND EthnicGroup LIKE '%Asian%'	View information about <i>UID</i> , <i>PatientName</i> , <i>PatientID</i> , <i>BirthDate</i> and <i>EthnicGroup</i> of male patients and <i>Asian Ethic</i> .
Q ₃	SELECT p.UID, p.PatientID, p.PatientName, p.PatientBirthDate, p.PatientSex, p.EthnicGroup, p.SmokingStatus, s.PatientAge, s.PatientWeight, s.PatientSize, i.GeneralNames, i.GeneralValues, q.UID, q.SequenceTags, q.SequenceVRs, q.SequenceNames, q.SequenceValues FROM Patient p, Study s, GeneralInfoTable i, SequenceAttributes q WHERE p.UID = s.UID AND p.UID = i.UID AND p.UID = q.UID AND p.PatientSexR = 'M' AND p.SmokingStatus = 'NO' AND s.PatientAge >= x AND q.SequenceNames LIKE '%X-ray%'	View detail information of X-ray images of male, non-smoking and over x-year-old patients.

We introduce the automated design approach for DICOM data in the next section.

4.3 Automated Design Approach for DICOM Data

In Section 4.2, we introduced the *expert-based design approach* to create data storage configurations for DICOM data. In this approach, experts manually decompose the entity tables into a number of vertically partitioned tables and then select suitable data layouts for them. Unfortunately, in practice, experts may be challenged to manually evaluate the similarity relationship among a large number of attributes based on both workload- and data-specific information at the same time as well as to determine which data layout is suitable for each column group. For this reason, in this section, we provide a formal representation of the automated design problem and cost models which are used to evaluate the quality of a data storage configuration in terms of storage and workload execution costs. All of this will be used as fundamentals to build an automated design approach for DICOM data.

First of all, we present our observations on the mixed use of both vertical partitioning and hybrid store to create data storage configurations.

4.3.1 Observations

We refer to a data storage configuration of a horizontal table T as a set of its vertically partitioned tables together with the corresponding data layouts (i.e., row- and column-oriented data layouts) applied to these tables. Based on given workload- and data-specific information, a large number of candidate data storage configurations can be created for T . An automated design approach can be used to support decision makers (e.g., database designers) in selecting a good data storage configuration with respect to expected requirements on storage space demand and workload execution time.

Workload:

q_1 : **SELECT** a_1, a_2, a_3, a_4 **FROM** T
 q_2 : **SELECT** a_1 **FROM** T
 q_3 : **SELECT** a_2 **FROM** T
 q_4 : **SELECT** a_3 **FROM** T
 q_5 : **SELECT** a_4 **FROM** T
 q_6 : **SELECT** a_1, a_2 **FROM** T
 q_7 : **SELECT** a_3, a_4 **FROM** T
 q_8 : **SELECT** a_1, a_2, a_3 **FROM** T

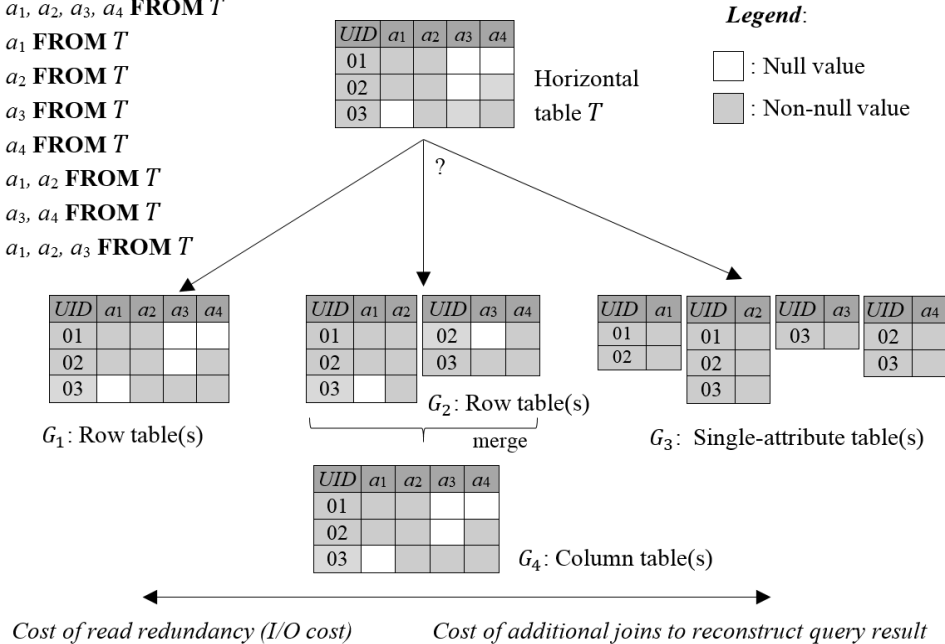


Figure 4.5: Combined use of vertical partitioning and a hybrid store

Figure 4.5 shows an example of the combined use of vertical partitioning and a hybrid store to create candidate data storage configurations for a horizontal table T . Here, we consider 4 different configurations: (1) G_1 : the entire T is stored in a single row table; (2) G_2 : T is decomposed into two vertically partition tables, stored in a row store; (3) G_3 : T is stored in single-attribute tables, stored in a row store; (2) G_4 : two vertically partition tables of G_2 are merged and stored in a single column table. Unlike the DSM layout, shown in Section 3.4.2, from this point henceforth, we assume that a column table still keeps null values in rows (like G_4) unless all values of these rows have null values.

Given a workload of 8 queries q_1 - q_8 , our observations can be described as follows: Using Configuration G_1 (just a row table) is beneficial for q_1 because it avoids the tuple reconstruction cost. In contrast, using Configuration G_3 (single-attribute tables) can help queries q_2 , q_3 , q_4 and q_5 avoid redundant data accesses because only relevant single-attribute tables are read for these queries. Similarly, using Configuration G_2 (two vertically partitioned tables) is beneficial for two queries q_6 and q_7 because only relevant tables are read. However, choosing a suitable configuration for q_8 is challenged because this query accesses overlapping attribute sets: q_8 incurs cost of redundant data accesses if using Configuration G_1 ; in contrast, it has to perform additional join operations if using Configurations G_2 or G_3 . According to these observations, our hypothesis is that if merging two vertically partitioned tables, e.g., the ones in Configuration G_2 , to create a merged table and then store this table in a single column store, e.g., the one in Configuration G_4 , the performance of a query, e.g., q_8 , will be improved because the query incurs neither irrelevant data access (only reading required attributes) nor extra join. Assume that the tuple reconstruction cost of a query is trivial when using a single column table, but this cost is slightly higher than that of a single row table.

The query performance is negatively impacted if the query execution needs to perform many join operations or to access irrelevant attributes. Besides, the storage space demand of the horizontal table T may be varied in different configurations. In general, null rows can be removed from vertically partitioned tables. However, additional storage space may be required to store the surrogate attribute UID in the vertically partitioned tables; this attribute is used to reconstruct result tuples.

It is not difficult to compute the storage space size of a table by depending on the size of its attributes; however, in this study, for simplicity, we assume that the storage space size is represented by the total number of data cells of the table (a data cell is defined as an intersection point between a row and a column of the table). As such, we have assumed that all the attributes of T have the same size. In addition, the storage space size of a data storage configuration is computed as the total number of data cells used to store all of vertically partitioned tables of that configuration. In our future work, in order to increase the accuracy of the storage cost estimation, we intend to take into account the varied sizes of the attributes in the vertical partitioning process; in order to obtain this, the storage space size of a table can be computed as the sum of the number of data cells of each attribute multiplied by its corresponding size (in bytes). Such a method was introduced in [109]. For instance, the total number of data cells used for Configurations G_1 , G_2 , G_3 , and G_4 are 15, 15, 16, and 15, respectively; thus, in this example, the vertical partitioning have not reduced the storage space size.

In a nutshell, given a horizontal table T and a workload, if most of the attributes of T are frequently accessed together by the same queries, storing the entire T in a single row table will reduce tuple reconstruction cost. In contrast, if each attribute of T is often accessed separately, storing each attribute of T in a single-attribute table will reduce I/O cost. If only a few of the attributes of T are frequently accessed together by the same queries, splitting T into multiple vertically partitioned tables and then merging some of these tables into column tables may provide a trade-off between the I/O cost and the tuple reconstruction cost. Besides the improvement of workload performance, if T is very sparse, storing T in multiple vertically partitioned tables can reduce storage space size because many null rows can be removed from these tables.

In the next section, we present the formulation of the automated design problem.

4.3.2 Formal Representation

In this section, we present the formal representation of the automated design problem, including representations of workload-specific information, data-specific information and objective function used to search the best data storage configuration.

Workload-specific Information

Formally, we describe a workload $W = (A, Q, AUM, F)$ with four components:

- $A = \{UID, a_1, a_2, \dots, a_n\}$ is a set of attributes of a horizontal table T . (UID is a unique identifier attribute).
- $Q = \{q_1, q_2, \dots, q_m\}$ is a set of queries executed over T .
- AUM is an Attribute Usage Matrix of size $m \times n$. Each row represents a query and each column represents an attribute: if a query q_i accesses an attribute a_j , the entry $AUM[i, j]$ is equal to 1; otherwise, $AUM[i, j]$ is equal to 0. Each query in AUM is unique, i.e., there are no two queries accessing to the same subset of the attributes.
- $F = \{f_1, f_2, \dots, f_m\}$ is a set of query frequencies. It consists of a set of total frequency counts of the most frequently-used queries in the workload.

	a_1	a_2	a_3	a_4	a_5	a_6
q_1	0	1	1	1	1	1
q_2	0	0	1	1	1	1
q_3	0	0	0	1	1	1
q_4	1	1	0	0	0	0
q_5	1	1	1	0	0	0
q_6	0	1	1	0	0	0

	F
q_1	600
q_2	500
q_3	700
q_4	1000
q_5	200
q_6	400

Attribute Usage Matrix (AUM) Query frequencies (F)

Figure 4.6: Example of Attribute Usage Matrix and query frequencies

Figures 4.6(a) and (b) respectively illustrate two data structures AUM and F of a sample workload of a horizontal table T . This workload consists of 6 queries q_1, q_2, \dots , and q_6 accessing 6 attributes a_1, a_2, \dots , and a_6 . Each query accesses to a subset of the attributes with a particular frequency. For instance, q_1 needs to access 5 attributes a_2, a_3, a_4, a_5 and a_6 , with a frequency of 600. In our study, as default, the attribute UID is included in all vertically partition tables, thus it is not included in the AUM .

Data-specific Information

Characteristics of data can be directly derived from data stored in a horizontal table T . Figure 4.7 shows an example of the horizontal table T which consists of 10 tuples for a set of 7 attributes, $A = \{UID, a_1, a_2, a_3, a_4, a_5, a_6\}$. In this table, an empty data cell stands for a null value. The data-specific information includes the sparseness of data and column groups which simultaneously have non-null values in the same rows. These information can be used as inputs for the vertical partitioning in order to determine decisions which attributes should be grouped and stored together in a way to reduce storage space demand.

UID	a_1	a_2	a_3	a_4	a_5	a_6
01	V1,1	V1,2	V1,3	V1,4	V1,5	
02	V2,1	V2,2	V2,3	V2,4		V2,6
03	V3,1	V3,2	V3,3	V3,4		
04	V4,1	V4,2	V4,3	V4,4		
05	V5,1	V5,2	V5,3	V5,4		
06	V6,1	V6,2	V6,3	V6,4		
07	V7,1	V7,2	V7,3	V7,4	V7,5	
08	V8,1	V8,2	V8,3	V8,4	V8,5	V8,6
09	V9,1	V9,2	V9,3			
10	V10,1	V10,2	V10,3			

Figure 4.7: Example of the horizontal table T

Representation of a Data Storage Configuration

Let $S = \{\text{"row - store"}, \text{"column - store"}\}$ denote a set of available data layouts. Without loss of generality, we denote a set of candidate data storage configurations for the horizontal table T as $G = \{G_1, G_2, \dots, G_K\}$, where G_i is a candidate data storage configuration and K is the number of possible candidate data storage configurations. Each configuration $G_i = (C_i, L_i)$ consists of two components: a set of column groups (i.e., vertical partitions) $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$ and a set of data layouts $L_i = \{L_{d_1}(C_{i,1}), L_{d_2}(C_{i,2}), \dots, L_{d_z}(C_{i,z})\}$ applied to those column groups. Here, $L_{d_x}(C_{i,x})$ denotes that the column group $C_{i,x}$ is stored in data layout d_x , where $d_x \in S$. The set C_i is produced as a result of applying a data storage strategy (as presented in Section 4.2.2) to the horizontal table T . Each column group $C_{i,x} = \{UID, a_1, a_2, \dots, a_h\}$ is a subset of the attributes in T such that $\cup_{C_{i,x} \in C_i} C_{i,x} = A$ and $C_{i,x} \cap C_{i,y} = \{UID\}$ for any $x \neq y$. (The column groups are non-overlapping, meaning that they share no common attribute except the attribute UID .)

For example, using the horizontal table T in Figure 4.7, we can create some data storage configurations as follows: Configuration $G_1 = (C_1, L_1)$, where $C_1 = \{C_{1,1}\}$, $C_{1,1} = \{UID, a_1, a_2, a_3, a_4, a_5, a_6\}$ and $L_1 = \{L^{\text{"row-store"}}(C_{1,1})\}$, means that the entire T is stored in a single row table. Alternatively, Configuration $G_2 = (C_2, L_2)$, where $C_2 = \{C_{2,1}, C_{2,2}\}$, where $C_{2,1} = \{UID, a_1, a_2, a_3, a_4\}$, $C_{2,2} = \{UID, a_5, a_6\}$ and $L_2 = \{L^{\text{"row-store"}}(C_{2,1}), L^{\text{"column-store"}}(C_{2,2})\}$. This configuration implies that T has been vertically partitioned into two column groups $C_{2,1}$ and $C_{2,2}$: the first is stored in a row table while the second is stored in a column table.

Objective Function

The problem of the automated design can be formulated as follows: *Given a horizontal table T and a workload W , find a data storage configuration G_i for T in order to minimize the value of both cost functions: $STORAGE_COST(W, G_i)$ and $EXECUTION_COST(W, G_i)$. The objective function is described as follows:*

$$\begin{cases} STORAGE_COST(W, G_i) \rightarrow \min \\ EXECUTION_COST(W, G_i) \rightarrow \min \end{cases} \quad (4.3.1)$$

where the cost $STORAGE_COST(W, G_i)$ is the total number of data cells used to store all column groups of G_i while the cost $EXECUTION_COST(W, G_i)$ is the execution cost of all queries in the workload W using G_i .

Each candidate data storage configuration $G_i = (C_i, L_i)$ is produced as the result of applying a particular data storage strategy to generate a set of column groups C_i and a set of corresponding data layouts L_i applied to C_i . For instance, in Section 4.2.2, we introduced the expert-based design approach to achieve such a storage configuration.

An alternative way to represent the above objective function is to use a cost-benefit function. Initially, we create a baseline data storage configuration for the given horizontal table T by storing all the attributes UID , a_1 , a_2 , ..., and a_n of T in just a single row table. This configuration can be represented as $G_1 = (C_1, L_1)$, where $C_1 = \{C_{1,1}\}$, $C_{1,1} = \{UID, a_1, a_2, \dots, a_n\}$ and $L_1 = \{L_{"row-store"}(C_{1,1})\}$. By this way, we can find the best data storage configuration within a set of possible data storage configurations $G = \{G_1, G_2, \dots, G_K\}$, where K is the number of possible data storage configurations, by estimating cost-benefit of each G_i compared with the baseline G_1 :

$$\begin{cases} Storage_Benefit(W, G_i) = \max(0, STORAGE_COST(W, G_1) - STORAGE_COST(W, G_i)) \\ Time_Benefit(W, G_i) = \max(0, EXECUTION_COST(W, G_1) - EXECUTION_COST(W, G_i)) \end{cases} \quad (4.3.2)$$

The best data storage configuration is the one giving the most beneficial values in terms of both the storage space demand and the workload execution cost. In the next section, we show how to estimate the costs.

4.3.3 Configuration Cost Estimation

It is less likely that all the attributes of the horizontal table T are required once per query. In typical cases, only a subset of the attributes in T is used once per query. This causes irrelevant attribute accesses if T is stored in a single row table. Moreover, if T is highly sparse, a large number of null values may result in waste of storage space. Although the vertical partitioning of T into several tables can help to reduce the number of irrelevant attributes accesses as well as null values, this approach may needs extra joins to reconstruct result tuples as well as additional storage space for a surrogate attribute, e.g., UID , added to each vertically partitioned tables. Therefore, selecting a data storage configuration should take into consideration of the storage space demand, the number of null values, the number of irrelevant attribute accesses and the number of extra joins needed to reconstruct result tuples. In general, a data storage configuration can be evaluated based on two main costs: *storage cost* and *workload execution cost*. The mathematical expression of these costs is presented below.

Storage Cost

We estimate the storage cost of a data storage configuration in terms of the number of data cells. It is easy to observe that there is a general trend toward the decrease in the number of null values if we split the given horizontal table T into multiple vertically partitioned tables; this is because null rows can be removed from the vertically partitioned tables. However, this is followed by adding a surrogate attribute to each vertically partitioned table; thus, the storage space demand may be increased if the number of removed null values has not been large enough. Therefore, the overall storage cost of a data storage configuration needs to include storage space demand required for that surrogate attribute.

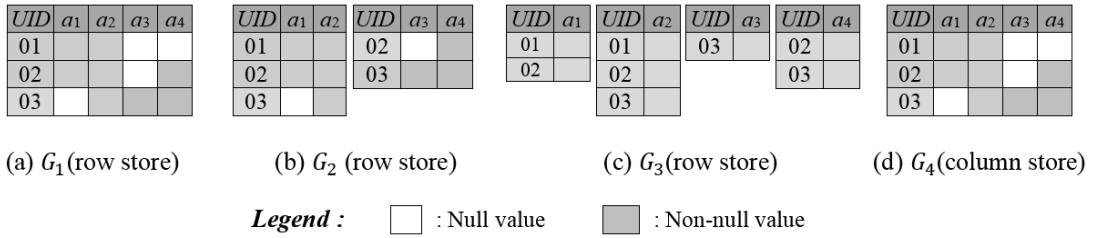


Figure 4.8: Four different configurations of the horizontal table T

Figure 4.8 re-presents four different data storage configurations G_1 , G_2 , G_3 , and G_4 of the horizontal table T , shown in Figure 4.5. If all the attributes of T are stored in a single row table, i.e., G_1 in Figure 4.8(a), or a single column table, i.e., G_4 in Figure 4.8(d), the storage cost is the same, i.e., 15 (data cells). If T is decomposed into two vertically partitioned tables and stored in a row store, i.e., G_2 in Figure 4.8(b), the storage cost is 15 (data cells). If T is decomposed into four single-attribute tables, stored in a row store, i.e., G_3 in Figure 4.8(c), the storage cost is 16 (data cells).

Without loss of generality, given a horizontal table T and its data storage configuration $G_i = (C_i, L_i)$, the size of each column group $C_{i,x} = \{UID, a_1, a_2, \dots, a_h\}$, where $C_{i,x} \in C_i$, can be approximately estimated by using Formula (4.3.3). This estimation has included the cost to store the surrogate attribute UID in $C_{i,x}$.

$$COLUMNGROUP_SIZE(C_{i,x}) = \lceil Length(T) \times (1 - NullRowRatio(C_{i,x})) \times |C_{i,x}| \rceil, \quad (4.3.3)$$

where $Length(T)$ is the length of T and is computed as the number of tuples (rows); $NullRowRatio(C_{i,x})$ is the null-ratio and is computed as the number of null rows divided by $Length(T)$; $\lceil \cdot \rceil$ is a ceiling function; $Length(T) \times (1 - NullRowRatio(C_{i,x}))$ represents the length of $C_{i,x}$ after removing all null rows; $|C_{i,x}|$ represents the number of attributes of $C_{i,x}$ which includes the attribute UID . Hence, $\lceil Length(T) \times (1 - NullRowRatio(C_{i,x})) \times |C_{i,x}| \rceil$ gives the total size of $C_{i,x}$.

Assume the null ratio of each attribute is independent from others, and the distribution of null values within the same attribute is uniform. The null-row ratio of the column group $C_{i,x}$ can be estimated approximately as follows:

$$NullRowRatio(C_{i,x}) = \prod_{a_k \in C_{i,x}} NullRatio(a_k), \quad (4.3.4)$$

where $NullRatio(a_k)$ is the null ratio of an attribute a_k with $a_k \neq UID$ (the attribute UID always has non-null value).

The storage cost of a data storage configuration G_i is assimilated to the total number of data cells of all column groups $C_{i,x}$ of G_i (after removing all null rows):

$$STORAGE_COST(G_i) = \sum_{C_{i,x} \in G_i} COLUMNNGROUP_SIZE(C_{i,x}) \quad (4.3.5)$$

Now, we can estimate the reduction in the storage space size of the data storage configuration G_i when compared with the baseline G_1 (where all the attributes of the horizontal table T are stored in a single row table) by Formula (4.3.6):

$$REDUCTION_SIZE(G_i, G_1) = \left(1 - \frac{STORAGE_COST(G_i)}{STORAGE_COST(G_1)}\right) \times 100 \quad (4.3.6)$$

Formula (4.3.6) returns the percentage decrease of the storage space size.

The above approximate estimation can be used to rapidly estimate the storage cost of a data storage configuration without scanning vertically partitioned tables (storing data for column groups) because providing an accurate estimate for a large number of candidate storage configurations over a large number of attributes may consume time.

For example, given the horizontal table T as shown in Figure 4.8(a), we can estimate the storage space size for Configuration G_2 , shown in Figure 4.8(b) as follows: First of all, based on the horizontal table T as shown in Figure 4.8(a), we obtain the length of T is $Length(T) = 3$ and the null ratios of its attributes are: $NullRatio(a_1) = NullRatio(a_4) = 1/3$; $NullRatio(a_2) = 0$ and $NullRatio(a_3) = 2/3$. Next, using these values, the null-row ratios of two column groups $\{a_1, a_2\}$ and $\{a_3, a_4\}$ of Configuration G_2 can be estimated by Formula (4.3.4):

- $NullRowRatio(\{a_1, a_2\}) = \frac{1}{3} \times 0 = 0$.
- $NullRowRatio(\{a_3, a_4\}) = \frac{2}{3} \times \frac{1}{3} = \frac{2}{9}$.

Then, the storage space size of the above two column groups (including their surrogate attribute) can be estimated by Formula (4.3.3):

- $COLUMNNGROUP_SIZE(\{a_1, a_2\}) = [3 \times (1 - 0) \times 3] = 9$ (the actual value is 9).
- $COLUMNNGROUP_SIZE(\{a_3, a_4\}) = \left[3 \times \left(1 - \frac{2}{9}\right) \times 3\right] = 7$ (the actual value is 6).

After that, using the above results, the storage cost of the configuration G_2 can be estimated by Formula (4.3.5):

$$STORAGE_COST(G_2) = COLUMNNGROUP_SIZE(\{a_1, a_2\}) + COLUMNNGROUP_SIZE(\{a_3, a_4\}) = 16.$$

Finally, we use Configuration G_1 as a baseline configuration. Its storage cost is 15 (data cells). We can estimate the reduction in the storage space size of G_2 when compared with G_1 by using Formula (4.3.6) as follows:

$$\begin{aligned} REDUCTION_SIZE(G_2, G_1) &= \left(1 - \frac{STORAGE_COST(G_2)}{STORAGE_COST(G_1)}\right) \times 100 = \left(1 - \frac{16}{15}\right) \times 100. \\ &= -7\%. \end{aligned}$$

The above result implies that the storage cost of G_2 is 7% larger than that of G_1 . That is, there is no benefit in terms of storage space demand when applying G_2 .

In short, if a data storage configuration is created by applying the vertical partitioning and if the number of removed null values is not large enough, that configuration would not benefit in terms of storage cost due to the additional storage cost required for the surrogate attribute. However, it may benefit from the reduction in the query execution time because of avoiding to expensive reconstruction cost and/or irrelevant attribute accesses.

Reconstruction Cost of a Configuration

Reading Cost: Before measuring the reading cost of a data storage configuration, we assume that scanning a data cell needs a uniform cost of 1 unit. This is because we target to compare the benefit among different candidate storage configurations rather than to obtain accurate estimates of their physical storage sizes.

Given a horizontal table T with a set of attributes A , a query q , a data storage configuration $G_i = (C_i, L_i)$ of T , let $A^q \subseteq A$ be a set of the attributes that the query q actually requires, and let $C_i^q \subseteq C_i$ denote a set of column groups required to answer q if q is using G_i , i.e., $C_i^q = \{C_{i,x} \in C_i \mid C_{i,x} \cap A^q \neq \emptyset\}$.

We define a new intersection operation \cap_L between two attribute sets X and Y so that it can take into consideration the impact of the data layout d_x used to store the left argument (i.e., X) on the result of this intersection operation:

$$L_{d_x}(X) \cap_L Y = \begin{cases} X & \text{if } X \text{ is stored in a row table (i. e., } d_x = \text{"row-store"}) \text{ and } X \cap Y \neq \emptyset \\ \emptyset & \text{if } X \text{ is stored in a row table (i. e., } d_x = \text{"row-store"}) \text{ and } X \cap Y = \emptyset \\ X \cap Y & \text{if } X \text{ is stored in a column table (i. e., } d_x = \text{"column-store"}) \end{cases}, \quad (4.3.7)$$

where X stands for a column group stored using the data layout d_x while Y stands for the attributes required by the query q . Hence, the Formula (4.3.7) will return a set of attributes that are actually scanned for answering the query q . The result is based on the data layout used to store X .

Now, we apply the Formula (4.3.7) to estimate the reading cost for a data storage configuration G_i . For each column group $C_{i,x} \in C_i^q$ of G_i , the number of attributes of the column group $C_{i,x}$ that is scanned by the query q depends on the data layout d_x used to store $C_{i,x}$ and is expressed by the Formula (4.3.7) as follows: $|L_{d_x}(C_{i,x}) \cap_L A^q|$. Let $r_{i,x}^q$ denote the number of tuples (rows) in $C_{i,x}$ that the query q has to scan. We assume that we do not use indexes and horizontal partitioning, thus all tuples of $C_{i,x}$ need to be read. In this case, $r_{i,x}^q$ is exactly equal to the number of rows in $C_{i,x}$, i.e., $r_{i,x}^q = r_{i,x}$. Additionally, the surrogate attribute is always added to each column group $C_{i,x}$, thus the additional reading cost required for this attribute is also included into the

total reading cost. The reading cost required for the query q to read $C_{i,x}$ is $(|L_{d_x}(C_{i,x}) \cap_L A^q|) \times r_{i,x}^q$.

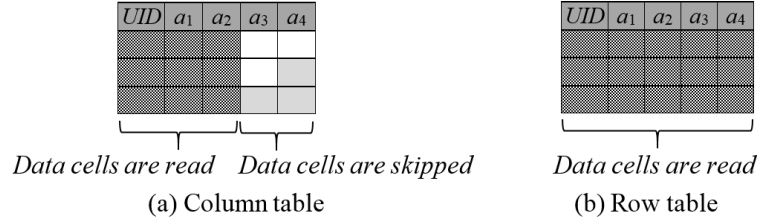


Figure 4.9: Reading effectiveness in (a) a column store and (b) a row store

Figure 4.9 shows an example of reading effectiveness. Here, the given horizontal table consists of 5 attributes UID, a_1, a_2, a_3, a_4 and 3 tuples (this horizontal table is the same as the one given in Figure 4.5); it contains 15 data cells. Assume that a query q needs to access only 3 attributes UID, a_1 and a_2 . If this table is stored in a single column table, as shown in Figure 4.9(a), only 9 data cells are read to answer q while the remaining data cells are ignored. In contrast, if it is stored in a single row table, as shown in Figure 4.9(b), all of the 15 data cells are read (q has to access all the attributes, including two irrelevant attributes a_3 and a_4).

The total reading cost for the query q when using the data storage configuration $G_i = (C_i, L_i)$ is estimated as follows:

$$READ_COST(q, G_i) = \sum_{C_i \in G_i, C_i^q \in C_i, C_{i,x} \in C_i^q, L_{d_x} \in L_i} [(|L_{d_x}(C_{i,x}) \cap_L A^q|) \times r_{i,x}^q], \quad (4.3.8)$$

where $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$, $L_i = \{L_{d_1}(C_{i,1}), L_{d_2}(C_{i,2}), \dots, L_{d_z}(C_{i,z})\}$, $d_x \in S$ and $S = \{\text{"row-store"}, \text{"column-store"}\}$.

Formula (4.3.8) computes the total reading cost for the query q when the data storage configuration G_i is used. It shows clearly that the reading effectiveness is achieved mainly when a column store is used:

- If $C_{i,x}$ is stored using a row table, all the attributes of $C_{i,x}$ have to be read by q : $|L^{\text{"row-store"}}(C_{i,x}) \cap_L A^q| = |C_{i,x}|$.
- If $C_{i,x}$ is stored using a column table, only the relevant attributes of $C_{i,x}$ are read by q : $|L^{\text{"column-store"}}(C_{i,x}) \cap_L A^q| \leq |C_{i,x}|$.

Tuple Reconstruction Cost: As mentioned earlier, when a user write a query q , names of entity tables (e.g., *Patient*, *Study*, *Series*, etc.) are used in q . We refer to these tables as horizontal tables. Then, each horizontal table may be decomposed into a number of vertically partitioned tables. As a result, if the query q needs to access attributes across several column groups, i.e., $|C_i^q| > 1$, HYTORMO has to perform additional join operations to reconstruct the original tuples from the relevant vertically partitioned tables. Thus, the tuple reconstruction cost needs to be taken into consideration when selecting a data storage configuration.

HYTORMO will automatically rewrite the query q into a sequence of inner and/or left-outer joins between relevant vertically partitioned. The attribute UID will be used as a join attribute to join the vertically partitioned tables together. A similar approach

has been presented by B. Cui et al. [14]. In general, given a data storage configuration G_i of the horizontal table T and a set of column groups C_i^q that is required to answer the query q , the query q can be easily translated into a relational algebraic expression as given in Formula (4.3.9):

$$q = \pi_{a_1, \dots, a_m} \left[\pi_{UID}(T) \bowtie \left(\bigbowtie_{x=1}^{|C_i^q|} \sigma_{P_{i,x}}(C_{i,x}) \right) \right], \quad (4.3.9)$$

where the selection operation $\sigma_{P_{i,x}}(C_{i,x})$ returns only tuples of the table storing data for the column group $C_{i,x}$ for which the predicate (or condition) $P_{i,x}$ is fulfilled. The projection operation $\pi_{UID}(T)$ returns a list of all UID s of the horizontal table T . However, this projection operation may not be required if this join sequence begins with a column group $C_{i,x}$ containing mandatory attributes of DICOM data. This is because, in this case, $C_{i,x}$ already consists of a list of all UID s. The projection operation $\pi_{a_1, \dots, a_m}[\dots]$ returns all tuples of the query result, where only the attributes a_1, \dots, a_m listed behind the keyword `SELECT` of the query q appear.

For example, given the data storage configuration G_3 of the horizontal table T , as given in Figure 4.8(c), the query $q = \text{SELECT } a_1, a_2, a_4 \text{ FROM } T$ can be translated into a relational algebraic expression as follows:

$$q = \pi_{a_1, a_2, a_4} \left(\pi_{UID}(T) \bowtie C_{3,1} \bowtie C_{3,2} \bowtie C_{3,4} \right),$$

where $C_{3,1}$, $C_{3,2}$ and $C_{3,4}$, respectively, represent three row tables storing data of three column groups $\{UID, a_1\}$, $\{UID, a_2\}$ and $\{UID, a_4\}$ of G_3 . Here, the result tuples of q are reconstructed by using a sequence of left-outer joins.

A complete estimate for the tuple reconstruction cost is quite complex due to a mixed use of both inner and left-outer joins in the same join sequence. Furthermore, the tuple reconstruction cost is aimed to be used for comparing several data storage configurations at the time the query execution plan may not well defined. Therefore, the scope of our study is limited to two cases: (1) left-outer join operations in a join sequence can be rewritten as inner join operations and (2) left-deep plans are used. (These two cases are mentioned in Chapter 5.) With these two cases, given a data storage configuration $G_i = (C_i, L_i)$ and a query q that needs to access a set of relevant column groups C_i^q , the tuple reconstruction cost is estimated by Formula (4.3.10):

$$\begin{aligned} & RECONSTRUCTION_COST(q, G_i) \\ &= \sum_{C_i \in G_i, C_i^q \in C_i, C_{i,x} \in C_i^q} JOIN_SIZE(C_{i,1}, C_{i,2}, \dots, C_{i,|C_i^q|}) \end{aligned} \quad (4.3.10)$$

The tuple reconstruction cost is estimated as the total size of the intermediate results yielded by the execution of a sequence of join operations applied on the relevant vertically partitioned tables. The size of the intermediate result of a join operation between two vertically partitioned tables storing two column groups $C_{i,x}$ and $C_{i,y}$ can be estimated by Formula (4.3.11):

$$\begin{aligned} & JOIN_SIZE(C_{i,x}, C_{i,y}) = \\ & \left(|L_{d_x}(C_{i,x}) \cap_L A^q| \times r_{i,x}^q \right) \times \left(|L_{d_y}(C_{i,y}) \cap_L A^q| \times r_{i,y}^q \right) \times \text{Sel}(C_{i,x}, C_{i,y}), \end{aligned} \quad (4.3.11)$$

where $|L_{d_x}(C_{i,x}) \cap_L A^q| \times r_{i,x}^q$ and $|L_{d_y}(C_{i,y}) \cap_L A^q| \times r_{i,y}^q$ denote the sizes of inputs that are actually read from two vertically partitioned tables storing two column groups $C_{i,x}$ and $C_{i,y}$, respectively; d_x and $d_y \in S = \{\text{"row - store"}, \text{"column - store"}\}$ (see Formula (4.3.7)) are data layouts used to store $C_{i,x}$ and $C_{i,y}$, respectively. We assume that all tuples (rows) of $C_{i,x}$ and $C_{i,y}$ will be read by q , i.e., $r_{i,x}^q = r_{i,x}$ and $r_{i,y}^q = r_{i,y}$, respectively. $Sel(C_{i,x}, C_{i,y})$ represents the join selectivity associated with two vertically partitioned tables storing $C_{i,x}$ and $C_{i,y}$.

When a query only requires data from a single column group, the tuple reconstruction cost is zero. Let us now consider the case where the query needs to access multiple column groups. As mentioned earlier, the reading cost for the query q to read all tuples of the column group $C_{i,x}$ is $(|L_{d_x}(C_{i,x}) \cap_L A^q|) \times r_{i,x}^q$, where $|L_{d_x}(C_{i,x}) \cap_L A^q|$ represents the number of attributes accessed from $C_{i,x}$. Therefore, the tuple reconstruction cost of the query q when applying data storage configuration $G_i = (C_i, L_i)$ can be rewritten in detail as follows:

$$RECONSTRUCTION_COST(q, G_i) = \begin{cases} \sum_{C_i \in G_i, C_i^q \in C_i, C_{i,x} \in C_i^q, y=2..|C_i^q|, L_{d_x} \in L_i} \prod_{x=1}^y [(|L_{d_x}(C_{i,x}) \cap_L A^q|) \times r_{i,x}^q] \times \prod_{t < x} Sel(C_{i,t}, C_{i,x}) & \text{if } |C_i^q| > 1 \\ 0 & \text{otherwise} \end{cases}, \quad (4.3.12)$$

where $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$, $L_i = \{L_{d_1}(C_{i,1}), L_{d_2}(C_{i,2}), \dots, L_{d_z}(C_{i,z})\}$, $d_x \in S$, $S = \{\text{"row - store"}, \text{"column - store"}\}$.

Using Formulas (4.3.8) and (4.3.12), the execution cost of the query q when using the configuration G_i is denoted by the cost function $EXECUTION_COST(q, G_i)$:

$$EXECUTION_COST(q, G_i) = READ_COST(q, G_i) + RECONSTRUCTION_COST(q, G_i) \quad (4.3.13)$$

The execution cost of a workload W when applying the configuration G_i can be estimated by adding the execution cost of each query q in this workload as follows:

$$EXECUTION_COST(W, G_i) = \sum_{q \in W} EXECUTION_COST(q, G_i) \quad (4.3.14)$$

Intuitively, the workload execution cost when using a data storage configuration can be reduced when the storage cost, the number of irrelevant attributes and the number of relevant column groups are reduced. Relying on how sparse the given horizontal table is and how often the attributes of this table are frequently accessed together, an automated design approach can vertically partition this table into multiple tables of various widths, and suggest suitable data layouts for them. A good data storage configuration needs to reduce both the storage space demand and the workload execution time.

However, the solution search space for an optimal data storage configuration that can minimize both storage cost and execution cost, as shown in Formula (4.3.1), is very large due to the need of exploring all possible combinations of the column groups and the data layouts. In practice, it is infeasible to discover all possible data storage configurations. To overcome this limitation, in the next section, we propose a new

hybrid automated design framework that uses a heuristic approach to assist experts in rapidly obtaining a good data storage configuration for a given horizontal table.

4.4 Hybrid Automated Design Framework

As mentioned in Section 3.5.1, some studies proposed different algorithms to design schemas for sparse datasets such as HoVer approach [14], data-centric approach [15], wide-table approach [16]. These algorithms have benefits in reducing the search space of solutions while automatically finding schemas from sparse datasets. However, they still exist some limitations: First, they do not distinguish clearly between the impact of workload- and data-specific information on the quality of vertical partitioning results. The reason is that they are typically based on an assumption that co-occurring attributes (i.e., having non-null values in the same rows of a given horizontal table) are also frequently accessed together by the same queries. However, this assumption does not strictly hold in the context of DICOM data where many non-null attributes may not be frequently accessed together and vice versa. Second, they also assume that all the vertical partitioning results will be stored using the same data layout, e.g., a row-oriented data layout, instead of both row- and column-oriented data layouts. To overcome these limitations, we propose a new *hybrid automated design framework*, called HADF.

4.4.1 Overview of the Framework

In this section, we introduce HADF that is a heuristic approach using both workload- and data-specific information to automatically produce data storage configurations for DICOM data. For this reason, we say that HADF is a *workload- and data-based automated design approach*.

Figure 4.10 shows an overall HADF that uses given inputs to perform two phases, namely *clustering* and *merging-selecting*, to automatically generate a candidate data storage configuration $G_i = (C_i, L_i)$ that consists of a set of column groups $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$ and a set of data layouts $L_i = \{L_{d_1}(C_{i,1}), L_{d_2}(C_{i,2}), \dots, L_{d_z}(C_{i,z})\}$ applied for these column groups. $L_{d_x}(C_{i,x})$ represents that a column group $C_{i,x}$ is stored by using a data layout d_x , where $d_x \in S$ and $S = \{"row - store", "column - store"\}$.

To achieve a candidate configuration G_i , three groups of inputs are required for HADF: (1) *Workload-specific inputs* include *AUM* (Attribute Usage Matrix) and *F* (query frequencies). (2) *Data-specific input* includes the horizontal table *T*. (3) *Parameters* include a weight α for prioritizing similarity measures, a threshold β for clustering attributes, a threshold θ for merging a pair of clusters together, and a threshold λ for selecting a suitable data layout.

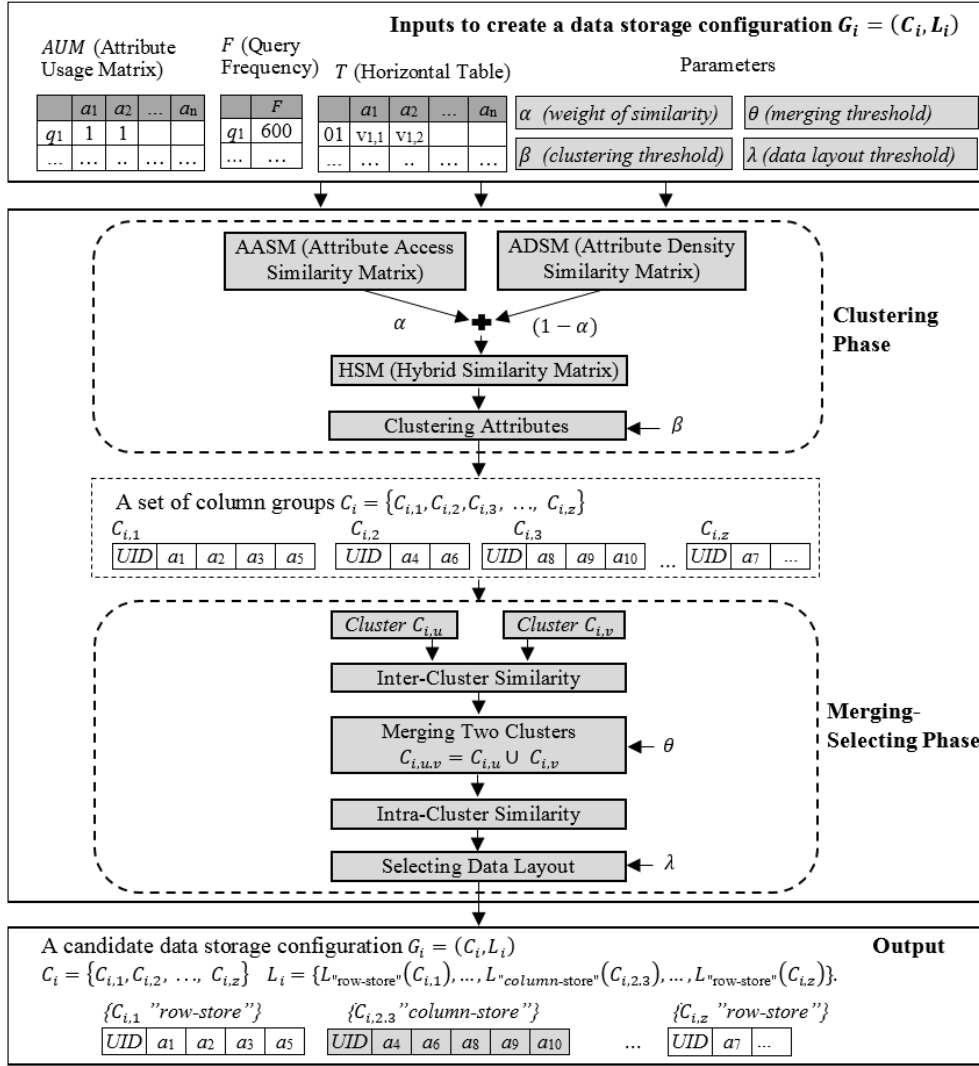


Figure 4.10: Overview of HADF

Using the above inputs, HADF, in turn, performs two phases *clustering* and *merging-selecting* as follows.

- Clustering Phase:** This phase aims at decreasing storage space demand (by reducing null values) and improving query performance (by reducing irrelevant attribute accesses). In order to achieve these aims, the clustering phase takes into consideration the combined impact of both workload- and data specific information on the quality of vertical partitioning results. First, we compute two similarity measures *Attribute Access Similarity* and *Attribute Density Similarity* between every pair of the attributes of the given horizontal table T . The former measure will capture the workload-specific information, while the later measure will capture the data-specific information. The Attribute Access Similarity between two attributes is computed using information about attribute usage, given in *Attribute Usage Matrix (AUM)* and *query frequencies (F)*. *Attribute Access Similarity Matrix (AASM)* is built to represent the Attribute Access Similarity of every pair of the attributes. In general, two attributes has a high value of the Attribute Access Similarity if they are frequently accessed together in the same

queries. On the other hand, the Attribute Density Similarity is computed by exploiting the information about co-occurrence of two attributes, shown in the given horizontal table T ; *Attribute Density Similarity Matrix (ADSM)* is built to represent the Attribute Density Similarity of every pair of the attributes. Two attributes has a high value of the Attribute Density Similarity if they simultaneously occur (i.e., non-null values) in all (or most) rows in T . Next, the *Hybrid Similarity* between each pair of attributes is computed by combining their Attribute Access Similarity and Attribute Density Similarity with the weight α . *Hybrid Similarity Matrix (HSM)* is built to represent the Hybrid Similarity between every pair of the attributes. Finally, using this *HSM*, the clustering phase will cluster the attributes into subspaces (i.e., column groups) so that the Hybrid Similarity between every pair of attributes in the same subspace is greater than or equal to the threshold β . (We say that all the attributes in the same subspace are similar with each other). The output of the clustering phase is a set of resulting column groups $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$.

- **Merging-Selecting Phase:** This phase aims at further improving the query performance by reducing both the tuple reconstruction cost (by reducing the number of additional joins) and the number of irrelevant attribute accesses. To achieve these, the underlying solution idea is to take into consideration the use of a hybrid store instead of just a row store or a column store. The resulting column groups yielded by the clustering phase are used as an initial input for this phase. Additionally, by the default, at initial time, all these column groups are regarded as using a row store. The merging-selecting phase begins with the computation of *Inter-Cluster Similarity* depending on the Attribute Access Similarity. It measures the overlapping access ratio between every pair of column groups (how often two column groups are simultaneously accessed by the same queries). A pair of column groups are chosen and merged together to create a new column group if their *Inter-Cluster Similarity* is greater than or equal to the threshold θ . Furthermore, a column group is stored in a row store if its *Intra-Cluster Similarity* that measures the attribute access ratio to the same column group (over the overall workload) is greater than or equal to the threshold λ ; otherwise, it is stored in a column store. As illustrated in Figure 4.10, two column groups $C_{i,2}$ and $C_{i,3}$ are merged into a new column group $C_{i,2,3}$. Then, $C_{i,2,3}$ is stored in a column store. This procedure is repeated similarly until all pairs of the column groups are considered. The output of the merging-selecting phase is a candidate data storage configuration $G_i = (C_i, L_i)$. For example, two components C_i and L_i in Figure 4.10 are represented as follows: $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$, and $L_i = \{L^{\text{row-store}}(C_{i,1}), \dots, L^{\text{column-store}}(C_{i,2,3}), \dots, L^{\text{row-store}}(C_{i,z})\}$.

In the followings, we provide more details about four parameters used by HADF: α (weight of similarity), β (clustering threshold), θ (merging threshold) and λ (data layout threshold).

- α ranges from 0 to 1. It is used in the clustering phase to control the combined impact of the Attribute Access Similarity and the Attribute Density Similarity on the Hybrid Similarity between two attributes a_x and a_y : $\text{HybridSim}(a_x, a_y) =$

$$\alpha \times \text{AttributeAccessSim}(a_x, a_y) + (1 - \alpha) \times \text{AttributeDensitySim}(a_x, a_y).$$

As such, the higher value α gets, the more impact of the Attribute Access Similarity on the clustering result and vice-versa.

- β ranges from 0 to 1. It is a clustering threshold used in the clustering phase. It is regarded as a threshold of similarity degree between two attributes (in terms of the Hybrid Similarity). The lower value β gets, the larger number of attributes (with low similarity degree) each resulting cluster can have and vice-versa.
- θ ranges from 0 to 1. It is a merging threshold used in the merging-selecting phase. It indicates how often two clusters are accessed together by the same queries in a given workload. When the Inter-Cluster Access Similarity of two clusters has a value of 0, it means that these two clusters have not been accessed together by any query; in contrast, a value of 1 indicates that these two clusters are always used together by all the queries. In general, if two clusters are frequently accessed together by the majority of the queries, their Inter-Cluster Access Similarity will be high; in this case, they should be merged into a new cluster to reduce the number of additional joins. We use the threshold θ for the Inter-Cluster Access Similarity to indicate whether two clusters will be merged together or not.
- λ ranges from 0 to 1. It is a data layout threshold used in the merging-selecting phase. It indicates how often the attributes of the same cluster are accessed together by the same queries in a given workload. If all (or most) attributes of a cluster are frequently accessed together, the value of the Intra-Cluster Access Similarity will be high, thus the cluster should be stored in a row table in order to reduce the tuple reconstruction cost; otherwise, it should be stored in a column table in order to reduce irrelevant attribute accesses. We apply the threshold λ for the Intra-Cluster Access Similarity to determine which data layout will be applied to a cluster.

To deal with the evolution of data (i.e., adding new attributes), new attributes can be stored temporarily in a separated column group. After that, HADF can be used to determine where to store them. However, this work is beyond the scope of this thesis.

In the following section, we present how to compute the similarity measures.

4.4.2 Similarity Measures

In this section, we present in detail the mathematic formalization of the used similarity measures. These similarity measures are computed using the following inputs:

- The workload-specific inputs include components used to represent information about a workload. A workload $W = (A, Q, AUM, F)$ contains the following components: (1) $A = \{a_1, a_2, \dots, a_n\}$ is a set of attributes of a horizontal table T . (2) $Q = \{q_1, q_2, \dots, q_m\}$ is a set of queries. (3) AUM is an Attribute Usage Matrix of size $m \times n$. (4) $F = \{f_1, f_2, \dots, f_m\}$ represents a set of total frequency counts f_i 's of queries q_i 's.
- The data-specific input includes the horizontal table T .

Attribute Access Similarity

We define the notion of *Attribute Access Similarity* between two attributes a_x and a_y based on the Jaccard's coefficient [110, 111] as follows:

$$\text{AttributeAccessSim}(a_x, a_y) = \frac{\sum_{i=1}^m [(AUM[i][a_x] \wedge AUM[i][a_y]) \times f_i]}{\sum_{i=1}^m (AUM[i][a_x] \times f_i) - \sum_{i=1}^m [(AUM[i][a_x] \wedge AUM[i][a_y]) \times f_i] + \sum_{i=1}^m (AUM[i][a_y] \times f_i)}, \quad (4.4.1)$$

where \wedge is a binary bitwise AND operator and m represents the number of queries in the given workload W . An entry $AUM[i][a_x]$ (resp. $AUM[i][a_y]$) indicates whether the attribute a_x (resp. a_y) is accessed by the query q_i or not. In particular, if the attribute a_x (resp. a_y) is accessed by the query q_i , $AUM[i][a_x]$ (resp. $AUM[i][a_y]$) is equal to 1; otherwise, $AUM[i][a_x]$ (resp. $AUM[i][a_y]$) is equal to 0. m represents the number of queries in the given workload W . Hence, $\sum_{i=1}^m (AUM[i][a_x] \times f_i)$ (resp. $\sum_{i=1}^m (AUM[i][a_y] \times f_i)$) represents the total number of times in which the attribute a_x (resp. a_y) is accessed by all the queries. $\sum_{i=1}^m [(AUM[i][a_x] \wedge AUM[i][a_y]) \times f_i]$ represents the total number of times in which both attributes a_x and a_y are accessed simultaneously by all the queries. $\sum_{i=1}^m (AUM[i][a_x] \times f_i) - \sum_{i=1}^m [(AUM[i][a_x] \wedge AUM[i][a_y]) \times f_i] + \sum_{i=1}^m (AUM[i][a_y] \times f_i)$ is the total number of times in which at least one of the two attributes a_x and a_y is accessed. In general, we can depict the relationship among three components, $\sum_{i=1}^m (AUM[i][a_x] \times f_i)$, $\sum_{i=1}^m (AUM[i][a_y] \times f_i)$ and $\sum_{i=1}^m [(AUM[i][a_x] \wedge AUM[i][a_y]) \times f_i]$, in a Venn diagram as shown in Figure 4.11.

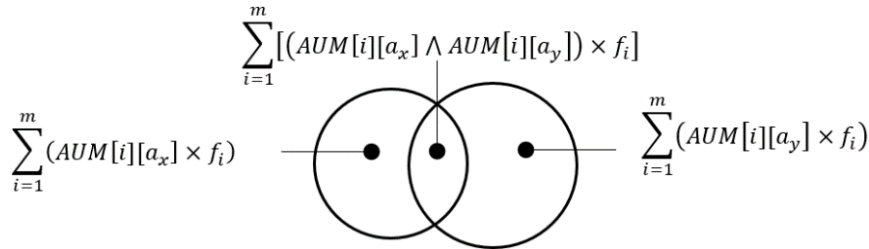


Figure 4.11: Venn diagram

Therefore, the Attribute Access Similarity is defined as the ratio between the total number of times in which a pair of attributes a_x and a_y are simultaneously accessed by the same queries q_i 's in the workload W and the total number of times in which at least one of the two attributes is accessed. Its value ranges from 0 to 1. It returns a value of 1 if two attributes a_x and a_y are always accessed together by the same queries and 0 if these two attributes have never been accessed together by any query.

Using Formula (4.4.1), we construct the Attribute Access Similarity Matrix, $AASM \in \mathbb{R}^{|A| \times |A|}$, to represent the Attribute Access Similarity of all pairs of the attributes.

Attribute Density Similarity

Similarly to the Attribute Access Similarity, we also define *Attribute Density Similarity* based on the Jaccard's Coefficient as follows:

$$\begin{aligned} & \text{AttributeDensitySim}(a_x, a_y) \\ &= \frac{\sum_{i=1}^{|T|} (isNotNull(T[i][a_x]) \wedge isNotNull(T[i][a_y]))}{\sum_{i=1}^{|T|} T.isNotNull(T[i][a_x]) - \sum_{i=1}^{|T|} (isNotNull(T[i][a_x]) \wedge isNotNull(T[i][a_y])) + \sum_{i=1}^{|T|} isNotNull(T[i][a_y])}, \end{aligned} \quad (4.4.2)$$

where $isNotNull(T[i][a_x])$ (resp. $isNotNull(T[i][a_y])$) represents a Boolean function which returns 1 if the attribute a_x (resp. a_y) in i -th row of the horizontal table T has a non-null value; otherwise, 0. We use $|T|$ to denote the number of rows in T .

Therefore, the Attribute Density Similarity is defined as the ratio of the total number of rows in which both attributes a_x and a_y simultaneously have non-null values and the total number of rows in which at least one of these two attributes has a non-null value. Its value ranges from 0 to 1. It returns a value of 1 when both two attributes a_x and a_y always co-occur (i.e., having non-null values) in the same rows of T and 0 when they have never co-occurred in any row of T .

Using Formula (4.4.2), we construct the Attribute Density Similarity Matrix $ADSM \in \mathbb{R}^{|A| \times |A|}$ to represent the Attribute Density Similarity of all pairs of the attributes.

Hybrid Similarity

We propose to measure the *Hybrid Similarity* between two attributes a_x and a_y by using a weighted combination between the Attribute Access Similarity and the Attribute Density Similarity of these two attributes as follows:

$$\begin{aligned} \text{HybridSim}(a_x, a_y) &= \alpha \times \text{AttributeAccessSim}(a_x, a_y) + \\ &\quad (1 - \alpha) \times \text{AttributeDensitySim}(a_x, a_y), \end{aligned} \quad (4.4.3)$$

where α is a user-specified weight parameter that controls the combined impact of the Attribute Access Similarity and the Attribute Density Similarity on the result of the clustering phase. Its value is between 0 and 1.

Now, we can construct the Hybrid Similarity Matrix, $HSM \in \mathbb{R}^{|A| \times |A|}$, to represent the Hybrid Similarity of all pairs of the attributes as follows:

$$HSM = \alpha \times AASM + (1 - \alpha) \times ADSM \quad (4.4.4)$$

The matrix HSM is used in the clustering phase, as shown in Figure 4.10.

Intra- and Inter-cluster Access Similarity

The *Intra-Cluster Access Similarity* of a single cluster $C_{i,u}$ of a data storage configuration G_i is defined as an average access similarity of all pairs of the attributes within this cluster:

$$\begin{aligned} & \text{IntraClusterAccessSim}(C_{i,u}) \\ &= \begin{cases} \frac{\sum_{a_x \in C_{i,u}, a_y \in C_{i,u}, x \neq y} \text{AttributeAccessSim}(a_x, a_y)}{|C_{i,u}| \times (|C_{i,u}| - 1)}, & \text{if } |C_{i,u}| > 1 \\ 1, & \text{if } |C_{i,u}| = 1 \end{cases} \end{aligned} \quad (4.4.5)$$

The value of the Intra-Cluster Access Similarity ranges from 0 to 1. We set the Intra-Cluster Access Similarity to 1 if $C_{i,u}$ contains only one attribute, i.e., $|C_{i,u}| = 1$.

The *Inter-Cluster Access Similarity* between two clusters $C_{i,u}$ and $C_{i,v}$ of a data storage configuration G_i is computed as the average of the total access similarity over every pair of the attributes between these two clusters:

$$\begin{aligned} & \text{InterClusterAccessSim}(C_{i,u}, C_{i,v}) \\ &= \frac{\sum_{a_x \in C_{i,u}, a_y \in C_{i,v}} \text{AttributeAccessSim}(a_x, a_y)}{|C_{i,u}| \times |C_{i,v}|}, \end{aligned} \quad (4.4.6)$$

where $C_{i,u} \neq C_{i,v}$.

Similarly to the Intra-Cluster Access Similarity, the value of the Inter-Cluster Access Similarity ranges from 0 to 1.

The Intra-Cluster Access Similarity and the Inter-Cluster Access Similarity are used in the merging-selecting phase, as shown in Figure 4.10. Because the goal of this phase is to improve the query performance by reducing the number of joins and the number of irrelevant attribute accesses, we only use the Attribute Access Similarity between every pair of the attributes (given in the matrix *AASM*) to compute these two similarity measures.

The Intra-Cluster Access Similarity and the Inter-Cluster Access Similarity have been widely applied to determine the quality of a clustering result. In general, the objective of the clustering is to maximize the Intra-Cluster Access Similarity and minimize the Inter-Cluster Access Similarity [112]. By applying these measures, in the following section, we describe the implementation of HADF.

4.4.3 Implementation of the Framework

HADF applies algorithms to produce data storage configurations from given inputs. The implementation of HADF is described through five algorithms, Algorithms 1 – 5, to compute inputs and to perform two phases *clustering* and *merging-selecting*.

Algorithm 1: Generating a Candidate Storage Configuration

Algorithm 1: GenerateStorageConfiguration

Input : AUM : Attribute Usage Matrix; F : Query frequencies; T : Horizontal table;
 S : A set of available data layouts; α : Weight parameter;
 β : Clustering threshold; θ : Merging threshold; λ : Data layout threshold;
 n : Number of attributes;

Output : G_i : A candidate data storage configuration G_i , consisting of column groups and their corresponding data layouts;

- 1: $AACM = \text{ConstructAttributeAccessCorrelationMatrix}(AUM, m, n, F)$;
- 2: $ADCM = \text{ConstructAttributeDensityCorrelationMatrix}(T, m, n)$;
- 3: $AASM = \text{ConstructAttributeAccessSimilarityMatrix}(AUM, m, n, F)$;
- 4: $ADSM = \text{ConstructAttributeDensitySimilarityMatrix}(T, m, n)$;
- 5: $HSM = \text{ConstructHybridSimilarityMatrix}(AASM, ADSM, n)$;
- 6: $C_i = \text{ClusterAttributes}(AACM, ADCM, HSM, \alpha, \beta, n)$;
- 7: $G_i = \text{MergeAndSelectStores}(C_i, S, AASM, \theta, \lambda, n)$;
- 8: **return** G_i ;

Algorithm 1 is implemented in the function *GenerateStorageConfiguration()*. It computes the inputs related to workload- and data-specific information and calls other

functions to perform two phases clustering and merging-selecting. Its pseudo code is described as follows: First of all, the function *ConstructAttributeAccessCorrelationMatrix()* (line 1) is performed to create the *Attribute Access Correlation Matrix* (AACM) that describes the correlation between every pair of attributes in terms of the number of times in which two attributes are simultaneously accessed. Next, the function *ConstructAttributeDensityCorrelationMatrix()* (line 2) is called to compute the *Attribute Density Correlation Matrix* (ADCM) that describes the correlation between every pair of attributes in terms of the number of times in which two attributes simultaneously have non-null values. Then, Algorithm 1 calls two functions *ConstructAttributeAccessSimilarityMatrix()* (line 3) and *ConstructAttributeDensitySimilarityMatrix()* (line 4) in order to respectively compute two matrices *Attribute Access Similarity Matrix* (AASM) and *Attribute Density Similarity Matrix* (ADSM), which present the similarity between every pair of the attributes in terms of the *Attribute Access Similarity* and the *Attribute Density Similarity*. After that, these two matrices are combined to construct the *Hybrid Similarity Matrix* (HSM) (line 5).

Following that, Algorithm 1 employs the function *ClusterAttributes()* (line 6) to perform the clustering phase which uses a clustering threshold β and the matrix *HSM* to group the attributes of the given horizontal table *T* into clusters such that the Hybrid Similarity between every pair of the attributes in the same cluster is greater than or equal to the given clustering threshold β . By this way, the function *ClusterAttributes()* will return a set of resulting clusters $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$. Finally, Algorithm 1 calls the function *MergeAndSelectStores()* (line 7) to perform the merging-selecting phase which depends on the Inter-Cluster Access Similarity between two clusters in C_i and the given merging threshold θ to determine which pair of clusters is merged together to create a new cluster. This procedure is repeated to consider every pair of clusters in C_i . After that, the merging-selecting phase depends on the Intra-Cluster Access Similarity and the given data layout threshold λ to decide whether a cluster will be stored in a row or a column store. The function *MergeAndSelectStores()* returns a candidate data storage configuration $G_i = (C_i, L_i)$, including a set of clusters C_i and a set of suggested data layouts L_i . This configuration is returned as the result of the function *GenerateStorageConfiguration()* (line 8).

A new candidate data storage configuration G_i will be generated corresponding to a new set of values of the input parameters: α , β , θ and λ . It is worthy to note that some real DICOM datasets can be used as sample data to obtain the inputs (i.e., Attribute Usage Matrix *AUM*, query frequencies *F*, and horizontal table *T*) for this algorithm.

Algorithm 2: Constructing an Attribute Access Correlation Matrix

This algorithm is used to implement the function *ConstructAttributeAccessCorrelationMatrix()* that computes the *Attribute Access Correlation Matrix* (AACM), a square matrix of size $n \times n$, to represent the correlation between every pair of n attributes of the given horizontal table *T* in terms of concurrent access degree. An element $AACM[i][j]$ ($i \leq j$) represents the total number of times in which both attributes i and j are simultaneously accessed. Because the attribute *UID* is always needed in all vertical partitions, we do not need to add it into the matrix *AUM*.

Algorithm 2: ConstructAttributeAccessCorrelationMatrix

Input : AUM : Attribute Usage Matrix (size is $m \times n$);
 m : Number of rows in AUM ; n : Number of attributes; F : Query frequencies;

Output : $AACM$: Attribute Access Correlation Matrix (a square matrix of size $n \times n$);

```

1: for  $i = 1$  to  $n$  do
2:   for  $j = i$  to  $n$  do
3:      $AACM[i][j] = 0$ ;
4:   end for
5: end for
6: for  $q = 1$  to  $m$  do
7:   for  $k = 1$  to  $n$  do
8:     if  $AUM[q][k] = 1$  then
9:        $row[k] = 1$ ;
10:    else
11:       $row[k] = 0$ ;
12:    end if
13:  end for
14:  for  $i = 1$  to  $n$  do
15:    if  $rowl[i] = 1$  then
16:      for  $j = i$  to  $n$  do
17:         $AACM[i][j] = AACM[i][j] + row[j] * F[q]$ ;
18:      end for
19:    end if
20:  end for
21: end for
22: return  $AACM$ ;

```

The pseudo code of Algorithm 2 is described as follows: First, we initialize the matrix $AACM$ by setting its elements to 0 (lines 1 - 5). Next, the matrix AUM is read row by row from top to bottom (lines 6 - 21). For each row q , we store it into an array row : each element $row[k]$ ($1 \leq k \leq n$) is assigned the value of 1 if the attribute k -th is used by the query q ; otherwise, it is assigned a value of 0. Then, we compute the total number of times in which both attributes i and j are simultaneously accessed: for each attribute i that is being used by the query q , we increase $AACM[i][j]$ by $rowl[j] * F[q]$ if both attributes i and j are simultaneously accessed by the query q ($F[q]$ is the frequency of the query q) (lines 14 - 20). Finally, the matrix $AACM$ is returned as the result of the function *ConstructAttributeAccessCorrelationMatrix()* (line 22).

For example, given the matrix AUM and F as presented in Figure 4.6, the matrix $AACM$ is computed and shown in Figure 4.12. The element $AACM[1][1] = 1200$ means that the attribute a_1 is accessed 1200 times while element $AACM[3][5] = 1100$ means that both attributes a_3 and a_5 are simultaneously accessed 1100 times.

	a_1	a_2	a_3	a_4	a_5	a_6
a_1	1200	1200	200	0	0	0
a_2		2200	1200	600	600	600
a_3			1700	1100	1100	1100
a_4				1800	1800	1800
a_5					1800	1800
a_6						1800

Figure 4.12: Attribute Access Correlation Matrix

Algorithm 3: Constructing an Attribute Density Correlation Matrix

Algorithm 3 is used to implement the function *ConstructAttributeDensityCorrelationMatrix()* that computes the *Attribute Density Correlation Matrix (ADCM)*, a square matrix of size $n \times n$, to describe the correlation between every pair of n attributes of the given horizontal table T in terms of concurrent occurrence degree. An element $ADCM[i][j]$ ($i \leq j$) represents the total number of times in which both attributes i and j concurrently have non-null values.

Algorithm 3: ConstructAttributeDensityCorrelationMatrix

Input : T : Horizontal table; m : Number of tuples (rows) in T ; n : Number of attributes of T ;
Output : $ADCM$: Attribute Density Correlation Matrix (a square matrix of size $n \times n$);

```

1: for  $i = 1$  to  $n$  do
2:   for  $j = i$  to  $n$  do
3:      $ADCM[i][j] = 0$ ;
4:   end for
5: end for
6: for  $t = 1$  to  $m$  do
7:   for  $k = 1$  to  $n$  do
8:     if isNotNull( $T[t][k]$ ) then
9:        $row[k] = 1$ ;
10:    else
11:       $row[k] = 0$ ;
12:    end if
13:  end for
14:  for  $i = 1$  do
15:    if  $row[i] = 1$  then
16:      for  $j = i$  to  $n$  do
17:         $ADCM[i][j] = ADCM[i][j] + row[j]$ ;
18:      end for
19:    end if
20:  end for
21: end for
22: return  $ADCM$ ;

```

The pseudo code of Algorithm 3 is described as follows: First of all, we initialize the matrix $ADCM$ by setting its elements to 0 (lines 1 - 5). Next, the horizontal table T is read row by row from top to bottom (lines 6 - 21). For each row, we store it into an array row : each element $row[k]$ ($1 \leq k \leq n$) is assigned the value of 1 if the attribute k -th is not a null value; otherwise, it is assigned a value of 0. After that, we count the total number of times in which both attributes i and j simultaneously have non-null values: for each attribute i having a non-null value, we increase $ADCM[i][j]$ by $row[j]$ if both attributes i and j simultaneously have non-null values (lines 14 - 20). Finally, the matrix $ADCM$ is returned as the result of the function *ConstructAttributeDensityCorrelationMatrix* (line 22).

For example, given the horizontal table T as shown in Figure 4.7, the corresponding matrix $ADCM$ is computed and presented in Figure 4.13. The element $ADCM[1][1] = 10$ means that the attribute a_1 has non-null values in 10 tuples while the element $ADCM[2][4] = 8$ indicates that both attributes a_2 and a_4 simultaneously have non-null values in 8 tuples.

	a_1	a_2	a_3	a_4	a_5	a_6
a_1	10	10	10	8	3	2
a_2		10	10	8	3	2
a_3			10	8	3	2
a_4				8	3	2
a_5					3	1
a_6						2

Figure 4.13: Attribute Density Correlation Matrix

Algorithm 4: Clustering Attributes

Algorithm 4 is used to implement the function *ClusterAttributes()* that performs the clustering phase. We implement it by extending the clustering algorithm proposed by B. Cui et al. [14]. Instead of only taking into consideration the impact of the data-specific information on the clustering result as proposed in [14], our algorithm takes into account the combined impact of both workload- and data-specific information. Algorithm 4 tries to group attributes of a given horizontal table T into a set of clusters C_i in a way to reduce both storage space demand and improve workload performance at the same time. In order to achieve this, given three matrices *AACM*, *ADCM* and *HSM* and two parameters α and β , Algorithm 4 starts by creating an initial (empty) set of clusters of attributes. Next, it repeatedly adds a new cluster into this set. Such a new cluster is created in a way so that the Hybrid Similarity between any two attributes in the same cluster is not less than the clustering threshold β . This procedure is repeated until all unclustered attributes are added into clusters.

The above clusters will be created in the descending order of the importance level of the attributes. This importance level is specified in terms of either attribute access frequency or data density. To achieve this, first of all, we look at the value of weight parameter α to determine whether the attribute access frequency or the data density should be used: the former is chosen if α is greater than or equal to 0.5; otherwise, the latter is chosen. Next, we create a new empty cluster. Then, an attribute having the highest value of the importance level among the unclustered attributes will be selected to become the first element of that new cluster. After that, each of other unclustered attributes will be added into this current new cluster if the Hybrid Similarity between it and every attribute in this cluster is not less than the clustering threshold β . By this way, the attributes having a high value of the importance level will be clustered before the others. Therefore, the important attributes will have more chances to be stored together. This heuristic way can help to reduce search space and create good results. For instance, it can avoid storing dense and spare attributes together to reduce the number of null values; or it can avoid storing frequently-used and seldom-used attributes together to decrease redundant data accesses.

The pseudo code of Algorithm 4 is described as follows: First of all, depending on the value of α , one of two matrices *AACM* and *ADCM* will be used as the priority matrix, i.e., *PriM*, from which the attributes are selected one after another to be considered for clustering attributes (lines 1-6). Next, we create a new empty set of clusters, i.e., $C_i = \emptyset$, and a new empty cluster, i.e., $C_{i,x} = \emptyset$ (line 7). Because we will create non-overlapping clusters, only the attributes that have not been clustered are considered (line 8). Then, we find the most important attribute a_{im} in terms of either the attribute access frequency or the data density (lines 10 - 16) and add it into the new

cluster $C_{i,x}$ (line 18). After that, for each of other unclustered attributes a , if the Hybrid Similarity between a and every attribute $a' \in C_{i,x}$ is not less than β , we add a into $C_{i,x}$ (lines 20 - 31). Once all unclustered attributes have been considered, the resulting cluster $C_{i,x}$ will be added into the set of clusters C_i (line 33). Finally, C_i is returned as the clustering result of the function *ClusterAttributes*() (line 35).

Algorithm 4: ClusterAttributes

Input : AACM: Attribute Access Correlation Matrix;
 ADCM: Attribute Density Correlation Matrix; HSM: Hybrid Similarity Matrix;
 α : Weight parameter; β : Clustering threshold; n : Number of attributes;

Output : C_i : A set of resulting clusters (column groups);

```

1: //choose the priority matrix from which an ordered list of its attributes will be considered
2: if  $\alpha \geq 0.5$  then
3:   PriM = AACM;
4: else
5:   PriM = ADCM;
6: end if
7:  $C_i = \emptyset$ ;  $x = 1$ ;  $C_{i,x} = \emptyset$ ;
8: while there exists an unclustered attribute do
9:   //find the most important attribute,  $a_{im}$ , in terms of workload or data density
10:   $a_{im} = 0$ ; PriM_max = 0;
11:  for each unclustered attribute  $a$  do
12:    if PriM[a][a] > PriM_max then
13:      PriM_max = PriM[a][a]; // an element on the main diagonal of the PriM
14:       $a_{im} = a$ ;
15:    end if
16:  end for
17:  //create a new cluster
18:   $C_{i,x} = C_{i,x} \cup \{a_{im}\}$ ;
19:  //generate a cluster  $c$  that contains highly similarity attributes
20:  for each unclustered attribute  $a$  do
21:    similarity = true;
22:    foreach attribute  $a'$  in  $C_{i,x}$  do
23:      if ( $a' < a$  and  $HSM[a'][a] < \beta$ ) or ( $a' > a$  and  $HSM[a][a'] < \beta$ ) then
24:        similarity = false;
25:        break;
26:      end if
27:    end for
28:    if similarity = true then
29:       $C_{i,x} = C_{i,x} \cup \{a\}$ ;
30:    end if
31:  end for
32:  //add cluster  $C_{i,x}$  into the set of clusters  $C_i$ 
33:   $C_i = C_i \cup \{C_{i,x}\}$ ;  $x = x + 1$ ;
34: end while
35: return  $C_i$ ;

```

The value of β can be chosen based on experiments. In general, if the value of β is small, a large number of attributes having a low value of the Hybrid Similarity will be clustered into the same cluster. This results in a small number of large clusters such that we will create wide tables to store those resulting clusters. Consequently, such wide tables may cause a large number of null values or a large number of irrelevant

attribute accesses. In contrast, if the value of β is large, only pairs of the attributes that have a high similarity value are grouped into the same cluster. This results in a large number of narrow tables created to store the resulting clusters. This helps to reduce the number of null values; however, multiple expensive join operations may be needed to reconstruct result tuples from the attributes stored across narrow tables.

Algorithm 5: Merging and Selecting Stores

Algorithm 5 is used to implement the function *MergeAndSelectStores()*. It aims to improve the query performance. Because an attribute is only clustered into one cluster (i.e., non-overlapping clustering), it is usually impossible to avoid joining vertically partitioned tables to answer queries. As a consequence, the additional join operations may reduce the query performance. In response to this problem, the merging-selecting phase tries to improve the query performance by reducing both the number of additional join operations and the number of irrelevant attribute accesses.

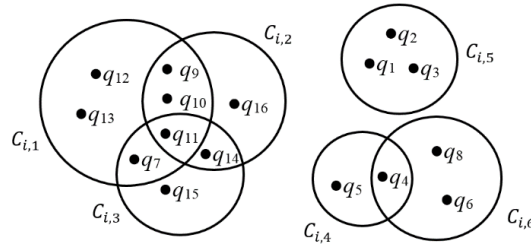


Figure 4.14: Example of cluster usage of a workload

Figure 4.14 presents an example of a set of resulting clusters $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,6}\}$ and a set of queries $Q = \{q_1, q_2, \dots, q_{16}\}$ in a workload. We assume that these clusters have been generated by the clustering phase such that the attributes within each cluster are grouped together based on the Hybrid Similarity. Here, the clusters are being viewed based on cluster usage of the queries: size of a circle representing a cluster denotes the total frequency count of all queries accessing that cluster; a point in a cluster denotes a query accessing one or more attributes of the cluster. It is clear that some queries need to access only one cluster while others may need to access several clusters. A query in a common intersection part of two or more clusters implies that it is accessing the attributes of these clusters. For instance, three queries q_1 , q_2 and q_3 access only the attributes of cluster $C_{i,5}$; thus, no join operation is required to answer these queries. Similarly, each of the queries q_5 , q_6 , q_8 , q_{12} , q_{13} , q_{15} and q_{16} requires to access the attributes of single clusters. When a query needs to use the attributes of several different clusters, it has to perform join operations across these clusters. For example, q_4 requires to join two tables of clusters $C_{i,4}$ and $C_{i,6}$ while q_{11} has to join three tables of three clusters $C_{i,1}$, $C_{i,2}$ and $C_{i,3}$.

The authors in [11] proposed a two-phase algorithm, called *AutoPart*, to reduce I/O costs and the number of additional joins. This algorithm can be described as follows: First of all, a categorical partitioning is performed to produce a set of resulting fragments that can reduce the unnecessary data accesses from a given workload. Next, the resulting fragments are passed through a heuristic procedure of pair-wise merges of the most used fragments in the given workload to reduce the number of joins across

fragments. This thus improves query performance. The merging procedure is repeated until the impact of merging pairs of fragments cannot further improve the overall workload performance. Besides, to remove the overhead joins caused by the need of accessing attributes in different fragments, some attributes are replicated across different fragments. However, this approach has some limitations: the merging of fragments would help to reduce the joining overheads, but queries may access more irrelevant attributes and thus I/O costs may be increased again; additionally, replicating the same attributes in different fragments certainly requires more space storage.

To circumvent the above limitations, our merging-selecting phase uses an alternative heuristic way to reduce the number of additional joins and irrelevant attribute accesses. It performs the following two steps: In the first step, it is based on the Inter-Cluster Access Similarity between two clusters to decide whether these two clusters are merged together or not. In the second step, it suggests a suitable data layout (i.e., a row- or a column-oriented data layout) for each resulting cluster.

The pseudo-code of Algorithm 5 is described as follows:

Algorithm 5: MergeAndSelectStores

Input : C_i : A set of clusters $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$;
 S : A set of available data layouts $S = \{\text{"row - store"}, \text{"column - store"}\}$;
AASM: Attribute Access Similarity Matrix;
 θ : Merging threshold; λ : Data layout threshold; n : Number of attributes;

Output : G_i : A candidate data storage configuration $G_i = (C_i, L_i)$;

- 1: //Step 1: Merge two clusters together based on its Inter-Cluster Access Similarity
- 2: **do**
- 3: $MaxSim = 0.0$; $found = 0$;
- 4: **for** $u = 1$ **to** $|C_i| - 1$ **do**
- 5: **for** $v = u + 1$ **to** $|C_i|$ **do**
- 6: $CurrentSim = InterClusterAccessSimFunc(C_{i,u}, C_{i,v}, AASM)$;
- 7: **if** $CurrentSim \geq \theta$ **and** $CurrentSim > MaxSim$ **then**
- 8: $MaxSim = CurrentSim$;
- 9: $u_{max} = u$; $v_{max} = v$;
- 10: $found = 1$;
- 11: **end if**
- 12: **end for**
- 13: **end for**
- 14: **if** $found = 1$ **then**
- 15: $C_{i,u,v} = Merge(C_{i,u}, C_{i,v})$; $C_i = C_i \cup \{C_{i,u,v}\}$;
- 16: $C_i = C_i \setminus \{C_{i,u}\}$; $C_i = C_i \setminus \{C_{i,v}\}$;
- 17: **end if**
- 18: **while** $found \neq 0$;
- 19: //Step 2: Select a data layout for each cluster based on its Inter-Cluster Access Similarity
- 20: $L_i = \emptyset$;
- 21: **for** $x = 1$ **to** $|C_i|$ **do**
- 22: **if** $IntraClusterAccessSimFunc(C_{i,x}, AASM) \geq \lambda$ **then**
- 23: $L_i = L_i \cup L^{\text{"row-store"}}(C_{i,x})$;
- 24: **else** $L_i = L_i \cup L^{\text{"column-store"}}(C_{i,x})$;
- 25: **end if**
- 26: **end for**
- 27: $G_i = (C_i, L_i)$;
- 28: **return** G_i ;

- Step 1:** This step aims at reducing the number of additional joins. To achieve this goal, it performs a repeated procedure of pair-wise merges of clusters. In particular, given a set of clusters (i.e., output of the clustering phase), each pair of clusters will be merged together if their Inter-Cluster Similarity is greater than or equal to a given merging threshold θ . In Algorithm 5, this step is presented in lines 1 - 18. Using the given set of clusters $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$, Step 1 will find a pair of clusters $C_{i,u}$ and $C_{i,v}$ where $C_{i,u} \in C_i$ and $C_{i,v} \in C_i$ ($u \neq v$) so that they satisfy the following merging criteria: the Inter-Cluster Access Similarity between $C_{i,u}$ and $C_{i,v}$ has the highest value among all possible pairs of clusters and this value is greater than or equal to θ (lines 3 - 13). Here, we implement the function *InterClusterAccessSimFunc*($C_{i,u}, C_{i,v}, AASM$) that applies Formula (4.4.6) and uses the matrix *AASM* to compute the Inter-Cluster Access Similarity between two clusters $C_{i,u}$ and $C_{i,v}$. We also define the function *Merge*($C_{i,u}, C_{i,v}$) that merges two clusters $C_{i,u}$ and $C_{i,v}$ together and returns a new cluster. Thus, if the merging criteria is satisfied, two clusters $C_{i,u}$ and $C_{i,v}$ are merged together to form a new cluster $C_{i,u.v}$. Then, $C_{i,u.v}$ is used to replace both $C_{i,u}$ and $C_{i,v}$ in C_i (lines 14 - 17). This procedure is repeated until we cannot find a pair of clusters that satisfy the above merging criteria.
- Step 2:** This step aims at reducing the number of irrelevant attribute accesses. To achieve goal, each of resulting clusters received from Step 1 is considered to determine whether it will be stored in a row store or a column store. In particular, we first compute the Intra-Cluster Access Similarity for each cluster, and then compare it with the given data layout threshold λ . To achieve this, we build the function *IntraClusterAccessSimFunc*($C_{i,x}, AASM$) that applies Formula (4.4.5) and uses the matrix *AASM* to compute the Intra-Cluster Access Similarity for the cluster $C_{i,x}$. If this Intra-Cluster Access Similarity is greater than or equal to λ , the corresponding cluster is stored in a row store (in this case, the attributes in the same cluster are very frequently accessed together); otherwise, a column store is used for it (in this case, most attributes in the same resulting cluster are not very frequently accessed together). In Algorithm 5, Step 2 is presented in lines 19 - 26. A list of suggested data layouts L_i for the corresponding clusters is created during this step. Finally, a candidate data storage configuration $G_i = (C_i, L_i)$ is returned as a result of the merging-selection phase (lines 27 and 28).

In short, given a set of inputs, Algorithm 1 first calls functions to compute matrices that represent the correlation and the similarity between attributes. Next, it calls Algorithm 4 for clustering attributes, and it calls Algorithm 5 for merging pairs of clusters and selecting suitable data layouts for them. Algorithm 5 will return a candidate data storage configuration $G_i = (C_i, L_i)$, where C_i represents a set of resulting clusters and L_i represents the corresponding data layouts. The values of parameters α , β , θ and λ can be chosen based on observations on experiments.

4.4.4 Examples

Given workload-and data-specific information, we will demonstrate the application of HADF to generate different data storage configurations for the same horizontal table.

We also perform a quantitative evaluation of results in terms of storage space and workload performance.

	a_1	a_2	a_3	a_4	a_5	a_6
q_1	0	1	1	1	1	1
q_2	0	0	1	1	1	1
q_3	0	0	0	1	1	1
q_4	1	1	0	0	0	0
q_5	1	1	1	0	0	0
q_6	0	1	1	0	0	0

	F
q_1	600
q_2	500
q_3	700
q_4	1000
q_5	200
q_6	400

UID	a_1	a_2	a_3	a_4	a_5	a_6
01	$v_{1,1}$	$v_{1,2}$	$v_{1,3}$	$v_{1,4}$	$v_{1,5}$	
02	$v_{2,1}$	$v_{2,2}$	$v_{2,3}$	$v_{2,4}$		$v_{2,6}$
03	$v_{3,1}$	$v_{3,2}$	$v_{3,3}$	$v_{3,4}$		
04	$v_{4,1}$	$v_{4,2}$	$v_{4,3}$	$v_{4,4}$		
05	$v_{5,1}$	$v_{5,2}$	$v_{5,3}$	$v_{5,4}$		
06	$v_{6,1}$	$v_{6,2}$	$v_{6,3}$	$v_{6,4}$		
07	$v_{7,1}$	$v_{7,2}$	$v_{7,3}$	$v_{7,4}$	$v_{7,5}$	
08	$v_{8,1}$	$v_{8,2}$	$v_{8,3}$	$v_{8,4}$	$v_{8,5}$	$v_{8,6}$
09	$v_{9,1}$	$v_{9,2}$	$v_{9,3}$			
10	$v_{10,1}$	$v_{10,2}$	$v_{10,3}$			

(a) Attribute Usage Matrix (AUM) (b) Query frequency (F) (c) Horizontal table T

Figure 4.15: Workload- and data-specific information of the horizontal table T

The workload-specific information is re-presented in the matrix AUM and the frequency F , shown in Figures 4.15(a) and (b), respectively, while the data-specific information is re-presented in the horizontal table T , given in Figures 4.15(c). (These information have been respectively given in Figure 4.6 and 4.7 in Section 4.3.2.) In this workload, some attribute access patterns can be expressed as follows: the query q_1 accesses five attributes a_2 , a_3 , a_4 , a_5 and a_6 with a frequency of 600; and the query q_2 accesses to four attributes a_3 , a_4 , a_5 and a_6 with a frequency of 500. Data is expressed in the horizontal table T with 10 tuples, each of which is presented in a single row; three attributes a_1 , a_2 and a_3 always have non-null values; the attribute a_4 has two null values; and two attributes a_5 and a_6 are very sparse.

In practice, in order to easily apply HADF, a user interface can be designed to enable users to explore different data storage configurations corresponding to different values of four parameters α , β , θ and λ . For instance, in [113], Sellam and Kersten introduced an user interface for cluster-driven navigation. Due to space limitations, below we only present three different data storage configurations created for T .

Data Storage Configuration 1: We create a baseline data storage configuration by storing the entire horizontal table T in a single row table. This configuration can be obtained by setting $\beta = 0$, $\lambda = 0$ and using arbitrary values for α and θ , e.g., $\alpha = 0$ and $\theta = 0$. The clustering phase produces two clusters $C_{1,1}$ and $C_{1,2}$:

- $C_{1,1} = \{UID, a_1, a_2\}$;
- $C_{1,2} = \{UID, a_3, a_4, a_5, a_6\}$.

Then, the merging-selecting phase merges the above two clusters into a single cluster $C_{1,1.2}$ and suggest to store it in a row store:

- $C_{1,1.2} = \{UID, a_1, a_2, a_3, a_4, a_5, a_6\} \Rightarrow \text{row store}$.

Figure 4.16 presents the single row table T_1 created to store the cluster $C_{1,1.2}$. Since only one table is used, no join is required to execute the given workload. However, the number of irrelevant attributes is relatively large.

UID	a_1	a_2	a_3	a_4	a_5	a_6
01	v1,1	v1,2	v1,3	v1,4	v1,5	
02	v2,1	v2,2	v2,3	v2,4		v2,6
03	v3,1	v3,2	v3,3	v3,4		
04	v4,1	v4,2	v4,3	v4,4		
05	v5,1	v5,2	v5,3	v5,4		
06	v6,1	v6,2	v6,3	v6,4		
07	v7,1	v7,2	v7,3	v7,4	v7,5	
08	v8,1	v8,2	v8,3	v8,4	v8,5	v8,6
09	v9,1	v9,2	v9,3			
10	v10,1	v10,2	v10,3			

Table T_1 (row store)**Figure 4.16:** Table created for Configuration 1

Particularly, we achieve the following statistics from the workload execution:

- The storage cost (in terms of the number of data cells): 70.
- NullRatio = 28.33% (the ratio between the total number of null values, i.e., 17, and the total number of possible values, except the UID attribute, i.e., 60).
- The total number of joins (between two tables over the given workload): 0.
- The total number of scanned data cells (over the given workload): 238,000.

Data Storage Configuration 2: The clustering phase is performed with the following settings: $\alpha = 0$ and $\beta = 0.4$. Thus, it only takes into account the impact of the data-specific information and creates three clusters $C_{2,1}$, $C_{2,2}$, and $C_{2,3}$:

- $C_{2,1} = \{UID, a_1, a_2, a_3, a_4\}$;
- $C_{2,2} = \{UID, a_5\}$;
- $C_{2,3} = \{UID, a_6\}$.

Now, we apply the merging-selecting phase with the settings: $\theta = 0.5$ and $\lambda = 0.6$. It keeps the cluster $C_{2,1}$, but merges $C_{2,2}$ and $C_{2,3}$ together into a new cluster $C_{2,2,3}$. Additionally, it suggests to store $C_{2,1}$ in a column store, but $C_{2,2,3}$ in a row store:

- $C_{2,1} = \{UID, a_1, a_2, a_3, a_4\} \Rightarrow$ column store;
- $C_{2,2,3} = \{UID, a_5, a_6\} \Rightarrow$ row store.

UID	a_1	a_2	a_3	a_4
01	v1,1	v1,2	v1,3	v1,4
02	v2,1	v2,2	v2,3	v2,4
03	v3,1	v3,2	v3,3	v3,4
04	v4,1	v4,2	v4,3	v4,4
05	v5,1	v5,2	v5,3	v5,4
06	v6,1	v6,2	v6,3	v6,4
07	v7,1	v7,2	v7,3	v7,4
08	v8,1	v8,2	v8,3	v8,4
09	v9,1	v9,2	v9,3	
10	v10,1	v10,2	v10,3	

(a) Table T_1 (column store)

UID	a_5	a_6
01	v1,5	
02		v2,6
07	v7,5	
08	v8,5	v8,6

(b) Table T_2 (row store)**Figure 4.17:** Two tables created for Configuration 2

Figure 4.17 provides two tables T_1 and T_2 created to store two clusters $C_{2,1}$ and $C_{2,2,3}$. The table T_1 is stored in a column store, whereas the table T_2 is stores in a row store. This configuration use less storage space than Configurations 1 because the most

two sparse columns a_5 and a_6 have been stored together in a separate table, i.e., T_2 , from which null rows are removed. This result shows that when we set the parameters to get the highest impact of the data-specific information on the clustering result, the number of null values is reduced. Besides, by storing the table T_1 in a column store, the given workload avoid accessing to irrelevant attributes. However, some queries such as q_1 , q_2 and q_3 require additional joins between two tables T_1 and T_2 .

This data storage configuration gives us the following statistics:

- *The storage cost: 62.*
- *NullRatio = 8.33%.*
- *The total number of joins: 1,800.*
- *The total number of scanned data cells: 71,600.*

Data Storage Configuration 3: The clustering phase is performed with the settings: $\alpha = 0.5$ and $\beta = 0.4$. Thus, it will take into consideration the combined impact of both the workload- and data-specific information and create two clusters $C_{3,1}$ and $C_{3,2}$:

- $C_{3,1} = \{UID, a_1, a_2, a_3\}$;
- $C_{3,2} = \{UID, a_4, a_5, a_6\}$.

Then, the merging-selecting phase is performed with the settings: $\theta = 0.5$ and $\lambda = 0.6$. The above two clusters are kept. The data layouts are suggested as follows:

- $C_{3,1} = \{UID, a_1, a_2, a_3\} \Rightarrow$ *column store*;
- $C_{3,2} = \{UID, a_4, a_5, a_6\} \Rightarrow$ *row store*.

UID	a_1	a_2	a_3
01	v1.1	v1.2	v1.3
02	v2.1	v2.2	v2.3
03	v3.1	v3.2	v3.3
04	v4.1	v4.2	v4.3
05	v5.1	v5.2	v5.3
06	v6.1	v6.2	v6.3
07	v7.1	v7.2	v7.3
08	v8.1	v8.2	v8.3
09	v9.1	v9.2	v9.3
10	v10.1	v10.2	v10.3

(a) Table T_1 (column store)

UID	a_4	a_5	a_6
01	v1.4	v1.5	
02	v2.4		v2.6
03	v3.4		
04	v4.4		
05	v5.4		
06	v6.4		
07	v7.4	v7.5	
08	v8.4	v8.5	v8.6

(b) Table T_2 (row store)

Figure 4.18: Two tables created for Configuration 3

Figure 4.18 presents two tables T_1 and T_2 created to store $C_{3,1}$ and $C_{3,2}$. The table T_1 is stored in a column store, whereas the table T_2 is stored in a row store. Compared to Configuration 2, the combined impact of both the workload- and data-specific information has helped Configuration 3 to reduce the number of null values as well as the number of additional joins at the same time.

This data storage configuration gives us the following statistics:

- *The storage cost: 72.*
- *NullRatio = 18.33%.*
- *The total number of joins: 1,100.*
- *The total number of scanned data cells: 107,600.*

In conclusion, we can say that HADF can provide good support for designing DICOM data. It is able to take into consideration the combined impact of both workload- and data-specific information on the quality of suggested data storage configurations. From the HADF-generated data storage configurations, we can choose a good one in terms of storage space and/or workload execution time. In Chapter 6, we will again analyze these impacts through experiments.

4.5 Summary and Conclusion

The characteristics of DICOM data and workloads have posed challenges on how to represent and manage data in a manner to reduce storage space demand and workload execution time. This chapter has presented the architecture of a novel hybrid storage model, called HYTORMO, and strategies for efficiently storing DICOM data.

The HYTORMO architecture is designed and built on top of an in-memory cluster computing framework, Spark, which can provide high performance, huge storage capability, scalability and elasticity. The combined use of row and column stores is to offer high performance for mixed workloads. DICOM data is organized based on the relational data model that facilitates the use of entity tables and SQL language.

The data storage strategy aims at reducing both storage space demand and workload execution time. The overall data storage strategy is based on the combined use of both vertical partitioning and a hybrid store in order to generate data storage configurations. To obtain a data storage configuration according to this data storage strategy, one of two different design approaches can be applied: *expert-based* and *automated*. The formal approach has been proposed by B. Mohamad, L. d'Orazio and L. Gruenwald [56, 57] where DICOM attributes are classified into three categories: *mandatory* and *frequently-accessed-together* attributes are stored in row store, whereas *optional/private/seldom-accessed* are stored in column store. In this thesis, we use the entity tables in the DICOM information model (e.g., *Patient*, *Study*, *Series*, etc.) as a starting point from which to create data storage configurations. However, when applying the expert-based design approach, it is difficult for experts to manually determine what attributes should be grouped together and what data layout should be applied for those column group so that both workload execution time and storage space demand are decreased, especially when the number of attributes is very large. To overcome this limitation, we formulate the *automated design problem* as the problem of selecting a data storage configuration to minimize both *storage cost* and *workload execution cost*. However, the solution search space for an optimal data storage configuration that minimizes both storage cost and execution cost is very large. Therefore, we proposed a hybrid automated design framework, called HADF.

HADF aims to support experts (e.g., database designers) in choosing good data storage configurations. It can fill the gaps between the workload-based and data-based partitioning approaches by taking into account the combined impact of both workload- and data-specific information as well as the use of a hybrid store. It includes two phases *clustering* and *merging-selecting*. The clustering phase is to reduce storage space demand and irrelevant attribute accesses. To achieve this, it groups high similar attributes in terms of Hybrid Similarity into the same clusters. The merging-selecting phase is to reduce both tuple reconstruction cost and irrelevant attribute accesses. It

contains two steps: first, the Inter-Cluster Access Similarity is used to determine whether a pair of clusters should be merged into a new cluster or not; then, the Inter-Cluster Access Similarity is applied to determine a suitable data layout for each cluster.

Besides, a suitable query processing strategy needs to be built on the top of HYTORMO. In the next chapter, we present in detail our approaches to create correct answers for queries and to improve the performance of the queries in distributed query processing environment.

Key Points

- We introduce a hybrid storage model, called HYTORMO.
- We introduce a data storage strategy: a mixed use of vertical partitioning and a hybrid store to reduce both storage space size and workload execution time.
- We present the application of the expert-based design approach.
- We provide a formal representation for the automated design approach.
- We describe the details of HADF.

Query Processing for HYTORMO

5.1 Overview

This chapter presents the proposed methods to improve the performance of queries for HYTORMO. An overview of the chapter is given in Table 5.1.

Table 5.1: Overview over Chapter 5

5.2 Query Rewriting	
5.2.1 Examples	5.2.2 Query Execution Plan
5.2.3 Determining Left-Outer Joins	5.2.4 Reducing the Number of Left-outer Joins
5.3 Intersection Bloom Filter	
5.3.1 Query Execution Plan with the IBF	5.3.2 Cost-effectiveness Analysis
5.3.3 Incremental Intersection Bloom Filter	
5.4 Summary and Conclusion	

In the previous chapter, on a high level, the query processing strategy built on top of HYTORMO was introduced. In general, entity tables in users' queries will be decomposed into sub-queries to be able to access relevant vertically partitioned tables. In order to correctly answer queries, in some cases, left-outer join operations are applied to prevent data loss that may occur if using only inner join operations. However, this may negatively impact on the query performance because this join type does not remove any tuple from their left tables. In this chapter, first of all, we analyze the cases where to use left-outer join operations. Next, we depict an execution plan. Then, we propose heuristic rules that are used to determine where to apply left-outer join operations and to reduce the number of left-outer join operations.

Besides, although the vertical partitioning and the hybrid store can help to improve the performance of queries by reducing I/O cost at attribute level (because of decreasing the number of irrelevant attribute accesses), they cannot reduce I/O cost at tuple level. A large number of irrelevant tuples are still read and propagated through a sequence of joins in queries before removed due to not satisfying join predicates. This can dramatically decrease performance of the queries due to expensive data transmission cost over the network in distributed query processing environments. This motivated us to integrate an IBF into query processing to reduce the size of intermediate results and network I/Os. We provide a cost-effectiveness analysis for the IBF. Additionally, we also propose an incremental IBF as an alternative to the IBF.

5.2 Query Rewriting

5.2.1 Examples

In our examples, the following four entity tables are used in users' queries: *Patient*, *Study*, *GeneralInfoTable* and *SequenceAttributes*. We assume that the expert-based design approach, described in Chapter 4, has been applied to generate data storage configurations (including schemas and their data layouts). The details of data storage configurations for these entity tables are as follows:

- **Patient**(UID^{RC}, PatientName^{Rm}, PatientID^{Rm}, PatientBirthDate^{Rm}, PatientSex^{Rm}, EthnicGroup^{Rm}, IssuerOfPatientID^C, PatientBirthTime^C, PatientInsurancePlanCode-Sequence^C, PatientPrimaryLanguageCodeSequence^C, PatientPrimaryLanguageModifier-CodeSequence^C, OtherPatientIDs^C, OtherPatientNames^C, PatientBirthName^C, PatientTelephoneNumbers^C, SmokingStatus^C, PregnancyStatus^{Rf}, LastMenstrualDate^{Rf}, PatientReligiousPreference^C, PatientComments^C, PatientAddress^C, PatientMotherBirthName^C, InsurancePlanIdentification^C)
- **Study**(UID^{RC}, StudyInstanceUID^{Rm}, StudyDate^{Rm}, StudyTime^{Rm}, ReferringPhysicianName^{Rm}, StudyID^{Rm}, AccessionNumber^{Rm}, StudyDescription^{Rm}, PatientAge^C, PatientWeight^C, PatientSize^C, Occupation^C, AdditionalPatientHistory^C, MedicalRecordLocator^C, MedicalAlerts^C)
- **GeneralInfoTable**(UID^{RC}, GeneralTags^C, GeneralVRs^C, GeneralNames^C, GeneralValues^C)
- **SequenceAttributes**(UID^{RC}, SequenceTags^{Rm}, SequenceVRs^{Rm}, SequenceNames^{Rm}, SequenceValues^{Rm})

In the above schemas, we use superscripts *Rm*, *Rf*, and *C* to denote that the corresponding attribute is stored in a *row table of mandatory attributes*, a *row table of frequently-accessed-together attributes* and a *column table of optional attributes*, respectively. Additionally, a superscript *RC* is used to denote that the corresponding attribute is stored in both row and column tables; however, in our DICOM data, only the attribute *UID* has been marked with *RC* because this attribute appears in all (vertically partitioned) tables to be used for tuple reconstruction. In any cases, all of these superscripts are hidden from end users.

According to the above suggested data storage configurations, in Table 5.2, we show the corresponding row and column tables used to store the above entity tables.

Table 5.2: Row and column tables used to store the entity tables

Entity	Row table of “Rm” attributes	Row table of “Rf” attributes	Column table of “C” attributes
<i>Patient</i>	<i>RowPatient</i>	<i>RowPregnancy</i>	<i>ColPatient</i>
<i>Study</i>	<i>RowStudy</i>	-	<i>ColStudy</i>
<i>GeneralInfoTable</i>	-	-	<i>ColGeneralInfoTable</i>
<i>SequenceAttributes</i>	<i>RowSequenceAttributes</i>	-	-

The schema of each entity table has not changed or has been decomposed into several vertically partitioned tables. Each vertically partitioned table is then suggested to be stored into a row or a column store: (1) The entity table *Patient* is decomposed into three vertically partitioned tables: *RowPatient*, *RowPregnancy* and *ColPatient*;

the first two tables are stored in a row store, whereas the last one is stored in a column store. (2) The entity table *Study* is decomposed into two vertically partitioned tables: *RowStudy* and *ColStudy*; the first one is stored in a row store while the second one is stored in a column store. (3) The two entity tables *GeneralInfoTable* and *SequenceAttributes* have not been decomposed; they are respectively stored in a column and a row store with names *ColGeneralInfoTable* and *RowSequenceAttributes*.

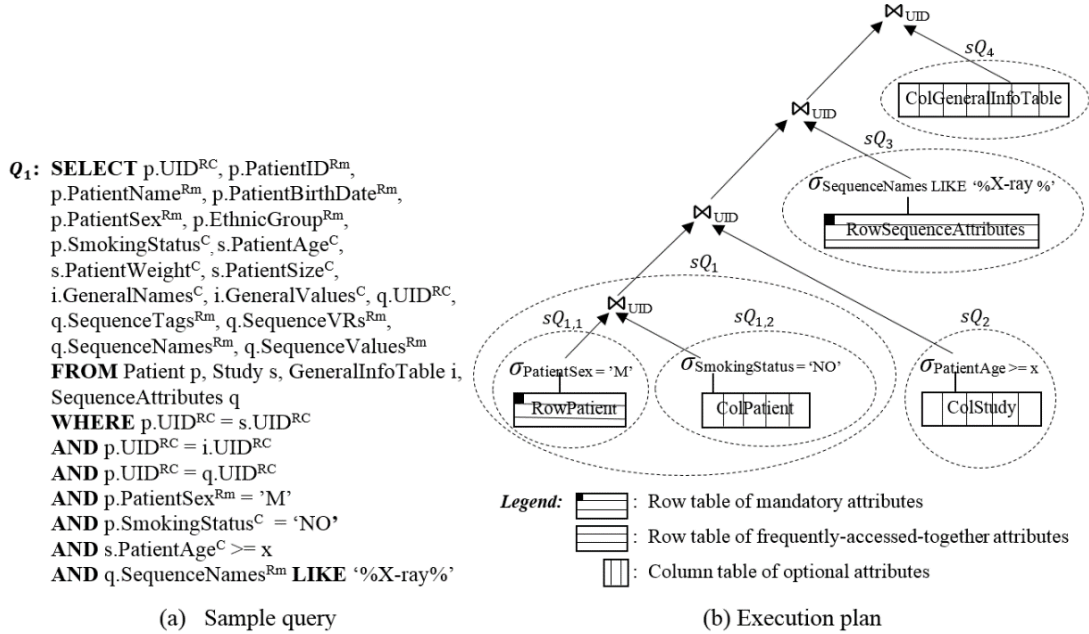


Figure 5.1: Representation of (a) the query Q_1 and (b) its execution plan tree

Figure 5.1(a) presents the query Q_1 which uses four entity tables: *Patient*, *Study*, *GeneralInfoTable* and *SequenceAttributes*. The attributes appearing in the SELECT and WHERE clauses are marked by superscripts *Rm*, *C*, and *RC* to indicate the corresponding data layout used to store the corresponding attribute (i.e., a row table, a column table or both types of tables). This query aims at retrieving the information stored in *X-ray* DICOM files of *non-smoking men* who are greater than or equal to *x years old*. The query has been based on TPC-H query 3 and 4 [114].

In Chapter 4, we mentioned that when a user writes a query, names of entity tables, e.g., *Patient*, *Study*, etc., are used. Then, HYTORMO will automatically decompose the query into multiple sub-queries so that each sub-query can only access relevant vertically partitioned tables. A left-deep sequential tree plan whose leaf nodes represent sub-queries is applied to execute the query. The join order is heuristically chosen in a way to keep intermediate results as small as possible. Once the execution plan is completely determined, the sub-queries will be processed using this plan. Finally their intermediate results will be integrated.

Applying the above query processing strategy, we achieve an execution plan for the query Q_1 , as shown in Figure 5.1(b). First, Q_1 is decomposed into a set of sub-queries sQ_1 , sQ_2 , sQ_3 , and sQ_4 accessing the entity tables *Study*, *Patient*, *SequenceAttributes* and *GeneralInfoTable*, respectively. Then, each of these sub-queries can be further decomposed into the deeper level sub-queries for directly accessing the underlying row and column tables stored in the hybrid store of

HYTORMO: sQ_1 is decomposed into $sQ_{1,1}$ and $sQ_{1,2}$ in order to access two tables *RowPatient* and *ColPatient*, respectively, because it uses only the attributes of these two tables; on the other hand, each of the sub-queries sQ_2 , sQ_3 , and sQ_4 is not decomposed, but is rewritten for only accessing the relevant tables *ColStudy*, *RowSequenceAttributes* and *ColGeneralInfoTable*, respectively. This execution plan tree is a left-deep processing tree in which relational operations are scheduled to be executed step by step while trying to keep intermediate results as small as possible. During the execution of a sequence of joins, the results of the sub-queries are joined together over the common join attribute *UID*. Finally, all the attributes listed behind the keyword *SELECT* of Q_1 will be presented in the final query result.

The Need to Use Left-Outer Join

The execution plan for query Q_1 in Figure 5.1(b) does not contain any left-outer join because every right-hand side table of join operations in this plan already belongs to one of three following cases: (1) It is either a row table of frequently-accessed-together attributes or a column table of optional attributes and there exist non-null constraints (predicates) on the attributes of these tables. For instance, there are non-null constraints on two attributes *SmokingStatus* and *PatientAge* of two tables *ColPatient* and *ColStudy*, respectively. (2) It is a row table of mandatory attributes (containing all values of the attribute *UID* of the entity table), e.g., *RowSequenceAttributes*. (3) It is the only sub-table that is decomposed from the entity table (containing the original schema of the entity table), e.g., *ColGeneralInfoTable*. For these three cases, the use of inner joins does not cause data loss in the results of the queries.

Table 5.3: Sample data of the table *RowPatient*

UID	PatientName	PatientID	PatientBirthDate	PatientSex	EthnicGroup
1440034811466	Smith	P40028	19610712	F	Whites
1440108680455	Muller	P40029	19500101	M	Whites
1440108686946	Young	P40030	19700509	M	Asians
1440108686950	Carol	P40031	19900122	(null)	(null)
1440108680459	Garcia	P40032	19990515	(null)	Blacks

Table 5.4: Sample data of the table *RowPregnancy*

UID	PregnancyStatus	LastMenstrualDate
1440108686950	4	20140212
1440108680459	4	20160511

However, in some other cases, several join operations in the execution plan need to be evaluated as left-outer join operations in order to prevent data loss in the results of the queries. For example, assume that sample data of two tables *RowPatient* and *RowPregnancy* (i.e., vertically partitioned tables) is given in Tables 5.3 and 5.4, respectively. Now, let us consider the user query Q_{2a} , shown in Figure 5.2(a), to view the content of the entity table *Patient*. Because there does not exist a physical table storing all the required attributes, in order to answer Q_{2a} , HYTORMO will decompose Q_{2a} into two sub-queries $sQ_{1,1}$ and $sQ_{1,2}$ to respectively access two tables *RowPatient* and *RowPregnancy*. Q_{2a} also needs to be rewritten into Q'_{2a} using a left-outer join, as shown in Figure 5.2(a). The corresponding execution plan is given in Figure 5(b).

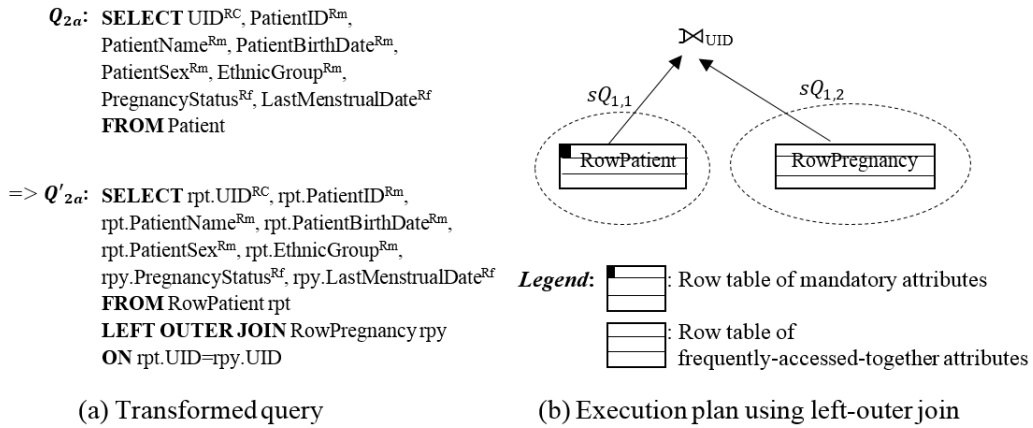


Figure 5.2: Transformation of the query Q_{2a} using a left-outer join

With the use of a left-outer join, Q_{2a} returns all the rows from the table *RowPatient* and just the rows from the table *RowPregnancy* in which the join predicate $rpt.UID = rpy.UID$ is satisfied. Additionally, rows from the left-hand side table that do not match any row in the right-hand side table will be still returned, but null values are inserted into each column of the right-hand side table. Table 5.5 shows the query result.

Table 5.5: Result of the query Q_{2a} when using a left-outer join

UID	PatientID	Patient-Name	Patient-BirthDate	Patient-Sex	Ethnic-Group	Pregnancy-Status	LastMenstrual-Date
1440108686950	P40031	Carol	19900122	(null)	(null)	4	20140212
1440108680459	P40032	Garcia	19990515	(null)	Blacks	4	20160511
1440108686946	P40030	Young	19700509	M	Asians	(null)	(null)
1440034811466	P40028	Smith	19610712	F	Whites	(null)	(null)
1440108680455	P40029	Muller	19500101	M	Whites	(null)	(null)

It is worth to note that if HYTORMO does not use a left-outer join for Q_{2a} , the query result cannot consist of many rows of the entity table *Patient* if the corresponding values of both attributes *PregnancyStatus* and *LastMenstrualDate* are null or empty. In Figure 5.3(a), we rewrite query Q_{2a} into Q''_{2a} using an inner join. Its corresponding execution plan is given in Figure 5.3(b). Table 5.6 presents the result of query Q_{2a} when using this execution plan. This is a wrong result.

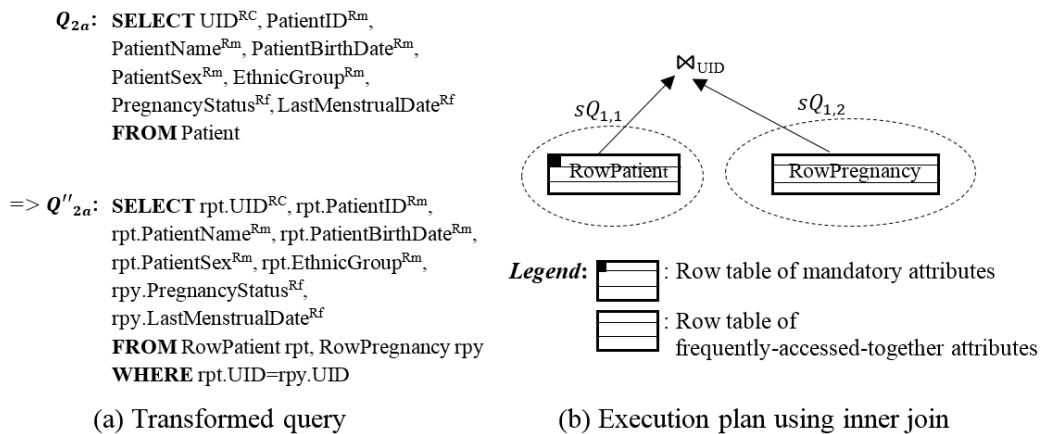


Figure 5.3: Transformation of the query Q_{2a} using an inner join

Table 5.6: The wrong result of the query Q_{2a} when using an inner join

UID	PatientID	Patient-Name	Patient-BirthDate	Patient-Sex	Ethnic-Group	Pregnancy-Status	LastMenstrual-Date
1440108686950	P40031	Carol	19900122	(null)	(null)	4	20140212
1440108680459	P40032	Garcia	19990515	(null)	Blacks	4	20160511

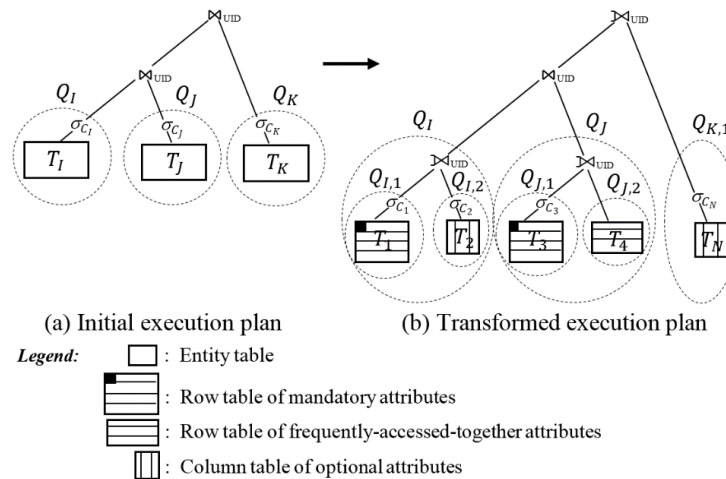
The reason of the result in Table 5.6 is that null rows are not stored in the table *RowPregnancy*, and thus when the query processing evaluates the join predicate $rpt.UID = rpy.UID$, there is not existing a $rpy.UID$ for a null row of the table *RowPregnancy*. Therefore, this execution plan will not applied to HYTORMO.

In short, HYTORMO uses a left-deep sequential tree plan to join intermediate results of sub-queries that access entity tables. However, these sub-queries are usually further decomposed into smaller sub-queries in order to access to relevant vertical partitioning tables. Thus, although join operations between entity tables are explicitly determined in users' queries, some join operations need to be rewritten to left-outer joins in order to avoid data loss caused by tuples discarded by only using inner joins.

Impact of Irrelevant Input Tuples on Query Performance

We can re-express the execution plan of query Q_1 , shown in Figure 5.1(b), in form of a join sequence: $Q_1 = (((sQ_{1,1} \bowtie_{UID} sQ_{1,2}) \bowtie_{UID} sQ_2) \bowtie_{UID} sQ_3) \bowtie_{UID} sQ_4$. In this join sequence, the execution of a sub-query is distributed across computer nodes and it is independent from others. The results of the sub-queries will be integrated during the execution of the join sequence. We assume that in this join sequence, a tuple t has been produced by first sub-query, $sQ_{1,1}$, and after that t is passed through the next two join operations $sQ_{1,1} \bowtie_{UID} sQ_{1,2}$ and $(...) \bowtie_{UID} sQ_2$. However, finally it is discarded since it does not satisfy a join predicate of the third join operation $(...) \bowtie_{UID} sQ_3$. It is clear that such propagations of t through the join sequence have caused a waste of disk and network I/Os. Besides, irrelevant data also cause many wasted CPU cycles. Hence, in order to improve query performance, irrelevant data should be discarded as early as possible.

5.2.2 Query Execution Plan


Figure 5.4: Execution plan transformation for the query Q

This section presents the query execution plan with main focus on how row and column tables are used in join operations. We describe this execution plan for a user query Q , which is represented in a general form, given in Figure 4.4 in Section 4.2.3.

Figure 5.4(a) presents the initial execution plan for Q . Here, Q is decomposed into sub-queries Q_I , Q_J and Q_K to respectively access entity tables T_I , T_J and T_K . (In DICOM data, these entity tables may be *Patient*, *Study*, *Series*, etc.) This execution plan can be mathematically written as $Q = Q_I \bowtie_{UID} Q_J \bowtie_{UID} Q_K$. As such, the names of the entity tables are used in the user query, and the types of join operations between any two entity tables are also explicitly identified by the user. In this example, the user is using only inner joins to join three entity tables T_I , T_J and T_K together. We assume that each of the entity tables has been vertically partitioned into several sub-tables and stored in row or column stores by applying the expert-based design approach or the automated design approach, as introduced in Chapter 4. We also assume that only few attributes of the entity tables are used by Q such that the sub-queries Q_I , Q_J and Q_K need to be further decomposed into smaller sub-queries $Q_{I,1}$, $Q_{I,2}$, $Q_{J,1}$, $Q_{J,2}$ and $Q_{K,1}$ to only access the sub-tables containing the attributes relevant to Q . Figure 5.4(b) presents the transformed execution plan for Q : $Q_{I,1}$ and $Q_{I,2}$ access respectively T_1 and T_2 (which are sub-tables of T_I); $Q_{J,1}$ and $Q_{J,2}$ access respectively T_3 and T_4 (which are sub-tables of T_J); similarly, $Q_{K,1}$ accesses only T_N (which is a sub-table of T_K). Some join operations between sub-tables need to be evaluated as left-outer joins.

In a nutshell, the query processing strategy can be described as follows: *HYTORMO will decompose the user query using entity tables into sub-queries to be able to access necessary row and column tables. A left-deep sequential tree plan is applied. Some join operations between the result tables of the sub-queries need to be evaluated as left-outer joins to prevent data loss caused by the tuples discarded by inner joins. HYTORMO will automatically determine the types of join operations.*

5.2.3 Determining Left-Outer Joins

We propose the heuristic rules to determine when a left-outer join is used.

Rule 1: *In a join between two sub-tables of the same entity table, if the left-hand side table is a row table of mandatory attributes while the right-hand side table is either a column table of optional attributes or a row table of frequently-accessed-together attributes, this join needs to be evaluated as a left-outer join.*

In Figure 5.4(b), both sub-queries $Q_I = Q_{I,1} \bowtie_{UID} Q_{I,2}$ and $Q_J = Q_{J,1} \bowtie_{UID} Q_{J,2}$ are evaluated as left-outer joins because $Q_{I,1}$ and $Q_{J,1}$, respectively, access two row tables of mandatory attributes T_1 and T_3 while $Q_{I,2}$ and $Q_{J,2}$ access a column table of optional attributes T_2 and a row table of frequently-accessed-together attributes T_4 , respectively.

Rule 2: *In a join between two entity tables, if the right-hand side table has been replaced with a sub-table that is either a row table of frequently-accessed-together or a column table of optional attributes (because the user query uses only the attributes from this sub-table) and this sub-table is not the only child of its parent table, this join needs to be evaluated as a left-outer join.*

For instance, in the query $Q = Q_I \bowtie_{UID} Q_J \bowtie_{UID} Q_K$, given in Figure 5.4(a), we focus on the join operation related to Q_K , i.e., $(...) \bowtie_{UID} Q_K$. Q_K has been changed (rewritten) to $Q_{K,1}$ accessing the column table of optional attributes, T_N . Assume that T_N is not the only child of its parent table, T_K , by applying Rule 2, the join operation using the result of $Q_{K,1}$ will be rewritten to a left-outer join, as illustrated in Figure 5.4(b).

As such, Rule 1 is applied to consider a join operation between two sub-tables of the same entity table. On the other hand, Rule 2 is applied to consider a join operation between two entity tables in which the right-hand side table has been changed to a sub-table. In the scope of our study, we only concern on the execution plans using the inner joins and the above-mentioned two cases of left-outer joins. Optimization for queries with left-outer joins can be referenced in [115].

5.2.4 Reducing the Number of Left-Outer Joins

In the previous section, we introduced two heuristic rules, Rule 1 and 2, to determine whether a join operation needs to be evaluated as a left-outer join or not. In order to improve the query performance, the number of left-outer joins should be minimized as small as possible. Below, we introduce another heuristic rule - Rule 3, used for deciding whether or not a left-outer join should be rewritten to an inner join:

Rule 3: *Given a left-outer join $T_1 \bowtie_{UID} T_2$, if there are not any non-null constraints on attributes of the right-hand side table T_2 , this left-outer join should be rewritten to an inner join in order to improve query performance.*

This heuristic rule is based on the fact that, in the left-outer join $T_1 \bowtie_{UID} T_2$, if there are not any non-null constraints on attributes of T_2 , the left-outer join returns all the matching tuples between T_1 and T_2 , like an inner join. Additionally, the unmatched tuples are also preserved from T_1 and are supplied with nulls for the attributes from T_2 . Thus, in this case, the left-outer join is kept (no change). However, if there is a non-null constraint on an attribute of the right-hand-side table, i.e., T_2 , this constraint must be evaluated to be TRUE to form a tuple in the query result. They also remove any null rows from T_2 . Therefore, in this case, it is unnecessary to use left-outer join. The join operation should be rewritten to an inner join.

Let us consider the user query Q_{2b} as shown in Figure 5.5(a), which will display information about *Patient*, including *UID*, *PatientID*, *PatientName*, *PatientBirthDate*, *PatientSex*, *EthnicGroup*, *PregnancyStatus* and *LastMenstrualDate*. This query is similar to the query Q_{2a} , given in Figure 5.2, but has a constraint *LastMenstrualDate* \geq '2016' in WHERE clause to find all the patients whose *LastMenstrualDate* from year 2016 onwards. There does not exist a physical table with all the attributes relevant to this query, thus Q_{2b} is decomposed into two sub-queries $sQ_{1,1}$ and $sQ_{1,2}$ to access two vertically partitioned tables *RowPatient* and *RowPregnancy*, respectively, to obtain the required attributes. Q_{2b} is written to Q'_{2b} , presented in Figure 5.5(a), which uses an left-outer join according to Rule 1. The corresponding execution plan tree is given in Figure 5.5(b).

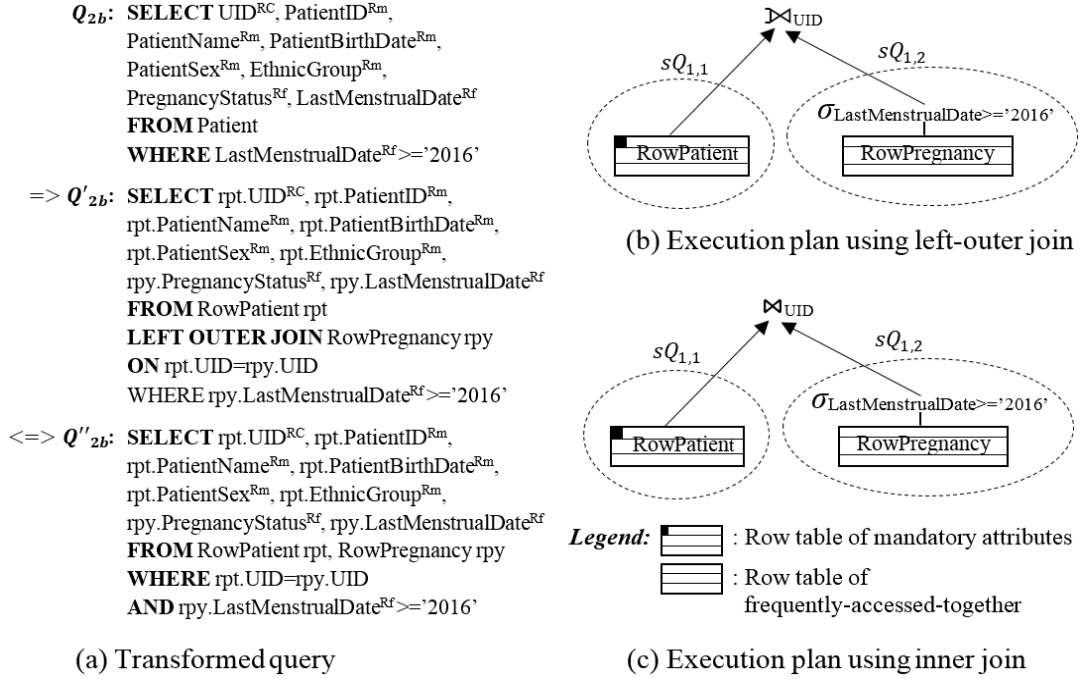


Figure 5.5: Transformation of the query Q_{2b} to two equivalent execution plans

Using the execution plan given in Figure 5.5(b), the results of two sub-queries $sQ_{1,1}$ and $sQ_{1,2}$ is joined together by using a left-outer join. The constraint $LastMenstrualDate \geq '2016'$ in WHERE clause must be evaluated to TRUE to form a row in the result of Q_{2b} . If there is an unmatched row in this left-outer, the columns from the right-hand side table, i.e., $RowPregnancy$, are inserted by null values. That is, on the unmatched rows, the column $LastMenstrualDate$ also gets null values that cannot make the constraint in WHERE clause become TRUE. Thus, those unmatched rows will be removed from the result of Q_{2b} . Clearly, in this case, a left-outer join is unnecessary, and Q'_{2b} should be rewritten to Q''_{2b} , as shown in Figure 5.5(a), which uses an inner join as suggested by Rule 3. Its corresponding execution plan tree in Figure 5.5(c). Table 5.7 presents the correct result of query Q_{2b} . (Both execution plans given respectively in Figures 5.5(b) and (c) give the same result.)

Table 5.7: Correct result of the query Q_{2b}

UID	PatientID	Patient-Name	Patient-BirthDate	Patient-Sex	Ethnic-Group	Pregnancy-Status	LastMenstrual-Date
1440108680459	P40032	Garcia	19990515	(null)	Blacks	4	20160511

In general, given an execution plan, we will apply Rule 1, then Rule 2, followed by Rule 3. Below we present how the execution plan of the user query, $Q = Q_I \bowtie_{UID} Q_J \bowtie_{UID} Q_K$, is transformed when applying these rules. Figure 5.6(a) presents the execution plan tree of Q over three entity tables T_I , T_J and T_K . Because these entity tables has been decomposed into several sub-tables and stored in row or column stores, Q needs to be decomposed into sub-queries Q_I , Q_J and Q_K which then are further decomposed into smaller sub-queries $Q_{I,1}$, $Q_{I,2}$, $Q_{J,1}$, $Q_{J,2}$ and $Q_{K,1}$ to be able to access, respectively, the row and column tables, T_1 , T_2 , T_3 , T_4 and T_N ,

containing attributes relevant to Q . Besides, some join operations need to be evaluated as left-outers: (1) by applying Rule 1 to consider joins between two sub-tables, we determine two left-outer joins: $Q_{I,1} \bowtie_{UID} Q_{I,2}$ and $Q_{J,1} \bowtie_{UID} Q_{J,2}$; and (2) by applying Rule 2 to consider join operations between two entity tables, we determine a left-outer join for third join: $(Q_I \bowtie_{UID} Q_J) \bowtie_{UID} Q_{K,1}$, as shown in Figure 5.6(b).

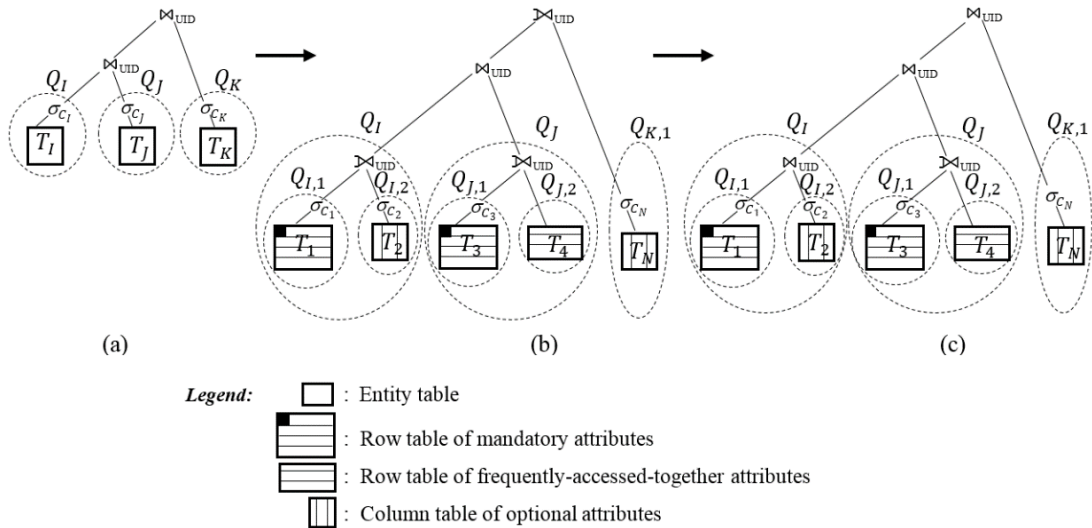


Figure 5.6: Transformation of the execution plan after applying Rule 3

Furthermore, we apply the Rule 3 to transform the execution plan tree in Figure 5.6(b) to the one in Figure 5.6(c) as follows: First, we check whether there exist non-null constraints on the attributes of the right-hand side table of each left-outer join. Here, we assume that σ_{C_2} and σ_{C_N} are non-null constraints on the attributes of the tables T_2 and T_N , respectively (as shown in Figure 5.6(b)). Thus, we replace two left-outer joins $Q_{I,1} \bowtie_{UID} Q_{I,2}$ and $(Q_I \bowtie_{UID} Q_J) \bowtie_{UID} Q_{K,1}$ with two inner joins $Q_{I,1} \bowtie_{UID} Q_{I,2}$ and $(Q_I \bowtie_{UID} Q_J) \bowtie_{UID} Q_{K,1}$, respectively (as shown in Figure 5.6(c)).

5.3 Intersection Bloom Filter

In this section, we first describe how to integrate an IBF into the query processing. Then we present in detail the cost-effectiveness analysis of the IBF. Finally, we introduce an alternative form of the IBF, called incremental IBF.

5.3.1 Query Execution Plan with the IBF

In Chapter 3, we presented background about BF and IBF which are used to remove irrelevant tuples out of input tables of join operations. In this section, we propose a method to integrate an IBF into the query processing strategy built on top of HYTORMO. An IBF is used instead of BFs because of its benefits. For instance, its error probability is significantly less than that of the BF, or the application of the IBF in a distributed query processing environment can reduce more network I/Os than the BF [23-25, 116]. In our study, the application of the IBF is considered in two phases, namely (1) *build phase*, where the IBF is built for input tables, and (2) *probe phase*,

where the IBF is applied to remove irrelevant tuples from input tables of multiple-table join queries. In the followings, we depict how to perform these phases.

In order to avoid loss of generality, we consider the integration of the IBF into the query processing for a user query Q which is written in a general form supported by HYTORMO, as presented in Figure 4.4 in Section 4.2.3: Q is a multi-way join query on common join attributes. We assume that Q can be decomposed into a set of sub-queries Q_I , Q_J and Q_K , each of which can be further decomposed into smaller sub-queries to be able to access, respectively, the underlying row and column tables, T_1 , T_2 , ..., T_N , containing the attributes relevant to Q . Because HYTORMO has used a *left-deep sequential tree plan*, we focus on the application of the IBF to this query execution plan. Although the underlying input tables T_1, T_2, \dots, T_N might have some common join attributes, in the scope of our study, we assume that these tables share only a common join attribute UID (which is an unique identification attribute in DICOM database tables). Under this assumption, we can build and probe a common IBF on the join attribute UID of the input tables.

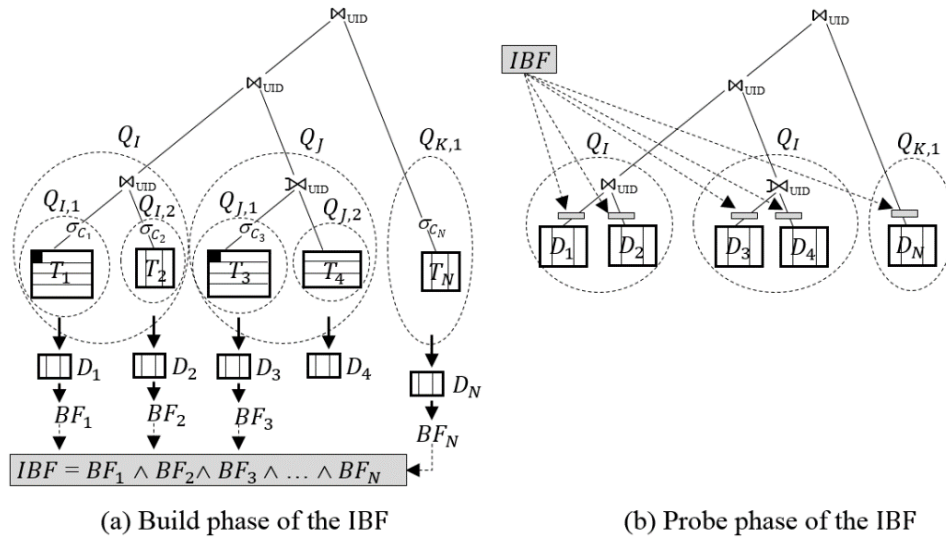


Figure 5.7: Query execution plan with the IBF

Figure 5.7(a) and (b) describes the build and probe phases of the IBF, respectively. First of all, we assume that three heuristic rules, Rules 1, 2 and 3 (as introduced in the previous section), have been applied to determine the suitable join types and to reduce the number of left-outer joins in the query execution plan. This results in the execution plan tree as shown in Figure 5.7(a). Then, in order to build the IBF for this execution plan tree, we need to compute a set of Bloom filters BF_i 's (using the same configuration: the same size and the same set of hash functions) on the join attribute UID for the intermediate result tables D_1, D_2, D_3, D_4 and D_N , which are generated as the results of the sub-queries $Q_{I,1}, Q_{I,2}, Q_{J,1}, Q_{J,2}$ and $Q_{K,1}$, respectively. After obtaining the set of BF_i 's, the IBF is computed by applying bitwise AND operations on these BFs. It is worth noting that, in our study, a Bloom filter is only applied for inner joins. Hence, during the build phase of the IBF, we do not create a Bloom filter for the right-hand side table of a left-outer join if there does not exist any non-null constraint on the attributes of this table. For instance, we do not create a BF for the table D_4 (i.e., the

result of $Q_{J,2}$), as shown in Figure 5.7(a), because it is a right-hand side table of a left-outer join and there are not any non-null constraints on its attributes. This is due to the fact that $Q_{J,1} \bowtie_{UID} Q_{J,2}$ is not equivalent to $Q_{J,1} \bowtie_{UID} Q_{J,2}$, thus using a BF for D_4 will cause data loss caused by tuples discarded when *ANDing* this BF with others in order to compute the IBF. Once the IBF is completely computed, the probe phase of the IBF is started, as given in Figure 5.7(b). The IBF is applied to filter irrelevant input tuples out of the input tables of joins before these joins are performed. It is worthy to note that although a BF have not built on the table D_4 , the IBF is still applied to this table.

5.3.2 Cost-effectiveness Analysis

In this section, we provide a cost-effectiveness analysis of the IBF when applied for HYTORMO. Our objective is to evaluate the benefit of the IBF in terms of query performance. Although there exist several research works that attempted to integrate the BFs in the processing of distributed queries [117] and MapReduce framework [118-120], our application context differs from theirs since we use the IBF instead of the BFs. Besides these research works, P. Koutris [116] theoretically made cost-effectiveness analyses of using the BFs within a single MapReduce but did not provide specific detail costs in practice; moreover, the impacts of the BFs on the execution cost of particular operations, such as disk and network I/Os, have not been evaluated clearly. More recently, the authors in [25] proposed several approaches for integrating the IBF into the MapReduce framework. They also presented cost models for join operations for the application of the IBF in MapReduce environment. However, in the context of HYTORMO, the query processing is performed on top of an in-memory cluster computing framework, called Spark, where the detailed execution of Map and Reduce phases will not concerned, instead we mainly focused on join operations to integrate the intermediate result tables. Therefore, we need to determine how to perform build and probe phases of the IBF and how to build cost models that provide detailed analysis of disk and network I/Os corresponding to this context.

Because there are many cases in which the IBF can be applied, in our study, we focus on the cases where the IBF is used for a sequential join sequence of N tables joined. We assume that the IBF is created by applying bitwise AND operations on the BFs on all input tables. Additionally, although the type of each join operation in the join sequence may be either an inner join or a left-outer join, to make the cost models simple, we assume that all left-outer join operations in the join sequence have been successfully transformed to the corresponding inner join operations, i.e., the join sequence now only consists of inner join operations. Formally, with these assumptions, we will provide a cost-effectiveness analysis of the application of the IBF to execute the multi-way join query $Q = D_1 \bowtie_{UID} D_2 \bowtie_{UID} \dots \bowtie_{UID} D_N$, where D_1, D_2, \dots, D_N are input tables. Besides, since the execution cost of the query Q depends on the processing order of its input tables, we assume that $|D_i| \leq |D_{i+1}|$, where $i \in [1, N - 1]$, such that the join sequence of the input tables can be expressed as

$$Q = (((D_1 \bowtie_{UID} D_2) \bowtie_{UID} \dots) \bowtie_{UID} D_{N-1}) \bowtie_{UID} D_N.$$

The left-deep sequential tree plan with the application of the IBF for the above join sequence is presented in Figure 5.8. The input tables D_1, D_2, \dots, D_N and the

intermediate result tables I_1, I_2, \dots, I_{N-1} are used as inputs of join operations. Here, we are setting $I_1 = D_1$ and $I_N = \langle \text{final query result} \rangle$.

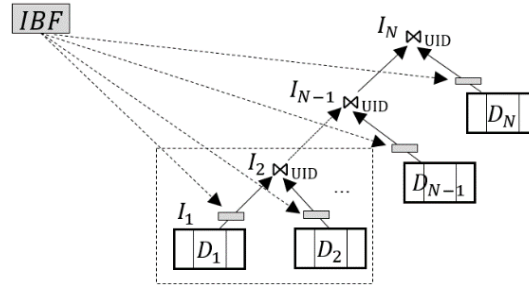


Figure 5.8: Left-deep sequential execution plan with the application of the IBF

Definitions and Basic Mathematical Concepts

Before building the cost models, we provide some definitions and basic mathematical concepts related to the configuration of the IBF.

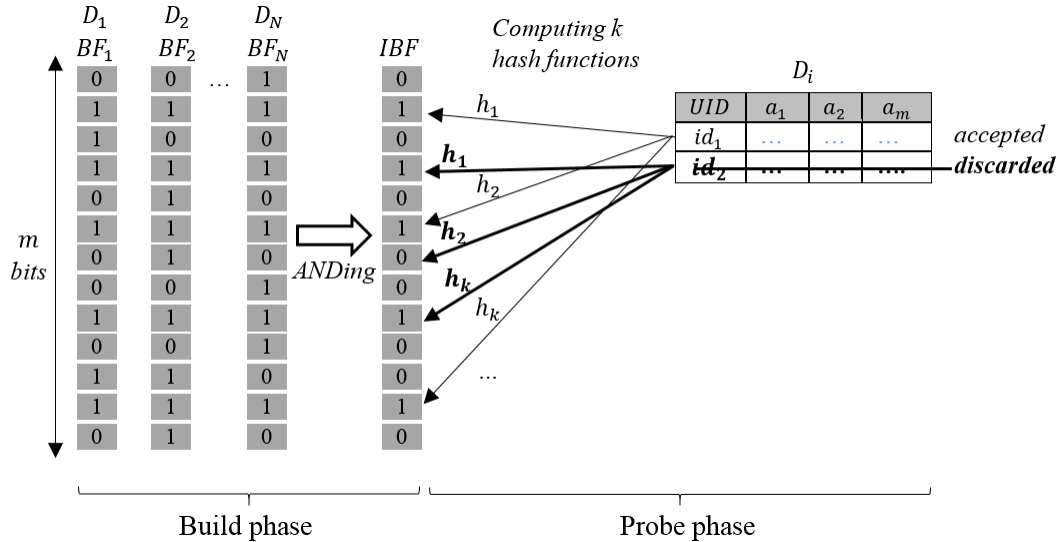


Figure 5.9: Phases of the IBF with component BF_i 's and hash functions

The build and probe phases of the IBF are quite simple and can be briefly described as follows: First of all, the IBF is computed by *ANDing* all BF_i 's created on the join attribute UID of the input tables, as illustrated in Figure 5.9. Next, the IBF is probed to filter the irrelevant input tuples out of these input tables. In this phase, the following steps are performed: checking the membership of a value v of the join attribute UID in each input table D_i , where $i \in [1, N]$, will only require to compute k hash functions and to access k bits of the IBF; if the value v makes all of k hash functions $h_1(v)$, $h_2(v)$, ..., $h_k(v)$ true ($= 1$), the corresponding tuple is accepted; otherwise it is discarded from the input table D_i . Figure 5.9 shows that when checking the input table D_i , value $v = id_1$ makes all of k hash functions true, so the corresponding tuple is accepted. In contrast, value $v = id_2$ does not make all of k hash functions true, e.g., $h_2(id_2) = 0$, thus the corresponding tuple is discarded.

In Table 5.8, we summarize notations used in the cost models.

Table 5.8: Notations

Notation	Explanation
D_i	Table that is used as either a build or a probe table.
I_i	Intermediate result table of the sequential join sequence.
BF_i	Bloom filter that is built on the table D_i .
IBF	Intersection Bloom filter.
ρ_{D_j, D_i}	Selectivity of the table D_j on the table D_i in the join $D_i \bowtie_{UID} D_j$.
ρ_{BF_j, D_i}	Selectivity of the BF_j that is built on the table D_j and then is probed on the table D_i .
ρ_{IBF, D_i}	Selectivity of the IBF that is computed on the input tables D_1, D_2, \dots, D_N and probed on the table D_i .
P_{BF_i}	False positive (due to hash collisions) of the BF_i that is built on the table D_i .
P_{IBF}	False positive of the IBF that is computed from the Bloom filters BF_1, BF_2, \dots, BF_N of the input tables D_1, D_2, \dots, D_N , respectively.

Probability of a false positive of the Bloom filter

The *probability of a false positive* of a Bloom filter BF_i due to hash collisions can be computed by Formula (5.3.1) [24].

$$P_{BF_i} = (1 - (1 - m^{-1})^{kn_i})^k \approx \left(1 - e^{-\frac{kn_i}{m}}\right)^k, \quad (5.3.1)$$

where the Bloom filter BF_i represents a set of n_i values (i.e., the number of values of the join attribute *UID* of the input table D_i) in a vector of m bits and using k independent hash functions. The value of P_{BF_i} ranges from 0 to 1.

According to [24], to store a set of n_i values in a m -bit Bloom filter BF_i , the larger m the smaller probability of a false positive is. If the m is fixed, in order to minimize the probability of a false positive, we can choose the number of hash functions k and the minimum probability of the false positive P_{BF_i} by using the Formulas (5.3.2) and (5.3.3), respectively:

$$k = \ln(2) \times \frac{m}{n_i}. \quad (5.3.2)$$

$$P_{BF_i} = \left(\frac{1}{2}\right)^k = (0.6185)^{m/n_i}. \quad (5.3.3)$$

Furthermore, based on Formulas (5.3.2) and (5.3.3), we conduct Formula (5.3.4) to compute the number of bits m needed for the Bloom filter BF_i :

$$m = \frac{-n_i \times \ln(P_{BF_i})}{(\ln(2))^2}. \quad (5.3.4)$$

For example, given a set of 200,000 values ($n_i = 200,000$) and an acceptable probability of false positive $P_{BF_i} = 0.01$, the number of bits m and the number of hash functions k (required to achieve such a probability of the false positive P_{BF_i} of the BF_i) can be respectively computed using Formulas (5.3.4) and (5.3.2) as follows:

- $m = \frac{-200,000 \times \ln(0.01)}{(\ln(2))^2} = 1,917,011.68$ bits (1,917,012 bits \approx 2 MB).
- $k = \ln(2) \times \frac{1,917,012}{200,000} = 6.64$ hash functions (7 hash functions)

Selectivity of a Bloom filter

We define the selectivity of a Bloom filter BF_j that is created from the input table D_j and then probed on the input table D_i is the probability in which a tuple t will be accepted by the Bloom filter BF_j . This selectivity is computed by Formula (5.3.5):

$$\rho_{BF_j, D_i} = \rho_{D_j, D_i} + (1 - \rho_{D_j, D_i}) \times P_{BF_j}, \quad (5.3.5)$$

where:

- ρ_{D_j, D_i} : selectivity of the table D_j on the table D_i in the join $D_i \bowtie_{UID} D_j$;
- P_{BF_j} : error probability of the Bloom filter BF_j that is created from the table D_j ;
- $(1 - \rho_{D_j, D_i}) \times P_{BF_j}$: fraction of tuples from the probe table D_i that are not discarded by BF_j and do not join with any tuples in the build table D_j ; we set $\rho_{D_j, D_i} + (1 - \rho_{D_j, D_i}) \times P_{BF_j} = 1$ when $j = i$.

Selectivity of an Intersection Bloom filter

The selectivity of the IBF, which is built from a set of Bloom filters BF_j 's of the input tables D_j 's, where $j \in [1, N]$, and then probed on each input table D_i , can be determined by Formula (5.3.6):

$$\rho_{IBF, D_i} = \prod_{j=1}^N \rho_{BF_j, D_i}, \quad (5.3.6)$$

The selectivity of the IBF is regarded as the probability in which a tuple t of the input table D_i will be accepted by all of its component Bloom filters BF_j 's ($j \in [1, N]$).

Comparing two Formulas (5.3.5) and (5.3.6), it is easy to see that ρ_{IBF, D_i} is usually much less than ρ_{BF_j, D_i} . This means that using an IBF can help to remove more irrelevant tuples than using just a single Bloom filter.

False positive of an Intersection Bloom filter

We can compute the *false positive* P_{IBF} of the IBF by Formula (5.3.7) as follows:

$$P_{IBF} = \prod_{i=1}^N P_{BF_i} = \prod_{i=1}^N (1 - e^{-kn_i/m})^k, \quad (5.3.7)$$

where N is the number of component Bloom filters BF_i 's with an assumption that there exists a BF_i on each table D_i .

A comparison between Formula (5.3.1) and Formula (5.3.7) shows that P_{IBF} is much less than P_{BF_i} . This implies that applying an IBF offers a lower amount of false positive errors than only applying a single Bloom filter.

Network I/O cost and disk I/O cost

The execution cost of a multi-way join query in a cluster is usually determined by *network I/O cost* and *disk I/O cost*, we thus will use these costs to analysis the cost-effectiveness of the IBF.

In order to estimate the *network I/O cost* and the *disk I/O cost*, we depend on the steps performed in the build and probe phases of the IBF, as shown in Figures 5.7(a) and (b). These steps include:

- Step 1.** Execute sub-queries to create intermediate result tables D_1, D_2, \dots, D_N .
- Step 2.** Compute BF_1, BF_2, \dots, BF_N on values of UID's of the input tables D_1, D_2, \dots, D_N , respectively.
- Step 3.** Compute $IBF = BF_1 \wedge BF_2 \wedge \dots \wedge BF_N$ (each \wedge is a binary bitwise AND operator).
- Step 4.** Broadcast the IBF to all slave nodes of the cluster.
- Step 5.** Apply the IBF to the input tables D_1, D_2, \dots, D_N to obtain the filtered input tables $D_{1(filtered)}, D_{2(filtered)}, \dots, D_{N(filtered)}$.
- Step 6.** Execute the sequential join sequence using the filtered input tables $D_{1(filtered)}, D_{2(filtered)}, \dots, D_{N(filtered)}$ as input tables.

The first three steps are in the build phase, whereas the rest of the steps are in the probe phase. We assume that the first step has been performed and we will start to estimate the costs from the second step.

Network I/O Cost: Since each join operation in the sequential join sequence will join an intermediate result table (created by the previous join operation) with an input table D_i , the network I/O cost when the IBF is not used, C_{Net}^{NoIBF} , can be computed by Formula (5.3.8):

$$C_{Net}^{NoIBF} = \sum_{i=1}^N size(D_i) + \sum_{i=1}^{N-1} size(I_i) \times size(D_{i+1}) \times \rho_{D_{i+1}, I_i}, \quad (5.3.8)$$

where:

- $size(D_i) = |D_i| \times size(tuple_{D_i})$: size of the input table D_i ;
- $\sum_{i=1}^N size(D_i)$: cost of sending the input tables;
- ρ_{D_{i+1}, I_i} : selectivity of the table D_{i+1} on the table I_i in the join $I_i \bowtie_{UID} D_{i+1}$;
- $size(I_i) = |I_i| \times size(tuple_{I_i})$: size of the intermediate result table I_i ;
- $\sum_{i=1}^{N-1} size(I_i) \times size(D_{i+1}) \times \rho_{D_{i+1}, I_i}$: cost of sending the intermediate results.

The cost C_{Net}^{NoIBF} consists of cost of sending the input tables and the intermediate result tables over the network. Assume that no replication is done on the input tables.

The *network I/O cost when the IBF is used*, C_{Net}^{IBF} , is computed by Formula (5.3.9):

$$C_{Net}^{IBF} = c \times size(IBF) + \sum_{i=1}^N size(D_{i(filtered)}) + \sum_{i=1}^{N-1} size(I_i) \times size(D_{i+1(filtered)}) \times \rho_{D_{i+1(filtered)}, I_i}, \quad (5.3.9)$$

where:

- $c \times size(IBF)$: cost of sending the IBF to c slave nodes;
- $size(D_{i(filtered)}) = |D_{i(filtered)}| \times size(tuple_{D_i})$: size of the filtered input table D_i ;
- $size(I_i)$: size of the intermediate result table I_i .

The cost C_{Net}^{IBF} consists of the cost of sending (broadcast) the IBF to all slave nodes of the cluster and the cost of sending the filtered input tables and intermediate result tables over the network. Here, we do not apply the IBF to filter intermediate results.

A comparison between Formula (5.3.8) and Formula (5.3.9) shows that $c \times size(IBF)$ is usually small and $size(D_{i(filtered)}) \ll size(D_i)$; therefore, C_{Net}^{IBF} is usually less than C_{Net}^{NoIBF} .

Disk I/O Cost: The disk I/O cost without using the IBF, $C_{I/O}^{NoIBF}$, is computed by Formula (5.3.10).

$$C_{I/O}^{NoIBF} = \sum_{i=1}^{N-1} [size(I_i) + size(D_{i+1})] + \sum_{i=2}^N size(I_i), \quad (5.3.10)$$

where:

- $\sum_{i=1}^{N-1} [size(I_i) + size(D_{i+1})]$: cost of reading the inputs for join operations;
- $\sum_{i=2}^N size(I_i)$: cost of writing the intermediate results of join operations to disks (here, we are setting $I_1 = D_1$).

The *disk I/O cost with the use of the IBF*, $C_{I/O}^{IBF}$, is computed by Formula (5.3.11).

$$C_{I/O}^{IBF} = 2 \times \sum_{i=1}^N size(D_i) + \sum_{i=1}^N size(D_{i(filtered)}) + \sum_{i=1}^{N-1} [size(I_i) + size(D_{i+1(filtered)})] + \sum_{i=2}^N size(I_i), \quad (5.3.11)$$

where:

- $2 \times \sum_{i=1}^N size(D_i)$: cost of reading the input tables two times (to build and probe the IBF);
- $\sum_{i=1}^N size(D_{i(filtered)})$: cost of writing the filtered input tables to disks after they are filtered by using the IBF;
- $\sum_{i=1}^{N-1} [size(I_i) + size(D_{i+1(filtered)})]$: cost of reading the intermediate results and the filtered input tables to be used as inputs of join operations;
- $\sum_{i=2}^N size(I_i)$: cost of writing the intermediate results to disks (here, we set $I_1 = D_1$).

We assume that the IBF and the BFs are small enough to be stored in internal memories of the slave nodes. Thus, no disk I/O costs are needed for them. A comparison between Formula (5.3.10) and Formula (5.3.11) shows that $C_{I/O}^{IBF}$ includes extra costs to read and to write the input tables during the build and probe phases. However, after that the joins will use filtered input tables as their inputs. Therefore, if $size(D_{i(filtered)}) \approx size(D_i)$, there is no benefit when applying the IBF; otherwise, if $size(D_{i(filtered)}) \ll size(D_i)$, we can achieve $C_{I/O}^{IBF} \approx C_{I/O}^{NoIBF}$.

Cost-Effectiveness Analysis

According to the above analysis, we can observe that, on one side, the IBF may help to reduce the network I/O cost, C_{Net}^{IBF} , and the disk I/O cost, $C_{I/O}^{IBF}$, because it may remove irrelevant tuples from the input tables. However, on another side, the IBF needs the disk and network I/O costs to build and send it to the slave nodes of the cluster and to probe it. Therefore, its benefit is only achieved when it can remove a large number of the irrelevant tuples. Below, we further analysis the cost-effectiveness of the IBF.

As mentioned earlier, we assumed that the IBF is computed from the BF_i 's created from N input tables D_i 's, where $i \in [1, N]$, and then it is applied to filter each input table D_i to produce a corresponding filtered input table $D_{i(filtered)}$. The number of tuples of each filtered input table $D_{i(filtered)}$ can be computed by Formula (5.3.12):

$$|D_{i(filtered)}| = |D_i| \times \rho_{IBF, D_i}, \quad (5.3.12)$$

where:

- $|D_i|$: the number of tuples in the i -th input table D_i ;
- ρ_{IBF} : selectivity of the IBF.

Based on Formulas (5.3.5) and (5.3.6), we rewrite Formula (5.3.12) as below:

$$|D_{i(filtered)}| = |D_i| \times \prod_{j=1}^N \left[\rho_{D_j, D_i} + (1 - \rho_{D_j, D_i}) \times P_{BF_j} \right], \quad (5.3.13)$$

where:

- $|D_i|$: the number of tuples in the i -th input table D_i ;
- ρ_{D_j, D_i} : selectivity of the table D_j on the table D_i in the join $D_i \bowtie_{UID} D_j$;
- P_{BF_j} : error probability of the Bloom filter BF_j that is built on the table D_j ;
- $(1 - \rho_{D_j, D_i}) \times P_{BF_j}$: fraction of tuples from the probe table D_i that are not discarded by the BF_j and do not join with any tuples in the build table D_j .

To reduce the network I/O cost and disk I/O cost, we need to apply the IBF if it is beneficial. Formula (5.3.13) shows that in order to achieve $|D_{i(filtered)}| \ll |D_i|$, the value of $\prod_{j=1}^N \left[\rho_{D_j, D_i} + (1 - \rho_{D_j, D_i}) \times P_{BF_j} \right]$ needs to be low. This means that the given join sequence needs to contain one or more joins between two input tables $D_i \bowtie_{UID} D_j$ in which the selectivity ρ_{D_j, D_i} of table D_j on D_i and the error probability P_{BF_j} of their Bloom filter BF_j are low; otherwise, the IBF may give no benefit.

Besides the above condition, no matter whether the IBF is applied or not, the size of intermediate results, i.e., $size(I_i)$ in Formulas (5.3.8) – (5.3.11), should be minimized by choosing a suitable join processing order for the input tables.

Example of Evaluating the Benefit of the IBF

The objective of the example is to show how the cost models can be applied to evaluate the cost-effectiveness of applying the IBF to a multi-way join query. For this objective, we consider a simple query Q having join operations across three input tables D_1, D_2

and D_3 , i.e., $Q = D_1 \bowtie_{UID} D_2 \bowtie_{UID} D_3$. For simplicity, we follow the following assumptions: (1) the query Q has a common join attribute UID on its input tables; (2) the input tables are using a row-oriented data layout; (3) the sizes of the input tables and the selectivity factors of join operations between each pair of input tables are given in Table 5.9; and (4) the number of slave nodes $c = 10$.

Table 5.9: Example of table sizes and selectivity factors of join operations

Input table (D_i)	Number of tuples (n_i)	Tuple size (MB)	Table size (MB)	Selectivity factor (ρ_{D_j, D_i})
D_1	200,000	0.05	10,000	$\rho_{D_1, D_1} = 1; \rho_{D_2, D_1} = 0.3; \rho_{D_3, D_1} = 0.4$
D_2	100,000	0.05	5,000	$\rho_{D_1, D_2} = 0.3; \rho_{D_2, D_2} = 1; \rho_{D_3, D_2} = 0.2$
D_3	50,000	0.16	8,000	$\rho_{D_1, D_3} = 0.4; \rho_{D_2, D_3} = 0.5; \rho_{D_3, D_3} = 1$

Under the above assumptions, we can compute the IBF by *ANDing* Bloom filters BF_1 , BF_2 and BF_3 created on the common join attribute UID of the input tables D_1 , D_2 and D_3 , respectively. The IBF and all the BF_i 's ($i = 1, 2, 3$) need to follow the same configuration: the number of bits of vectors and the number of hash functions. There is a challenge to build the IBF and all the BF_i 's with the same configuration because each of them may have its own optimal configuration corresponding to its own number of tuples. To overcome this challenge, we first find an optimal configuration for the Bloom filter BF_1 created from the attribute UID of the biggest input table, i.e., D_1 , and then apply this configuration to the IBF and the other Bloom filters BF_i 's ($i = 2, 3$). In particular, given that the table D_1 contains a set of 200,000 tuples (i.e., $n_1 = 200,000$) and assumedly we want to achieve a probability of false positives bounded by $P_{BF_1} = 0.01$, the number of bits m and the number of hash functions k that are required for BF_1 to obtain the above bounded value of P_{BF_1} can be computed using two Formulas (5.3.4) and (5.3.2), respectively: $m = 1,917,012$ (≈ 2 MB) and $k = 7$. (This is identical to the example shown at the start of this section).

Using the above configuration, we compute the error probabilities for BF_2 , BF_3 and IBF according to Formulas (5.3.1) and (5.3.7):

- $P_{BF_2} = \left(1 - e^{-7 \times \frac{100,000}{1,917,012}}\right)^7 = 0.00025$.
- $P_{BF_3} = \left(1 - e^{-7 \times \frac{50,000}{1,917,012}}\right)^7 = 0.0000036$.
- $P_{IBF} = \prod_{i=1}^3 P_{BF_i} = 0.01 \times 0.00025 \times 0.0000036 = 0.00000000009$.

Up to this point, we have already obtained the following information of the IBF and BF_i 's ($i = 1, 2, 3$): the number of bits, the number of hash functions and the error probabilities of the IBF and BF_i 's. In the next step, we need to estimate the number of tuples in filtered input tables $D_{i(filtered)}$ ($i = 1, 2, 3$) by using Formula (5.3.12). This step, in turn, requires us to compute the selectivity ρ_{BF_j, D_i} of the Bloom filter BF_j on the input table D_i by using Formula (5.3.5) and the selectivity ρ_{IBF, D_i} of the IBF on D_i by using Formula (5.3.6). The results of our calculations are shown below:

- The selectivity of BF_i 's on each input table computed using Formula (5.3.5):
 - $\rho_{BF_1, D_1} = 1$.

- $\rho_{BF_2,D_1} = \rho_{D_2,D_1} + (1 - \rho_{D_2,D_1}) \times P_{BF_2} = 0.3 + 0.7 \times 0.00025 = 0.300175$.
 - $\rho_{BF_3,D_1} = \rho_{D_3,D_1} + (1 - \rho_{D_3,D_1}) \times P_{BF_3} = 0.4 + 0.6 \times 0.0000036 = 0.40000216$.
 - $\rho_{BF_1,D_2} = \rho_{D_1,D_2} + (1 - \rho_{D_1,D_2}) \times P_{BF_1} = 0.3 + 0.7 \times 0.01 = 0.307$.
 - $\rho_{BF_2,D_2} = 1$.
 - $\rho_{BF_3,D_2} = \rho_{D_3,D_2} + (1 - \rho_{D_3,D_2}) \times P_{BF_3} = 0.2 - 0.8 \times 0.0000036 = 0.19999712$.
 - $\rho_{BF_1,D_3} = \rho_{D_1,D_3} + (1 - \rho_{D_1,D_3}) \times P_{BF_1} = 0.4 - 0.6 \times 0.01 = 0.394$.
 - $\rho_{BF_2,D_3} = \rho_{D_2,D_3} + (1 - \rho_{D_2,D_3}) \times P_{BF_2} = 0.5 - 0.5 \times 0.00025 = 0.499875$.
 - $\rho_{BF_3,D_3} = 1$.
- The selectivity of the IBF on each input table computed using Formula (5.3.6):
 - $\rho_{IBF,D_1} = \prod_{j=1}^3 \rho_{BF_j,D_1} = \rho_{BF_1,D_1} \times \rho_{BF_2,D_1} \times \rho_{BF_3,D_1} = 0.120070648$.
 - $\rho_{IBF,D_2} = \prod_{j=1}^3 \rho_{BF_j,D_2} = \rho_{BF_1,D_2} \times \rho_{BF_2,D_2} \times \rho_{BF_3,D_2} = 0.06139911584$.
 - $\rho_{IBF,D_3} = \prod_{j=1}^3 \rho_{BF_j,D_3} = \rho_{BF_1,D_3} \times \rho_{BF_2,D_3} \times \rho_{BF_3,D_3} = 0.19695075$.
 - The number of tuples of the filtered input tables computed using Formula (5.3.12):
 - $|D_{1(filtered)}| = |D_1| \times \rho_{IBF,D_1} = 200,000 \times 0.120070648 = 24,014$ (reduced 88%).
 - $|D_{2(filtered)}| = |D_2| \times \rho_{IBF,D_2} = 100,000 \times 0.06139911584 = 6,140$ (reduced 94%).
 - $|D_{3(filtered)}| = |D_3| \times \rho_{IBF,D_3} = 50,000 \times 0.19695075 = 9,848$ (reduced 80%).

Since the given query may have a large number of candidate execution plans due to different join ordering possibilities, an exhaustive search for an optimal execution plan is too expensive. Hence, we adopt to apply the *minimum selectivity heuristic strategy* introduced by M. Steinbrunn et al. [108] to build a left-deep processing tree step by step by attempting to keep intermediate results as small as possible. First of all, a table D_i that has the smallest cardinality will be chosen to become an initial intermediate result, i.e., $I_1 = D_i$. Then, for each step, a table D_j having the smallest selectivity factor ρ_{D_j,I_i} for the join operation $I_i \bowtie_{UID} D_j$ is chosen. Our study takes into account the integration of the IBF into this left-deep sequential tree plan of N tables.

Now, we will apply the above execution plan to the query $Q = D_1 \bowtie_{UID} D_2 \bowtie_{UID} D_3$. Based on the information given in Table 5.9, we see that $|D_3| \leq |D_2| \leq |D_1|$, thus we select the initial intermediate result $I_1 = D_3$. Besides, by looking at the selectivity factors of other tables on table D_3 , we see that $\rho_{D_1,D_3} = 0.4 < \rho_{D_2,D_3} = 0.5$, hence we choose table D_1 instead of D_2 to join with the current intermediate result I_i . By continuing in this way, we achieve a left-deep sequential tree plan for Q . Finally, we integrate the IBF into this execution plan, as shown in Figure 5.10.

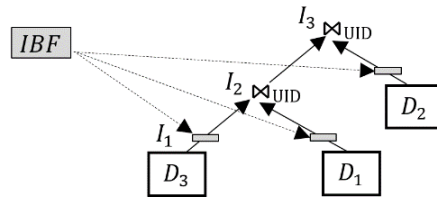


Figure 5.10: Left-deep processing tree of the query Q with the use of the IBF

The evaluation of the cost-effectiveness of applying the IBF to the query Q in terms of network I/O cost and disk I/O cost is given as follows.

Network I/O Cost:

- When not using the IBF, to compute the network I/O cost, we use Formula (5.3.8):

$$\begin{aligned}
 C_{Net}^{NoIBF} &= \sum_{i=1}^3 (|D_i| \times size(tuple_{D_i})) + |D_3| \times |D_1| \times \rho_{D_1, D_3} \\
 &\quad \times \left((size(tuple_{D_3}) + size(tuple_{D_1})) + |D_2| \times \rho_{D_2, D_3} \times \rho_{D_2, D_1} \right. \\
 &\quad \left. \times (size(tuple_{D_3}) + size(tuple_{D_1}) + size(tuple_{D_2})) \right). \\
 &= 23,000 + 50,000 \times 200,000 \times 0.4 \times \\
 &\quad ((0.16 + 0.05) + 100,000 \times 0.5 \times 0.3 \times (0.16 + 0.05 + 0.05)). \\
 &= 15,600,840,023,000 (MB).
 \end{aligned}$$

- When using the IBF, to compute the network I/O cost, we use Formula (5.3.9):

$$\begin{aligned}
 C_{Net}^{IBF} &= 10 * size(IBF) + \\
 &\quad \sum_{i=1}^3 (|D_{i(filtered)}| \times size(tuple_{D_i})) + |D_{3(filtered)}| \times |D_{1(filtered)}| \times \rho_{D_1, D_3} \\
 &\quad \times \left((size(tuple_{D_3}) + size(tuple_{D_1})) + |D_{2(filtered)}| \times \rho_{D_2, D_3} \times \rho_{D_2, D_1} \right. \\
 &\quad \left. \times (size(tuple_{D_3}) + size(tuple_{D_1}) + size(tuple_{D_2})) \right). \\
 &= 20 + (24,014 \times 0.05 + 6,140 \times 0.05 + 9,848 \times 0.16) + 9,848 \times 24,014 \times 0.4 \\
 &\quad \times ((0.16 + 0.05) + 6,140 \times 0.5 \times 0.3 \times (0.16 + 0.05 + 0.05)). \\
 &= 22,671,814,152(MB)(reduced 99\%).
 \end{aligned}$$

Disk I/O Cost:

- When not using the IBF, we can compute the disk I/O cost using Formula (5.3.10):

$$\begin{aligned}
 C_{I/O}^{NoIBF} &= (|D_3| \times size(tuple_{D_3}) + |D_3| \times |D_1| \times \rho_{D_1, D_3} \times (size(tuple_{D_3}) + size(tuple_{D_1})) \\
 &\quad + (|D_1| \times size(tuple_{D_1}) + size(D_3) + |D_2| \times size(tuple_{D_2}))) \\
 &\quad + \left(|D_3| \times |D_1| \times \rho_{D_1, D_3} \right. \\
 &\quad \times \left((size(tuple_{D_3}) + size(tuple_{D_1})) + |D_2| \times \rho_{D_2, D_3} \times \rho_{D_2, D_1} \right. \\
 &\quad \left. \left. \times (size(tuple_{D_3}) + size(tuple_{D_1}) + size(tuple_{D_2})) \right) \right). \\
 &= (50,000 \times 0.16 + 50,000 \times 200,000 \times 0.4 \times (0.16 + 0.05) \\
 &\quad + (200,000 \times 0.05 + 100,000 \times 0.05)) \\
 &\quad + (50,000 \times 200,000 \times 0.4 \\
 &\quad \times ((0.16 + 0.05) + 100,000 \times 0.5 \times 0.3 \times (0.16 + 0.05 + 0.05))). \\
 &= 15,601,680,023,000 (MB).
 \end{aligned}$$

- When using the IBF, we can compute the disk I/O cost by using Formula (5.3.11):

$$\begin{aligned}
 C_{I/O}^{IBF} &= 2 \times \sum_{i=1}^3 |D_i| \times \text{size}(\text{tuple}_{D_i}) + \sum_{i=1}^3 |D_{i(\text{filtered})}| \times \text{size}(\text{tuple}_{D_i}) \\
 &\quad + \left(|D_{3(\text{filtered})}| \times \text{size}(\text{tuple}_{D_3}) + |D_{3(\text{filtered})}| \times |D_{1(\text{filtered})}| \right. \\
 &\quad \times \rho_{D_1, D_3} \times (\text{size}(\text{tuple}_{D_3}) + \text{size}(\text{tuple}_{D_1})) \\
 &\quad + \left(|D_{1(\text{filtered})}| \times \text{size}(\text{tuple}_{D_1}) + \text{size}(D_3) + |D_{2(\text{filtered})}| \right. \\
 &\quad \times \text{size}(\text{tuple}_{D_2})) \left. \right) \\
 &\quad + \left(|D_{3(\text{filtered})}| \times |D_{1(\text{filtered})}| \times \rho_{D_1, D_3} \right. \\
 &\quad \times \left((\text{size}(\text{tuple}_{D_3}) + \text{size}(\text{tuple}_{D_1})) + |D_{2(\text{filtered})}| \times \rho_{D_2, D_3} \times \rho_{D_2, D_1} \right. \\
 &\quad \times \left. \left. (\text{size}(\text{tuple}_{D_3}) + \text{size}(\text{tuple}_{D_1}) + \text{size}(\text{tuple}_{D_2})) \right) \right). \\
 &= 2 \times (23,000) + (24,014 \times 0.05 + 6,140 \times 0.05 + 9,848 \times 0.16) \\
 &\quad + (9,848 \times 0.16 + 9,848 \times 24,014 \times 0.4 \times (0.16 + 0.05) \\
 &\quad + (24,014 \times 0.05 + 6,140 \times 0.05)) \\
 &\quad + (9,848 \times 24,014 \times 0.4 \\
 &\quad \times ((0.16 + 0.05) + 6,140 \times 0.5 \times 0.3 \times (0.16 + 0.05 + 0.05))). \\
 &= 22,691,728,365 \text{ (MB)} (\text{reduced } 99\%).
 \end{aligned}$$

Based on the above estimation results, the impact of the application of the IBF on both the network I/O cost and the disk I/O cost has been shown clearly. The estimation results show that the benefit of the IBF is achieved when the given join sequence contains one or more join operations whose selectivity factors are very highly selective

5.3.3 Incremental Intersection Bloom Filter

In the previous section, we showed that the disk I/O cost required for building and probing the IBF is quite high because of a large number of reading and writing operations on intermediate result tables and filtered intermediate result tables. To reduce this cost, instead of building a complete IBF from all Bloom filters of all input tables before probing it, we can build and probe the IBF incrementally during the execution of join operations. For simplicity, we refer to this IBF as an *incremental IBF*. Figure 5.11(a) illustrates the integration of the incremental IBF into the execution plan of the query $Q = Q_I \bowtie_{UID} Q_J \bowtie_{UID} Q_K$, which was given in Figure 5.6(c).

The steps of build and probe phases of the incremental IBF given in Figure 5.11(a):

- Execute the sub-query $Q_{I,1}$ and create the intermediate result table D'_1 .
- Compute the Bloom filter BF_1 on values of the attribute UID of D'_1 and then compute an incremental IBF: $IBF_1 = BF_1$.
- Execute the sub-query $Q_{I,2}$ with the application of the current incremental IBF_1 as a local predicate on the input table T_2 and create the intermediate result table D'_2 (i.e., D'_2 only consists of the tuples whose values are already represented in IBF_1).
- Compute the Bloom filter BF_2 on values of the attribute UID of D'_2 and recompute the incremental IBF: $IBF_2 = IBF_1 \wedge BF_2$.

- ...
- The above steps are similarly performed for the next sub-queries. Finally, we obtain all the intermediate result tables D'_1, D'_2, \dots, D'_N of the sub-queries. We also achieve the newest incremental IBF: IBF_N , i.e., in general, $IBF_N = IBF_{N-1} \wedge BF_N$. It is worthy to note that a Bloom filter is not computed on a right-hand side table of a left-outer join and no non-null constraint found on it, e.g., D'_4 . Besides, it is unnecessary to recompute the IBF_N from the intermediate result table D'_N of the uppermost sub-query $Q_{K,1}$ because it will not be used any more.

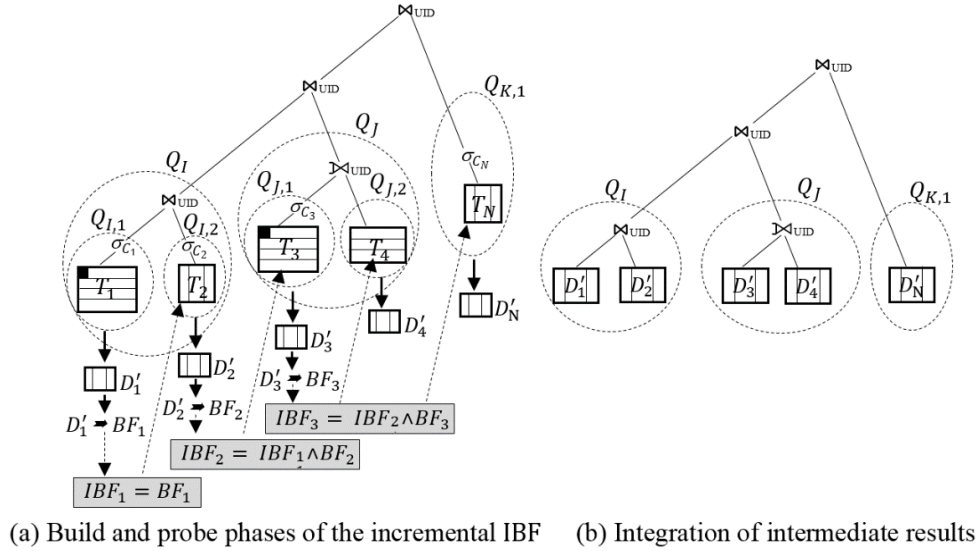


Figure 5.11: Query execution plan with the incremental IBF

Finally, the intermediate result tables D'_1, D'_2, \dots, D'_N are used as input tables for join operations in the execution plan, as illustrated in Figure 5.11(b). This means that these intermediate result tables do not need to be re-filtered as in the case of the IBF approach presented in Section 5.3.1. Thus, the disk I/O cost is significantly saved when the incremental IBF is applied. However, when using the incremental IBF, the sizes of input tables of join operations is generally larger than those in the case of the IBF approach (presented in Section 5.3.1). This is due to that fact that when using the incremental IBF, an input table is filtered by the incremental IBF that is being incrementally computed by just using the BFs of the lower input tables in the execution plan rather than using all the BFs of all the input tables. Therefore, approximately, we have $size(D_{i(filtered)}) \leq size(D'_i) \leq size(D_i)$ ($i = 1, \dots, N$), where D'_i is an intermediate result table in the case the incremental IBF is applied, whereas D_i and $D_{i(filtered)}$, respectively, are the intermediate result table and the filtered input table in the case of the IBF approach. Moreover, when the incremental IBF is applied, each sub-query produces only one intermediate result table D'_i , instead of two intermediate result tables, i.e., D_i and $D_{i(filtered)}$, as in the case of the IBF approach. Due to $size(D'_i) \leq size(D_i) + size(D_{i(filtered)})$, the disk I/O cost is required for the build and probe phases of the incremental IBF is less than that of the IBF.

The incremental IBF can offer the best benefit when it is incrementally built from intermediate result tables of highly selective sub-queries of lowermost join operations

of the execution plan tree and then used as a local predicate to filter irrelevant inputs out of the intermediate result tables of the sub-queries of upper join operations.

5.4 Summary and Conclusion

Querying DICOM data from the hybrid store of HYTORMO has posed some challenges. Entity tables of the DICOM data have been decomposed into a number of vertically partitioned tables and are stored using row or column data layouts, thus the query processing strategy needs to be designed to suit with such a data storage strategy. Besides, although the proposed data storage strategy is able to reduce the I/O cost at attribute level by attempting to reduce the number of irrelevant attribute accesses, it cannot reduce the I/O cost at tuple level. The irrelevant tuples that will not pass join predicates in multiple-table join queries are still read and sent over the network before discarded from join results. This decreases query performance.

To address the above problems, we designed and implemented a query processing strategy built on top of HYTORMO: a query execution plan with inner and left-outer joins on vertically partitioned tables. The left-outer joins are used to prevent the data loss when the inner joins are performed on the vertically partitioned tables. Furthermore, we proposed heuristic rules to determine when a left-outer join operation needs to be used and when a left-outer join operation should be rewritten into an inner-join operation. On the other hand, the integrating of the IBF into the query processing aims to minimize the irrelevant input data and the intermediate results during the query execution; this helps to reduce network communication cost. We presented a cost-effectiveness analysis of the application of the IBF. Finally, we introduce an alternative IBF approach, called incremental IBF for saving the disk I/O cost required for build and probe phases of the IBF approach. Experimental evaluation of the benefit of the IBF will be presented in the next chapter.

Key Points
<ul style="list-style-type: none">• We provide heuristic rules to choose the suitable join types.• We propose a query processing with IBF and give cost-effectiveness analysis.• We propose an alternative IBF approach, called incremental IBF.

Performance Evaluation

6.1 Overview

This chapter presents the evaluation results and lessons learned from applying HYTORMO, the data storage strategy, HADF and the query processing strategy with the use of an IBF. An overview of the chapter is given in Table 6.1.

Table 6.1: Overview over Chapter 6

6.2 Experimental Environment			
6.2.1 Spark Cluster	6.2.2 Datasets	6.2.3 Workloads	
6.3 Experiment Execution			
6.3.1 Experiment 1	6.3.2 Experiment 2	6.3.3 Experiment 3	6.3.4 Experiment 4
6.4 Analysis and Interpretation			
6.4.1 H1 - Effectiveness of HYTORMO	6.4.2 H2 - Usefulness of HADF		
6.4.3 H3 - Effectiveness of the Query Processing Strategy			
6.5 Summary and Conclusion			

Our experiments aim at providing empirical evidences that the proposed methods are helpful as well as isolating the lessons learned and determining the critical aspects of successful applying the proposed methods. The experiments concentrate on answering the following questions:

- Does the combined use of the hybrid storage model, HYTORMO, offer a better workload performance than only using a pure row store or a pure column store?
- Does HADF with taking into account the combined impact of both workload- and data-specific information as well as the use of both row and column stores help us to generate better data storage configurations in terms of storage space size and workload execution time?
- Does the query processing strategy with the integration of an IBF improve query performance?

The above questions are respectively related to three hypotheses H1 – H3 proposed to evaluate the proposed methods, shown in Section 1.7 in Chapter 1: the first one concerns on the benefit of HYTORMO; the second one concerns on the benefit of HADF; and the last one concerns on the benefit of the IBF. To get the answers for these questions, we first describe the experimental environment. Next, we execute the experiments. After that, we analyze and interpret the results to evaluate the hypotheses.

6.2 Experimental Environment

6.2.1 Spark Cluster

We used Hadoop 2.7.1, Hive 1.2.1 and Spark 1.6.0 to install a cluster. The hardware of the cluster consists of 9 different nodes: 1 \times *Master node*: Intel(R) core(TM) i7-3770 CPU @ 3.40GHz, 8GB RAM, 2TB hard disk and 1Gbit network connection; and 8 \times *Slave nodes*: Intel(R) core(TM) i7-3770 CPU @ 3.40GHz, 6GB RAM, 500GB hard disk and 1Gbit network connection (GALACTICA: <https://horizon.isima.fr>). HDFS was used for the hybrid store of HYTORMO. We ran 1 Namenode and 8 Datanodes using the standard configuration of HDFS with a modification: we set the replication factor of HDFS to 2 (instead of 3 as default) in order to save storage space. We implemented execution plans for queries in workloads using Spark [21].

6.2.2 Datasets

We used real DICOM datasets [121-126] whose statistics (including the number of DICOM files, the number of extracted attributes, the size of extracted metadata in text format and the total size of files) are given in Table 6.2. We performed four different experiments in order to validate the benefits of the hybrid store, HADF and IBF. Each experiment used different parts of the DICOM datasets. For simplicity, we created two mixed datasets: (1) MDB1 consisted of DICOM files of the first five DICOM datasets: CTColonography, Dclunie, Idoimaging, LungCancer and MIDAS; and (2) MDB2 consisted of all DICOM datasets: CTColonography, Dclunie, Idoimaging, LungCancer, MIDAS and CIAD. In order to reduce the complexity of processing and analyzing a large amount of data, Experiment 1 used only MDB1 as a sample dataset to provide data-specific information for HADF. This is because the distribution of null ratios of attributes in MDB1 is similar to that of MDB2. Experiment 2 used MDB1 and MDB2 separately. Experiment 3 and 4 used only MDB2.

Table 6.2: Mixed DICOM datasets used in the experiments

No.	Datasets	No. of DICOM files	No. of extracted attributes	Size of extracted metadata	Total size of files	Mixed dataset	
1	CTColonography	98,737	86	7.76 GB	48.6 GB	MDB1	MDB2
2	Dclunie	541	86	86.0 MB	45.7 GB		
3	Idoimaging	1,111	86	53.9 MB	369 MB		
4	LungCancer	174,316	86	1.17 GB	76.0 GB		
5	MIDAS	2,454	86	63.4 MB	620 MB		
6	CIAD	3,763,894	86	61.5 GB	1.61 TB		

Metadata and pixel data were extracted from the DICOM files by using a Java program that calls methods in the library dcm4che-2.0.29 [127]. The extracted attributes were grouped together and stored in suitable storage layouts (a row- or a column-oriented data layout). This can be performed by using one of two design approaches: expert-based and automated. In the former approach, first of all, experts (e.g., database designers) manually group the attributes of each entity table into column groups (mandatory, frequently-accessed-together or optional/private/seldom-

accessed attributes), then select a suitable data layout for each column group. In [107], we showed that the query performance when applying this approach is improved when compared with two other approaches: using a pure row store and a pure column store. In this chapter, we only present the experiments to assess the benefit of the later approach.

HADF is applied to generate multiple data storage configurations depending on the combined impact of both workload- and data-specific information as well as the mixed use of both row and column stores. The HADF-generated data storage configurations were stored as follows: sequence files and ORC files in Hive [128] were used to store row and column tables, respectively.

In order to manage completely DICOM data, many entity tables, such as *Patient*, *Study*, *GeneralInfoTable*, *SequenceAttribute*, *ClinicalTrial*, *GeneralSeries*, *FileMetaElement* and *ImageInformation* and so on, need to be stored. However, our experiments only concern on the following four entity tables:

- **Patient**(UID, PatientName, PatientID, PatientBirthDate, PatientSex, EthnicGroup, IssuerOfPatientID, PatientBirthTime, PatientInsurancePlanCodeSequence, PatientPrimaryLanguageCodeSequence, PatientPrimaryLanguageModifierCodeSequence, OtherPatientIDs, OtherPatientNames, PatientBirthName, PatientTelephoneNumbers, SmokingStatus, PregnancyStatus, LastMenstrualDate, PatientReligiousPreference, PatientComments, PatientAddress, PatientMotherBirthName, InsurancePlanIdentification)
- **Study**(UID, StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, AccessionNumber, StudyDescription, PatientAge, PatientWeight, PatientSize, Occupation, AdditionalPatientHistory, MedicalRecordLocator, MedicalAlerts)
- **GeneralInfoTable**(UID, GeneralTags, GeneralVRs, GeneralNames, GeneralValues)
- **SequenceAttributes**(UID, SequenceTags, SequenceVRs, SequenceNames, SequenceValues)

Table 6.3 shows the sizes of the entity tables extracted from the dataset MDB1.

Table 6.3: Sizes of the entity tables of the dataset MDB1

Entity Table	Total size	
	Number of tuples	Size
Patient	120,306	20.788 MB
Study	120,306	19.183 MB
GeneralInfoTable	16,226,762	4,845,042 MB
SequenceAttributes	4,149,395	389.433 MB

The null ratios of the attributes of the entity tables of MDB1 are listed below:

- The null ratios of the attributes of the entity table *Patient*:
 - 1 *PatientName*: 0.00%
 - 2 *PatientID*: 0.00%
 - 3 *PatientBirthDate*: 83.55%
 - 4 *PatientSex*: 1.48%
 - 5 *EthnicGroup*: 100%
 - 6 *IssuerOfPatientID*: 100%
 - 12 *OtherPatientNames*: 100%
 - 13 *PatientBirthName*: 100%
 - 14 *PatientTelephoneNumbers*: 100%
 - 15 *SmokingStatus*: 97.48%
 - 16 *PregnancyStatus*: 90.01%
 - 17 *LastMenstrualDate*: 97.72%

- 7 *PatientBirthTime*: 96.32%
- 8 *PatientInsurancePlanCodeSequence*: 100%
- 9 *PatientPrimaryLanguageCodeSequence*: 100%
- 10 *PatientPrimaryLanguageModifierCodeSequence*: 100%
- 11 *OtherPatientIDs*: 100%
- 18 *PatientReligiousPreference*: 100%
- 19 *PatientComments*: 99.64%
- 20 *PatientAddress*: 100%
- 21 *PatientMotherBirthName*: 100%
- 22 *InsurancePlanIdentification*: 100%

- The null ratios of the attributes of the entity table *Study*:

- 1 *StudyInstanceUID*: 0.00%
- 2 *StudyDate*: 0.07%
- 3 *StudyTime*: 0.07%
- 4 *ReferringPhysicianName*: 16.44%
- 5 *StudyID*: 15.65%
- 6 *AccessionNumber*: 93.93%
- 7 *StudyDescription*: 0.48%
- 8 *PatientAge*: 11.23%
- 9 *PatientWeight*: 14.18%
- 10 *PatientSize*: 90.34%
- 11 *Occupation*: 99.63%
- 12 *AdditionalPatientHistory*: 71.64%
- 13 *MedicalRecordLocator*: 100%
- 14 *MedicalAlerts*: 100%

- The null ratios of the attributes of the entity table *GeneralInfoTable*:

- 1 *GeneralTags*: 0.00%
- 2 *GeneralVRs*: 0.00%
- 3 *GeneralNames*: 0.00%
- 4 *GeneralValues*: 13.97%

- The null ratios of the attributes of the entity table *SequenceAttributes*:

- 1 *SequenceTags*: 0.00%
- 2 *SequenceVRs*: 0.00%
- 3 *SequenceNames*: 0.00%
- 4 *SequenceValues*: 0.34%

Table 6.4 shows the sizes of the entity tables extracted from the dataset MDB2.

Table 6.4: Sizes of the entity tables of the dataset MDB2

Entity Table	Total size	
	Number of tuples	Size
Patient	1,802,376	324 MB
Study	1,856,892	384 MB
GeneralInfoTable	337,730,322	39.2 GB
SequenceAttributes	75,314,902	7.64 GB

The null ratios of the attributes of the entity tables of MDB2 are listed below.

- The null ratios of the attributes of the entity table *Patient*:

- 1 *PatientName*: 19.25%
- 2 *PatientID*: 0%
- 3 *PatientBirthDate*: 96.70%
- 4 *PatientSex*: 11.99%
- 5 *EthnicGroup*: 78.29%
- 6 *IssuerOfPatientID*: 100%
- 7 *PatientBirthTime*: 99.75%
- 8 *PatientInsurancePlanCodeSequence*: 100%
- 9 *PatientPrimaryLanguageCodeSequence*: 100%
- 10 *PatientPrimaryLanguageModifierCodeSequence*: 100%
- 11 *OtherPatientIDs*: 100%
- 12 *OtherPatientNames*: 100%
- 13 *PatientBirthName*: 100%
- 14 *PatientTelephoneNumbers*: 100%
- 15 *SmokingStatus*: 79.33%
- 16 *PregnancyStatus*: 36.36%
- 17 *LastMenstrualDate*: 99.85%
- 18 *PatientReligiousPreference*: 100%
- 19 *PatientComments*: 83.23%
- 20 *PatientAddress*: 100%
- 21 *PatientMotherBirthName*: 100%
- 22 *InsurancePlanIdentification*: 100%

- The null ratios of the attributes of the entity table *Study*:

1 <i>StudyInstanceUID</i> : 2.72%	8 <i>PatientAge</i> : 29.78%
2 <i>StudyDate</i> : 2.94%	9 <i>PatientWeight</i> : 27.11%
3 <i>StudyTime</i> : 23.43 %	10 <i>PatientSize</i> : 33.12%
4 <i>ReferringPhysicianName</i> : 87.63%	11 <i>Occupation</i> : 97.11%
5 <i>StudyID</i> : 86.57%	12 <i>AdditionalPatientHistory</i> : 79.38%
6 <i>AccessionNumber</i> : 24.58%	13 <i>MedicalRecordLocator</i> : 98.36%
7 <i>StudyDescription</i> : 19.83%	14 <i>MedicalAlerts</i> : 98.21%
- The null ratios of the attributes of the entity table *GeneralInfoTable*:

1 <i>GeneralTags</i> : 0%	3 <i>GeneralNames</i> : 0%
2 <i>GeneralVRs</i> : 0%	4 <i>GeneralValues</i> : 10.19%
- The null ratios of the attributes of the entity table *SequenceAttributes*:

1 <i>SequenceTags</i> : 0.2%	3 <i>SequenceNames</i> : 0.36%
2 <i>SequenceVRs</i> : 0.36%	4 <i>SequenceValues</i> : 0.69%

As shown above, the null ratios of the attributes of two entity tables *Patient* and *Study* are very high, thus we can refer to them as sparse tables. Conversely, two entity tables *GeneralInfoTable* and *SequenceAttributes* are regarded as dense tables because the null ratios of their attributes are very low.

6.2.3 Workloads

We simulated various workloads, each of which includes a set of queries and their occurrence frequency. There are the following types of the workloads: (1) *OLAP-like workload* contains queries using only a few attributes from each entity table; (2) *OLTP-like workload* consists of queries using most (or all) attributes from each entity table; and (3) *mixed OLTP and OLAP workload* includes queries using an arbitrary number of attributes from the entity tables.

Workload W1: This is an OLAP-like workload that mainly contains queries using only a few attributes of the entity table *GeneralInfoTable* (which is the largest entity table in terms of storage space size). Workload W1 aims at demonstrating the benefit of HADF when used for OLAP workloads. The set of queries and their occurrence frequency in this workload is given in Table 6.5.

Table 6.5: Queries and their occurrence frequency in Workload W1

Query	Query	Freq
Q _{1,1}	SELECT UID, GeneralTags, GeneralVRs, GeneralNames, GeneralValues FROM GeneralInfoTable	100
Q _{1,2}	SELECT GeneralTags, count(GeneralValues) FROM GeneralInfoTable GROUP BY GeneralTags	100
Q _{1,3}	SELECT UID, GeneralNames FROM GeneralInfoTable WHERE GeneralNames = 'Modality'	100
Q _{1,4}	SELECT UID, GeneralVRs FROM GeneralInfoTable WHERE GeneralVRs = 'DA'	100

Workload W2: This is an OLTP-like workload that consists of queries using the majority of the attributes from the entity table *SequenceAttributes*. It aims at showing the application of HADF to OLTP workloads. We present this workload in Table 6.6.

Table 6.6: Queries and their occurrence frequency in Workload W2

Query	Query	Freq
Q2.1	SELECT UID, SequenceTags, SequenceVRs, SequenceNames, SequenceValues FROM SequenceAttributes WHERE SequenceNames LIKE '%X-Ray%'	100
Q2.2	SELECT SequenceTags, SequenceVRs, SequenceNames FROM SequenceAttributes WHERE SequenceVRs = 'CS'	100

Workload W3: This is a mixed OLTP and OLAP workload that uses an arbitrary number of attributes from the entity table *Patient* (which is the widest and sparsest entity table). Some attributes of the entity table *Patient* are frequently accessed together by the same queries (OLTP-like workload) while the others are seldom accessed together (OLAP-like workload). Workload W3 is shown in Table 6.7.

Table 6.7: Queries and their occurrence frequency in Workload W3

Query	Query	Freq
Q3.1	SELECT UID, PatientName, PatientID, PatientBirthDate, PatientTelephoneNumbers, PatientSex, PatientBirthName, SmokingStatus, PatientComments, PatientMotherBirthName FROM Patient WHERE PatientID = 'P30013'	300
Q3.2	SELECT UID, PatientName, PatientID, PatientBirthDate, PatientSex, EthnicGroup, IssuerOfPatientID, OtherPatientNames, PatientMotherBirthName, InsurancePlanIdentification FROM Patient	100
Q3.3	SELECT UID, PatientID, PatientName, PatientBirthDate, PatientSex, EthnicGroup, SmokingStatus FROM Patient WHERE PatientSex = 'M' AND SmokingStatus = 'NO'	100
Q3.4	SELECT UID, PatientName, PatientID, PatientBirthDate, EthnicGroup, PatientPrimaryLanguageModifierCodeSequence, OtherPatientIDs, PatientAddress FROM Patient	100
Q3.5	SELECT UID, PatientName, PatientID, PatientBirthDate, PatientBirthTime, PatientInsurancePlanCodeSequence, PregnancyStatus, LastMenstrualDate, PatientReligiousPreference FROM Patient	100
Q3.6	SELECT UID, PatientName, PatientID, PatientBirthDate, EthnicGroup, PregnancyStatus, LastMenstrualDate FROM Patient	100

This workload aims at illustrating the application of HADF to a mixed OLAP and OLTP workload and showing whether the combined use of both workload- and data-specific information is helpful in reducing the storage space demand and the workload execution time.

Workload W4: Similarly to Workload W3, this is a mixed OLTP and OLAP workload, using an arbitrary number of attributes from multiple entity tables *Patient*, *Study*, *GeneralInfoTable* and *SequenceAttributes*. Additionally, it contains multiple table join queries. Hence, it is used not only to demonstrate the application of HADF to mixed workloads but also to show the support of HYTORMO for multiple-table join queries. We introduce this workload in Table 6.8.

Table 6.8: Queries and their occurrence frequency in Workload W4

Query	Query	Freq
Q4.1	SELECT Patient.UID, Patient.PatientName, Patient.PatientID, Patient.PatientBirthDate, Patient.PatientTelephoneNumbers, Patient.PatientSex, Patient.PatientBirthName, Patient.SmokingStatus, Patient.PatientComments, Patient.PatientMotherBirthName, Study.StudyInstanceUID, Study.StudyDate, Study.StudyTime, Study.ReferringPhysicianName, Study.StudyID, Study.AccessionNumber, Study.MedicalAlerts FROM Patient, Study WHERE Patient.UID = Study.UID AND and Patient.PatientID = 'P30013' AND Study.StudyDate >= '20000101' AND Study.StudyDate <= '20150101'	300
Q4.2	SELECT Patient.UID, Patient.PatientName, Patient.PatientID, Patient.PatientBirthDate, Patient.PatientSex, Patient.EthnicGroup, Patient.IssuerOfPatientID, Patient.OtherPatientNames, Patient.PatientMotherBirthName, Patient.InsurancePlanIdentification, Study.StudyInstanceUID, Study.StudyDate, Study.StudyTime, Study.ReferringPhysicianName, Study.StudyID, Study.MedicalRecordLocator FROM Patient, Study WHERE Patient.UID = Study.UID AND Study.StudyID = '20050920'	100
Q4.3	SELECT Patient.UID, Patient.PatientID, Patient.PatientName, Patient.PatientBirthDate, Patient.PatientSex, Patient.EthnicGroup, Patient.SmokingStatus, Study.PatientAge, Study.PatientWeight, Study.PatientSize, GeneralInfoTable.GeneralNames, GeneralInfoTable.GeneralValues, SequenceAttributes.UID, SequenceAttributes.SequenceTags, SequenceAttributes.SequenceVRs, SequenceAttributes.SequenceNames, SequenceAttributes.SequenceValues FROM Patient, Study, GeneralInfoTable, SequenceAttributes WHERE Patient.UID = Study.UID AND Study.UID = GeneralInfoTable.UID AND Patient.UID = SequenceAttributes.UID AND Patient.PatientSex = 'M' AND Patient.SmokingStatus = 'NO' AND Study.PatientAge >= 90 AND SequenceAttributes.SequenceNames LIKE '%X-Ray%'	100
Q4.4	SELECT Patient.UID, Patient.PatientName, Patient.PatientID, Patient.PatientBirthDate, Patient.EthnicGroup, Patient.PatientPrimaryLanguageModifierCodeSequence, Patient.OtherPatientIDs, Patient.PatientAddress, Study.UID, Study.StudyInstanceUID, Study.StudyDate, Study.StudyTime, Study.ReferringPhysicianName, Study.StudyID, Study.AccessionNumber, Study.PatientWeight, Study.AdditionalPatientHistory, GeneralInfoTable.GeneralTags, GeneralInfoTable.GeneralValues SequenceAttributes.SequenceTags, SequenceAttributes.SequenceVRs, SequenceAttributes.SequenceNames FROM Patient, Study, GeneralInfoTable, SequenceAttributes WHERE Patient.UID = Study.UID AND Patient.UID = GeneralInfoTable.UID AND Patient.UID = SequenceAttributes.UID AND SequenceAttributes.SequenceVRs = 'CS' AND GeneralInfoTable.GeneralTags LIKE '0008%'	100
Q4.5	SELECT Patient.UID, Patient.PatientName, Patient.PatientID, Patient.PatientBirthDate, Patient.PatientBirthTime, Patient.PatientInsurancePlanCodeSequence, Patient.PregnancyStatus, Patient.LastMenstrualDate, Patient.PatientReligiousPreference, Study.StudyInstanceUID, Study.StudyDate, Study.StudyTime, Study.StudyID, Study.PatientSize, Study.Occupation, GeneralInfoTable.GeneralNames FROM Patient, Study, GeneralInfoTable WHERE Patient.UID = Study.UID AND Patient.UID = GeneralInfoTable.UID AND GeneralInfoTable.GeneralNames = 'Modality'	100
Q4.6	SELECT Patient.UID, Patient.PatientName, Patient.PatientID, Patient.PatientBirthDate, Patient.EthnicGroup, Patient.PregnancyStatus, Patient.LastMenstrualDate, Study.StudyInstanceUID, Study.StudyDate, Study.StudyTime, Study.ReferringPhysicianName, Study.StudyID, Study.StudyDescription, Study.PatientAge, GeneralInfoTable.GeneralVRs FROM Patient, Study, GeneralInfoTable WHERE Patient.UID = Study.UID AND Patient.UID = GeneralInfoTable.UID AND Study.StudyDate >= '20000101' AND Study.StudyDate <= '20150101' AND GeneralInfoTable.GeneralVRs = 'DA'	100

6.3 Experiment Execution

This section presents four different experiments used to evaluate the hypotheses.

6.3.1 Experiment 1: Evaluating the Effectiveness of HYTORMO and the Usefulness of HADF

Experiment 1 aims at assessing Hypotheses H1 and H2 in order to show the effectiveness of HYTORMO and the usefulness of HADF, respectively. Besides, a good data storage configuration for each entity table will be chosen from a set of HADF-generated data storage configurations.

Table 6.9: Major steps of Experiment 1

Conf	Typical candidate data storage configuration	Execution	Measures	Selection Criteria
G_1	- Settings: $\alpha = 0$; $\beta = 0$; $\theta = 0$; and $\lambda = 0$. - HADF-generated data storage configurations: The entity table T_i is stored in a single row table.	- Run Workload W_j ($j = 1, \dots, 4$) relevant to the entity table T_i five time for each configuration. - Using the dataset MDB1.	- Storage space size of T_i . - Workload execution time.	- A good configuration is chosen for T_i according to: (1) storage space size; and/or (2) workload execution time.
G_2	- Settings: $\alpha = 0$; $\beta = 0$; $\theta = 0$; and $\lambda = 1$. - HADF-generated data storage configurations: The entity table T_i is in a single column table.			
G_3 - G_7	- Settings: $\alpha = 0, 0.3, 0.5, 0.7, 1$; $\beta = 0.4$; $\theta = 0.5$; and $\lambda = 0.6$. - HADF-generated data storage configurations: vertically partitioned tables and their data layouts.			

The major steps of the experiment are presented in detail in Table 6.9. First of all, to obtain a good configuration for each entity table T_i (i.e., *GeneralInfoTable*, *SequenceAttributes*, *Patient* and *Study*) in the given workloads W_j , where $j = 1, \dots, 4$, we apply HADF to generate a set of 7 typical candidate data storage configurations corresponding to 7 different settings of the input parameters α , β , θ and λ . Each HADF-generated data storage configuration will be represented in the form of a set of clusters together with their corresponding data layouts. Next, we build these configurations in HYTORMO. For each configuration, we run the relevant workload(s) five times; the average workload execution time is calculated. To reduce experiment time, this experiment uses only the dataset MDB1. A good configuration is chosen for T_i based on storage space size and workload execution time.

Below, we present the experimental results of four workloads W1 - W4.

Execution of Workload W1

Workload W1 uses the entity table *GeneralInfoTable* and a set of queries given in Table 6.5. We first build the corresponding matrix *AUM* for this entity table, as shown in Figure 6.1. This is an OLAP-like workload since only a few of the attributes of the

entity table are accessed together by the same queries. As default, the attribute *UID* is added to all vertically partitioned tables, thus we do not need to add it into the *AUM*.

	i_1	i_1	i_2	i_4	Freq
$Q_{1,1}$	1	1	1	1	100
$Q_{1,2}$	1	0	0	1	100
$Q_{1,3}$	0	0	1	0	100
$Q_{1,4}$	0	1	0	0	100

i_1	<i>GeneralTags</i>
i_2	<i>GeneralVRs</i>
i_3	<i>GeneralNames</i>
i_4	<i>GeneralValues</i>

Figure 6.1: *AUM* of the entity table *GeneralInfoTable* in Workload W1

Table 6.10: Typical candidate configurations for *GeneralInfoTable*

Conf	Input						Output					
	Parameters				Entity table		Typical candidate data storage configuration	No. of stored data cells	Null ratio	No. of joins	No. of scanned data cells	Exec. time (sec)
	α	β	θ	λ	No. of data cells	Null ratio						
G_1	0	0	0	0	81,135,145	3.49%	$C_{1,1} = \{UID, i_1, i_2, i_3, i_4\}$ => row store	81,135,145	3.49%	0	32,454,058,000	15,180
G_2	0	0	0	1	81,135,145	3.49%	$C_{2,1} = \{UID, i_1, i_2, i_3, i_4\}$ => column store	81,135,145	3.49%	0	19,472,434,800	13,790
G_3	0	0.4	0.5	0.6	81,135,145	3.49%	like G_2	like G_2	like G_2	like G_2	like G_2	like G_2
G_4	0.3	0.4	0.5	0.6	81,135,145	3.49%	like G_2	like G_2	like G_2	like G_2	like G_2	like G_2
G_5	0.5	0.4	0.5	0.6	81,135,145	3.49%	like G_2	like G_2	like G_2	like G_2	like G_2	like G_2
G_6	0.7	0.4	0.5	0.6	81,135,145	3.49%	like G_2	like G_2	like G_2	like G_2	like G_2	like G_2
G_7	1	0.4	0.5	0.6	81,135,145	3.49%	$C_{7,1} = \{UID, i_1, i_4\}$ => row store $C_{7,2} = \{UID, i_3\}$ => row store $C_{7,2} = \{UID, i_2\}$ => row store	113,589,203	3.49%	200	22,717,840,600	19,020

Table 6.10 presents a set of 7 HADF-generated data storage configurations $G_1 - G_7$ and their statistics for the entity table *GeneralInfoTable*, corresponding to 7 different settings of the input parameters (i.e., α , β , θ and λ). Each row in the table describes a configuration. The columns represent: (1) the values for input parameters (2) the number of stored data cells of the original entity table; (3) the null ratio of the entity table; (4) the configuration represented in the forms of a set of column groups and their corresponding data layouts; (5) the number of stored data cells of the configuration; (6) the null ratio of the configuration; (7) the number of join operations needed by Workload W1; (8) the number of data cells scanned for Workload W1; and (9) the workload execution time (in second).

Here, the null ratio of a table is computed by Formula (6.4.1):

$$NullRatio = \frac{\text{The number of null data cells in table}}{M \times N}, \quad (6.4.1)$$

where M and N respectively represent the number of rows and columns in the table (not including the attribute *UID*). Similarly, the null ratio of a configuration is the ratio

between the total number of null data cells stored in all vertically partitioned tables and the total number of data cells stored for that configuration.

Configurations $G_1 - G_7$ are described as follows:

- Configuration G_1 :** This configuration is referred to as a baseline configuration in which all the attributes of *GeneralInfoTable* are grouped into single cluster $C_{1,1} = \{UID, i_1, i_2, i_3, i_4\}$, stored in a single row table. Some relevant statistics are as follows: (1) the number of stored data cells is 81,135,145; (2) the overall null ratio is 3.49%; (3) no join operation is required because the workload access only one table; (4) the number of data cells scanned by the workload is 32,454,058,000; and (5) the workload execution time is 15,180 seconds.
- Configuration G_2 :** This configuration is similar to Configuration G_1 : it groups all the attributes of *GeneralInfoTable* into a single cluster $C_{2,1} = \{UID, i_1, i_2, i_3, i_4\}$, but uses a column table, instead of a row table. Compared to G_1 , G_2 also does not decrease the null ratio, but helps the workload significantly reduces the number of scanned data cells: 19,472,434,800 data cells are scanned. The reason is when a column store is used, only the columns relevant to the queries are read. Similarly to G_1 , no join operation is needed. As a result, the workload execution time is low: the workload is performed in 13,790 seconds.
- Configuration $G_3 - G_6$:** *GeneralInfoTable* is a dense table (its null ratio is very low: 3.49%) and most of its attributes are seldom accessed together (except query $Q_{1,1}$). Thus, when the weight parameter α is set, respectively, to 0, 0.3, 0.5 and 0.7, the clustering phase of HADF found that all the attributes of *GeneralInfoTable* are highly correlated with each other with respect to Hybrid Similarity in which Attribute Density Similarity has more impact on the result of the clustering than Attribute Access Similarity. Recall that the Hybrid Similarity between two attributes a_x and a_y is computed by Formula (4.4.3): $HybridSim(a_x, a_y) = \alpha \times AttributeAccessSim(a_x, a_y) + (1 - \alpha) \times AttributeDensitySim(a_x, a_y)$. Therefore, the clustering phase groups all the attributes into a single cluster, i.e., $C_{3,1} = C_{4,1} = C_{5,1} = C_{6,1} = \{UID, i_1, i_2, i_3, i_4\}$. If this cluster is stored in a single row table, the number of redundant accesses from queries $Q_{1,2}$, $Q_{1,3}$ and $Q_{1,4}$ will be large. However, the merging-selecting phase of HADF suggests to store the cluster in a single column table so that no join operation is required for $Q_{1,1}$ while the number of redundant attribute accesses from queries $Q_{1,2}$, $Q_{1,3}$ and $Q_{1,4}$ is reduced as well. Therefore, the workload execution time is low: 13,790 seconds.
- Configuration G_7 :** When α is set to 1, the clustering phase of HADF only takes into account the impact of workload-specific information while the data-specific information has no impact on the clustering result, thus it decomposes *GeneralInfoTable* into multiple clusters: $C_{7,1} = \{UID, i_1, i_4\}$, $C_{7,2} = \{UID, i_3\}$, and $C_{7,2} = \{UID, i_2\}$. Besides, the Inter-cluster Access Similarity between these clusters is not high enough such that the merging-selecting phase does not merge any pair of clusters together. Furthermore, Intra-cluster Access Similarity within each of those clusters is high enough such that they are stored in row tables. Using this configuration, the queries needs to scan a lightly higher number of data cells (i.e., 22,717,840,600) than Configurations $G_2 - G_6$ due to the need to scan the

attribute *UID* in each vertically partitioned tables. Besides, *GeneralInfoTable* is a dense entity table, storing it in multiple vertically partitioned tables does not help to reduce the storage space demand (i.e., not removing many null values) while additional data cells needed to store the attribute *UID* increase the storage space size of G_7 : 113,589,203 data cells are used. Moreover, a large number of additional join operations is needed to join the vertically partitioned tables together: 200 joins are performed. All of this result in high workload execution time: 19,020 seconds.

Configurations $G_2 - G_6$ are the same: storing *GeneralInfoTable* in a column table, thus we only need to compare the effectiveness of three distinct configurations G_1 , G_2 and G_7 . Our experiments put more focus on the workload execution time than the storage space demand, thus we choose G_3 , using a single column table, as a good configuration to store *GeneralInfoTable*.

Execution of Workload W2

	e_1	e_2	e_3	e_4	Freq
$Q_{2,1}$	1	1	1	1	100
$Q_{2,2}$	1	1	1	0	100

e_1	<i>SequenceTags</i>
e_2	<i>SequenceVRs</i>
e_3	<i>SequenceNames</i>
e_4	<i>SequenceValues</i>

Figure 6.2: *AUM* of the entity table *SequenceAttributes* in Workload W2

Workload W2 uses the entity table *SequenceAttributes* and a set of queries given in Table 6.6. We first build the matrix *AUM* for *SequenceAttributes*, as presented in Figure 6.2. This is an OLTP workload since most of the attributes of the entity table are frequently accessed together. As default, the attribute *UID* is required in all vertically partitioned tables, so we do not need to include it into the *AUM*.

Table 6.11 describes a set of 7 HADF-generated data storage configurations $G_1 - G_7$ and their statistics.

Table 6.11: Typical candidate configurations for *SequenceAttributes*

Conf	Input						Output					
	Parameters				Entity Table		Candidate Data Storage Configuration	No. of stored data cells	Null ratio	No. of joins	No. of scanned data cells	Exec. Time (sec)
	α	β	θ	λ	No. of data cells	Null ratio						
G_1	0	0	0	0	20,746,975	0.086%	$C_{1,1} = \{UID, e_1, e_2, e_3, e_4\}$ \Rightarrow row store	20,746,975	0.086%	0	4,149,395,000	5,620
G_2	0	0	0	1	20,746,975	0.086%	$C_{2,1} = \{UID, e_1, e_2, e_3, e_4\}$ \Rightarrow column store	20,746,975	0.086%	0	3,734,455,500	5,780
G_3	0	0.4	0.5	0.6	20,746,975	0.086%	like G_1	like G_1	like G_1	like G_1	like G_1	like G_1
G_4	0.3	0.4	0.5	0.6	20,746,975	0.086%	like G_1	like G_1	like G_1	like G_1	like G_1	like G_1
G_5	0.5	0.4	0.5	0.6	20,746,975	0.086%	like G_1	like G_1	like G_1	like G_1	like G_1	like G_1
G_6	0.7	0.4	0.5	0.6	20,746,975	0.086%	like G_1	like G_1	like G_1	like G_1	like G_1	like G_1
G_7	1.0	0.4	0.5	0.6	20,746,975	0.086%	like G_1	like G_1	like G_1	like G_1	like G_1	like G_1

Configurations $G_1 - G_7$ in the table are described as follows:

- **Configuration G_1 :** This configuration is referred to as a baseline configuration in which all the attributes of *SequenceAttributes* is grouped into a single cluster $C_{1,1} = \{UID, e_1, e_2, e_3, e_4\}$, stored in a row table. Some statistics relevant to this configuration include: (1) the number of stored data cells is 20,746,975; (2) the null ratio is 0.086%; (3) no join operation is required; (4) the number of data cells scanned is 4,149,395,000; and (5) the workload execution time is 5,620 seconds.
- **Configuration G_2 :** This configuration is similar to G_1 ; it groups all the attributes of *SequenceAttributes* into a single cluster $C_{2,1} = \{UID, e_1, e_2, e_3, e_4\}$. However, it stores this cluster in a single column table, instead of a single row store. Compared to G_1 , it does not reduce the null ratio, but reduces the number of scanned data cells: 3,734,455,500 data cells are scanned. Like G_1 , no join operation is used, but the workload execution time when using G_2 is lightly higher than that of G_1 : this time is 5,780 seconds. This is due to the fact that G_2 is using a column store that incurs a high cost to reconstruct result tuples for an OLTP workload such as W2.
- **Configurations $G_3 - G_7$:** *SequenceAttributes* is a dense table and all of its attributes are frequently accessed together. When α is set, respectively, to 0, 0.3, 0.5, 0.7 and 1, the clustering phase of HADF found that all the attributes are highly correlated depending on the combined impact of both Attribute Density Similarity and Attribute Access Similarity. Thus, it groups the attributes into a single cluster. Additionally, the Intra-cluster Access Similarity between every pair of attributes within this cluster is high enough such that the merging-selecting phase decides to store it in a row table.

G_1, G_3, G_4, G_5, G_6 and G_7 are the same, thus we only need to compare two distinct configurations G_1 and G_2 . We choose G_3 , which stores *SequenceAttributes* in a single row table, because this configuration gives a faster workload execution time.

Execution of Workload W3

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}	p_{21}	p_{22}	Freq
$Q_{3,1}$	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	1	0	300
$Q_{3,2}$	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	100
$Q_{3,3}$	1	1	1	1	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	100
$Q_{3,4}$	1	1	1	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	100
$Q_{3,5}$	1	1	1	1	0	0	1	1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	100
$Q_{3,6}$	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	100

p_1	<i>PatientName</i>	p_8	<i>PatientInsurancePlanCodeSequence</i>	p_{15}	<i>SmokingStatus</i>
p_2	<i>PatientID</i>	p_9	<i>PatientPrimaryLanguageCodeSequence</i>	p_{16}	<i>PregnancyStatus</i>
p_3	<i>PatientBirthDate</i>	p_{10}	<i>PatientPrimaryLanguageModifierCodeSequence</i>	p_{17}	<i>LastMenstrualDate</i>
p_4	<i>PatientSex</i>	p_{11}	<i>OtherPatientIDs</i>	p_{18}	<i>PatientReligiousPreference</i>
p_5	<i>EthnicGroup</i>	p_{12}	<i>OtherPatientNames</i>	p_{19}	<i>PatientComments</i>
p_6	<i>IssuerOfPatientID</i>	p_{13}	<i>PatientBirthName</i>	p_{20}	<i>PatientAddress</i>
p_7	<i>PatientBirthTime</i>	p_{14}	<i>PatientTelephoneNumbers</i>	p_{21}	<i>PatientMotherBirthName</i>
				p_{22}	<i>InsurancePlanIdentification</i>

Figure 6.3: AUM of the entity table *Patient* in Workload W3

Workload W3 uses the entity table *Patient*. Figure 6.3 presents the matrix *AUM* built using the set of queries given in Table 6.7. This is a mixed workload. Table 6.12 presents a set of 7 HADF-generated data storage configurations and their statistics.

Table 6.12: Typical candidate configurations for *Patient*

Conf	Input						Output					
	Parameters				Entity Table		Typical Candidate Data Storage Configuration	No. of stored data cells	Null ratio	No. of joins	No. of scanned data cells	Exec. Time (sec)
	α	β	θ	λ	No. of data cells	Null ratio						
G ₁	0	0	0	0	2,767,038	84.83%	$C_{1,1} = \{UID, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{19}, p_{20}, p_{21}, p_{22}\} \Rightarrow \text{row store}$	2,767,038	84.83%	0	2,213,630,400	26,731
G ₂	0	0	0	1	2,767,038	84.83%	$C_{2,1} = \{UID, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{19}, p_{20}, p_{21}, p_{22}\} \Rightarrow \text{column store}$	2,767,038	84.83%	0	878,233,800	24,260
G ₃	0	0.4	0.5	0.6	2,767,038	84.83%	$C_{3,1} = \{UID, p_1, p_2, p_3, p_4, p_{16}\} \Rightarrow \text{row store}$ $C_{3,2} = \{UID, p_7, p_{15}, p_{17}\} \Rightarrow \text{column store}$ $C_{3,3} = \{UID, p_{13}, p_{14}, p_{19}, p_{21}\} \Rightarrow \text{row store}$ $C_{3,4} = \{UID, p_5\} \Rightarrow \text{row store}$ $C_{3,5} = \{UID, p_6, p_{12}, p_{22}\} \Rightarrow \text{row store}$ $C_{3,6} = \{UID, p_{10}, p_{11}, p_{20}\} \Rightarrow \text{row store}$ $C_{3,7} = \{UID, p_8, p_{18}\} \Rightarrow \text{row store}$ $C_{3,8} = \{UID, p_9\} \Rightarrow \text{row store}$	741,743	8.12%	1,800	584,102,300	29,482
G ₄	0.3	0.4	0.5	0.6	2,767,038	84.83%	$C_{4,1} = \{UID, p_1, p_2, p_3, p_4\} \Rightarrow \text{row store}$ $C_{4,2} = \{UID, p_{16}, p_{17}\} \Rightarrow \text{row store}$ $C_{4,3} = \{UID, p_7, p_{15}\} \Rightarrow \text{column store}$ $C_{4,4} = \{UID, p_{13}, p_{14}, p_{19}, p_{21}\} \Rightarrow \text{row store}$ $C_{4,5} = \{UID, p_5\} \Rightarrow \text{row store}$ $C_{4,6} = \{UID, p_6, p_{12}, p_{22}\} \Rightarrow \text{row store}$ $C_{4,7} = \{UID, p_{10}, p_{11}, p_{20}\} \Rightarrow \text{row store}$ $C_{4,8} = \{UID, p_8, p_{18}\} \Rightarrow \text{row store}$ $C_{4,9} = \{UID, p_9\} \Rightarrow \text{row store}$	653,075	4.31%	1,900	493,742,000	26,140
G ₅	0.5	0.4	0.5	0.6	2,767,038	84.83%	$C_{5,1} = \{UID, p_1, p_2, p_3, p_4\} \Rightarrow \text{row store}$ $C_{5,2} = \{UID, p_{13}, p_{14}, p_{19}, p_{21}\} \Rightarrow \text{row store}$ $C_{5,3} = \{UID, p_{15}, p_{17}\} \Rightarrow \text{column store}$ $C_{5,4} = \{UID, p_5\} \Rightarrow \text{row store}$ $C_{5,5} = \{UID, p_7, p_8, p_{16}, p_{18}\} \Rightarrow \text{row store}$ $C_{5,6} = \{UID, p_6, p_{12}, p_{22}\} \Rightarrow \text{row store}$ $C_{5,7} = \{UID, p_{10}, p_{11}, p_{20}\} \Rightarrow \text{row store}$ $C_{5,8} = \{UID, p_9\} \Rightarrow \text{row store}$	674,840	5.18%	1,900	498,143,000	27,140
G ₆	0.7	0.4	0.5	0.6	2,767,038	84.83%	$C_{6,1} = \{UID, p_1, p_2, p_3, p_4\} \Rightarrow \text{row store}$ $C_{6,2} = \{UID, p_{13}, p_{14}, p_{15}, p_{19}, p_{21}\} \Rightarrow \text{row store}$ $C_{6,3} = \{UID, p_5\} \Rightarrow \text{row store}$ $C_{6,4} = \{UID, p_7, p_8, p_{16}, p_{17}, p_{18}\} \Rightarrow \text{row store}$ $C_{6,5} = \{UID, p_6, p_{12}, p_{22}\} \Rightarrow \text{row store}$ $C_{6,6} = \{UID, p_{10}, p_{11}, p_{20}\} \Rightarrow \text{row store}$ $C_{6,7} = \{UID, p_9\} \Rightarrow \text{row store}$	696,782	6.01%	1,400	506,520,200	24,120
G ₇	1	0.4	0.5	0.6	2,767,038	84.83%	$C_{7,1} = \{UID, p_1, p_2, p_3, p_4, p_{13}, p_{14}, p_{15}, p_{19}, p_{21}\} \Rightarrow \text{row store}$ $C_{7,2} = \{UID, p_5\} \Rightarrow \text{row store}$ $C_{7,3} = \{UID, p_7, p_8, p_{16}, p_{17}, p_{18}\} \Rightarrow \text{row store}$ $C_{7,4} = \{UID, p_6, p_{12}, p_{22}\} \Rightarrow \text{row store}$ $C_{7,5} = \{UID, p_{10}, p_{11}, p_{20}\} \Rightarrow \text{row store}$ $C_{7,6} = \{UID, p_9\} \Rightarrow \text{row store}$	1,277,498	28.08%	900	977,337,200	25,140

Configurations $G_1 - G_7$, as shown in the table, are described as follows:

- **Configuration G_1 :** In this configuration, all the attributes of *Patient* is grouped into a single cluster $C_{1,1} = \{UID, p_1, p_2, \dots, p_{22}\}$, stored in a single row table. Some statistics relevant to this configuration are as follows: (1) the number of stored data cells is 2,767,038; (2) the null ratio is 84.83%; (3) no join operation is performed due to using only one table; (4) the number of data cells scanned by the workload is 2,213,630,400; and (5) the workload execution time is 26,731 seconds.
- **Configuration G_2 :** This configuration is similar to G_1 since all the attributes of *Patient* is grouped into a single cluster $C_{2,1} = \{UID, p_1, p_2, \dots, p_{22}\}$; however, it uses a single column table instead of a single row table. Like G_1 , it does not reduce the null ratio, but it helps the workload to significantly decrease the number of scanned data cells: only 878,233,800 data cells are scanned. The workload execution time when using G_2 is less than using G_1 : G_2 takes 24,260 seconds.
- **Configuration $G_3 - G_6$:** *Patient* is a wide and sparse table. In addition, some of its attributes are frequently accessed together by the same queries, while others are seldom accessed together. This is clearly shown in the matrix *AUM*, given in Figure 6.3. When α is respectively set to 0, 0.3, 0.5 and 0.7, the clustering phase of HADF groups the attributes that are highly correlated with each other based on Hybrid Similarity. When $\alpha = 0$, only Attribute Density Similarity has impact on the clustering result. However, when α is respectively set to 0.3, 0.5 and 0.7, there is a combined impact of both Attribute Density Similarity and Attribute Access Similarity on the clustering result such that the clustering phase groups the attributes into separate clusters. Storing *Patient* in multiple vertically partitioned tables will help to reduce the number of stored data cells, null values and scanned data cells. However, in general, Configurations $G_3 - G_6$ need a large number of join operations for tuple reconstruction, thus their workload execution time is not significantly reduced when compared to G_1 and G_2 . Among Configurations $G_3 - G_6$, G_6 gives the smallest number of joins (i.e., 1,400 joins); G_6 also gives the shortest workload execution time as well (i.e., 24,120 seconds).
- **Configuration G_7 :** This configuration is close to G_6 , but only takes into consideration the impact of the workload-specific information (due to setting $\alpha = 1$). It uses a less number of vertically partitioned tables than Configurations $G_3 - G_6$, thus needs a less number of joins than these configurations. Because the data-specific information has not been used, the null ratio and the number of stored data cells of G_7 could not be reduced to as low as those of Configurations $G_3 - G_6$.

We choose G_6 , which stores *Patient* in 7 different vertically partitioned tables, as a good configuration for the entity table *Patient* in terms of both the storage space size and the workload execution time.

Execution of Workload W4

Workload W4 is a mixed workload containing multiple-table join queries on the entity tables *Patient*, *Study*, *GeneralInfoTable* and *SequenceAttributes*. A good data storage configuration for this workload is created by combining the good ones of the entity tables. To achieve this, the following two steps are performed: (1) separate Workload

W4 into four sub-workloads, each of which is relevant to only one entity table; and (2) apply HADF to find a good data storage configuration for each entity table.

The sub-workloads relevant to three entity tables *GeneralInfoTable*, *SequenceAttributes* and *Patient* are the same as Workloads W1, W2 and W3, respectively; their good configurations have been chosen above. Therefore, below we only describe the steps to find a good configuration for *Study*.

Let us denote sW4 as the sub-workload including only the queries relevant to *Study*. In Table 6.13, we present the set of queries $Q_{4,1s} - Q_{4,6s}$ in sW4, which are respectively separated from the original queries $Q_{4,1} - Q_{4,6}$ in Workload W4 (given in Table 6.8).

Table 6.13: Workload sW4 for the entity table *Study*

Query	Query	Freq
Q _{4,1s}	SELECT StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, AccessionNumber, MedicalAlerts FROM Study WHERE StudyDate >= '20000101' AND StudyDate <= '20150101'	300
Q _{4,2s}	SELECT StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, MedicalRecordLocator FROM Study WHERE StudyID = '20050920'	100
Q _{4,3s}	SELECT PatientAge, PatientWeight, PatientSize FROM Study WHERE PatientAge >= 90	100
Q _{4,4s}	SELECT UID, StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, AccessionNumber, PatientWeight, AdditionalPatientHistory FROM Study	100
Q _{4,5s}	SELECT StudyInstanceUID, StudyDate, StudyTime, StudyID, PatientSize, Occupation FROM Study	100
Q _{4,6s}	SELECT StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, StudyDescription, PatientAge FROM Study WHERE StudyDate >= '20000101' AND StudyDate <= '20150101'	100

Figure 6.4 presents the matrix *AUM* of the entity table *Study* in Workload sW4.

	s ₁	s ₂	s ₃	s ₄	s ₅	s ₆	s ₇	s ₈	s ₉	s ₁₀	s ₁₁	s ₁₂	s ₁₃	s ₁₄	Freq
Q _{4,1s}	1	1	1	1	1	1	0	0	0	0	0	0	0	1	300
Q _{4,2s}	1	1	1	1	1	0	0	0	0	0	0	0	1	0	100
Q _{4,3s}	0	0	0	0	0	0	0	1	1	0	0	0	0	0	100
Q _{4,4s}	1	1	1	1	1	1	0	0	1	0	0	1	0	0	100
Q _{4,5s}	1	1	1	0	1	0	0	0	0	1	1	0	0	0	100
Q _{4,6s}	1	1	1	1	1	0	1	1	0	0	0	0	0	0	100

s ₁	<i>StudyInstanceUID</i>	s ₈	<i>PatientAge</i>
s ₂	<i>StudyDate</i>	s ₉	<i>PatientWeight</i>
s ₃	<i>StudyTime</i>	s ₁₀	<i>PatientSize</i>
s ₄	<i>ReferringPhysicianName</i>	s ₁₁	<i>Occupation</i>
s ₅	<i>StudyID</i>	s ₁₂	<i>AdditionalPatientHistory</i>
s ₆	<i>AccessionNumber</i>	s ₁₃	<i>MedicalRecordLocator</i>
s ₇	<i>StudyDescription</i>	s ₁₄	<i>MedicalAlerts</i>

Figure 6.4: *AUM* of the entity table *Study* in Workload sW4

Similarly to the cases of Workloads W1, W2 and W3, we apply HADF to produce a set of 7 typical candidate data storage configurations $G_1 - G_7$ for the entity table *Study*. Table 6.14 shows these configurations together with their statistics.

Configurations $G_1 - G_7$ in the table are explained as follows:

- **Configuration G₁:** In this configuration, all the attributes of *Study* is grouped into a single cluster $C_{1,1} = \{UID, p_1, p_2, \dots, p_{14}\}$ which is stored in a row table.

- **Configuration G2:** Similar to G₁, this configuration groups all the attributes of *Study* into a single column table, but stores it in a row table.

Table 6.14: Typical candidate configurations for *Study*

Conf	Input						Output					
	Parameters				Entity Table		Candidate Data Storage Configuration	No. of stored data cells	Null ratio	No. of joins	No. of scanned data cells	Exec. Time (sec)
	α	β	θ	λ	No. of data cells	Null ratio						
G ₁	0	0	0	0	1,804,590	43.83%	$C_{1,1} = \{UID, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}\} \Rightarrow \text{row store}$	1,804,590	43.83%	0	1,443,672,000	25,220
G ₂	0	0	0	1	1,804,590	43.83%	$C_{2,1} = \{UID, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}\} \Rightarrow \text{column store}$	1,804,590	43.83%	0	697,774,800	23,440
G ₃	0	0.4	0.5	0.6	1,804,590	43.83%	$C_{3,1} = \{UID, s_1, s_2, s_3, s_4, s_5, s_7, s_8, s_9\} \Rightarrow \text{column store}$ $C_{3,2} = \{UID, s_{12}\} \Rightarrow \text{row store}$ $C_{3,3} = \{UID, s_{10}, s_{11}\} \Rightarrow \text{row store}$ $C_{3,4} = \{UID, s_6, s_{14}\} \Rightarrow \text{row store}$ $C_{3,5} = \{UID, s_{13}\} \Rightarrow \text{row store}$	1,207,777	5.25%	700	584,516,800	26,600
G ₄	0.3	0.4	0.5	0.6	1,804,590	43.83%	like G ₃	like G ₃	like G ₃	like G ₃	like G ₃	like ₃
G ₅	0.5	0.4	0.5	0.6	1,804,590	43.83%	like G ₃	like G ₃	like G ₃	like G ₃	like G ₃	like G ₃
G ₆	0.7	0.4	0.5	0.6	1,804,590	43.83%	$C_{6,1} = \{UID, s_1, s_2, s_3, s_4, s_5, s_6, s_{14}\} \Rightarrow \text{row store}$ $C_{6,2} = \{UID, s_8, s_9\} \Rightarrow \text{column store}$ $C_{6,3} = \{UID, s_{13}\} \Rightarrow \text{row store}$ $C_{6,4} = \{UID, s_{12}\} \Rightarrow \text{row store}$ $C_{6,5} = \{UID, s_{10}, s_{11}\} \Rightarrow \text{row store}$ $C_{6,6} = \{UID, s_7\} \Rightarrow \text{row store}$	1,665,910	18.63%	600	792,180,600	25,400
G ₇	1.0	0.4	0.5	0.6	1,804,590	43.83%	$C_{7,1} = \{UID, s_1, s_2, s_3, s_4, s_5, s_6, s_{14}\} \Rightarrow \text{row store}$ $C_{7,2} = \{UID, s_9, s_{12}\} \Rightarrow \text{column store}$ $C_{7,3} = \{UID, s_7, s_8\} \Rightarrow \text{column store}$ $C_{7,4} = \{UID, s_{13}\} \Rightarrow \text{row store}$ $C_{7,5} = \{UID, s_{10}, s_{11}\} \Rightarrow \text{row store}$	1,676,028	22.08%	500	790,319,200	25,980

- **Configuration G₃ - G₆:** When α is respectively set to 0, 0.3, 0.5 and 0.7, the clustering phase of HADF takes into account the impact of both workload- and data-specific information on the clustering result. The entity table *Study* is stored in several vertically partitioned tables using both row and column stores. The number of stored data cells, null values and scanned data cells are reduced when compared with G₁ and G₂. However, their workload execution time is lightly higher than that of G₁ and G₂ due to the costs needed for additional join operations.
- **Configuration G₇:** This configuration only takes into account the workload-specific information on the clustering result (due to $\alpha = 1$). Therefore, although its number of join operations has been decreased to lower than that of Configurations G₃ - G₆, its number of null values and scanned data cells are still high. Its workload execution time is also lightly higher than that of Configurations G₁, G₂ and G₆.

We choose G₂ which stores the entity table *Study* in single column table because it has the lowest workload execution time.

6.3.2 Experiment 2: Evaluating HYTORMO and HADF using More Data and Multiple-table Joins

Similarly to Experiment 1, Experiment 2 also aims at evaluating Hypotheses H1 and H2, but it uses more data and multiple-table join queries.

Table 6.15: Major steps of Experiment 2

Conf	Typical candidate configuration	Execution	Measures
G*	Good HADF-generated data storage configuration, i.e., the one is composed of good configurations of all the entity tables T_i 's chosen in Experiment 1.	- Run Workload W4 five times for each configuration.	- Storage space size.
G ₁	Pure row tables (all T_i 's are stored in row tables).	- Using two datasets MDB1 and MDB2 separately.	- Execution time of W4.
G ₂	Pure column tables (all T_i 's are stored in column tables).		

Table 6.15 presents the major steps of Experiment 2. First, we create three different configurations: (1) G^* : a good HADF-generated data storage configuration, composed of good configurations of all the entity tables T_i 's chosen in Experiment 1 (T_i is *Patient*, *Study*, *GeneralInfoTable* or *SequenceAttributes*); (2) G_1 : pure row tables; and (3) G_2 : pure column tables. Next, we apply these configurations to execute Workload W4 using two datasets MDB1 and MDB2, separately: for each configuration, W4 is run five times; its average execution time is calculated. Finally, we compare these configurations in terms of storage space size and/or workload execution time.

Tables 6.16, 6.17 and 6.18 present Configurations G^* , G_1 and G_2 , respectively, in the forms of vertically partitioned tables, instead of clusters as in Experiment 1.

Table 6.16: Configuration G^* of Experiment 2

No.	Entity Table	Data Storage Configuration
		PatientP1P2P3P4(UID, PatientName, PatientID, PatientBirthDate, PatientSex) => <i>row store</i>
		PatientP13P14P15P19P21(UID, PatientBirthName, PatientTelePhoneNumbers, SmokingStatus, PatientComments, PatientMotherBirthName) => <i>row store</i>
		PatientP5(UID, EthnicGroup) => <i>row store</i>
		PatientallP7P8P16P17P18(UID, PatientBirthTime, PatientInsurancePlanCodeSequence, PregnancyStatus, LastMenstrualDate, PatientReligiousPreference) => <i>row store</i>
		PatientP6P12P22(UID, IssuerOfPatientID, OtherPatientNames, InsurancePlanIdentification) => <i>row store</i>
		PatientP10P11P20(SOPInstanceUID, PatientPrimaryLanguageModifier-CodeSequence, OtherPatientIDs, PatientAddress) => <i>row store</i>
		PatientP9(UID, PatientPrimaryLanguageCodeSequence) => <i>row store</i>
2	Study	Study(UID, StudyInstanceUID, StudyDate, StudyTime, Referring-PhysicianName, StudyID, AccessionNumber, StudyDescription, PatientAge, PatientWeight, PatientSize, Occupation, AdditionalPatient-History, MedicalRecordLocator, MedicalAlerts) => <i>column store</i>
3	GeneralInfoTable	GeneralInfoTable(UID, GeneralTags, GeneralVRs, GeneralNames, GeneralValues) => <i>column store</i>
4	SequenceAttributes	SequenceAttributes(UID, SequenceTags, SequenceVRs, SequenceNames, SequenceValues) => <i>row store</i>

Table 6.17: Configuration G_1 of Experiment 2

No.	Entity Table	Data Storage Configuration
1	Patient	Patient (UID, PatientName, PatientID, PatientBirthDate, PatientSex, EthnicGroup, IssuerOfPatientID, PatientBirthTime, PatientInsurancePlanCodeSequence, PatientPrimaryLanguageCodeSequence, PatientPrimaryLanguageModifierCodeSequence, OtherPatientIDs, OtherPatientNames, PatientBirthName, PatientTelePhoneNumbers, SmokingStatus, PregnancyStatus, LastMenstrualDate, PatientReligiousPreference, PatientComments, PatientAddress, PatientMotherBirthName, InsurancePlanIdentification) => <i>row store</i>
2	Study	Study(UID, StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, AccessionNumber, StudyDescription, PatientAge, PatientWeight, PatientSize, Occupation, AdditionalPatientHistory, MedicalRecordLocator, MedicalAlerts) => <i>row store</i>
3	GeneralInfoTable	GeneralInfoTable(UID, GeneralTags, GeneralVRs, GeneralNames, GeneralValues) => <i>row store</i>
4	SequenceAttributes	SequenceAttributes(UID, SequenceTags, SequenceVRs, SequenceNames, SequenceValues) => <i>row store</i>

Table 6.18: Configuration G_2 of Experiment 2

No.	Entity Table	Data Storage Configuration
1	Patient	Patient (UID, PatientName, PatientID, PatientBirthDate, PatientSex, EthnicGroup, IssuerOfPatientID, PatientBirthTime, PatientInsurancePlanCodeSequence, PatientPrimaryLanguageCodeSequence, PatientPrimaryLanguageModifierCodeSequence, OtherPatientIDs, OtherPatientNames, PatientBirthName, PatientTelePhoneNumbers, SmokingStatus, PregnancyStatus, LastMenstrualDate, PatientReligiousPreference, PatientComments, PatientAddress, PatientMotherBirthName, InsurancePlanIdentification) => <i>column store</i>
2	Study	Study(UID, StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, AccessionNumber, StudyDescription, PatientAge, PatientWeight, PatientSize, Occupation, AdditionalPatientHistory, MedicalRecordLocator, MedicalAlerts) => <i>column store</i>
3	GeneralInfoTable	GeneralInfoTable(UID, GeneralTags, GeneralVRs, GeneralNames, GeneralValues) => <i>column store</i>
4	SequenceAttributes	SequenceAttributes(UID, SequenceTags, SequenceVRs, SequenceNames, SequenceValues) => <i>column store</i>

Tables 6.19 and 6.20 respectively present the average workload execution time obtained over five runs when applying three Configurations G^* , G_1 and G_2 corresponding to two different cases: (1) using the dataset MDB1; and (2) using the dataset MDB2. The experimental results show that G^* provides the shortest workload execution time among these three configurations: it takes 35,940 seconds to perform W4 on MDB1, and 118,940 seconds to perform W4 on MDB2. In addition, two data storage configurations G_1 and G_2 have the same storage space requirement, whereas Configuration G^* has the smallest storage space size because the entity table *Patient* has been reduced by 75% (as shown in Experiment 1) after removing null rows from its vertical partitions.

Table 6.19: Execution time of Workload W4 over 3 configurations using MDB1

Conf	Data Storage Configuration	Exec. Time (sec)
G*	Good HADF-generated data storage configuration	35,940
G ₁	Pure row tables	37,860
G ₂	Pure column tables	36,960

Table 6.20: Execution time of Workload W4 over 3 configurations using MDB2

Conf	Data Storage Configuration	Exec. Time (sec)
G*	Good HADF-generated data storage configuration	118,940
G ₁	Pure row tables	161,040
G ₂	Pure column tables	120,120

6.3.3 Experiment 3: Comparison between HADF and HoVer

Experiment 3 aims at further evaluating Hypothesis H2a that shows the usefulness of the combined use of both workload- and data-specific information in HADF. To obtain this, we compare HADF and HoVer approach that was proposed by B. Cui et al. [14]. The experiment is performed according to the major steps as given in Table 6.21.

Table 6.21: Major steps of Experiment 3

Conf	Typical candidate configuration	Execution	Measures
G*	Good HADF-generated data storage configuration that is chosen for the entity table T_i in workload W_j 's, where $j = 1, 2$.	- Run Workload W_j ($j = 1, 2$) five times for each configuration. - Using the dataset MDB2.	- Storage space size of T_i . - Workload execution time.
G ₁ - G ₆	- Setting: $\beta = 0, 0.2, 0.4, 0.6, 0.8, 1$. - HoVer-generated data storage configuration for the entity table T_i is stored in row tables.		

First of all, we prepare the following configurations: (1) *Configuration G** is a good HADF-generated data storage configuration, obtained in Experiment 1, for the entity table T_i , where T_i is used to refer to *GeneralInfoTable* or *Sequenceattributes*, in workload W_j 's, where $j = 1, 2$; (2) *Configurations G₁ - G₆* are configurations generated by applying the HoVer approach. It is worthy to remind that the HoVer approach is similar to the clustering phase of HADF; it is a clustering algorithm that groups the similar attributes into the same column groups. However, the HoVer approach only uses Attribute Density Similarity, a clustering threshold β and a row store, instead of using a Hybrid Similarity, $\alpha, \beta, \theta, \lambda$ and a hybrid store as the clustering phase of HADF. In other words, the HoVer approach uses only the data-specific information and the row store instead of a combined use of both workload- and data-specific information together with a hybrid store. Therefore, to achieve a set of 6 data storage configurations, we will set β to 0, 0.2, 0.4, 0.6, 0.8 and 1 for the HoVer approach. Next, we build these configurations in HYTORMO and execute the workloads W_j 's ($j = 1, 2$) using the dataset MDB2. Each workload is also run five time for each configurations; its average execution time is calculated. Finally, we compare these configurations.

Following the above steps, the results of the executions of two workloads W1 and W2 are given below.

Execution of Workload W1

Workload W1 uses only the entity table *GeneralInfoTable* and its queries are shown in Table 6.5. First, we apply the good HADF-generated configuration, i.e., Configuration G* of *GeneralInfoTable* obtained in Experiment 1, to execute this workload. This configuration store *GeneralInfoTable* in a single column table. Table 6.22 presents the result obtained from the execution.

Table 6.22: Good HADF-generated configuration for *GeneralInfoTable*

Conf	Input						Output					
	Parameters				Entity Table		Typical Candidate Data Storage Configuration	No. of stored data cells	Null ratio	No. of joins	No. of scanned data cells	Exec. Time (sec)
	α	β	θ	λ	No. of data cells	Null ratio						
G*	0	0.4	0.5	0.6	1,688,651,610	2.55%	$C_{3,1} = \{UID, i_1, i_2, i_3, i_4\}$ \Rightarrow column store	1,688,651,610	2.55%	0	405,276,386,400	23,520

(i_1 : GeneralTags; i_2 : GeneralVRs; i_3 : GeneralNames; i_4 : GeneralValues)

Next, we apply the Hover approach [14] to generate a set of 6 typical candidate data storage configurations $G_1 - G_6$ for the entity table *GeneralInfoTable*. These configurations are described in Table 6.23. These configurations can be also obtained by applying HADF with the following values of its parameters: (1) $\beta = 0, 0.2, 0.4, 0.6, 0.8$ and 1; (2) $\alpha = 0$ (i.e., only taking into account the impact of data-specific information); (3) $\theta = 1$ (i.e., not merging any pair of clusters together); and (4) $\lambda = 0$ (i.e., column groups are always stored in a row store).

Table 6.23: HoVer-generated configurations for *GeneralInfoTable*

Conf	Input				Output						
	Parameter β	Entity Table		Typical Candidate Data Storage Configuration	No. of stored data cells	Null ratio	No. of joins	No. of scanned data cells	Exec. Time (sec)		
		No. of data cells	Null ratio								
G ₁	0	1,688,651,610	2.55%	$C_{1,1} = \{UID, i_1, i_2, i_3, i_4\}$ \Rightarrow row store	1,688,651,610	2.55%	0	675,460,644,000	26,940		
G ₂ - G ₅	0.2;0.4;0.6;0.8	1,688,651,610	2.55%	like G ₁	like G ₁	like G ₁	like G ₁	like G ₁	like G ₁		
G ₆	1	1,688,651,610	2.55%	$C_{6,1} = \{UID, i_1\} \Rightarrow$ row store $C_{6,2} = \{UID, i_2\} \Rightarrow$ row store $C_{6,3} = \{UID, i_3\} \Rightarrow$ row store $C_{6,4} = \{UID, i_4\} \Rightarrow$ row store	2,633,036,392	0.00%	400	526,607,278,400	475,990		

(i_1 : GeneralTags; i_2 : GeneralVRs; i_3 : GeneralNames; i_4 : GeneralValues)

When applying HADF, we found that: *GeneralInfoTable* is a dense table, the similarity between any pair of two attributes in terms of Attribute Density Similarity is high, while the similarity between any pair of two attributes in terms of Attribute Access Similarity is low (because the attributes of *GeneralInfoTable* are seldom access together). However, since HADF can take into account the combined impact of both workload- and data-specific information on the clustering result, it found that Hybrid Similarity between any pair of two attributes is high enough such that all the attributes are grouped into a single cluster. Besides, Intra-Cluster Similarity of this cluster is not

high enough such that HADF decides to store it cluster in a column table, which is G^* in Table 6.22. On the other hand, when applying the HoVer approach, if the values of β is set to 0, 0.2, 0.4, 0.6 or 0.8 (i.e., $G_1 - G_5$ in Table 6.23), the clustering algorithm of the Hover approach also found that Attribute Density Similarity between any pair of two attributes is always greater than or equal to the corresponding value of β such that all the attributes of *GeneralInfoTable* are grouped and stored together in a single row table. When β is set to 1, the entity table *GeneralInfoTable* is decomposed and stored in single-attribute tables in row store (i.e., G_6 in Table 6.23).

With regards to data storage space demand and workload execution time, we can clearly see that with the combined use of both the workload-specific and the data-specific information together with a hybrid store, HADF can provide a better data storage configuration than the HoVer approach. It can suggest to store the piece of data used for an OLAP workload as Workload W1 in a column store. The execution time of Workload W1 when using the good HADF-generated data storage configuration is 23,520 seconds (G^* in Table 6.22), whereas this time when using the good HoVer-generated data storage configuration is 26,940 second (G_1 in Table 6.23).

Execution of Workload W2

Workload W2 uses only the entity table *Sequenceattributes* and its queries are shown in Table 6.6. First of all, we execute this workload using the good HADF-generated configuration, i.e., Configuration G^* of *Sequenceattributes* obtained in Experiment 1. Table 6.24 shows the result of this execution.

Table 6.24: Good HADF-generated configurations for *Sequenceattributes*

Conf	Input						Output					
	Parameters				Entity Table		Typical Candidate Data Storage Configuration	No. of stored data cells	Null ratio	No. of joins	No. of scanned data cells	Exec. Time (sec)
	α	β	θ	λ	No. of data cells	Null ratio						
G^*	0	0.4	0.5	0.6	376,574,510	0.082%	$C_{3,1} = \{UID, e_1, e_2, e_3, e_4\}$ => row store	376,574,510	0.082%	0	75,314,902,000	5,640

(e_1 : SequenceTags; e_2 : SequenceVRs; e_3 : SequenceNames; e_4 : SequenceValues)

Sequenceattributes is a dense table, thus the similarity between any pair of two attributes in terms of either Attribute Density Similarity or Attribute Access Similarity is high. As a result, when HADF takes into account the combined impact of both workload- and data-specific information, it found that Hybrid Similarity between any pair of two attributes is very high such that all the attributes are grouped into a single cluster. Furthermore, Intra-Cluster Similarity of this cluster is high enough such that HADF decides to store the cluster in a row table.

Alternatively, Table 6.25 presents the HoVer-generated configurations $G_1 - G_6$ and their statistics. When β is set to 0, 0.2, 0.4, 0.6 or 0.8 (i.e., $G_1 - G_5$ in Table 6.25), the clustering algorithm of the HoVer approach found that the Attribute Density Similarity between any pair of two attributes is always greater than or equal to the corresponding value of β , thus it groups and stores all the attributes of *Sequenceattributes* together in the same row table. In contrast, if β is set to 1, the entity table *Sequenceattributes* is decomposed and stored in four single-attribute tables in a row store (i.e., G_6 in Table 6.25).

Table 6.25: HoVer-generated configurations for *Sequenceattributes*

Conf	Input			Output					
	Parameter β	Entity Table		Typical Candidate Data Storage Configuration	No. of stored data cells	Null ratio	No. of joins	No. of scanned data cells	Exec. Time (sec)
		No. of data cells	Null ratio						
G ₁	0	376,574,510	0.082%	$C_{1,1} = \{UID, e_1, e_2, e_3, e_4\}$ \Rightarrow row store	376,574,510	0.082 %	0	75,314,902,000	5,640
G ₂ - G ₅	0.2; 0.4;0.6;0.8	376,574,510	0.082%	like G ₁	like G ₁	like G ₁	like G ₁	like G ₁	like G ₁
G ₆	1	376,574,510	0.082%	$C_{6,1} = \{UID, e_1\} \Rightarrow$ row store $C_{6,2} = \{UID, e_2\} \Rightarrow$ row store $C_{6,3} = \{UID, e_3\} \Rightarrow$ row store $C_{6,4} = \{UID, e_4\} \Rightarrow$ row store	602,026,842	0.00%	500	105,391,543,000	31,280

(e_1 : SequenceTags; e_2 : SequenceVRs; e_3 : SequenceNames; e_4 : SequenceValues)

Therefore, in the case of an OLTP workload as Workload W2, HADF is able to provide a data storage configuration that is as good as the configurations generated by the clustering algorithm of the HoVer approach. A row store is used to store the piece of data used for the OTLP workload. The execution time of Workload W2 when using the good HADF- or HoVer-generated data storage configuration is 5,640 seconds (G* in Table 6.24 and G₁ in Table 6.25).

6.3.4 Experiment 4: Evaluating the Effectiveness of the IBF

Experiment 4 aims at evaluating Hypothesis H3 that shows the benefit of the IBF. To achieve this, it compares the execution time of a query with and without using the IBF.

Table 6.26: Major steps of Experiment 4

Query	Selectivity	Execution	Measures
- Choosing a t -th query $Q_{W,j,t}$ in Workload W_j .	- Specifying predicate sets and their selectivity ratios used in $Q_{W,j,t}$.	- Executing $Q_{W,j,t}$ five times for each predicate using the good configuration G* when (1) using IBF and (2) not using IBF. - Using the dataset MDB2.	- Execution time of $Q_{W,j,t}$.

Table 6.26 presents the major steps of Experiment 4: (1) Choosing a query $Q_{W,j,t}$. (2) Specifying predicate sets and their selectivity ratios in $Q_{W,j,t}$. (3) Executing $Q_{W,j,t}$ five times using the good configuration G* (chosen in Experiment 1) and the dataset MDB2 with respect to a particular predicate set for two cases: using and not using an IBF. The average query execution time obtained over five runs is calculated. (4) Comparing the query execution time. We use $Q_{4,3}$ in Workload W4 for $Q_{W,j,t}$:

$Q_{4,3}$: SELECT Patient.UID, Patient.PatientID, Patient.PatientName, Patient.PatientBirthDate, Patient.PatientSex, Patient.EthnicGroup, Patient.SmokingStatus, Study.PatientAge, Study.PatientWeight, Study.PatientSize, GeneralInfoTable.GeneralNames, GeneralInfoTable.GeneralValues, SequenceAttributes.UID, SequenceAttributes.SequenceTags, SequenceAttributes.SequenceVRs, SequenceAttributes.SequenceNames, SequenceAttributes.SequenceValues FROM Patient, Study, GeneralInfoTable, SequenceAttributes WHERE Patient.UID = Study.UID AND Patient.UID = GeneralInfoTable.UID AND Patient.UID = SequenceAttributes.UID AND Patient.PatientSex = 'M' AND Patient.SmokingStatus = 'NO' AND Study.PatientAge >= 60 AND SequenceAttributes.SequenceNames LIKE '%X-Ray%'

We use the query $Q_{4,3}$ because it is a typical multiple-table join query, where four entity tables *Patient*, *Study*, *SequenceAttributes* and *GeneralInfoTable* are joined with each other; additionally, the effectiveness of the query processing with the use of the IBF in other cases, e.g., using only one entity table, is similar to this case.

The entity tables are stored in the hybrid store of HYTORMO according to the good data storage configuration G^* , described in Table 6.16: *Patient* is decomposed into vertically partitioned tables and stored in a row store; *Study* and *GeneralInfoTable* are stored in a column store; and *SequenceAttributes* is stored in a row store.

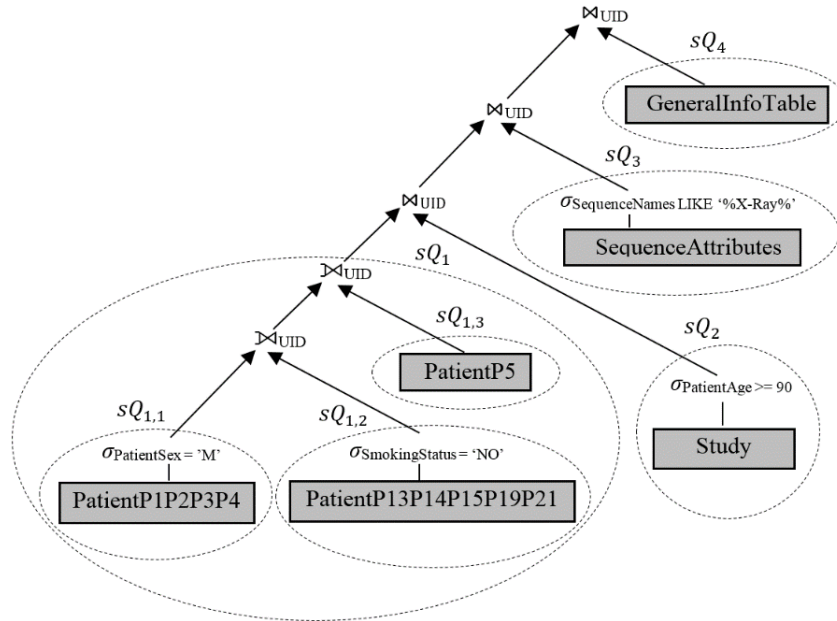


Figure 6.5: Execution plan for the query $Q_{4,3}$

Figure 6.5 shows the execution plan tree used for the query $Q_{4,3}$. (This execution plan tree is different to the one presented in Figure 5.1 in Chapter 5, where the expert-based design approach was applied to create data storage configurations.) Here, $Q_{4,3}$ first is decomposed into a set of sub-queries sQ_1 , sQ_2 , sQ_3 , and sQ_4 which access four entity tables *Patient*, *Study*, *SequenceAttributes* and *GeneralInfoTable*, respectively. Next, each of these sub-queries is further decomposed into smaller sub-queries to be able to access relevant row and column tables. For instance, the sub-query sQ_1 is decomposed into three sub-queries $sQ_{1,1}$, $sQ_{1,2}$ and $sQ_{1,3}$ to access three vertically partitioned tables *PatientP1P2P3P4*, *PatientP13P14P15P19P21* and *PatientP5*, respectively. On the other hand, the sub-queries sQ_2 , sQ_3 and sQ_4 are not further decomposed because they can directly access the single tables *Study*, *SequenceAttributes* and *GeneralInfoTable*, respectively. This execution plan tree is a left-deep processing tree whose relational operators are scheduled to be executed step by step while trying to keep intermediate results as small as possible. During the query execution, the results of the sub-queries are joined over the attribute *UID*. To prevent the data loss in the query result, the sub-query sQ_1 consists of two left-outer joins:

$$\begin{aligned}
 & sQ_1 \\
 & = \left(\left(\sigma_{PatientSex = 'M'}(PatientP1P2P3P4) \right) \bowtie_{UID} \left(\sigma_{SmokingStatus = 'NO'}(PatientP13P14P15P19) \right) \right) \\
 & \quad \bowtie_{UID} PatientP5.
 \end{aligned}$$

To improve the query performance, each left-outer join is rewritten to an inner join if there exists a non-null constraint on the right-hand side table of that left-outer join (applying Rule 3 given in Chapter 5). sQ_1 contains a predicate $\sigma_{SmokingStatus = 'NO'}$ on the table $PatientP13P14P15P19P2$, thus it is rewritten as follows:

$$sQ_1 = \left(\sigma_{PatientSex = 'M'}(PatientP1P2P3P4) \bowtie_{UID} \sigma_{SmokingStatus = 'NO'}(PatientP13P14P15P19) \right) \bowtie_{UID} PatientP5.$$

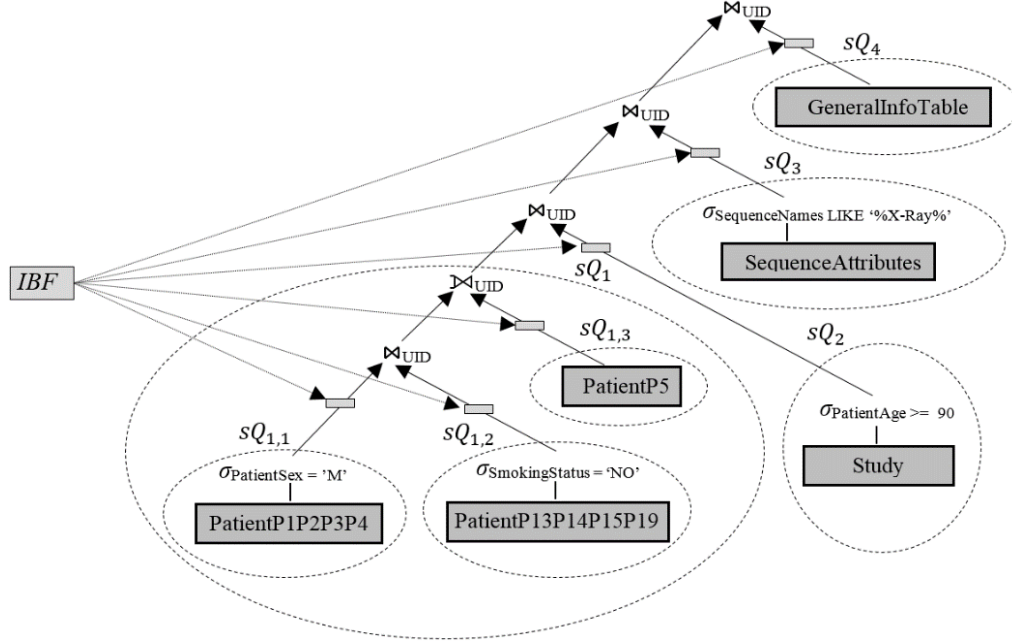


Figure 6.6: Execution plan for the query $Q_{4,3}$ with the IBF

Furthermore, an IBF is built from BFs created on the attribute UID of the result tables of the sub-queries $sQ_{1,1}$, $sQ_{1,2}$, sQ_2 , sQ_3 and sQ_4 . However, a BF will not be computed for a right-hand side table of a left-outer join, e.g., $PatientP5$. The IBF is computed by performing bitwise AND operations on all the BFs and applied to filter irrelevant tuples out of the input tables before joins occur. The new execution plan after reducing the number of left-outer joins and using the IBF is given in Figure 6.6.

All the BFs and the IBF have the same configuration, i.e., a bit vector with a length of m and a set of k hash functions, thus we need to choose a suitable configuration for them. The accuracy of a Bloom filter BF_i can be decided by ratio m/n_i where m is the length of bit vector and n_i is the size of set represented in BF_i . In our case, we already know the size n_i , which is the cardinality of the attribute UID of each input table T_i (i.e., $PatientP1P2P3P4$, $PatientP13P14P15P19P21$, $Study$, $SequenceAttributes$ and $GeneralInfoTable$), we thus need to determine the length m of each BF_i and the number of hash functions k to obtain a high accuracy for each BF_i . In particular, the false positive probability of BF_i is $P_{BF_i} \approx \left(1 - e^{-kn_i/m}\right)^k$ when BF_i is using k independent hash functions and a vector of m bits used to represent a set of n_i values (see Formula (5.3.1)). This probability can achieve the minimum $\left(\frac{1}{2}\right)^k$ or $(0.6185)^{m/n_i}$ (see Formula (5.3.3)) when $k = \ln(2) \times \frac{m}{n_i}$ (see Formula (5.3.2)).

We use the cardinality of attribute *UID* of the entity table *Patient* of the dataset MDB2 for n_i , i.e., $n_i = 1,802,277$. This is because the entity table *Patient* contains all values of the attribute *UID*. Besides, $m = 8 \times n_i$ is regarded as a good tradeoff between accuracy and space storage used for a Bloom filter [22]; thus, we set $m = 8 \times n_i$ bits, i.e., $m = 14,418,216 \approx 14\text{MB}$. Then, we use Formula (5.3.2) in order to compute the corresponding number k of hash functions: we get $k = \ln(2) \times 8 \approx 6$ hash functions; and the false positive probability becomes 0.0156.

Table 6.27: Sets of predicates on the attributes in the input tables

Pre. Set	PatientP1P2P3P4		PatientP13P14P15P19P21		Study		SequenceAttributes	
	Sel.	Predicate	Sel.	Predicate	Sel.	Predicate	Sel.	Predicate
1	1	No predicate	1	No predicate	1	No predicate	1	No predicate
2	1	No predicate	1	No predicate	0.6327	PatientAge >= 10	1	No predicate
3	0.4764	Patientsex = 'M'	1	No predicate	0.6327	PatientAge >= 10	1	No predicate
4	0.4764	Patientsex = 'M'	1	No predicate	0.2462	PatientAge >= 60	1	No predicate
5	0.4764	Patientsex = 'M'	0.0017	smokingstatus = 'NO'	0.2462	PatientAge >= 60	1	No predicate
6	0.4764	Patientsex = 'M'	0.0017	smokingstatus = 'NO'	0.0061	PatientAge >= 90	0.0019	SequenceNames LIKE '%X-Ray%'

To assess the benefit of the IBF, we compare the query performance difference between two cases: using and not using the IBF. The query $Q_{4,3}$ consists of the predicates on the attributes *PatientSex*, *SmokingStatus*, *PatientAge* and *SequenceNames* of the input tables *PatientP1P2P3P4*, *PatientP13P14P15P19P21*, *Study* and *SequenceAttributes*; however, to observe the impact of the IBF over a range of situations, we will modify the predicates to change the selectivity of the input tables. Table 6.27 presents six different sets of predicates (Pre. Set) on the attributes of the input tables. The selectivity (Sel.) of each individual predicate is also specified. (The query $Q_{4,3}$ in Figure 6.5 and 6.6 is corresponding to the 6th selectivity set in the table.)

Table 6.28: Comparison of the execution time of using and not using the IBF

Pre. Set	Execution time when not using IBF		Execution time when using IBF		Reduced time ratio (%)
	Average (sec)	Std. dev.	Average (sec)	Std. dev.	
1	1264.80	389.20	1007.20	176.89	20%
2	1209.20	234.63	748.00	92.29	38%
3	1068.40	438.10	962.80	197.97	10%
4	1122.80	330.83	908.80	202.48	19%
5	1215.80	407.01	964.80	189.23	21%
6	1452.40	421.58	930.40	127.05	36%

The IBF is computed and applied, no matter what set of predicates is used for the input tables. Table 6.28 presents a comparison of the execution time (obtained over five runs for each set of predicates) between using and not using the IBF: the average and the standard deviation (std. dev.) of the execution times are given. We also provide the reduced time ratio when using the IBF. This ratio is computed by Formula (6.4.2).

$$\text{Reduced time ratio} = \frac{\text{Execution time without using IBF} - \text{Execution time with using IBF}}{\text{Execution time without using IBF}} \quad (6.4.2)$$

The comparison of the query execution time between two cases, using and not using the IBF, shows that the performance query is significantly improved for all sets

of predicates. The query execution time is reduced to 10-38% of the time when the query processing strategy is not using the IBF.

Table 6.29: Comparison of the sizes of the input tables before and after using IBF

Pre. Set	Patient-P1P2P3P4		Patient-P13P14P15P19P21		PatientP5		Study		Sequence-Attributes		GeneralInfoTable	
	Size	RISR	Size	RISR	Size	RISR	Size	RISR	Size	RISR	Size	RISR
1	Before using IBF: 1,802,376 After using IBF: 543,963	70%	Before using IBF: 579,605 After using IBF: 543,708	58%	Before using IBF: 391,332 After using IBF: 243,241	38%	Before using IBF: 1,856,892 After using IBF: 543,963	71%	Before using IBF: 75,314,902 After using IBF: 28,750,207	62%	Before using IBF: 337,730,322 After using IBF: 129,188,521	62%
2	Before using IBF: 1,802,376 After using IBF: 392,871	78%	Before using IBF: 579,605 After using IBF: 392,661	60%	Before using IBF: 391,332 After using IBF: 231,159	41%	Before using IBF: 1,174,845 After using IBF: 392,381	67%	Before using IBF: 75,314,902 After using IBF: 21,926,259	71%	Before using IBF: 337,730,322 After using IBF: 92,537,771	73%
3	Before using IBF: 858,729 After using IBF: 179,414	79%	Before using IBF: 579,605 After using IBF: 179,275	69%	Before using IBF: 391,332 After using IBF: 179,275	54%	Before using IBF: 1,174,845 After using IBF: 179,066	85%	Before using IBF: 75,314,902 After using IBF: 10,414,222	86%	Before using IBF: 337,730,322 After using IBF: 41,966,822	88%
4	Before using IBF: 858,729 After using IBF: 74,868	91%	Before using IBF: 579,605 After using IBF: 74,893	90%	Before using IBF: 391,332 After using IBF: 57,018	85%	Before using IBF: 457,115 After using IBF: 74,904	84%	Before using IBF: 75,314,902 After using IBF: 4,512,373	94%	Before using IBF: 337,730,322 After using IBF: 17,798,449	95%
5	Before using IBF: 858,729 After using IBF: 0	100%	Before using IBF: 3,034 After using IBF: 0	100%	Before using IBF: 391,332 After using IBF: 0	100%	Before using IBF: 457,115 After using IBF: 0	100%	Before using IBF: 75,314,902 After using IBF: 0	100%	Before using IBF: 337,730,322 After using IBF: 0	100%
6	Before using IBF: 858,729 After using IBF: 0	100%	Before using IBF: 3,034 After using IBF: 0	100%	Before using IBF: 391,332 After using IBF: 0	100%	Before using IBF: 11,372 After using IBF: 0	100%	Before using IBF: 146,217 After using IBF: 0	100%	Before using IBF: 337,730,322 After using IBF: 0	100%

(RISR: the reduced input size ratio)

Table 6.29 provides a comparison of the sizes of the input tables before and after using the IBF. The reduced input size ratio (RISR) is computed by Formula (6.4.3); besides, the size of the input tables is measured in terms of the number of tuples (rows).

$$\text{Reduced input size ratio} = \frac{\text{Input size without using IBF} - \text{Input size with using IBF}}{\text{Input size without using IBF}} \quad (6.4.3)$$

The IBF has filtered out many irrelevant tuples from the input tables of joins. The reduced input size ratio of the input tables increases when the selectivity of predicates

in the query increases. For instance, as shown in the table, the size of the input table *GeneralInfoTable* is reduced to 62-100% of the size without using the IBF.

In this experiment, we also assess the effectiveness of an incremental IBF, introduced in Section 5.3.3 of Chapter 5. For this purpose, the incremental IBF is computed from Bloom filters created from the result tables of the sub-queries of the lower join operations in the execution plan. Then, it is applied as a local predicate of the sub-queries of the upper join operations of this execution plan.

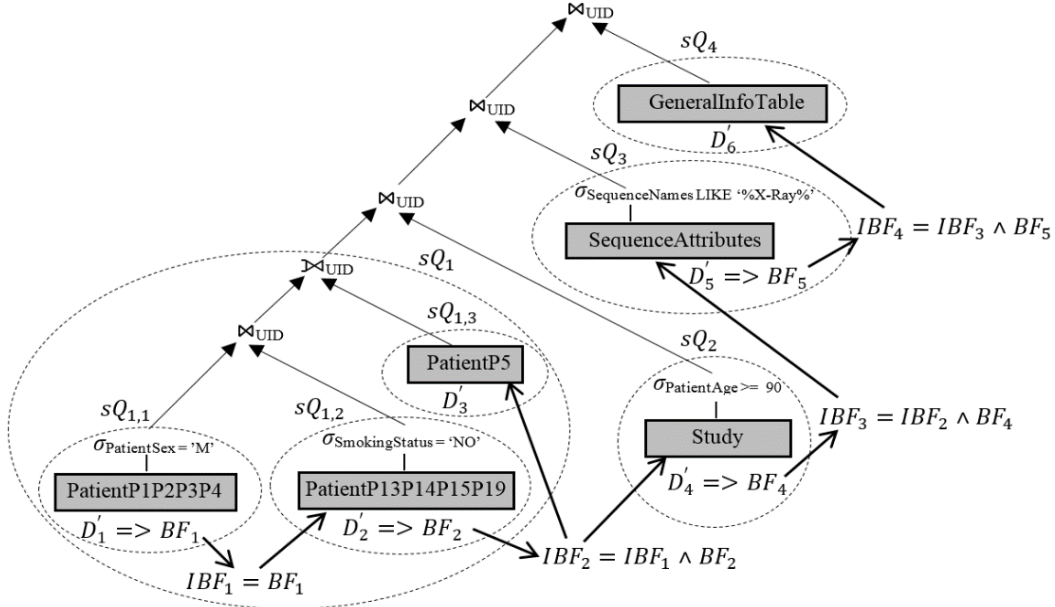


Figure 6.7: Execution plan for the query $Q_{4,3}$ with the incremental IBF

Figure 6.7 illustrates the application of the incremental IBF to the execution plan of the query $Q_{4,3}$. First, the sub-query $sQ_{1,1}$ is executed and produces the result table D'_1 . The Bloom filter BF_1 is created from values of the attribute UID of D'_1 , and the incremental IBF is computed from this Bloom filter: $IBF_1 = BF_1$. Next, the sub-query $sQ_{1,2}$ is executed with the application of IBF_1 as its local predicate and produces the result table D'_2 . The Bloom filter BF_2 is computed on the values of the attribute UID of D'_2 , and the incremental IBF is recomputed as follows: $IBF_2 = IBF_1 \wedge BF_2$. Then, the sub-query $sQ_{1,3}$ is executed with the application of IBF_2 as its local predicate in order to create intermediate result table D'_3 . Here, no Bloom filter is computed on the values of the attribute UID of D'_3 because this result table is a right-hand side table of a left-outer join. Thus, the incremental IBF are not recomputed. Next, the sub-query sQ_2 is executed with the application of IBF_2 as its local predicate and produces the result table D'_4 . The Bloom filter BF_4 is created from values of the attribute UID of D'_4 , and the incremental IBF is recomputed as follows: $IBF_2 = IBF_2 \wedge BF_4$. Similarly, in the next steps, the sub-query sQ_3 is executed with the application of IBF_3 as its local predicate and produces the result table D'_5 . The Bloom filter BF_5 is created from values of the attribute UID of D'_5 , and the incremental IBF is recomputed: $IBF_4 = IBF_3 \wedge BF_5$. After that, the sub-query sQ_4 is executed with the application of IBF_4 as its local predicate and produces the result table D'_6 . sQ_4 is the uppermost sub-query of the execution plan tree, thus the incremental IBF is not

recomputed. Finally, the join operations are executed using the above result tables as their inputs.

Table 6.30 presents a comparison of query execution time between two cases: using the IBF and the incremental IBF. It shows that for all cases of predicate sets the query execution time is reduced when using an incremental IBF.

Table 6.30: Comparison between the IBF and incremental IBF

Pre. Set	Execution time when using an IBF		Execution time when using an incremental IBF		Reduced time ratio (%)
	Average (sec)	Std. dev.	Average (sec)	Std. dev.	
1	1007.20	176.89	862.60	242.25	14%
2	748.00	92.29	925.40	198.97	-23%
3	962.80	197.97	995.40	167.60	-3%
4	908.80	202.48	901.80	216.55	1%
5	964.80	189.23	779.00	98.02	19%
6	930.40	127.05	729.80	202.91	22%

Besides, the incremental IBF outperforms the IBF for the majority of sets of predicates. More particularly, for the first and last three sets of the predicates, the reduced time ratios are 14%, 1%, 19% and 22%, respectively, when the incremental IBF is applied (given as Table 6.30). In these cases, the incremental IBF is only computed from the BFs built on the result tables of highly selective sub-queries of the lower join operations, i.e., $sQ_{1.1}$, $sQ_{1.2}$, $sQ_{1.3}$ and sQ_2 , but it can still filter a large number of irrelevant tuples out of the input tables of the upper join operations, e.g., *SequenceAttributes* and *GeneralInfoTable*, especially when these tables are very large. On other words, when applying the incremental IBF to these cases, the amount of filtered data is very large while the high costs of building and probing the incremental IBF are trivial. However, for the second and the third sets of predicates, the IBF outperforms the incremental IBF. This is probably because the high cost of building and probing the IBF has been compensated significantly by amount of filtered data.

6.4 Analysis and Interpretation

This section assesses and presents results of the hypotheses.

6.4.1 H1 - Effectiveness of HYTORMO

The results of Experiment 1 show that a hybrid store should be used for DICOM data because a row or a column store has its own benefits for a specific workload type:

- For OLAP workloads, a column store provides a higher performance than a row store. For instance, the performance of Workload W1 (OLAP-like workload) is improved when using the column table *GeneralInfoTable*.
- For OLTP workloads, a row store offers a higher performance than a column store. For instance, for Workload W2 (OLTP-like workload), storing the entity table *SequenceAttribute* in a row table improved the workload execution time.

Additionally, the results of Experiment 2 show that, for the mixed OLAP and OLTP workloads, a mixed use of both the row and column stores will improve the workload

execution time. For instance, Configuration G*, which stored the entity table *SequenceAttribute* and the vertical partitions of the entity table *Patient* in a row store, and two entity tables *Study* and *GeneralInfoTable* in a column store, gave a shorter workload execution time than Configuration G₁ (using a pure row store) and Configuration G₂ (using a pure column store).

The above results indicate that it is beneficial to use the hybrid storage strategy of both row and column stores to store DICOM data. Hence, Hypothesis H1 is accepted.

6.4.2 H2 - Usefulness of HADF

Due to the variety of DICOM data and its workloads, taking into account the combined impact of both workload- and data-specific information allows HADF to be able to well support in choosing a good data storage configuration for each entity table.

- In Experiments 1, 2 and 3, for the dense entity tables, e.g., *GeneralInfoTable* and *SequenceAttribute*, the data-specific information did not have a strong positive effect on reducing storage space size. In such cases, most of the attributes in the same entity table have low values of null ratios. Thus, if only depending on the data-specific information, most of the attributes are highly similar to each other in terms of Attribute Density Similarity such that they are grouped and stored together without the reduction of null values. However, for these cases, if the workload-specific information is also taken into account, the merging-selecting phase of HADF found that an OLAP-like workload is being used for *GeneralInfoTable* and an OLTP-like workload is being used for *SequenceAttribute*. Therefore, at the end, it suggests to store *GeneralInfoTable* in a column store and *SequenceAttribute* in a row store. This improved the overall performance of Workloads W1, W2 and W4.
- Conversely, in Experiments 1 and 2, for the wide and sparse entity tables, i.e., *Patient* and *Study*, the data-specific information had a strong effect on the vertical partitioning results. Multiple vertical partitions are created to remove null values.

Therefore, the combined use of both workload- and data-specific information has positive effects on creating good configurations. Hence, Hypothesis H2a is accepted.

Additionally, another important goal of HADF is to support decision makers in selecting data storage configurations where both the workload execution time and the storage space size are reduced at the same time. The results of Experiments 1 and 2 show that this goal was achieved for very wide and sparse entity tables such as *Patient*. HADF decomposed these tables into multiple vertical partitions from which null rows are removed; besides, the reduction of tuple reconstruction cost and I/Os speeded up the workload execution time as well. It seems easier to improve the workload performance than to reduce the storage space size because the storage space size is mainly reduced for very wide and sparse entity tables. However, such entity tables are popularly used in the context of DICOM data. Thus, Hypothesis H2b is accepted.

To the best of our knowledge, our work is among the first to consider a heuristic design approach that takes into consideration the combined impact of both workload- and data-specific information and the mixed use of both row and column stores while generating data storage configurations. Our HADF is inspired from the up-to-date vertical partitioning approach proposed by B. Cui et al. [14], which depends on only

the data-specific information in order to decompose a sparse table into multiple vertically partitioned tables and then stores these result tables in just a row store. However, their approach has been included as a part of our solution. In Experiments 1, 2 and 3, we showed that the combined use of both workload- and data-specific information and the use of a hybrid store is able to generate better data storage configurations than only using the data-specific information and row store.

6.4.3 H3 - Effectiveness of the Query Processing Strategy

The results of Experiment 4 show that both the IBF and the incremental IBF significantly speeded up the query processing. The reason for this improvement is that the IBF helps to filter the irrelevant tuples out of the input tables of join operations. This leads to reduction of network I/Os, disk I/Os and CPU cost (because less input data will be processed at nodes or sent on the network). Hypothesis H3 is accepted.

6.5 Summary and Conclusion

This chapter presented the results of the validation of the proposed methods. HYTOMO was implemented using a Spark cluster of 9 nodes. Real DICOM datasets were collected and their metadata and image data were extracted. The workloads were also determined. The experimental results can be summarized below.

The experimental results show that the hybrid storage strategy provides a better query performance than a pure row store and a pure column store in the context of DICOM data. The column store improves the performance of OLAP workloads while the row store improves the performance of OLTP workloads. Therefore, in order to improve the overall system performance, depending on the workloads associated with the attributes, we should apply a suitable data layout to store the particular attributes.

Additionally, taking into account the combined impact of both workload- and data-specific information is very helpful to generate a good data storage configurations in terms of storage space size and workload execution time. The experimental results show that, with the use of both sources of information, HADF can produce good data storage configurations. The workload-specific information has a strong effect on improving the workload performance while the data-specific information can help to reduce the storage space demand. Beside, HADF can generate a data storage configuration that decreases both storage space size and workload execution time; however, this is mainly achieved when an entity table is very wide and sparse.

Finally, the query processing strategy with the use of the IBF or the incremental IBF improves the query performance. They can filter irrelevant tuples out of the input tables of join operations. This helps to reduce network I/Os, disk I/Os and CPU cost.

Key Points

- We execute the experiments to validate HYTORMO.
- We execute the experiments to validate HADF.
- We execute the experiments to validate IBF and incremental IBF.

Conclusion and Future Works

7.1 Overview

The dissertation deals with the Big Data issues in DICOM data management from one big question: how to efficiently store and query DICOM data? This chapter summarizes and concludes the dissertation. We also give an outlook for future research. An overview of the chapter is presented in Table 7.1.

Table 7.1: Overview over Chapter 7

<p>7.2 Summary and Conclusion</p> <ul style="list-style-type: none"> 7.2.1 Existing DICOM Data Management Systems 7.2.2 Current Databases and Related Techniques 7.2.3 HYTORMO and DICOM Data Storage Strategy 7.2.4 HADF 7.2.5 Query Processing Strategy with the Use of an IBF 7.2.6 Validations of Proposed Methods
<p>7.3 Future Works</p> <ul style="list-style-type: none"> 7.3.1 Hybrid Storage Model 7.3.2 HADF 7.3.3 Query Processing Strategy 7.3.4 Non-precomputed and Precomputed BFs

There are six main contributions emerged from our study: First, we performed a comprehensive evaluation of the existing DICOM data management systems and addressed their strengths and weaknesses. As a response to the shortcomings, we specified the expected requirements for a new DICOM data management system. Second, we provided a state of the art review of the current databases (relational, NoSQL and NewSQL databases) and the related techniques (including cluster computing frameworks, data layouts, vertical partitioning, BF and IBF techniques). Third, we proposed a hybrid storage model, called HYTORMO, together with a data storage strategy. Fourth, we proposed a hybrid automated design framework, called HADF. Fifth, we introduced a query processing strategy with the use of an IBF for HYTORMO. Finally, we performed validations to demonstrate the benefits of the proposed methods.

7.2 Summary and Conclusion

7.2.1 Existing DICOM Data Management Systems

Based on the characteristics of DICOM data and workloads, we specified that a new DICOM data management system needs to satisfy three expected requirements: (R1) Flexible data; (R2) Flexible querying; and (R3) Efficiency of storage and CPU. The requirement R1 requires that the system is able to deal with the complexity and the variety of DICOM data. The requirement R2 requires that the system enables users to write SQL ad-hoc queries with joins. The requirement R3 requires that DICOM data will be organized based on workload and data-specific information to reduce storage space demand and execution time of queries in mixed OLTP and OLAP workloads; additionally, it is able to provide efficient query processing over large-scale datasets, huge storage capacity, scalability and elasticity.

We performed a comprehensive evaluation of the existing DICOM data management systems. With regards to data storage, the existing systems can be classified into four groups of solutions: row-oriented databases, vertically-decomposed row-oriented databases, NoSQL document-based databases and hybrid cloud-enabled storage system. First, the systems that are using a row-oriented database such as PACSs [129], eDiaMoND [42], and commercial RDBMSs (Oracle) [130] store DICOM data in tables. These systems are optimized for write-intensive (OLTP) workloads in which all (or most) attributes of each tuple are frequently accessed together by queries. Unfortunately, they waste I/O bandwidth because all attributes of a table have to be read into memory from disk even if only few attributes are needed once per query. Second, in the system using a vertically-decomposed row-oriented database such as DCMDISM [54], data is vertically decomposed and stored into multiple tables. This strategy can help the system reduce disk I/Os, but it needs more CPU cost due to multi-table joins required for tuple reconstruction. Moreover, the proposed system has not been designed to operate in a distributed query processing environment. Third, the system using a NoSQL document-based database [40] could handle the heterogeneous schemas due to sharing non-relational design, but it does not provide a standardized declarative query language, e.g., SQL. Finally, the hybrid (row-column) cloud-enabled storage system can reduce I/Os and tuple reconstruction cost and deal with the evolution of data. Nevertheless, it is hard to scale and has not provided an automated design approach to create data storage configurations.

Therefore, the document-based database and hybrid cloud-enabled storage system have shown their ability or potential to satisfy the above-mentioned respected requirements. However, they still lack the following features that are addressed in our thesis:

- An automated design approach that uses both workload and data-specific information to design and store DICOM data in a manner to reduce both workload execution time and storage space demand.
- Efficient solutions for query processing over large-scale datasets, especially, to reduce network I/Os in a distributed query processing environment.

7.2.2 Current Databases and Related Techniques

We performed a state of the art review of the current databases. Relational databases are based on the relational data model. They organize data in tables and provide users with SQL interfaces. The relational databases can handle the complexity of DICOM data because entities and relationships among the entities in the DICOM data model can be well represented by the entity-relationship model. However, they have some limitations in providing the following features: huge storage capacity, high query performance over high and ever-growing volume of data, scalability and elasticity. Thus, in general, they are not efficient to handle DICOM data. In contrast to the relational databases, NoSQL databases are designed to handle Big Data. They can deal with un/semi-structured data, process large amounts of data with high performance and scalability, provide huge storage capacity, elasticity and so on. However, they do not represent well data in tabular form, nor do they provide SQL support. Therefore, the relational and NoSQL databases alone do not provide all features required to manage DICOM data. For this reason, we move towards applying the concepts of NoSQL databases to build a data storage model that is able to support SQL and represent data in form of tables.

Besides, we performed reviews on cluster computing frameworks, data layouts, vertical partitioning, BF and IBF techniques. First, the batch-oriented processing technique of MapReduce is not suitable for processing interactive workloads because of its high latency. In contrast, the interactive ad-hoc query and analysis technique, e.g., Spark, is able to provide high performance for interactive workloads. Second, row stores (Oracle, DB2, etc.) are optimized for write-intensive (OLTP) workloads, whereas column stores (MonetDB, C-Store, etc.) are well-suited for read-intensive (OLAP) workloads. To fill the gap between these two types of stores, hybrid stores (e.g., HYRISE, SAP HANA, etc.) have aimed at optimizing the performance for both types of workloads. Third, the vertical partitioning algorithms show that they can be applied to improve the query performance or to reduce storage space size especially for sparse datasets. However, they have not taken into consideration the combined impact of both workload- and data-specific information on vertical partitioning results. Additionally, they have assumed that resulting schemas will be stored in tables using just one kind of data layout, e.g., row-oriented data layout, instead of hybrid data layout. Finally, to improve query performance, the IBF have shown that they are able to reduce the network I/O cost with a false positive probability less than the BF.

7.2.3 HYTORMO and DICOM Data Storage Strategy

We proposed a new hybrid storage model, called HYTORMO:

- To facilitate users, DICOM data in HYTORMO is organized based on the relational data model. Users can use entity tables in their SQL queries. HYTORMO will automatically decompose the users' queries into sub-queries to access only relevant tables archived in row or column stores.
- To provide huge storage capacity, high query performance, scalability and elasticity, we designed and implemented HYTORMO using an in-memory massively-parallel computation and storage techniques on large clusters of

nodes. In fact, HYTORMO was implemented on top of Spark: DICOM data is stored in a distributed file system (HDFS) and queries are processed in parallel.

- To achieve a data storage configuration for DICOM data, one of two design approaches can be applied: expert-based and automated.

7.2.4 HADF

HADF is proposed to assist decision makers in selecting a good data storage configuration for each entity table. It can take into account the combined impact of both workload- and data-specific information as well as the combined use of both row and column stores to generate a new data storage configuration. In particular, HADF works through two phases: clustering and merging-selecting. The clustering phase aims at reducing storage space size and tuple reconstruction cost. To achieve this, it depends on Hybrid Similarity (a weighted combination of Attribute Access Similarity and Attribute Density Similarity) between every pair of attributes to cluster attributes into column groups such that attributes in each particular column group are similar and attributes in different column groups are dissimilar. The merging-selecting phase aims at reducing the number of join operations across vertically partitioned tables and the number of irrelevant attribute accesses. It uses Inter-Cluster Access Similarity to determine whether two clusters are merged together or not and uses Intra-Cluster Access Similarity to determine a suitable data layout for each column group.

7.2.5 Query Processing Strategy with the Use of an IBF

We proposed a query processing strategy built on top of HYTORMO that includes the use of inner joins, left-outer joins and an IBF. We scoped our work to only consider a left-deep sequential tree plan with inner joins and left-outer joins. Our proposed query processing strategy is designed for: (1) working with tables archived in both row and column stores; (2) reducing the number of left-outer joins; and (3) reducing the network communication cost by applying the IBF.

7.2.6 Validations of Proposed Methods

We performed experiments to validate the proposed methods using real DICOM datasets. Experimental results show that performance of the hybrid store is better than either a pure row store or a pure column store because it can combine the fundamental advantages of both row and column stores: pieces of data used by OLTP workloads are stored in row tables while pieces of data required by OLAP workloads are stored in column tables. The combined use of both workload- and data-specific information is necessary for HADF to generate good data storage configurations. The workload-specific information has a strong effect on improving the workload performance while the data-specific information is helpful in reducing storage space demand. HADF is able to support in selecting a good data storage configuration that reduces both the storage space demand and the workload performance, but this is mainly achieved for wide and sparse tables.

The experimental results also show that the IBF or the incremental IBF help to improve the query performance. The query execution time was reduced to 10-38% of

the time when applying the IBF. Besides, the incremental IBF outperforms the IBF in the majority of cases of predicate sets. Filtering irrelevant tuples out of input tables of join operations results in the reduction of disk and network I/Os and CPU cost.

In short, the conclusions are as follows: using the hybrid storage model improves the workload execution time; taking into account the combined impact of both workload- and data-specific information is necessary to produce better data storage configurations; and the application of the IBF improves query performance.

7.3 Future Works

There are some open research axes that we can investigate and extend in future.

7.3.1 Hybrid Storage Model

HYTORMO was designed for storing and querying DICOM data. Nevertheless, we believe that it can be extended to be used for many various Big Data applications. Instead of just using row and column stores, we plan to extend the current model to support multiple stores: row store, column store, key-value store, etc. As such, it is well suited for the variety of data in many different applications.

Recently, some systems using multiple data models and data stores have been proposed. For instance, CloudMdsQL Multistore System [131, 132] provides a SQL-like language, called CloudMdsQL that is a common language for querying and integrating data from multiple heterogeneous cloud data stores. It can exploit the full performance of local data stores by allowing embedded invocations to each local data store native query interface. Another system, called BigDAWG Polystore System [133], also stores data in different storage engines by depending on the data access patterns. However, these systems lack an automated solution that is based on both the characteristics of data and workloads to determine the right stores for their data.

7.3.2 HADF

Some requirements would be performed to extend our work: We have based on experiments and experts' opinion to select suitable values for the parameters of HADF (including α , β , θ and λ), thus it would be necessary to develop a method to automatically determine these values. For this requirement, we would investigate the application of optimization techniques that may give better results than our approach. In [134], the authors proposed an agglomerative clustering algorithm to automatically generate property table schemas that can balance storage efficiency and query performance for a very large RDF dataset. Unfortunately, a hybrid storage system has not used to store the property tables. As a future work, the authors in [134] planned to develop a hybrid approach by combine the triple store, vertical database, and property table schemes to have their own advantages in different situations.

We will extend HADF to take into consideration the horizontal tables that may have different widths in their attributes. We plan to research the effect of compression on some pieces of data, e.g., column tables. We also plan to research how new attributes are added to an existing data storage configuration. For instance, HADF can

modify incrementally the existing configuration while still maintaining a trade-off between the storage space size and the query performance. In [135], the authors proposed an approach that consists of two phases: the vertical partitioning phase aims at reducing the number of join operations while the adjustment phase aims to maintain the query performance by adapting the underlying schema to react to the changes in the characteristics of the continuous query workload stream. However, the authors have not taken into account the storage space size during the adjustment phase.

7.3.3 Query Processing Strategy

We will explore the query execution plan with the use of inner joins and (full) outer joins (\bowtie), instead of inner joins and left-outer joins. In this way, given a data storage configuration G_i of a horizontal table T , a query q and a set C_i^q of column groups that are required to answer q , a relational algebraic expression using inner joins or left-outer joins can be given as follows (see Formula (4.3.9) in Chapter 4):

$$q = \pi_{a_1, \dots, a_m} \left[\pi_{UID}(T) \bowtie \left(\bowtie_{x=1}^{|C_i^q|} \sigma_{P_{i,x}}(C_{i,x}) \right) \right],$$

where the selection operation $\sigma_{P_{i,x}}(C_{i,x})$ returns only tuples of the table storing data of the column group $C_{i,x}$ for which the predicate (or condition) $P_{i,x}$ is fulfilled; the projection operation $\pi_{UID}(T)$ returns a list of all UID 's of the horizontal table T . We will consider the use of the full outer joins for q because when this join type is applied, the resulting tuples of T will be produced for each tuple in each joined vertical partition, no matter what the join order is. Thus, we can select a join order that can result in a better overall query performance:

$$q = \pi_{a_1, \dots, a_m} \left(\bowtie_{x=1}^{|C_i^q|} \sigma_{P_{i,x}}(C_{i,x}) \right).$$

Additionally, we will consider how to transform a left-deep tree plan to a bushy tree plan to increase parallelism in query processing. Although many studies have introduced different approaches for transforming a left-deep tree into a bushy tree [136-138], there is a lack of studies generating a bushy tree for a left-deep tree plan consisting of left or full outer joins as in our context.

All the above changes introduce new research challenges. For instance, how is tuple reconstruction cost modelled? How is an IBF applied to a query execution plan using outer joins in a bushy tree plan?

7.3.4 Non-precomputed and Precomputed BFs

We believe it would be beneficial to combine both types of BFs: non-precomputed and precomputed BFs. The non-precomputed BFs are computed from input tables during query processing as used in our thesis. Alternatively, the precomputed BFs are computed beforehand to avoid additional computation steps required during query processing. For instance, in [139], the precomputed BFs were used to speed up SPARQL processing in the cloud. Based on usage frequency of the input tables, first we can precompute BFs as many as possible. Then, BFs of these two types can be combined by using bitwise AND-operations to build a common IBF.

Bibliography

- [1] O. S. Pianykh, *Digital Imaging and Communications in Medicine (DICOM)*: Springer-Verlag Berlin Heidelberg, 2008.
- [2] I. Merelli, H. Pérez-Sánchez, S. Gesing, and D. D'Agostino, "Managing, Analysing, and Integrating Big Data in Medical Bioinformatics: Open Problems and Future Perspectives," *BioMed Research International*, vol. 2014, 2014.
- [3] N. Chandrashekar, S. M. Gautam, K. S. Srinivas, and J. Vijayananda, "Design Considerations for a Reusable Medical Database," in Proceedings of 19th IEEE Symposium on Computer-Based Medical Systems, 2006, pp. 69-74.
- [4] C. Parisot. "The Basic Structure of DICOM," 1 Oct 2017; <http://www.ssrpm.ch/old/dicom/parisot1.pdf>.
- [5] OECD, *Genetic Testing: A Survey of Quality Assurance and Proficiency Standards*: OECD Publishing, October 2007.
- [6] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, "Vertical partitioning algorithms for database design," *Transactions on Database Systems (TODS)*, vol. 9, no. 4, pp. 680-710, 1984.
- [7] W. W. Chu, and I. T. Jeong, "A transaction-based approach to vertical partitioning for relational database systems," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 804-812, 1993.
- [8] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in Proc. ACM SIGMOD, 2004, pp. 359-370.
- [9] M. Hammer, and B. Niamir, "A heuristic approach to attribute partitioning," in Proc. ACM SIGMOD, 1979, pp. 93-101.
- [10] R. A. Hankins, and J. M. Patel, "Data morphing: an adaptive, cache-conscious storage technique," in PVLDB, 2003, pp. 417-428.
- [11] S. Papadomanolakis, and A. Ailamaki, "AutoPart: automating schema design for large scientific databases using data partitioning," in Proceedings of 16th International Conference on Scientific and Statistical Database Management, 2004, pp. 383-392.
- [12] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "HYRISE: a main memory hybrid storage engine," in Proc. VLDB Endow., 2010, pp. 105-116.
- [13] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts: right shoes for a running elephant," in Proceedings of the 2nd ACM Symposium on Cloud Computing, 2011, pp. 1-14.
- [14] B. Cui, J. Zhao, and D. Yang, "Exploring Correlated Subspaces for Efficient Query Processing in Sparse Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 2, pp. 219-233, 2010.

- [15] J. J. Levandoski, and M. F. Mokbel, "RDF Data - Centric Storage," in Proceedings of 2009 IEEE International Conference on Web Services, 2009, pp. 911-918.
- [16] E. Chu, J. Beckmann, and J. Naughton, "The case for a wide-table approach to manage sparse relational data sets," in Proc. ACM SIGMOD, 2007, pp. 821-832.
- [17] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases," in Proceedings of the 32nd international conference on Very large data bases, Seoul, Korea, 2006, pp. 487-498.
- [18] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in CIDR, 2005.
- [19] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: a column-oriented DBMS," in PVLDB, 2005, pp. 553-564.
- [20] F. Färber, S. K. Cha, J. Primsch, C. Bornh, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45-51, 2012.
- [21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in Proceedings of SIGMOD, 2015, pp. 1383-1394.
- [22] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422-426, 1970.
- [23] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski, "Improving distributed join efficiency with extended bloom filter operations," in The 21st International Conference on Advanced Information Networking and Applications, 2007, pp. 187-194.
- [24] A. Broder, and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics* vol. 1, no. 4, pp. 485-509, 2003.
- [25] T.-C. Phan, L. d'Orazio, and P. Rigaux, "Toward intersection filter-based optimization for joins in MapReduce," in Proceedings of the 2nd International Workshop on Cloud Intelligence, Riva del Garda, Trento, Italy, 2013, pp. 1-2.
- [26] M. Hausenblas, and J. Nadeau, "Apache Drill: Interactive Ad-Hoc Analysis at Scale," *Big Data*, vol. 1, pp. 100-104, 2013.
- [27] T.-C. Phan, "Optimization for Big Joins and Recursive Query Evaluation using Intersection and Difference Filters in MapReduce," PhD Thesis, Blaise Pascal University - Clermont II, 2014.
- [28] B. Revet, *DICOM Cookbook for Implementations in Modalities (Technical Report)*: Philips medical systems, 1997.
- [29] NEMA, *Digital Imaging and Communications in Medicine (DICOM) - Part 5: Data Structures and Encoding*, National Electrical Manufacturers Association (NEMA), 2011.
- [30] D. Laney, *3D Data Management: Controlling Data Volume, Velocity, and Variety*, Technical Report, 2001.
- [31] M. A. Beyer, and D. Laney, "The Importance of 'Big Data': A Definition," Gartner, Stamford, CT, 2012.

- [32] NEMA, *Digital Imaging and Communications in Medicine (DICOM) - Part 6: Data Dictionary*, National Electrical Manufacturers Association (NEMA), 2017.
- [33] R. R. Carlton, and A. M. Adler, *Principles of Radiographic Imaging : An Art and a Science*, Clifton Park, New York: Delmar/Cengage Learning, 2013.
- [34] Oracle, *Performance Evaluation of Storage and Retrieval of DICOM Image Content in Oracle Database 11g Using HP Blade Servers and Intel Processors*, Oracle Ltd, July 2008.
- [35] H. Chen, R. H. L. Chiang, and V. C. Storey, "Business intelligence and analytics: from big data to big impact," *MIS Quarterly*, vol. 36, no. 4, pp. 1165-1188, 2012.
- [36] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi, "Big data and its technical challenges," *Commun. ACM*, vol. 57, no. 7, pp. 86-94, 2014.
- [37] D. R. Voellmy, O. Handgräter, S. Wildermuth, B. Fröhlich, and B. Marincek, "Total cost of high volume multi-detector CT data management," *International Congress Series*, vol. 1268, pp. 249-253, June 2004.
- [38] A. A. TOLE, "Big Data Challenges," *Database Systems Journal*, vol. IV, no. 3, 2013.
- [39] N. S. Ujgare, and S. P. Baviskar, "Conversion of DICOM Image in to JPEG, BMP and PNG Image Format," *International Journal of Computer Applications (0975 – 8887)*, vol. 62, no. 11, January 2012.
- [40] S. J. Rascovsky, J. A. Delgado, A. Sanz, V. D. Calvo, and G. Castrillón, "Informatics in Radiology: Use of CouchDB for Document-based Storage of DICOM Objects," *RadioGraphics*, vol. 32, no. 3, pp. 913-927, 2012.
- [41] H. Satoh, N. Niki, K. Eguchi, H. Ohmatsu, M. Kusumoto, M. Kaneko, and N. Moriyama, "Teleradiology network system on cloud using the web medical image conference system with a new information security solution," in *SPIE Medical Imaging*, 2013, pp. 9.
- [42] D. Power, E. Politou, M. Slaymaker, S. Harris, and A. Simpson, "A relational approach to the capture of DICOM files for Grid-enabled medical imaging databases," in *Proceedings of the 2004 ACM symposium on Applied computing*, Nicosia, Cyprus, 2004, pp. 272-279.
- [43] M. Brady, D. Gavaghan, A. Simpson, M. M. Parada, and R. Highnam, "eDiamond: A Grid-Enabled Federated Database of Annotated Mammograms," *Grid Computing*, pp. 923-943: John Wiley & Sons, Ltd, 2003.
- [44] "Open Grid Services Architecture -Data Access and Integration," 5 May 2017; <http://www.ogsadai.org.uk/>.
- [45] M. Oevers, B. M. Collins, A. Knox, and J. Williams, "The Use of OGSA-DAI with IBM DB2 Content Manager for Multiplatforms in the eDiaMoND Project," in *Proceedings of the Future of Grid Data Environments Workshop at the Global Grid Forum 10 meeting*, March 2004.
- [46] Oracle, *Oracle Database 10g Release 2: A Revolution in Database Technology*, An Oracle White Paper, Oracle Ltd, May 2005.
- [47] Oracle, *Oracle Multimedia DICOM Developer's Guide, 11g Release 1*, Oracle White Paper, Oracle Ltd, 2009.

- [48] Oracle, *Unstructured Data Management with Oracle Database 12c*, Oracle White Paper, Oracle Ltd, November 2016.
- [49] Oracle, *Oracle RAC: Oracle Real Application Clusters 10g*, Oracle Technical White Paper, Oracle Ltd, May 2005.
- [50] Oracle, *Oracle RAC: Oracle Real Application Clusters 11g Release 2*, Oracle White Paper, Oracle Ltd, November 2010.
- [51] Oracle, *Oracle RAC: Oracle Database 12c Release 2: Oracle Real Application Clusters*, Oracle White Paper, Oracle Ltd, March 2017.
- [52] E. Hewitt, *Cassandra: The Definitive Guide*: O'Reilly Media, November 2010.
- [53] K. Chodorow, *MongoDB: The Definitive Guide*, 2nd ed.: O'Reilly Media, May 2013.
- [54] A. Savaris, T. Härder, and A. von Wangenheim, "DCMDSM: a DICOM decomposed storage model," *Journal of the American Medical Informatics Association : JAMIA*, vol. 21, no. 5, pp. 917-924, 2014.
- [55] G. P. Copeland, and S. N. Khoshafian, "A decomposition storage model," in *Proc. ACM SIGMOD*, 1985, pp. 268-279.
- [56] B. Mohamad, L. d'Orazio, and L. Gruenwald, "Towards a hybrid row-column database for a cloud-based medical data management system," in *Proceedings of the 1st International Workshop on Cloud Intelligence*, Istanbul, Turkey, 2012, pp. 1-4.
- [57] B. Mohamad, "Medical Data Management on the Cloud," PhD Thesis, Blaise Pascal University - Clermont II, 2015.
- [58] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377-387, 1970.
- [59] R. Hecht, and S. Jablonski, "NoSQL evaluation: A use case oriented survey," in *Proceedings of 2011 International Conference on Cloud and Service Computing*, 2011, pp. 336-341.
- [60] A. B. M. Moniruzzaman, and S. Hossain, "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison," *International Journal of Database Theory and Application*, vol. 6, pp. 1-14, 2013.
- [61] A. Auradkar, C. Botev, S. Das, D. D. Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, S. Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, L. Qiao, T. Quiggle, J. Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang, "Data Infrastructure at LinkedIn," in *Proceedings of 2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 1370-1381.
- [62] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1-26, 2008.
- [63] "Welcome to Apache HBase," April 24, 2017; <https://hbase.apache.org>.
- [64] "Apache CouchDB," April 18, 2017; <http://couchdb.apache.org/>.
- [65] "Neo4j - What is a Graph Database? ," September 19, 2017; <http://www.neo4j.org/>.

- [66] "AllegroGraph," <http://www.franz.com/agraph/allegrograph>.
- [67] R. k. Kaliyar, "Graph databases: A survey," in Proceedings of International Conference on Computing, Communication & Automation, 2015, pp. 785-790.
- [68] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 1-22, 2013.
- [69] VoltDB, *Technical Overview: High performance, scalable RDBMS for Big Data and Real-time Analytics*, White paper (http://voltadb.com/downloads/datasheets_collateral/technical_overview.pdf), VoltDB.
- [70] "A New Approach: Clustrix Sierra Database Engine," April 20, 2017; <http://www.clustrix.com>.
- [71] NuoDB, *NuoDB Emergent Architecture: A 21st Century Transactional Relational Database Founded On Partial, On-Demand Replication*, Greenbook, NuoDB Inc., 2013.
- [72] A. Ribeiro, A. Silva, and A. R. d. Silva, "Data Modeling and Data Analytics: A Survey from a Big Data Perspective," *Journal of Software Engineering and Applications*, vol. 8, pp. 617-634, 2015.
- [73] J. Dean, and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in Proceedings of Sixth Symposium on Operating System Design and Implementation (OSDI'04), San Francisco, CA, 2004.
- [74] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in Proc. ACM SIGMOD, 2008, pp. 1099-1110.
- [75] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," in Proc. VLDB Endow., 2009, pp. 1626-1629.
- [76] S. Shaw, A. F. Vermeulen, A. Gupta, and D. Kjerrumgaard, *The Future of Hive*: Apress, Berkeley, CA, 2016.
- [77] "BigQuery," April 25, 2017; <https://developers.google.com/bigquery>.
- [78] "CitusDB," April 25, 2017; <http://citusdata.com/docs>.
- [79] "Hadapt," April 25, 2017; <http://hadapt.com/product>.
- [80] "HAWQ," April 25, 2017; <http://www.greenplum.com/blog/topics/hadoop/introducing-pivotal-hd>.
- [81] "Impala," April 25, 2017; <https://github.com/cloudera/impala>.
- [82] S. Akhtar, and R. Magham, *Pro Apache Phoenix: An SQL Driver for HBase*, 1st ed.: Apress, Berkeley, CA, 2017.
- [83] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "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, San Jose, CA, 2012, pp. 2-2.

- [84] R. Agrawal, A. Somani, and Y. Xu, "Storage and Querying of E-Commerce Data," in *PVLDB*, 2001, pp. 149-158.
- [85] J. Foley, *Comparison of Data Warehousing DBMS Platforms*, illuminate Solutions, Barcelona, Spain, 2013.
- [86] R. Ramamurthy, D. J. DeWitt, and Q. Su, "A case for fractured mirrors," *The VLDB Journal*, vol. 12, no. 2, pp. 89-101, 2003.
- [87] A. Kemper, and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proceedings of 2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 195-206.
- [88] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving Relations for Cache Performance," in *Proc. VLDB*, 2001, pp. 169-180.
- [89] A. Shatdal, C. Kant, and J. F. Naughton, "Cache Conscious Algorithms for Relational Query Processing," in *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 510-521.
- [90] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?," in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 266-277.
- [91] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman, "The Asilomar report on database research," *SIGMOD Rec.*, vol. 27, no. 4, pp. 74-80, 1998.
- [92] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access," in *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 54-65.
- [93] A. Kemper, T. Neumann, F. Funke, V. Leis, and H. Mühe, "HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing," *IEEE Data Eng. Bull. (DEBU)*, vol. 35 (1), pp. 46-51, 2012.
- [94] F. Farber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database - An Architecture Overview," *IEEE Data Eng. Bull.*, vol. 35(1), pp. 28-33, 2012.
- [95] C. Meyer, M. Boissier, A. Michaud, J. O. Vollmer, K. Taylor, D. Schwalb, M. Uflacker, and K. Roedszus, "Dynamic and Transparent Data Tiering for In-Memory Databases in Mixed Workload Environments," *ADMS@VLDB*, pp. 37-48, 2015.
- [96] C. Pai-Cheng, "A transaction-oriented approach to attribute partitioning," *Information Systems*, vol. 17, no. 4, pp. 329-342, 1992.
- [97] D. W. Cornell, and P. S. Yu, "A Vertical Partitioning Algorithm for Relational Databases," in *Proceedings of the Third International Conference on Data Engineering*, 1987, pp. 30-35.
- [98] C.-W. Fung, K. Karlapalem, and Q. Li, "Cost-driven vertical class partitioning for methods in object oriented databases," *The VLDB Journal*, vol. 12, no. 3, pp. 187-210, 2003.
- [99] L. Bellatreche, A. Cuzzocrea, and S. Benkrid, "Effectively and Efficiently Designing and Querying Parallel Relational Data Warehouses on Heterogeneous Database Clusters: The F&A Approach," *J. Database Manage.*, vol. 23, no. 4, pp. 17-51, 2012.

-
- [100] S. K. Song, and N. Gorla, "A Genetic Algorithm for Vertical Fragmentation and Access Path Selection," *The Computer Journal*, vol. 43, no. 1, pp. 81-93, 2000.
- [101] J. Pèrez, R. Pazos, J. Frausto, D. Romero, and L. Cruz, "Vertical Fragmentation and Allocation in Distributed Databases with Site Capacity Restrictions Using the Threshold Accepting Algorithm," *Parallel Distributed Comput Syst*, pp. 210-213, 1998.
- [102] J. A. Hoffer, and D. G. Severance, "The use of cluster analysis in physical data base design," in Proc. VLDB, 1975, pp. 69-86.
- [103] J. Muthuraj, S. Chakravarthy, R. Varadarajan, and S. B. Navathe, "A formal approach to the vertical partitioning problem in distributed database design," in Proceedings of the Second International Conference on Parallel and Distributed Information Systems, 1993, pp. 26-34.
- [104] S. B. Navathe, and M. Ra, "Vertical partitioning for database design: a graphical algorithm," in SIGMOD Record, 1989, pp. 440-450.
- [105] "CLUstering TOolkit (CLUTO)," April 10, 2017; <http://www.cs.umn.edu/karypis/cluto>.
- [106] J. J. Brito, T. Mosqueiro, R. R. Ciferri, and C. D. d. A. Ciferri, "Faster cloud Star Joins with Reduced Disk Spill and Network Communication," *Procedia Computer Science*, vol. 80, pp. 74-85, 2016/01/01/, 2016.
- [107] D. Nguyen-Cong, L. D'Orazio, N. Tran, and M.-S. Hacid, "Storing and Querying DICOM Data with HYTORMO," in Proceedings of the Second International Workshop on Data Management and Analytics for Medicine and Healthcare - Volume 10186, 2017, pp. 43-61.
- [108] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *The VLDB Journal*, vol. 6, no. 3, pp. 191-208, August 01, 1997.
- [109] L. Rodríguez-Mazahua, G. Alor-Hernández, J. Cervantes, A. López-Chau, and J. L. Sánchez-Cervantes, "A hybrid partitioning method for multimedia databases," *Dyna*, vol. 83, no. 198, pp. 59-67, 2016.
- [110] B. Markines, C. Cattuto, F. Menczer, D. Benz, A. Hotho, and G. Stumme, "Evaluating similarity measures for emergent semantics of social tagging," in Proceedings of the 18th international conference on World wide web, Madrid, Spain, 2009, pp. 641-650.
- [111] T. V. V. Kumar, and K. Devi, "Frequent queries identification for constructing materialized views," in 2011 3rd International Conference on Electronics Computer Technology, 2011, pp. 177-181.
- [112] A. Strehl, and J. Ghosh, "Value-based customer grouping from large retail data sets," in Proc. SPIE, 2000, pp. 33-42.
- [113] T. Sellam, and M. Kersten, "Cluster-Driven Navigation of the Query Space," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1118-1131, 2016.
- [114] "TPC-H specification 2.8.0," <http://www.tpc.org/tpch/>.
- [115] L. Byung Suk, and G. Wiederhold, "Outer joins and filters for instantiating objects from relational databases through views," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 108-119, 1994.

- [116] P. Koutris, "Bloom Filters in Distributed Query Execution," *University of Washington*, 2011.
- [117] L. F. Mackert, and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *VLDB*, 1986, pp. 149-159.
- [118] C. Zhang, L. Wu, and J. Li, "Efficient Processing Distributed Joins with Bloomfilter using MapReduce," *International Journal of Grid and Distributed Computing*, vol. 6, pp. 43-58, 2013.
- [119] T. Lee, K. Kim, and H.-J. Kim, "Join processing using Bloom filter in MapReduce," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, San Antonio, Texas, 2012, pp. 100-105.
- [120] F. N. Afrati, and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *Proceedings of Proceedings of EDBT '10*, 2010, pp. 99-110.
- [121] "CT Colonography," October 11, 2015; <https://idash.ucsd.edu>.
- [122] "David Clunie's Medical Image Format Site," October 15, 2015; <http://www.dclunie.com>.
- [123] "Sample Data," October 12, 2015; <http://idoimaging.com>.
- [124] "Lung Cancer Datasets," October 11, 2015; <http://giveascan.org>.
- [125] "MIDAS Datasets," October 12, 2015; <http://www.insight-journal.org>.
- [126] "Cancer Imaging Archive Database (CIAD)," October 28, 2017; <https://public.cancerimagingarchive.net>.
- [127] "Open Source Clinical Image and Object Management," October 02, 2015; <http://www.dcm4che.org>.
- [128] T. White, *Hadoop: The Definitive Guide, Fourth Edition*: O'Reilly Media, 2015.
- [129] M. Möller, and S. Mukherjee, "Context-Driven Ontological Annotations in DICOM Images - Towards Semantic Pacs," in *Proceedings of the Second International Conference on Health Informatics, HEALTHINF 2009*, Porto, Portugal, 2009, pp. 294-299.
- [130] M. Annamalai, D. Guo, M. Susan, and J. Steiner, "An oracle white paper: oracle database 11g DICOM medical image support," 2009.
- [131] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira, "CloudMdsQL: querying heterogeneous cloud data stores with a common language," *Distributed and Parallel Databases*, vol. 34, no. 4, pp. 463-503, 2016/12/01, 2016.
- [132] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, and J. Pereira, "The CloudMdsQL Multistore System," in *SIGMOD*, San Francisco, United States, 2016.
- [133] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The BigDAWG Polystore System," *SIGMOD Rec.*, vol. 44, no. 2, pp. 11-16, 2015.
- [134] Thomas Y. Lee, D. W., Cheung, Jimmy Chiu, S.D. Lee, Hailey Zhu, Patrick Yee, and W. Yuan, *Automating Relational Database Schema Design for Very Large Semantic Datasets*, TR-2013-02, Technical report, Department of Computer Science, University of Hong Kong, 2013.

-
- [135] M. Hooran, and S. Sherif, “AdaptRDF: adaptive storage management for RDF databases,” *International Journal of Web Information Systems*, vol. 8, no. 2, pp. 234-250, 2012.
 - [136] R. Ahmed, R. Sen, M. Poess, and S. Chakkappen, “Of snowstorms and bushy trees,” *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1452-1461, 2014.
 - [137] W. Du, M.-C. Shan, and U. Dayal, “Reducing multidatabase query response time by tree balancing,” in Proceedings of the 1995 ACM SIGMOD international conference on Management of data, San Jose, California, USA, 1995, pp. 293-303.
 - [138] G. Moerkotte, and T. Neumann, “Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products,” in Proceedings of the 32nd international conference on Very large data bases, Seoul, Korea, 2006, pp. 930-941.
 - [139] S. Groppe, T. Kiencke, S. Werner, D. Heinrich, M. Stelzner, and L. Gruenwald, “P-LUPOSDATE: Using Precomputed Bloom Filters to Speed Up SPARQL Processing in the Cloud,” *OJSW*, vol. 1, no. 2, pp. 25-55, 2014.

N° d'ordre : XXX

EDSPIC : XXX



Université Clermont Auvergne

École Doctorale des Sciences pour l'Ingénieur

Thèse

pour l'obtention du grade de

DOCTEUR D'UNIVERSITÉ

Discipline : Informatique

Présentée par

Cong-Danh NGUYEN

Conception Automatisée Basée sur la Charge de Travail et les Données pour un Modèle de Stockage Hybride Ligne-Colonne et le Traitement des Requêtes à l'aide de Filtres de Bloom pour la Gestion de Données DICOM à Grande Échelle

Soutenue publiquement le 4 Mai 2018

Composition du jury :

Rapporteurs :

Prof. Christine COLLET Institut polytechnique de Grenoble, France
Prof. Abdelkader HAMEURLAIN Université de Paul Sabatier, Toulouse, France

Examineur :

Prof. Farouk TOUMANI Université Clermont Auvergne, France

Directeurs de thèse :

Prof. Laurent d'Orazio Université Rennes 1, France
Prof. Mohand-Said HACID Université de Lyon, France
Msc. Nga TRAN Micro Focus - Vertica, Cambridge, Massachusetts, USA

Contents

Chapitre 1 Introduction	1
Chapitre 2 Systèmes et Exigences de Gestion des Données DICOM	3
2.1 Caractéristiques des données et des charges de travail.....	3
2.1.1 Complexité élevée	3
2.1.2 Haute variété	3
2.1.3 Volume élevé et en constante augmentation	4
2.1.4 Grande vitesse	4
2.1.5 Diverses charges de travail.....	4
2.2 Systèmes de gestion de données DICOM.....	4
2.2.1 Besoins attendus	4
2.2.2 Systèmes existants.....	5
2.2.3 Systèmes existants.....	6
Chapitre 3 Bases de Données et Techniques Associées	9
3.1 Classements	9
3.1.1 Charges de travail OLTP et OLAP	9
3.1.2 Bases de données.....	9
3.2 Cadres de calcul en grappe	10
3.2.1 MapReduce	10
3.2.2 Spark	10
3.3 Dispositions de données	10
3.3.1 Modèle de stockage orienté ligne.....	10
3.3.2 Modèle de stockage orienté colonne	10
3.3.3 Modèle de stockage hybride.....	11
3.4 Partitionnement vertical et filtres de Bloom.....	11
3.4.1 Partitionnement vertical	11
3.4.2 Filtre de Bloom et l'intersection de filtres de Bloom.....	11
3.5 Résumé et conclusion	12
Chapitre 4 HYTORMO et HADF	13
4.1 HYTORMO et stratégies	13
4.1.1 Architecture d'HYTORMO	13
4.1.2 Stratégie de stockage de données	13
4.2 Approche de conception automatisée pour les données DICOM.....	15
4.3 Cadre de conception automatisé hybride	17
4.3.1 Aperçu du cadre	17
4.3.2 Exemple.....	19
Chapitre 5 Traitement de Requête pour HYTORMO.....	21
5.1 Stratégie de traitement des requêtes	21

5.1.1 Plan d'exécution de requête	21
5.1.2 Détermination des jonctions gauches-extérieures	22
5.1.3 Réduire le nombre de jointures externes gauche.....	22
5.2 Intégration de filtres de Bloom et rapport coût-bénéfice.....	23
5.2.1 Intersection des filtres de Bloom.....	23
5.2.2 Analyse coût-bénéfice	24
5.2.3 Intersection incrémentale de Filtres de Bloom.....	25
Chapitre 6 Évaluation des Performances	27
6.1 Environnement expérimental.....	27
6.2 Exécution des expériences	28
6.3 Analyse et interprétation.....	31
6.3.1 Résultats de l'hypothèse H1	31
6.3.2 Résultats de l'hypothèse H2	32
6.3.3 Résultats de l'hypothèse H3	32
Chapitre 7 Conclusion et Travaux Futurs	33
7.1 Résumé et conclusion	33
7.2 Travaux futurs.....	33

Mots clés : DICOM, données volumineuses, données clairsemées, HYTORMO, modèle de stockage hybride, stockage en lignes, stockage en colonnes, similarité hybride, filtre de Bloom, conception automatisée, jointure.

Chapitre 1 Introduction

Dans le secteur de la santé, la norme DICOM (Digital Imaging and Communication in Medicine) est utilisée pour stocker les données d'imagerie médicale. Les données DICOM possèdent les caractéristiques du Big Data, telles que la haute complexité, la grande variété, de grands volumes, en augmentation constante, et une importante vélocité. De plus, il existe une variété de charges de travail, notamment le traitement transactionnel en ligne (Online Transaction Processing, abrégé en OLTP), le traitement analytique en ligne (Online Analytical Processing, abrégé en OLAP) et les charges de travail mixtes. Ces caractéristiques et charges de travail des données posent de nombreux problèmes dans la gestion des données. Les systèmes existants ont des limites concernant ces caractéristiques des données et des charges de travail. Dans cette thèse, nous proposons des méthodes efficaces pour stocker et interroger les données DICOM en termes de d'espace de stockage et de temps d'exécution.

Dans la communauté de recherche de base de données, nombreuses techniques ont été proposées pour réduire la demande d'espace de stockage et améliorer la performance de la charge de travail pour les données à grande échelle telles que: (1) le partitionnement vertical pour réduire le nombre de valeurs nulles dans les ensembles de données éparses ou pour améliorer les performances des requêtes; (2) les modèles hybrides de stockage en lignes et en colonnes proposés pour augmenter les performances des charges de travail mixtes ; ou (3) des bases de données NoSQL qui sont bien adaptées pour traiter la grande variété et les volumes élevés de données.

Cependant, il y a des manques :

- Un modèle de stockage de données avec haute performance, évolutivité, disponibilité et élasticité pour les volumes élevés de données DICOM.
- Une stratégie de stockage de données pour réduire à la fois l'espace de stockage et le temps d'exécution des requêtes.
- Un modèle d'aide à la décision pour les décideurs (par exemple, les concepteurs de bases de données) dans la conception de schémas et la sélection de dispositifs de stockage de données appropriés.
- Un traitement de requête adapté et efficace.

En réponse aux problèmes ci-dessus, les objectifs de cette thèse sont de proposer :

- Un nouveau modèle de stockage hybride appelé HYTORMO qui offre hautes performances, évolutivité, disponibilité et élasticité.
- Une stratégie efficace de stockage de données qui est un moyen systématique de regrouper les attributs en groupes de colonnes et de suggérer des dispositions de stockage de données appropriées.
- Un cadre de conception automatisé hybride, appelé HADF, qui prend en compte l'impact combiné des informations spécifiques à la charge de travail et aux données et un stockage hybride pour créer des configurations de stockage de données.
- Un traitement de requête adapté et efficace pour HYTORMO avec l'intégration de filtres de Bloom (IBF) pour réduire les entrées/sorties sur le réseau.

La thèse fournit également des expérimentations pour démontrer les avantages des méthodes proposées à l'aide de véritables ensembles de données DICOM.

Chapitre 2 Systèmes et Exigences de Gestion des Données DICOM

Dans ce chapitre, nous déterminons d'abord les principales caractéristiques des données DICOM et des charges de travail susceptibles de poser des problèmes de gestion des données. Ensuite, nous présentons les exigences attendues. Ensuite, nous passons en revue les systèmes de gestion de données DICOM existants en mettant l'accent sur leurs forces et leurs faiblesses. Après cela, nous comparons ces systèmes et concluons avec leurs limites à satisfaire les exigences attendues.

2.5 Caractéristiques des données et des charges de travail

2.2.3 Complexité élevée

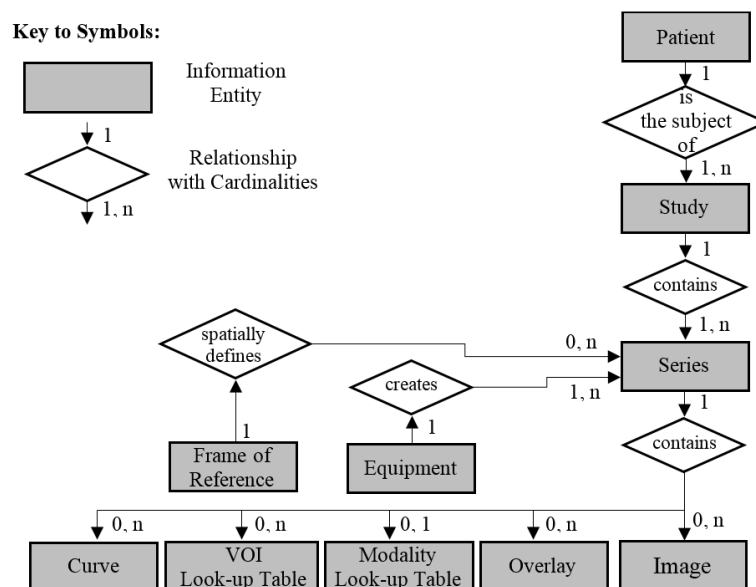


Figure 2.1: Modèle d'information détaillé

L'organisation des données DICOM est complexe. Le *modèle d'information* de la norme DICOM est utilisé pour décrire les informations sur les entités et leurs relations, comme illustré par la figure 2.1. Une *entité d'information* (IE : information entity) est utilisée pour représenter des informations pour un seul objet du monde réel tel que *Patient*, *Study*, etc. L'IE se compose à son tour d'une liste d'attributs. En outre, il existe des relations entre les IE. Par conséquent, les requêtes nécessitent généralement plusieurs jointures pour intégrer les informations des tables.

2.2.4 Haute variété

Les données DICOM sont très variées car elles incluent des données d'image et des métadonnées. En outre, il existe une hétérogénéité et une évolution des métadonnées.

- **Schéma hétérogène :** Le nombre d'attributs est très important (plus de 3,500). Ils incluent des attributs obligatoires et facultatifs. Cela peut conduire aux problèmes

suivant : espace de stockage peut être gaspillé en raison de valeurs nulles ; les performances des requêtes peuvent être réduites en raison d'accès aux attributs non pertinents.

- **Schéma évolutif** : L'évolution du schéma fait référence aux changements dans les schémas des métadonnées au cours du temps, à mesure que les attributs sont modifiés. Cela conduit à des problèmes de gestion des données. Par exemple, comment garder le système existant pour fonctionner normalement en présence de schémas de « nouvelle version » est un défi.

2.2.5 Volume élevé et en constante augmentation

Le volume élevé et toujours croissant de données a introduit des défis à la gestion moderne des données. Des stratégies efficaces de stockage à long terme et de traitement des données doivent être appliquées pour garantir la rapidité du traitement des données et réduire les entrées/sorties.

2.2.6 Grande vitesse

La vitesse est considérée comme la vitesse des flux de données à venir qui doivent être traités aussi rapidement que possible. Cependant, notre étude se concentre sur des méthodes efficaces pour améliorer la vitesse des requêtes OLAP, OLTP et des charges de travail mixtes, au lieu du traitement des flux de données.

2.2.7 Diverses charges de travail

Les charges de travail sont diverses (OLTP, OLAP, charges de travail mixtes). Les modèles d'accès aux attributs et les requêtes conduisent souvent à des opérations de jointure multi-tables. Certains attributs sont fréquemment accédés ensemble tandis que certains attributs sont rarement utilisés ensemble. De telles charges de travail peuvent avoir un impact négatif sur les performances des requêtes.

2.6 Systèmes de gestion de données DICOM

2.3.3 Besoins attendus

Ci-dessous, nous spécifions les exigences attendues pour un nouveau système de gestion de données DICOM:

R1) Données flexibles : le système est capable de gérer la complexité des données DICOM en permettant aux utilisateurs de représenter facilement les tables d'entités et leurs relations dans le modèle d'information DICOM. Des données normalisées doivent être créées. De plus, le système est capable de gérer la variété de données DICOM.

R2) Requêtes flexibles : le système permet aux utilisateurs d'écrire des requêtes SQL ad hoc avec des opérations de jointure.

R3) Efficacité du stockage et de traitement : tout d'abord, les données doivent être organisées en fonction d'informations sur la charge de travail et sur

les données afin de réduire la demande d'espace de stockage et l'exécution de requêtes dans des charges de travail OLTP et OLAP mixtes. Plus particulièrement, les données doivent être organisées et stockées de manière appropriée afin de réduire la redondance des données, le coût de reconstruction du n-uplet et les coûts d'E/S. Deuxièmement, le système est en mesure de fournir des solutions pour un traitement efficace des requêtes sur des jeux de données DICOM à grande échelle. Enfin, il est capable de fournir une capacité de stockage, une évolutivité et une élasticité énormes.

2.3.4 Systèmes existants

PACS : Les PACS (Picture Archiving and Communication System) utilisent principalement des SGBDR (systèmes de gestion de base de données) orientés ligne pour stocker, récupérer et distribuer des données d'images médicales. Ils fournissent des techniques d'index robustes pour l'accélération des opérations de récupération de données. Cependant, ils ne prennent en charge que les requêtes avec des attributs prédéfinis et ne gèrent pas les schémas hétérogènes.

eDiaMoND : eDiaMoND (Grid-enabled Medical Imaging Database) stocke les données DICOM à l'aide d'une base de données d'images médicales basée sur une grille informatique (grid computing) qui est construite à partir de SGBDR orientés ligne. Le système vise à assurer l'interopérabilité, l'évolutivité et la flexibilité. Cependant, le développement de techniques d'optimisation de requêtes n'a pas été introduit. De plus, eDiaMoND ne fournit aux utilisateurs que des requêtes prédéfinies (sous forme de document XML).

Oracle : Oracle est un SGBDR orienté ligne qui fournit des fonctionnalités pour stocker et gérer des référentiels à grande échelle de fichiers DICOM. Il ajoute un nouveau type de données qui permet à n'importe quelle colonne de ce type de stocker un contenu DICOM dans leur table de base de données. Depuis qu'un nouvel objet séparé est créé pour chaque fichier DICOM, l'espace de stockage est rapidement augmenté et diminue ainsi les performances globales du système. Oracle RACs permet de stocker et de gérer les données DICOM dans un environnement de type grappe (cluster) pour fournir disponibilité, performance, évolutivité et élasticité. Cette solution permet de fournir un débit élevé pour les charges de travail OLTP mais n'optimise pas les charges de travail OLAP. Cette approche est également moins évolutive que certaines bases de données NoSQL, telles que Cassandra et MongoDB.

DCMDSM : DCMDSM (DICOM Decomposed Storage Model) partitionne verticalement les métadonnées DICOM en plusieurs petites tables. La méthode est capable de gérer les schémas évolutifs/hétérogènes et d'économiser de la bande passante. Cependant, le modèle utilise une approche de base de données centralisée développée au sommet d'un SGBDR orienté ligne et n'a pas été conçue pour fonctionner dans un environnement parallèle. En outre, ils peuvent entraîner des coûts plus élevés en raison des jointures supplémentaires nécessaires pour la reconstruction des n-uplets.

Base de données documentaire : Une base de données basée sur des documents, telle que CouchDB, a été proposée pour stocker et interroger des données DICOM. Il partage la conception non relationnelle sans schéma des systèmes de stockage clé-

valeur standards et peut donc gérer l'évolution des métadonnées. Cependant, il n'existe pas de langage de requête standard pour le système proposé. En outre, il est difficile de représenter les tables et leurs relations dans le modèle d'information DICOM.

Système de stockage hybride compatible avec le cloud : Le système de stockage hybride basé sur le cloud stocke les données DICOM dans les magasins de lignes et de colonnes. Tout d'abord, les attributs DICOM sont classés en trois catégories : (1) attributs obligatoires ; (2) les attributs fréquemment accédés ensemble; et (3) les attributs facultatifs / privés / rarement accédés. Ensuite, les attributs sont regroupés manuellement selon ces catégories. Enfin, la sélection des dispositions de stockage de données pour les groupes de colonnes est décrite comme suit :

- Les attributs appartenant aux deux premières catégories sont regroupés et stockés dans des tables de lignes afin de réduire le coût de reconstruction des n-uplets.
- Les attributs appartenant à la dernière catégorie sont stockés dans des tables de colonnes afin d'économiser le coût d'E / S si seulement quelques attributs sont requis par une requête.

Ce système peut gérer la complexité et l'évolution des données. Les coûts de reconstruction des E / S et des n-uplets sont diminués. Cependant, il y a certaines limites. Tout d'abord, le regroupement des attributs et la sélection des dispositions de stockage de données appropriées sont effectués manuellement. Nous appelons cette méthode approche d'expert. Deuxièmement, le médiateur distribué doit décider du moteur de stockage le plus approprié pour effectuer une requête et déplacer des données entre les moteurs de stockage. Enfin, le système ne passe pas facilement à l'échelle.

2.3.5 Systèmes existants

Les exigences attendues énumérées à la section 2.2.1 sont utilisées comme critères de comparaison des systèmes existants. Le tableau 2.1 montre le résultat de la comparaison.

Table 2.1 : Comparaison des systèmes existants

Systèmes de gestion de données DICOM existants	Besoins attendus		
	R1	R2	R3
PACSs	0	-	-
eDiaMoND	+	-	-
Oracle/Oracle RAC	+	0	-
DCMDSM	+	0	-
Base de données documentaire	+	-	0
Système de stockage hybride compatible avec le cloud	+	+	0

+ pris en charge, 0 partiel, - non pris en charge

Les systèmes utilisant des bases de données relationnelles, notamment les systèmes PACS, eDiaMoND et Oracle / Oracle RAC, peuvent traiter des données extrêmement complexes, créer des données normalisées et prendre en charge SQL et les jointures. Cependant, ils ne disposent pas de solutions pour : (1) organiser les données en fonction de la charge de travail et d'informations spécifiques afin de réduire la demande

d'espace de stockage et le temps d'exécution de la charge de travail ; (2) fournir une stratégie de traitement de requête efficace ; et (3) fournir une énorme capacité de stockage, évolutivité et élasticité.

Le modèle DCMDSM peut aider à améliorer les requêtes OLAP et à réduire la demande d'espace de stockage en raison du modèle DSM. Néanmoins, le coût d'exécution des requêtes OLTP peut être élevé en raison des jointures multi-tables. De plus, le système existant a été conçu et validé en utilisant un environnement de traitement distribué.

La base de données basée sur les documents et le système de stockage hybride compatible avec le cloud possèdent de nombreuses fonctionnalités capables de gérer les caractéristiques des données et des charges de travail DICOM. La base de données basée sur les documents est une base de données NoSQL. Elle peut donc gérer la grande variété de données DICOM et fournir des performances de requête élevées, une capacité de stockage importante, une évolutivité et une élasticité naturelles. Le système de stockage hybride en nuage a fourni des solutions en fonction de la charge de travail et d'informations spécifiques aux données pour organiser et stocker les données DICOM. Cependant, ces deux systèmes ne disposent pas des fonctionnalités suivantes :

- Une approche de conception automatisée qui utilise des informations spécifiques à la charge de travail et aux données pour concevoir et stocker les données DICOM de manière à réduire à la fois le temps d'exécution des charges de travail et la demande d'espace de stockage.
- Des solutions efficaces pour le traitement des requêtes sur les jeux de données à grande échelle, en particulier pour réduire les E / S réseau dans un environnement de traitement de requêtes distribué.

Chapitre 3 Bases de Données et Techniques Associées

Ce chapitre fournit tout d'abord une analyse des types de charge de travail, des bases de données courantes, des structures d'informatique en cluster et des dispositions de données. Ensuite, nous nous concentrons sur les techniques de partitionnement vertical appliquées pour réduire l'espace de stockage (en particulier pour les jeux de données fragmentés) et pour améliorer les performances des requêtes. Après cela, nous introduisons des techniques de filtre de Bloom (BF) et de filtre de Bloom d'Intersection (IBF) qui peuvent être appliquées pour améliorer les performances des requêtes dans les environnements de traitement de requêtes distribués. Enfin, nous résumons et concluons le chapitre en sélectionnant des solutions pour les composants clés d'un nouveau système de gestion de données DICOM.

3.8 Classements

3.4.4 Charges de travail OLTP et OLAP

Les charges de travail OLTP contiennent des requêtes exigeantes en écriture qui doivent insérer, supprimer, mettre à jour ou extraire toutes les colonnes (ou la plupart des colonnes) d'une table. Les bases de données orientées lignes sont optimisées en écriture pour les applications OLTP.

En revanche, les charges de travail OLAP sont principalement constituées de requêtes nécessitant une lecture intensive qui doivent accéder ou être agrégées sur plusieurs lignes, mais uniquement sur quelques colonnes. Les bases de données orientées colonnes sont optimisées en lecture, elles sont donc généralement utilisées pour les applications OLAP.

3.4.5 Bases de données

Bases de données relationnelles : Les bases de données relationnelles organisent les données en fonction du *modèle de données relationnel* qui utilise des tables ou des schémas pour organiser et récupérer des données. Elles sont conçues pour stocker des données structurées.

Bases de données NoSQL : Les bases de données NoSQL sont basées sur des modèles de données flexibles sans avoir besoin de schémas prédéfinis. Elles peuvent donc gérer des données non structurées ou semi-structurées stockées dans des systèmes de stockage de clé-valeur, familles de colonnes, documents ou dans des systèmes de gestion de bases de données.

Bases de données NewSQL : Les bases de données NewSQL (telles que VoltDB, Clustrix, NuoDB et Google Spanner) sont considérées comme des SGBDR modernes. Ils sont basés sur le modèle de données relationnelles, mais peuvent fournir une évolutivité horizontale et des performances élevées en tant que bases de données NoSQL tout en garantissant les garanties ACID traditionnelles et en fournissant du SQL.

3.9 Cadres de calcul en grappe

Notre étude est axée sur la technique de requête et d'analyse interactive et ad hoc qui est généralement comparée à la technique de traitement par lots. Ainsi, dans cette section, nous nous concentrons uniquement sur deux modèles de calcul parallèles : MapReduce et Spark. La première technique utilisée est un modèle de programmation par lots performant, tandis que la dernière est une infrastructure informatique en grappe capable de fournir des performances élevées pour des charges de travail interactives.

3.2.1 MapReduce

MapReduce est un modèle de programmation par lots. Cette technique traite un grand volume de données en divisant un travail en plusieurs tâches qui sont effectuées en parallèle sur plusieurs nœuds (machines). Cependant, ce modèle entraîne des entrées/sorties disque et une latence réseau élevés car ses tâches doivent répliquer les données pour le calcul local au niveau des nœuds.

3.2.2 Spark

Le modèle de traitement par lots de MapReduce n'est pas adapté aux requêtes et analyses interactives ad-hoc en raison de sa latence élevée. En revanche, Spark est une infrastructure de calcul en mémoire pouvant s'exécuter sur Hadoop pour offrir des performances élevées aux charges de travail interactives. Pour réduire la latence, il essaie de conserver les données intermédiaires en mémoire autant que possible afin de réduire le besoin d'écrire les données sur des disques. De plus, les DataFrames dans Spark permettent aux utilisateurs de représenter des données sous forme de tables. Spark permet d'interroger les données à l'aide d'un langage de type SQL.

3.10 Dispositions de données

3.3.1 Modèle de stockage orienté ligne

Le modèle de stockage orienté ligne qui est utilisé dans les SGBDR orientés ligne (tels qu'Oracle, DB2, SQL Server, etc.) stocke les données ligne par ligne. Ce modèle est optimisé en écriture pour les charges de travail OLTP où tous les attributs d'un n-uplet sont écrits une fois par requête. Cependant, il gaspille les entrées/sorties disque pour les charges de travail OLAP car toute la table doit encore être lue en mémoire à partir du disque, même si seulement quelques attributs sont requis.

3.3.2 Modèle de stockage orienté colonne

Le modèle de stockage orienté colonne qui est utilisé dans les SGBDR à colonnes (tels que MonetDB et C-Store) stocke les données sur le disque colonne par colonne. Ce modèle est optimisé en lecture car il permet de lire uniquement les colonnes requises. Ceci est bien adapté aux charges de travail OLAP où seul un petit nombre d'attributs d'une table peut être utilisé.

3.3.3 Modèle de stockage hybride

Les modèles de stockage de données présentés dans les sections précédentes sont optimisés pour une charge de travail OLTP ou OLAP, mais pas pour les deux. Pour surmonter cette limite, des modèles hybrides de stockage de données ont été introduits, tels que des modèles de stockage de groupes de colonnes (PAX, Data Morphing, HYRISE), Mirror et Fractured Mirrors, HyPer, Colonnes de Trojan et SAP HANA. Cependant, ces modèles n'ont pas été conçus pour exploiter les données DICOM. Par exemple, il y a un manque de solutions pour réduire la demande d'espace de stockage et le temps d'exécution des requêtes.

3.4 Partitionnement vertical et filtres de Bloom

Les techniques de partitionnement vertical permettent de réduire le temps d'exécution de la charge de travail et la taille de l'espace de stockage pour les jeux de données fragmentés. En outre, les techniques de filtre de Bloom permettent de réduire les coûts d'E / S réseau et disque dans les environnements de traitement de requêtes distribuées.

3.5.3 Partitionnement vertical

Dans notre étude, nous classons les algorithmes de partitionnement vertical actuels en deux approches : *l'approche basée sur la charge de travail* et *l'approche basée sur les données*. La première catégorie utilise des informations sur l'utilisation des attributs des requêtes pour générer des partitions verticales de manière à améliorer les performances des requêtes. En revanche, la seconde approche utilise des informations spécifiques aux données (par exemple, l'écart des données) pour regrouper les attributs en grappes (c'est-à-dire, des partitions verticales). Cette approche vise principalement à réduire le nombre de valeurs nulles dans les ensembles de données épars.

Cependant, il y a un manque d'études qui prennent en considération l'impact combiné des informations spécifiques à la charge de travail et aux données sur le résultat du partitionnement vertical et l'utilisation d'une disposition de stockage de données différente pour les stocker.

3.5.4 Filtre de Bloom et l'intersection de filtres de Bloom

Le filtre de Bloom (BF: Bloom filter) est une structure de données probabiliste compacte qui est utilisée pour les tests d'appartenance avec peu d'erreurs permises. Alternativement, une intersection de filtres de Bloom (abrégé en IBF) qui est calculée en effectuant des opérateurs AND au niveau des bits sur les BF peut être utilisée pour représenter une intersection approximative d'ensembles. Le BF et l'IBF peuvent tous deux être appliqués pour améliorer les performances des requêtes en filtrant les données non pertinentes parmi les entrées des opérations de jointure. La probabilité de faux positifs de l'IBF a été prouvée inférieure à celle de ses composantes BF.

Dans le contexte de la gestion de données DICOM, les requêtes de jonction de tables multiples des utilisateurs peuvent impliquer une grande quantité ou des données d'entrée redondantes en raison de la grande sélectivité des prédicats. Par conséquent,

une stratégie de traitement de requête avec l'intégration de l'IBF est une solution potentielle pour améliorer la performance des requêtes. Cependant, il existe un manque d'études qui appliquent l'IBF qui est calculé à partir de fichiers BF non pré-calculés dans un environnement de traitement de requête distribué, par exemple, Spark. De plus, nous devons déterminer comment intégrer un IBF dans un plan d'exécution particulier et effectuer une analyse coûts-avantages pour cette application.

3.5 Résumé et conclusion

Les composants clés du nouveau système (modèle de données, modèle de stockage de données, schéma de données et traitement des requêtes) doivent être conçus de manière à satisfaire aux exigences attendues en matière de stockage et d'interrogation des données DICOM : (R1) données souples ; (R2) interrogation flexible ; et (R3) Efficacité du stockage et de la CPU :

Modèle de données : Le modèle de données relationnel doit être appliqué pour représenter facilement les entités et les relations du modèle d'information DICOM, pour fournir du code SQL et pour pouvoir créer des données normalisées. Cependant, par rapport aux bases de données NoSQL, les bases de données relationnelles ont des limites pour fournir des performances de requête élevées, un stockage de données énorme et une évolutivité horizontale. Il est clair qu'une base de données relationnelle pure et une base de données NoSQL pure ne fournissent pas toutes les fonctionnalités requises. Nous nous orientons donc vers une base de données NoSQL, mais nous devons prendre en charge l'utilisation efficace du SQL et la représentation des données sous forme de tables.

Modèle de stockage de données : En raison de la variété de la charge de travail (charge de travail mixte OLTP et OLAP), un modèle de stockage hybride en lignes et en colonnes est utilisé. En outre, les modèles de stockage hybride existants ont encore des limites pour gérer une grande quantité de données. Par conséquent, pour faire face au volume élevé et croissant de données DICOM, le nouveau modèle de stockage hybride doit être conçu et mis en œuvre.

Schéma de données : Les algorithmes de partitionnement verticaux existants ont montré leur utilité dans la conception de schémas pour réduire le temps d'exécution de la charge de travail ou la taille de l'espace de stockage, mais il manque une solution pour prendre en compte utilisation de systèmes lignes et colonnes. Par conséquent, pour soutenir la prise de décision dans la conception de bases de données pour les données DICOM, il est nécessaire d'adopter une nouvelle approche de partitionnement vertical pour surmonter ces limites.

Traitement des requêtes : Le traitement de la requête doit produire des réponses correctes et fournir des performances élevées pour les charges de travail interactives. Spark devrait être choisi pour traiter les requêtes ad hoc interactives en raison de sa capacité à offrir une faible latence, de hautes performances, de l'évolutivité et de l'élasticité. Les jointures interne et externe doivent être appliquées pour créer les réponses correctes pour les opérations de jointure entre des tables partitionnées verticalement. Les IBF ont montré leur capacité à réduire le coût des E / S réseau, et devraient donc être appliquées à la stratégie de traitement des requêtes.

Chapitre 4 HYTORMO et HADF

Dans ce chapitre, nous introduisons d'abord l'architecture d'HYTORMO. Ensuite, nous introduisons deux approches de conception de base de données différentes, à savoir *la conception par expert* et *la conception automatisée*, pour créer des configurations de stockage de données pour les données DICOM.

4.6 HYTORMO et stratégies

4.2.4 Architecture d'HYTORMO

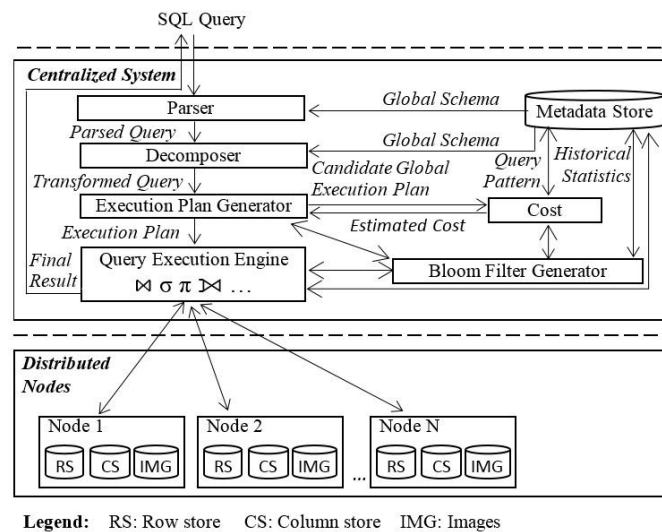


Figure 4.1: Architecture d'HYTORMO

La figure 4.1 décrit l'architecture de HYTORMO. Il existe deux composants clés : *le système centralisé* et *les nœuds distribués*. Les tâches de traitement des requêtes sont réparties entre plusieurs nœuds. Les données DICOM (métadonnées et données de pixel) sont stockées sur les nœuds distribués à l'aide d'un système de fichiers distribué, par exemple HDFS pouvant prendre en charge le stockage de données DICOM dans des dispositions de stockage en lignes et en colonnes.

4.2.5 Stratégie de stockage de données

Les objectifs de la stratégie de stockage des données sont d'optimiser les performances et l'espace de stockage des requêtes sur une charge de travail OLTP et OLAP mixte. Pour atteindre ces objectifs, les métadonnées et les données d'image des fichiers DICOM sont extraits, organisés et stockés de manière à réduire l'espace de stockage, le coût de construction des n-uplets et les coûts d'entrée / sortie.

La stratégie de stockage de données proposée est exécutée comme suit : *tout d'abord, les tables d'entités doivent être décomposées en plusieurs sous-tables (c'est-à-dire, des tables partitionnées verticalement). Ensuite, ces sous-tables seront stockées dans des systèmes lignes et colonnes du système hybride de HYTORMO (dans un système de fichiers distribué).*

Afin de réaliser une configuration de stockage de données conformément à la stratégie de stockage de données ci-dessus, l'une des deux approches de conception d'analyse est appliquée : basée sur des experts et automatisée. Dans cette section, nous présentons l'approche par experts.

Tout d'abord, nous étendons l'approche basée sur l'expertise proposée par B. Mohamad, L. d'Orazio et Gruenwald en fournissant des définitions claires de trois catégories d'attributs, y compris *Obligatoire*, *Fréquemment accédés ensemble* et *Optionnel/privé/rarement-accédé* (parfois appelé « *Optionnel* ») :

4. Les *attributs obligatoires* ne peuvent pas avoir pour valeur nulle et sont fréquemment accédés ensemble.
5. Les *attributs fréquemment accédés ensemble* peuvent avoir pour valeur nulle et sont fréquemment accédés ensemble.
6. Les *attributs optionnels* peuvent avoir la valeur nulle et ne sont pas fréquemment accédés ensemble.

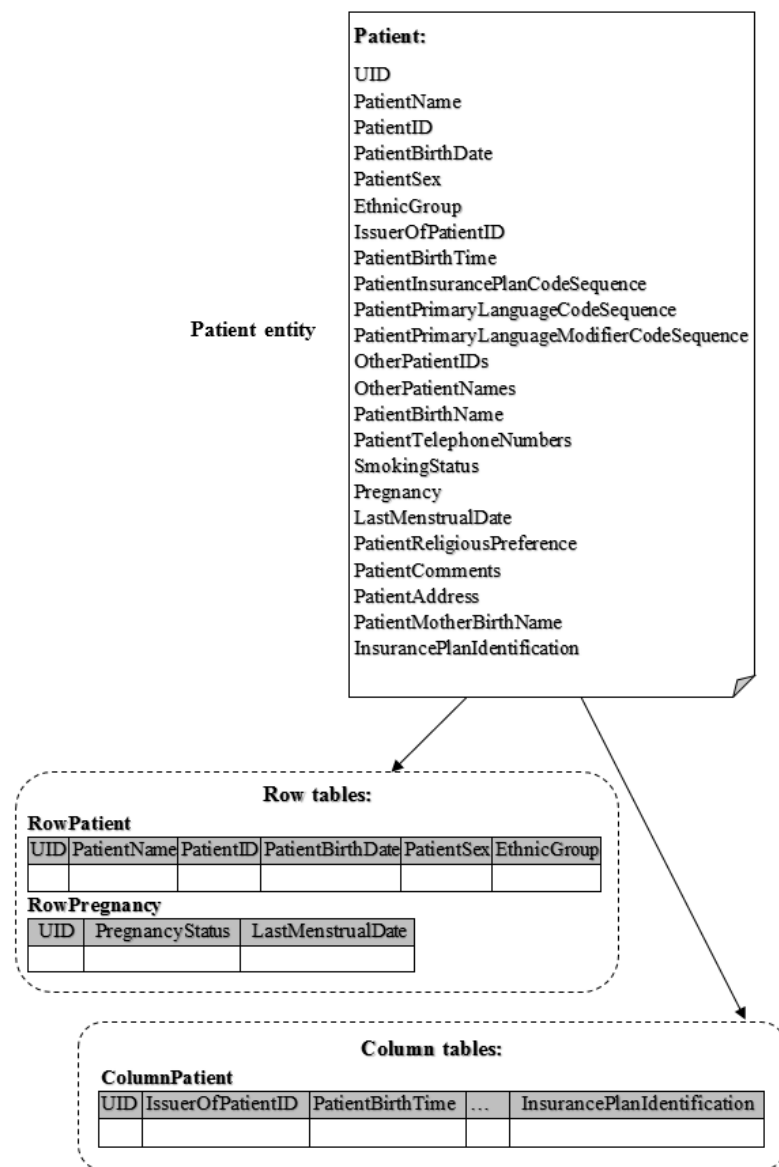


Figure 4.2: Tables en lignes et en colonnes de l'entité *Patient*

Ensuite, nous regroupons et stockons les attributs comme suit :

- 1) Les attributs des deux premières catégories sont regroupés et stockés dans des tables en lignes afin de réduire le coût de la reconstruction des n-uplets, car ils sont fréquemment interrogés ensemble.
- 2) Les attributs appartenant à la dernière catégorie (c'est-à-dire, optionnel) sont stockés dans des tables en colonnes afin d'économiser des coûts d'entrée/sortie si seulement quelques attributs sont accédés par requête à la fois.

La figure 4.2 montre que les attributs de l'entité *Patient* sont stockés comme suit : *RowPatient* et *RowPregnancy* sont des tables en lignes stockant respectivement des attributs obligatoires et des attributs fréquemment accédés. D'un autre côté, *ColumnPatient* est une table en colonnes qui stocke des attributs optionnels.

4.7 Approche de conception automatisée pour les données DICOM

En pratique, il peut être difficile pour les experts d'évaluer manuellement la relation de similarité parmi un grand nombre d'attributs en fonction à la fois des informations spécifiques à la charge de travail et aux données, ainsi que de déterminer le format de données approprié pour chaque groupe de colonnes. Pour cette raison, dans cette section, nous fournissons une représentation formelle du problème de conception automatisée et des modèles de coûts.

Informations spécifiques à la charge de travail

Nous décrivons une charge de travail $W = (A, Q, AUM, F)$ comme suit :

- $A = \{UID, a_1, a_2, \dots, a_n\}$ est un ensemble de tous les attributs d'une table horizontale T .
- $Q = \{q_1, q_2, \dots, q_m\}$ est un ensemble de requêtes exécutées sur T .
- AUM est une matrice d'utilisation des attributs (attribute usage matrix) de taille $m \times n$.
- $F = \{f_1, f_2, \dots, f_m\}$ est un ensemble de fréquences de requête (query frequency).

	a_1	a_2	a_3	a_4	a_5	a_6
q_1	0	1	1	1	1	1
q_2	0	0	1	1	1	1
q_3	0	0	0	1	1	1
q_4	1	1	0	0	0	0
q_5	1	1	1	0	0	0
q_6	0	1	1	0	0	0

	F
q_1	600
q_2	500
q_3	700
q_4	1000
q_5	200
q_6	400

(a) Attribute Usage Matrix (AUM) (b) Query frequency (F)

Figure 4.3: Exemple de matrice d'utilisation des attributs et fréquences de requête

Par exemple, la figure 4.3 présente deux composantes AUM et F de la charge de travail de la table horizontale T . Dans notre étude, par défaut, l'attribut UID est inclus dans toutes les tables de partition verticales, il n'est donc pas représenté dans l' AUM .

Informations spécifiques aux données

Les caractéristiques des données sont dérivées de la table horizontale T. La figure 4.4 montre un exemple de T avec 7 attributs, $A = \{UID, a_1, a_2, a_3, a_4, a_5, a_6\}$.

UID	a_1	a_2	a_3	a_4	a_5	a_6
01	V1.1	V1.2	V1.3	V1.4	V1.5	
02	V2.1	V2.2	V2.3	V2.4		V2.6
03	V3.1	V3.2	V3.3	V3.4		
04	V4.1	V4.2	V4.3	V4.4		
05	V5.1	V5.2	V5.3	V5.4		
06	V6.1	V6.2	V6.3	V6.4		
07	V7.1	V7.2	V7.3	V7.4	V7.5	
08	V8.1	V8.2	V8.3	V8.4	V8.5	V8.6
09	V9.1	V9.2	V9.3			
10	V10.1	V10.2	V10.3			

Figure 4.4: Exemple de la table horizontale T

Configuration Représentation d'une configuration de stockage de données

Soit $S = \{\text{"row-store"}, \text{"column-store"}\}$ qui désigne un ensemble de dispositions de stockage de données disponibles. On note un ensemble de configurations de stockage de données candidates pour la table horizontale T comme $G = \{G_1, G_2, \dots, G_K\}$. Chaque $G_i = (C_i, L_i)$ est constitué de deux composantes : un ensemble $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$ des groupes de colonnes (c'est-à-dire, partitions verticales) et un ensemble $L_i = \{L_{d_1}(C_{i,1}), L_{d_2}(C_{i,2}), \dots, L_{d_z}(C_{i,z})\}$ des dispositions de stockage de données suggérées. $L_{d_x}(C_{i,x})$ indique que le groupe de colonnes $C_{i,x}$ est stocké dans la structure de stockage de données $d_x \in S$.

Fonction objectif

Le problème de la conception automatisée peut être formulé comme suit : *À partir d'une table horizontale T et d'une charge de travail W, recherchez une configuration de stockage de données G_i pour T afin de minimiser la valeur des deux fonctions de coût: STORAGE_COST(W, G_i) et EXECUTION_COST(W, G_i). Cette fonction objectif est décrite comme suit :*

$$\begin{cases} \text{STORAGE_COST}(W, G_i) \rightarrow \min \\ \text{EXECUTION_COST}(W, G_i) \rightarrow \min \end{cases} \quad (4.2.1)$$

où le coût $\text{STORAGE_COST}(W, G_i)$ est le nombre total de cellules de données utilisées pour stocker tous les groupes de colonnes de G_i alors que le coût $\text{EXECUTION_COST}(W, G_i)$ est le coût d'exécution de la charge de travail W.

La configuration $G_i = (C_i, L_i)$ est produite à la suite de l'application de la stratégie de stockage de données proposée pour générer un ensemble C_i et un ensemble L_i .

Le coût de stockage d'une configuration de stockage de données G_i est assimilé au nombre total de cellules de données de tous les groupes de colonnes $C_{i,x}$ de G_i (après suppression de toutes les lignes nulles):

$$\text{STORAGE_COST}(G_i) = \sum_{C_i \in G_i, C_{i,x} \in C_i} \text{COLUMN_GROUP_SIZE}(C_{i,x}) \quad (4.2.2)$$

Le coût d'exécution d'une requête q lors de l'utilisation de la configuration G_i peut être noté par la fonction coût EXECUTION_COST(q, G_i) comme suit :

$$\begin{aligned} \text{EXECUTION_COST}(q, G_i) \\ = \text{READ_COST}(q, G_i) + \text{RECONSTRUCTION_COST}(q, G_i) \end{aligned} \quad (4.2.3)$$

Le coût d'exécution de la charge de travail W peut être estimé comme suit:

$$\text{COST}(W, G_i) = \sum_{q \in W} \text{COST}(q, G_i) \quad (4.2.4)$$

Motivation

L'espace de recherche de solution pour une configuration optimale qui peut minimiser la fonction d'objectif (montré dans la formule (4.2.1)) est très important. Pour pallier cette limite, nous proposons un cadre de conception automatisée hybride qui permet d'obtenir rapidement une bonne configuration.

4.8 Cadre de conception automatisé hybride

4.4.5 Aperçu du cadre

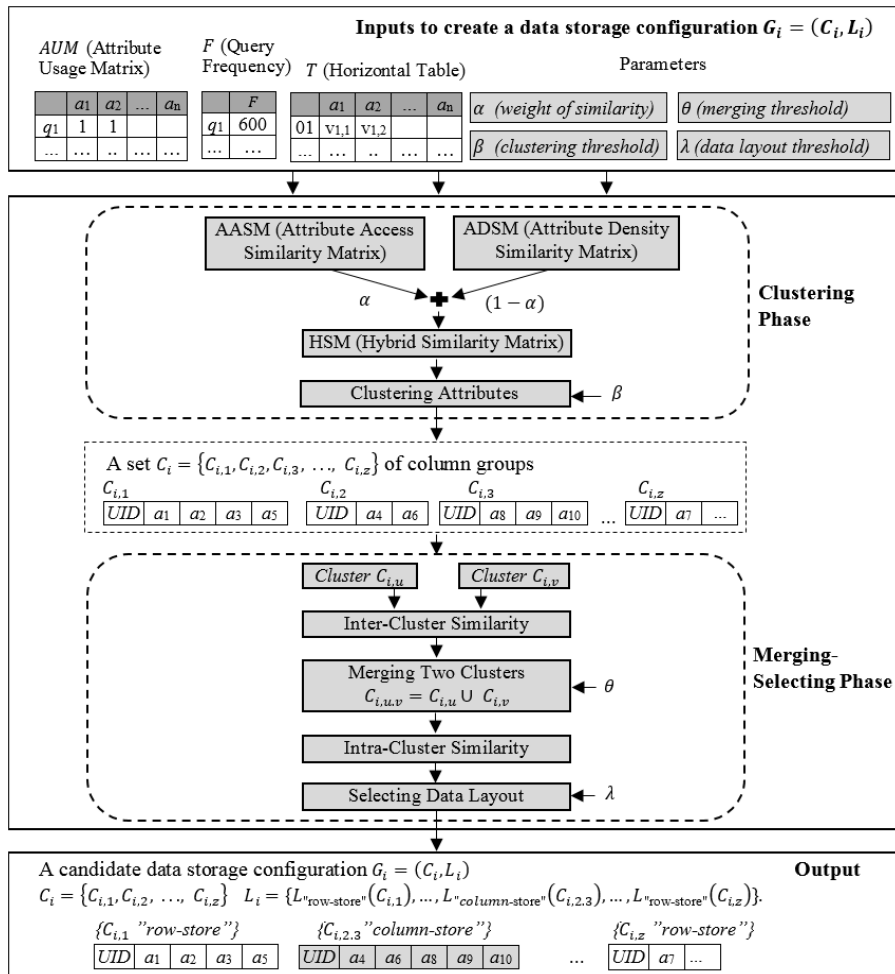


Figure 4.5: Vue d'ensemble de HADF

Dans cette section, nous présentons un cadre de conception automatisé hybride, appelé HADF. HADF est une approche heuristique basée à la fois sur des informations spécifiques à la charge de travail et aux données pour produire automatiquement des configurations de stockage de données pour les données DICOM. Pour cette raison, nous disons que HADF dépend d'une approche de conception automatisée basée sur la charge de travail et les données.

La figure 4.5 montre un HADF global qui utilise des entrées données pour effectuer deux phases, *phase de regroupement* (clustering phase) et *phase de fusion-sélection* (merging-selecting phase), pour générer automatiquement une configuration candidate $G_i = (C_i, L_i)$, où $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$ est un ensemble de groupes de colonnes et $L_i = \{L_{d_1}(C_{i,1}), L_{d_2}(C_{i,2}), \dots, L_{d_z}(C_{i,z})\}$ est un ensemble de dispositions de stockage de données suggérées.

Pour réaliser une configuration candidate G_i , trois groupes d'entrées sont requis pour l'exécution de HADF : (1) *Entrées spécifiques à la charge de travail* : AUM et F. (2) *Entrée spécifique aux données* : T. (3) *Paramètres* : poids α pour gérer la priorité sur la similarité ; seuil β pour les attributs de regroupement ; seuil θ pour fusionner une paire de groupes ; et le seuil λ pour sélectionner une disposition de stockage de données appropriée. Ces paramètres vont de 0 à 1.

Deux phases de HADF sont données ci-dessous :

- **Phase de regroupement** : Cette phase vise à réduire l'espace de stockage et à améliorer les performances des requêtes en réduisant les accès aux attributs non pertinents. Il prend en compte l'impact combiné des *informations spécifiques à la charge de travail* (workload-specific information) et *spécifiques aux données* (data-specific information) sur la qualité du résultat du partitionnement vertical en termes d'espace de stockage et de performance des requêtes. Il calcule d'abord deux matrices de similarité : *la matrice de similarité d'accès d'attribut AASM* (Attribute Access Similarity Matrix) en utilisant AUM et F et *la matrice de similarité de densité d'attribut ADSM* (Attribute Density Similarity Matrix) en utilisant T. Ensuite, *la matrice de similarité hybride HSM* (Hybrid Similarity Matrix) est calculée en combinant AASM et ADSM avec un poids α . Enfin, la phase de regroupement rapprochera les attributs en sous-espaces (c'est-à-dire des groupes de colonnes) de sorte que la similarité hybride (donnée dans HSM) entre deux attributs dans les mêmes sous-espaces soit supérieure ou égale à β . La sortie de cette phase est un ensemble $C_i = \{C_{i,1}, C_{i,2}, \dots, C_{i,z}\}$ des groupes de colonnes.
- **Phase de fusion-sélection** : Cette phase vise à améliorer encore les performances de la requête en réduisant à la fois le coût de reconstruction des n-uplets (le nombre de jointures supplémentaires) et les accès aux attributs non pertinents. Les groupes de colonnes résultants de la phase de regroupement sont utilisés comme entrée initiale pour cette phase. La phase de fusion-sélection commence par le calcul de la *similarité entre les groupes* (Inter-Cluster Similarity) qui mesure le rapport d'accès qui se chevauchent entre les paires de groupes de colonnes. Une paire de groupes de colonnes est fusionnée pour créer un nouveau groupe de colonnes si leur similarité entre les groupes est supérieure ou égale à θ . En outre, un groupe de colonnes est stocké dans un stockage en lignes si sa *similarité intra-cluster* (Intra-Cluster Similarity) qui mesure le rapport d'accès d'attribut à ce groupe de colonnes

est supérieure ou égale à λ ; sinon, un stockage de colonnes est utilisé. Comme l'illustre la figure 4.5, deux groupes de colonnes $C_{i,2}$ et $C_{i,3}$ sont fusionnés dans $C_{i,2.3}$ qui est stocké dans un stockage en colonnes. Cette procédure est répétée de manière similaire jusqu'à ce que toutes les paires de groupes de colonnes soient prises en compte. Cette phase renvoie une configuration candidate $G_i = (C_i, L_i)$.

4.4.6 Exemple

	a_1	a_2	a_3	a_4	a_5	a_6
q_1	0	1	1	1	1	1
q_2	0	0	1	1	1	1
q_3	0	0	0	1	1	1
q_4	1	1	0	0	0	0
q_5	1	1	1	0	0	0
q_6	0	1	1	0	0	0

	F
q_1	600
q_2	500
q_3	700
q_4	1000
q_5	200
q_6	400

UID	a_1	a_2	a_3	a_4	a_5	a_6
01	$v_{1,1}$	$v_{1,2}$	$v_{1,3}$	$v_{1,4}$	$v_{1,5}$	
02	$v_{2,1}$	$v_{2,2}$	$v_{2,3}$	$v_{2,4}$		$v_{2,6}$
03	$v_{3,1}$	$v_{3,2}$	$v_{3,3}$	$v_{3,4}$		
04	$v_{4,1}$	$v_{4,2}$	$v_{4,3}$	$v_{4,4}$		
05	$v_{5,1}$	$v_{5,2}$	$v_{5,3}$	$v_{5,4}$		
06	$v_{6,1}$	$v_{6,2}$	$v_{6,3}$	$v_{6,4}$		
07	$v_{7,1}$	$v_{7,2}$	$v_{7,3}$	$v_{7,4}$	$v_{7,5}$	
08	$v_{8,1}$	$v_{8,2}$	$v_{8,3}$	$v_{8,4}$	$v_{8,5}$	$v_{8,6}$
09	$v_{9,1}$	$v_{9,2}$	$v_{9,3}$			
10	$v_{10,1}$	$v_{10,2}$	$v_{10,3}$			

(a) Attribute Usage Matrix (AUM) (b) Query frequency (F) (c) Horizontal table T

Figure 4.6: Informations spécifiques à la charge de travail et aux données de T

Étant donné les informations spécifiques à la charge de travail et aux données de la table horizontale T , comme le montre la figure 4.6, nous appliquons ici HADF pour générer deux configurations candidates différentes correspondant aux différents paramètres des paramètres α , β , θ et λ .

Configuration 1 : Cette configuration peut également être obtenue en exécutant HADF avec $\beta = 0$, $\lambda = 0$ et en utilisant des valeurs arbitraires pour α et θ , par exemple, $\alpha = 0$ et $\theta = 0$.

La phase de regroupement produit les deux groupes suivants :

- $C_{1,1} = \{UID, a_1, a_2\}$
- $C_{1,2} = \{UID, a_3, a_4, a_5, a_6\}$.

Ensuite, la phase de fusion-sélection va fusionner les deux groupes ci-dessus en un seul et suggérer d'utiliser un stockage en lignes pour cela :

- $C_{1,1.2} = \{UID, a_1, a_2, a_3, a_4, a_5, a_6\} \Rightarrow$ stockage en lignes (row store).

UID	a_1	a_2	a_3	a_4	a_5	a_6
01	$v_{1,1}$	$v_{1,2}$	$v_{1,3}$	$v_{1,4}$	$v_{1,5}$	
02	$v_{2,1}$	$v_{2,2}$	$v_{2,3}$	$v_{2,4}$		$v_{2,6}$
03	$v_{3,1}$	$v_{3,2}$	$v_{3,3}$	$v_{3,4}$		
04	$v_{4,1}$	$v_{4,2}$	$v_{4,3}$	$v_{4,4}$		
05	$v_{5,1}$	$v_{5,2}$	$v_{5,3}$	$v_{5,4}$		
06	$v_{6,1}$	$v_{6,2}$	$v_{6,3}$	$v_{6,4}$		
07	$v_{7,1}$	$v_{7,2}$	$v_{7,3}$	$v_{7,4}$	$v_{7,5}$	
08	$v_{8,1}$	$v_{8,2}$	$v_{8,3}$	$v_{8,4}$	$v_{8,5}$	$v_{8,6}$
09	$v_{9,1}$	$v_{9,2}$	$v_{9,3}$			
10	$v_{10,1}$	$v_{10,2}$	$v_{10,3}$			

Table T_1 (row store)

Figure 4.7: Table créée pour la configuration 1

La figure 4.7 illustre le groupe ci-dessus stocké dans une table unique orientée ligne T_1 . Aucune jointure n'est nécessaire durant l'exécution de la charge de travail, cependant des attributs non pertinents sont manipulés.

Configuration 2 : La phase de regroupement est effectuée avec les paramètres suivants : $\alpha = 0.5$ et $\beta = 0.4$. Ainsi, cette phase prendra en compte l'impact combiné des informations spécifiques à la charge de travail et aux données pour aboutir à deux groupes :

- $C_{2,1} = \{UID, a_1, a_2, a_3\}$
- $C_{2,2} = \{UID, a_4, a_5, a_6\}$.

Ensuite, la phase de fusion-sélection est effectuée en utilisant les réglages suivants : $\theta = 0.5$ et $\lambda = 0.6$. Il suggère la configuration de stockage de données suivante :

- $C_{2,1} = \{UID, a_1, a_2, a_3\} \Rightarrow$ stockage en colonnes (column store);
- $C_{2,2} = \{UID, a_4, a_5, a_6\} \Rightarrow$ stockage en lignes (row store) .

La figure 4.8 présente deux tables T_1 et T_2 qui sont utilisées pour stocker les deux groupes ci-dessus dans différentes dispositions de stockage de données. La configuration 2 permet de réduire le nombre de valeurs nulles et le nombre d'opérations de jointure supplémentaires en même temps.

UID	a_1	a_2	a_3
01	V1,1	V1,2	V1,3
02	V2,1	V2,2	V2,3
03	V3,1	V3,2	V3,3
04	V4,1	V4,2	V4,3
05	V5,1	V5,2	V5,3
06	V6,1	V6,2	V6,3
07	V7,1	V7,2	V7,3
08	V8,1	V8,2	V8,3
09	V9,1	V9,2	V9,3
10	V10,1	V10,2	V10,3

(a) Table T_1 (column store)

UID	a_4	a_5	a_6
01	V1,4	V1,5	
02	V2,4		V2,6
03	V3,4		
04	V4,4		
05	V5,4		
06	V6,4		
07	V7,4	V7,5	
08	V8,4	V8,5	V8,6

(b) Table T_2 (row store)**Figure 4.8:** Deux tables créées pour la configuration 2

En conclusion, HADF peut fournir un bon support pour la conception de données DICOM qui peuvent prendre en compte l'impact combiné des informations spécifiques aux charges de travail et aux données sur la qualité des configurations de stockage de données suggérées.

Chapitre 5 Traitement de Requête pour HYTORMO

Dans ce chapitre, nous présentons une stratégie de traitement de requêtes adaptée et efficace pour HYTORMO. Nous introduisons d'abord un plan d'exécution de requête qui peut prendre en compte une utilisation mixte des tables en lignes et en colonnes (row table and column table). Des heuristiques sont introduites pour sélectionner un type de jointure approprié (c'est-à-dire, jointure interne ou jointure externe gauche) pour une jointure particulière et pour réduire le nombre de jointures externes gauches dans une séquence de jointures. Ensuite, nous présentons comment intégrer un IBF dans le traitement des requêtes afin de réduire les entrées/sorties réseau. Ensuite, une analyse coût-bénéfice de cette intégration est fournie. Enfin, nous décrivons une approche IBF alternative, appelée IBF incrémentale.

5.5 Stratégie de traitement des requêtes

5.2.5 Plan d'exécution de requête

La stratégie de traitement des requêtes peut être décrite comme suit : Une requête utilisateur sera décomposée en sous-requêtes pour pouvoir accéder aux tables en lignes et en colonnes nécessaires. HYTORMO utilise un *plan d'arbre séquentiel gauche profond* (*left-deep sequential tree plan*) pour joindre ensemble des tables pertinentes. Il est nécessaire d'évaluer certaines opérations de jointure entre ces sous-requêtes en tant que jointures externe gauche pour éviter la perte de données causée par les n-uplets rejetés par les jointures internes. HYTORMO déterminera automatiquement une jointure en tant que jointure interne ou externe gauche.

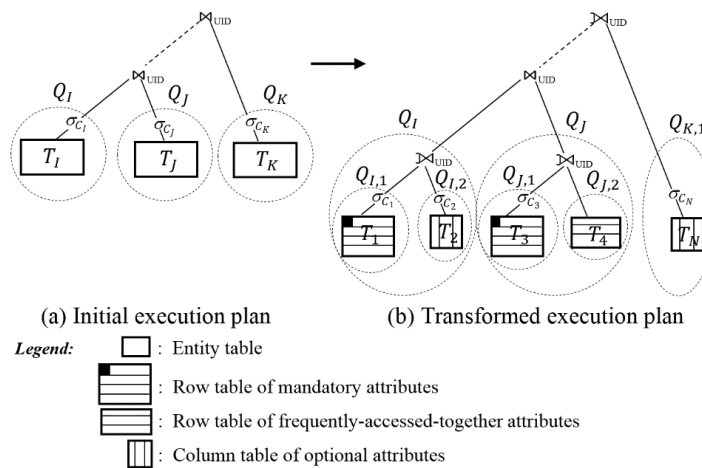


Figure 5.1: Transformation du plan d'exécution pour la requête Q

Dans la requête utilisateur, les types de jointures entre ces tables d'entités sont explicitement identifiés par les utilisateurs. Par exemple, dans la figure 5.1 (a), la requête utilisateur $Q = Q_I \bowtie_{\text{UID}} Q_J \bowtie_{\text{UID}} Q_K$ montre que les jointures internes sont utilisées pour joindre les tables d'entités T_I , T_J et T_K ensemble. Cependant, dans la figure 5.1 (b), certaines opérations de jointure entre les sous-tables de ces tables d'entités doivent être évaluées en tant que jointures externe gauche.

5.2.6 Détermination des jonctions gauches-extérieures

Nous proposons des heuristiques pour déterminer quand une jointure externe gauche est utilisée :

Règle 1 : *Dans une jointure entre deux sous-tables de la même table d'entités, si la table de gauche est une table en lignes d'attributs obligatoires alors que la table de droite est soit une table en colonnes d'attributs optionnels, soit une table en lignes attributs fréquemment accédés ensemble, cette jointure doit être évaluée en tant que jointure externe gauche.*

Par exemple, dans la figure 5.1 (b), les deux sous-requêtes $Q_{I,1} \bowtie_{\text{UID}} Q_{I,2}$ et $Q_{J,1} \bowtie_{\text{UID}} Q_{J,2}$ sont évaluées comme jointures externes gauches. C'est parce que $Q_{I,1}$ et $Q_{J,1}$, respectivement, accèdent à deux tables en lignes d'attributs obligatoires, T_1 et T_3 , tandis que $Q_{I,2}$ et $Q_{J,2}$ accèdent à une table en colonnes d'attributs optionnels T_2 et à une table en lignes d'attributs fréquemment accédés ensemble T_4 , respectivement.

Règle 2 : *Dans une opération de jointure entre deux tables d'entités, si la table de droite a été remplacée par une sous-table qui est soit une table en lignes d'accès d'attributs fréquemment accédés ensemble ou une table en colonnes d'attributs optionnels (parce que la requête utilisateur utilise uniquement les attributs de cette sous-table) et cette sous-table n'est pas le seul enfant de sa table parente, cette opération de jointure doit être évaluée en tant que jointure externe gauche.*

Par exemple, dans la requête $Q = Q_I \bowtie_{\text{UID}} Q_J \bowtie_{\text{UID}} Q_K$, présentée dans la figure 5.1 (a), nous nous intéressons à l'opération de jointure liée à Q_K , c'est-à-dire, $(\dots) \bowtie_{\text{UID}} Q_K$. La requête Q_K a été réécrite en $Q_{K,1}$ qui accède à la table en colonnes des attributs optionnels T_N . Supposons que T_N n'est pas le seul enfant de sa table parente, T_K . Ainsi, la jointure ci-dessus est réécrite en une jointure externe gauche, comme présenté dans la figure 5.1 (b).

Dans notre étude, seuls les cas de jointures gauches-externes sont concernés.

5.2.7 Réduire le nombre de jointures externes gauche

Afin d'améliorer les performances de la requête, le nombre de jointures externes gauches doit être réduit le plus possible. Nous présentons la règle 3 ci-dessous.

Règle 3 : *Étant donné une jointure externe gauche $T_1 \bowtie_{\text{UID}} T_2$, si la table de droite T_2 contient une contrainte non nulle sur ses attributs, cette jointure externe gauche doit être réécrite dans une jointure interne afin d'améliorer la requête performance.*

Par exemple, comme le montre la figure 5.2, nous appliquons la règle 3 pour transformer l'arbre du plan d'exécution de la figure 5.2 (b) en celui de la figure 5.2 (c) : Premièrement, nous vérifions s'il existe des contraintes non nulles sur la table de droite des jointures externes gauches. Ici, nous supposons que σ_{C_2} et σ_{C_N} sont des contraintes non nulles sur les attributs des tables T_2 et T_N , respectivement. Ainsi, deux jointures externes gauches $Q_{I,1} \bowtie_{\text{UID}} Q_{I,2}$ et $(Q_I \bowtie_{\text{UID}} Q_J) \bowtie_{\text{UID}} Q_{K,1}$ sont réécrites comme deux jointures internes $Q_{I,1} \bowtie_{\text{UID}} Q_{I,2}$ et $(Q_I \bowtie_{\text{UID}} Q_J) \bowtie_{\text{UID}} Q_{K,1}$, respectivement.

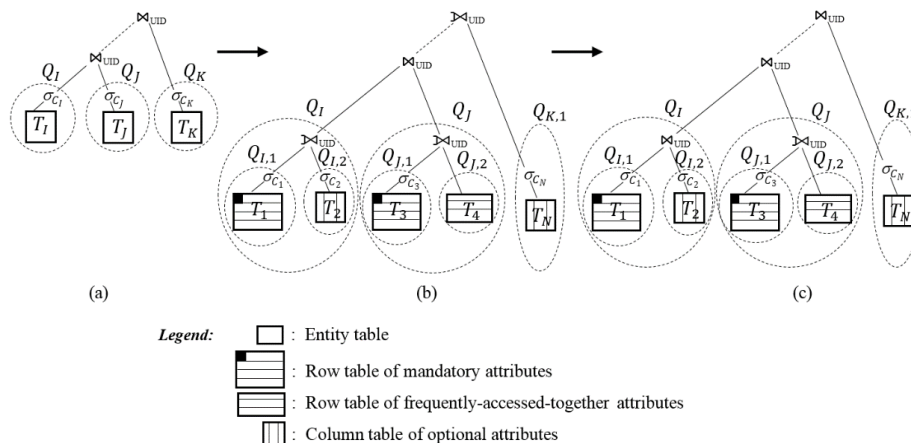


Figure 5.2: Transformation du plan d'exécution après application de la règle 3

5.6 Intégration de filtres de Bloom et rapport coût-bénéfice

Dans cette section, nous présentons comment intégrer un IBF dans le traitement des requêtes et son rapport coût-bénéfice.

5.3.4 Intersection des filtres de Bloom

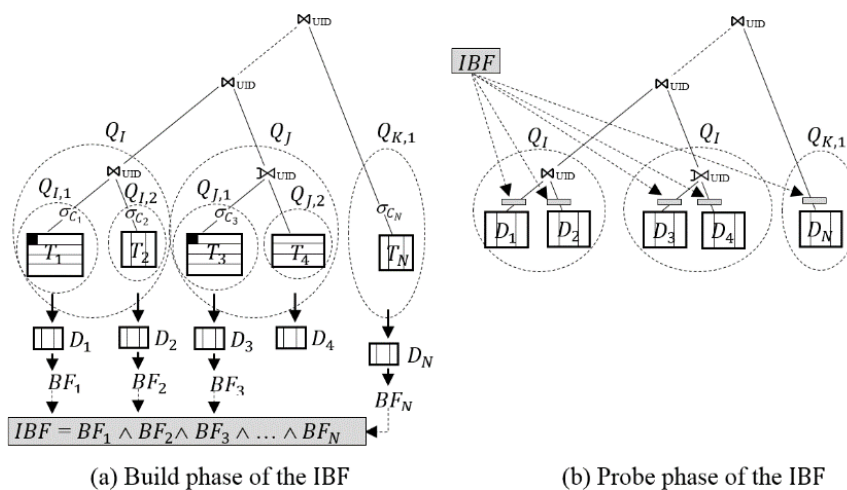


Figure 5.3: Plan d'exécution de requête avec l'IBF

Afin d'éviter la perte de généralité, nous considérons l'intégration d'un IBF dans le traitement de la requête pour la forme générale d'une requête Q supportée par HYTORMO. Nous supposons que la requête Q peut être décomposée en sous-requêtes Q_I , Q_J et Q_K , chacune pouvant être décomposée en sous-requêtes plus petites pour accéder respectivement aux tables en lignes et en colonnes T_1, T_2, \dots, T_N . Après les heuristiques (les règles 1 à 3) sont appliquées pour sélectionner les types de jointures appropriés et pour réduire le nombre de jointures externes dans la séquence de jointure, nous pouvons construire (build) et consulter (probe) un IBF commun sur l'attribut UID des tables d'entrée, comme l'illustre la figure 5.3.

5.3.5 Analyse coût-bénéfice

Comme il existe de nombreux cas dans lesquels l'IBF peut être appliqué, notre étude se concentre sur les cas où l'IBF est utilisé pour une séquence de jointures séquentielles de N tables jointes. Nous supposons que l'IBF est créé en croisant les BF sur toutes les tables d'entrée D_1, D_2, \dots, D_N . En outre, nous supposons que toutes les opérations de jointure externe gauche dans la séquence de jointures ont été transformées avec succès en opérations de jointure internes correspondantes. De plus, nous supposons également que $|D_i| \leq |D_{i+1}|$, où $i \in [1, N-1]$, de sorte que la séquence de jointure peut être exprimée comme : $Q = (((D_1 \bowtie_{\text{UID}} D_2) \bowtie_{\text{UID}} \dots) \bowtie_{\text{UID}} D_{N-1}) \bowtie_{\text{UID}} D_N$.

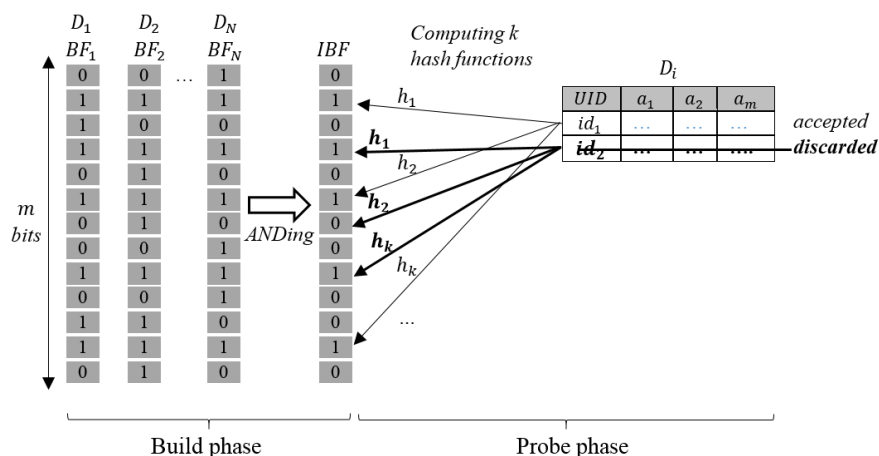


Figure 5.4: Phases de l'IBF

La figure 5.4 illustre les phases de construction et de consultation de l'IBF. Dans la phase de construction, l'IBF est construit en appliquant un AND à tous les composant BF_i créés à partir des attributs de jointure, par exemple, UID, des tables d'entrée D_1, D_2, \dots, D_N . Après cela, dans la phase de consultation, l'IBF est appliqué pour filtrer les n-uplets non pertinents hors de ces tables d'entrée. En particulier, les étapes suivantes sont effectuées : vérification de l'appartenance d'une valeur v d'UID dans chaque table d'entrée D_i ; si pour la valeur $v = id_i$ de UID toutes les fonctions de hachage $h_1(v), h_2(v), \dots, h_k(v)$ retourne vrai (= 1), le n-uplet correspondant est accepté ; sinon, il est ignoré de D_i . Par exemple, le tuple avec la valeur $v = id_1$ de l'UID est accepté (conservé), tandis que le n-uplet avec la valeur $v = id_2$ de l'UID est rejeté.

Nous supposons que IBF est calculé à partir des entrées BF_i créés à partir de N tables d'entrée D_i , où $i \in [1, N]$, puis il est appliqué pour filtrer chaque table d'entrée D_i pour produire une table d'entrée filtrée $D_{i(\text{filtered})}$. Le nombre de n-uplets de chaque table d'entrée filtrée $D_{i(\text{filtered})}$ peut être calculé par la formule (5.2.1):

$$|D_{i(\text{filtered})}| = |D_i| \times \rho_{\text{IBF}, D_i}, \tag{5.2.1}$$

où :

- $|D_i|$ nombre de n-uplets dans la i -ième table d'entrée D_i ;
- ρ_{IBF} : sélectivité de l'IBF.

Par ailleurs, nous pouvons réécrire la formule (5.2.1) comme suit :

$$|D_{i(\text{filtered})}| = |D_i| \times \prod_{j=1}^N \left[\rho_{D_j, D_i} + (1 - \rho_{D_j, D_i}) \times P_{BF_j} \right], \quad (5.2.2)$$

où :

- $|D_i|$: nombre de n-uplets dans la i -ième table d'entrée D_i ;
- ρ_{D_j, D_i} : sélectivité de la table D_j sur la table D_i dans la jointure $D_i \bowtie_{UID} D_j$;
- P_{BF_j} : probabilité d'erreur du filtre de Bloom BF_j qui est construit sur la table D_j ;
- $(1 - \rho_{D_j, D_i}) \times P_{BF_j}$: fraction de n-uplets de la table de consultation D_i qui ne sont pas rejetés par le BF_j et ne se correspondent à aucun n-uplet de la table D_j dans la construction.

Pour réduire le coût d'entrée/sortie du réseau et le coût d'entrée/sortie disque, nous devons appliquer l'IBF si cela est avantageux. La formule (5.2.2) montre que pour obtenir $|D_{i(\text{filtered})}| \ll |D_i|$, la valeur de $\prod_{j=1}^N \left[\rho_{D_j, D_i} + (1 - \rho_{D_j, D_i}) \times P_{BF_j} \right]$ doit être faible. Cela signifie que la séquence de jointure doit contenir une ou plusieurs jointures entre deux tables d'entrée $D_i \bowtie_{UID} D_j$ dans lesquelles la sélectivité ρ_{D_j, D_i} de la table D_j sur D_i et la probabilité d'erreur P_{BF_j} de BF_j sont faibles ; sinon, l'IBF peut ne pas être bénéfique au traitement de la requête.

5.3.6 Intersection incrémentale de Filtres de Bloom

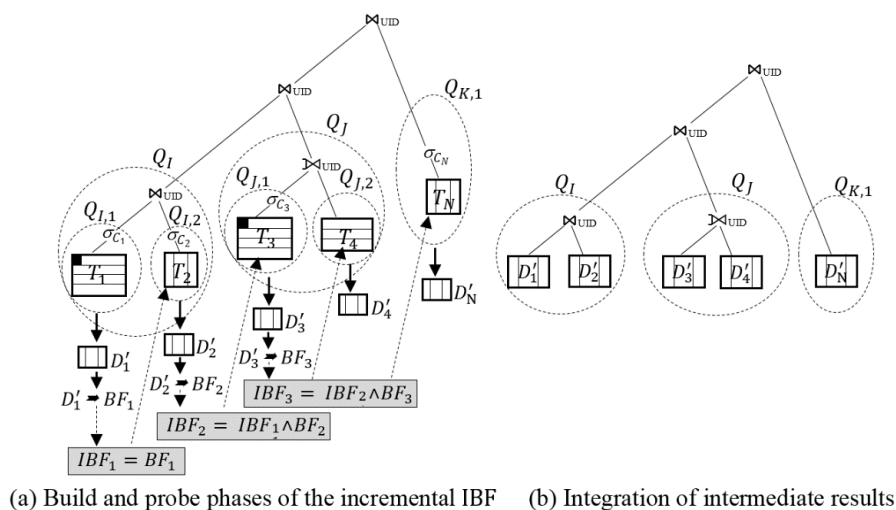


Figure 5.5: Plan d'exécution avec l'IBF incrémental

Pour réduire le coût d'entrée/sortie disque nécessaire pour construire et consulter l'IBF (causé par un grand nombre d'opérations de lecture et d'écriture sur les tables de résultats intermédiaires et sur les tables de résultats intermédiaires filtrés), nous pouvons construire et consulter l'IBF progressivement durant l'exécution des opérations de jointure dans le plan d'exécution. Nous appelons le nouvel IBF proposé *IBF incrémental*. La figure 5.5 (a) illustre l'intégration des phases de construction et de consultation de l'IBF incrémental dans le plan d'exécution de la requête

$Q = Q_I \bowtie_{UID} Q_J \bowtie_{UID} Q_K$ alors que la figure 5.5 (b) présente l'intégration des résultats intermédiaires des sous-requêtes.

Chapitre 6 Évaluation des Performances

Ce chapitre présente les résultats de l'évaluation et les leçons tirées de l'application de HYTORMO et des méthodes proposées.

6.6 Environnement expérimental

Centre de traitement de données de Spark

Nous avons utilisé Hadoop 2.7.1, Hive 1.2.1 et Spark 1.6.0 pour installer un centre de traitement de données. Ce centre est constitué de 9 nœuds différents : $1 \times$ *Nœud maître* et $8 \times$ *Nœuds esclaves*. Nous utilisons la configuration standard avec une modification : nous changeons le facteur de réplication de HDFS de 3 à 2 afin d'économiser de l'espace de stockage. Nous implémentons le plan d'exécution pour les requêtes utilisant un programme Spark.

Jeux de données

Nous avons utilisé les jeux de données DICOM réels : CTColonography, Dclunie, Idoimaging, LungCancer, MIDAS et CIAD. Leurs statistiques sont décrites dans le tableau 6.1. À partir de ces ensembles de données, nous créons deux ensembles de données mixtes : (1) MDB1; et (2) MDB2.

Table 6.1: Les jeux de données DICOM mixtes utilisés dans les expériences

N°	Jeux de données	Le nombre de fichiers DICOM	Le nombre d'attributs extraits	Taille des métadonnées extraites	Taille totale des fichiers	Jeux de données mixtes	
1	CTColonography	98,737	86	7.76 GB	48.6 GB	MDB1	MDB2
2	Dclunie	541	86	86.0 MB	45.7 GB		
3	Idoimaging	1,111	86	53.9 MB	369 MB		
4	LungCancer	174,316	86	1.17 GB	76.0 GB		
5	MIDAS	2,454	86	63.4 MB	620 MB		
6	CIAD	3,763,894	86	61.5 GB	1.61 TB		

Les métadonnées et les données de pixels ont été extraites des fichiers DICOM en utilisant la bibliothèque dcm4che-2.0.29. Les expériences de ce chapitre concernent uniquement quatre tables d'entités *Patient*, *Study*, *GeneralInfoTable* et *SequenceAttribute* comme indiqué ci-dessous :

- **Patient**(UID, PatientName, PatientID, PatientBirthDate, PatientSex, EthnicGroup, IssuerOfPatientID, PatientBirthTime, PatientInsurancePlanCodeSequence, PatientPrimaryLanguageCodeSequence, PatientPrimaryLanguageModifierCodeSequence, OtherPatientIDs, OtherPatientNames, PatientBirthName, PatientTelephoneNumbers, SmokingStatus, PregnancyStatus, LastMenstrualDate, PatientReligiousPreference, PatientComments, PatientAddress, PatientMotherBirthName, InsurancePlanIdentification)
- **Study**(UID, StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, AccessionNumber, StudyDescription, PatientAge, PatientWeight, PatientSize, Occupation, AdditionalPatientHistory, MedicalRecordLocator, MedicalAlerts)
- **GeneralInfoTable**(UID, GeneralTags, GeneralVRs, GeneralNames, GeneralValues)
- **SequenceAttributes**(UID, SequenceTags, SequenceVRs, SequenceNames, SequenceValues)

Les charges de travail

Nous utilisons quatre charges de travail différentes : La charge de travail W1 contient principalement des requêtes OLAP en utilisant la table d'entités *GeneralInfoTable*. La charge de travail W2 correspond principalement à des requêtes OLTP utilisant la table d'entités *SequenceAttributes*. La charge de travail W3 inclut à la fois les requêtes OLAP et OLTP à l'aide de la table d'entités *Patient*. La charge de travail W4 n'est pas seulement une charge de travail mixte (comme W3) mais elle inclut également des requêtes de jointure de tables multiples sur des tables d'entités.

6.7 Exécution des expériences

Expérience 1 : Évaluation de l'efficacité de HYTORMO et de l'utilité de HADF

Cette expérience vise à évaluer les avantages du modèle de stockage hybride et HADF. Le jeu de données MDB1 et les charges de travail W1 à W4 sont utilisés.

Nous exécutons HADF sur les charges de travail W1 - W4 l'une après l'autre pour choisir une bonne configuration de stockage de données en termes de demande d'espace de stockage et de temps d'exécution de charge de travail pour chaque table d'entités. Enfin, nous créons une configuration G* qui combine la bonne configuration de chaque table d'entités. Le tableau 6.2 présente la configuration G*.

Table 6.2: Configuration G*

N°	Table des entités	Configuration de stockage de données
1	Patient	PatientP1P2P3P4(UID, PatientName, PatientID, PatientBirthDate, PatientSex) => <i>stockage en lignes</i>
		PatientP13P14P15P19P21(UID, PatientBirthName, PatienttelePhoneNumbers, SmokingStatus, PatientComments, PatientMotherBirthName) => <i>stockage en lignes</i>
		PatientP5(UID, EthnicGroup) => <i>stockage en lignes</i>
		PatientallP7P8P16P17P18(UID, PatientBirthTime, PatientInsurancePlanCodeSequence, PregnancyStatus, LastMenstrualDate, PatientReligiousPreference) => <i>stockage en lignes</i>
		PatientP6P12P22(UID, IssuerOfPatientID, OtherPatientNames, InsurancePlanIdentification) => <i>stockage en lignes</i>
		PatientP10P11P20(SOPInstanceUID, PatientPrimaryLanguageModifier-CodeSequence, OtherPatientIDs, PatientAddress) => <i>stockage en lignes</i>
		PatientP9(UID, PatientPrimaryLanguageCodeSequence) => <i>stockage en lignes</i>
2	Study	Study(UID, StudyInstanceUID, StudyDate, StudyTime, ReferringPhysicianName, StudyID, AccessionNumber, StudyDescription, PatientAge, PatientWeight, PatientSize, Occupation, AdditionalPatientHistory, MedicalRecordLocator, MedicalAlerts) => <i>stockage en colonnes</i>
3	GeneralInfoTable	GeneralInfoTable(UID, GeneralTags, GeneralVRs, GeneralNames, GeneralValues) => <i>stockage en colonnes</i>
4	SequenceAttributes	SequenceAttributes(UID, SequenceTags, SequenceVRs, SequenceNames, SequenceValues) => <i>stockage en lignes</i>

Expérience 2 : Évaluation de HYTORMO et de HADF à l'aide de davantage de jointures de données et de tables multiples

Comme l'expérience 1, cette expérience vise à évaluer les avantages d'HYTORMO et de HADF. Cependant, il utilise plus de données et des requêtes de jointure à plusieurs tables. À ces fins, cette expérience compare l'efficacité de trois configurations : (1) G* qui est une bonne configuration obtenue à partir de l'expérience 1; (2) G1 qui stocke toutes les tables d'entités dans un stockage en lignes; et (3) G2 qui stocke toutes les tables d'entités dans un stockage en colonnes. La charge de travail W4 et deux jeux de données MDB1 et MDB2 sont utilisés.

Les tableaux 6.3 et 6.4, représentent respectivement le temps d'exécution moyen (5 exécutions) de la charge de travail W4 en utilisant les trois configurations ci-dessus par rapport à deux cas différents: (1) utiliser MDB1 et (2) MDB2. Dans les deux cas, la configuration G* nécessite la plus petite demande d'espace de stockage car de nombreuses valeurs nulles sont supprimées de la table d'entités *Patient*. G* offre également le temps d'exécution de la charge de travail le plus rapide.

Table 6.3: Temps d'exécution de la charge de travail W4 en utilisant MDB1

Conf.	Configuration de stockage de données	Temps d'exéc (s)
G*	Bonne configuration de stockage de données générée par HADF, créée en combinant toutes les bonnes configurations des tables d'entités.	35,940
G ₁	Toutes les tables d'entités sont stockées dans le stockage en lignes.	37,860
G ₂	Toutes les tables d'entités sont stockées dans le stockage en colonnes.	36,960

Table 6.4: Temps d'exécution de la charge de travail W4 en utilisant MDB2

Conf.	Configuration de stockage de données	Temps d'exéc (s)
G*	Bonne configuration de stockage de données générée par HADF, créée en combinant toutes les bonnes configurations des tables d'entités.	118,940
G ₁	All entity tables are stored in row stores.	161,040
G ₂	All entity tables are stored in column stores.	120,120

Expérience 3 : Comparaison entre HADF et HoVer

Cette expérience vise à évaluer plus avant le bénéfice de l'utilisation combinée d'informations spécifiques à la charge de travail et aux données dans HADF. Nous comparons HADF avec l'approche HoVer proposée par Bin Cui et al. L'approche HoVer est un algorithme de clustering identique à la phase de clustering de HADF. Cependant, l'approche HoVer est basée uniquement sur de similarité de densité d'attributs (au lieu de similarité d'accès d'attribut et de similarité de densité d'attribut); de plus, l'approche HoVer utilise uniquement le stockage en lignes (au lieu de stockage en lignes et en colonnes). L'approche HoVer génère une configuration de stockage de données correspondant à une valeur donnée du seuil de regroupement β .

Dans cette expérience, nous effectuons deux charges de travail W1 et W2, séparément, sur l'ensemble de données MDB2. Le résultat de l'expérience montre que : (1) Dans le cas d'une charge de travail OLAP (par exemple, W1), HADF peut fournir une meilleure configuration de stockage de données que l'approche HoVer. En effet, il est possible de suggérer de stocker la donnée utilisée pour une charge de travail OLAP dans un système colonnes. (2) Dans le cas d'une charge de travail OLTP (par exemple,

W2), HADF est capable de fournir une configuration aussi bonne que celles générées par l'approche HoVer. Un système lignes est utilisé pour stocker les données utilisées pour la charge de travail OTLP.

Expérience 4 : Evaluer l'efficacité de l'IBF

Cette expérience vise à évaluer l'efficacité de la stratégie de traitement des requêtes avec l'intégration d'un IBF. Nous utilisons l'ensemble de données MDB2 et la configuration G* pour stocker les tables d'entités. La requête suivante de jointure de plusieurs tables avec et sans utiliser un IBF est exécutée :

```
SELECT Patient.UID, Patient.PatientID, Patient.PatientName, Patient.PatientBirthDate,
Patient.PatientSex, Patient.EthnicGroup, Patient.SmokingStatus, Study.PatientAge,
Study.PatientWeight, Study.PatientSize, GeneralInfoTable.GeneralNames,
GeneralInfoTable.GeneralValues, SequenceAttributes.UID,
SequenceAttributes.SequenceTags, SequenceAttributes.SequenceVRs,
SequenceAttributes.SequenceNames, SequenceAttributes.SequenceValues
FROM Patient, Study, GeneralInfoTable, SequenceAttributes
WHERE Patient.UID = Study.UID AND Patient.UID = GeneralInfoTable.UID
AND Patient.UID = SequenceAttributes.UID AND Patient.PatientSex = 'M'
AND Patient.SmokingStatus = 'NO' AND Study.PatientAge >= 60
AND SequenceAttributes.SequenceNames LIKE '%X-Ray%'
```

Cependant, pour observer l'impact de l'IBF sur une gamme de situations, nous allons modifier la sélectivité (Sél.) des prédicats de la requête ci-dessus. Dans le tableau 6.5, nous fournissons six ensembles différents de prédicats (Ens. Pré.).

Table 6.5: Ensembles de prédicats sur les attributs des tables d'entrée

Ens. Pré.	PatientP1P2P3P4		PatientP13P14P15P19P21		Study		SequenceAttributes	
	Sél.	Prédicat	Sél.	Prédicat	Sél.	Prédicat	Sél.	Prédicat
1	1	Aucun prédicat	1	Aucun prédicat	1	Aucun prédicat	1	Aucun prédicat
2	1	Aucun prédicat	1	Aucun prédicat	0.6327	PatientAge >= 10	1	Aucun prédicat
3	0.4764	Patientsex = 'M'	1	Aucun prédicat	0.6327	PatientAge >= 10	1	Aucun prédicat
4	0.4764	Patientsex = 'M'	1	Aucun prédicat	0.2462	PatientAge >= 60	1	Aucun prédicat
5	0.4764	Patientsex = 'M'	0.0017	smokingstatus='NO'	0.2462	PatientAge >= 60	1	Aucun prédicat
6	0.4764	Patientsex = 'M'	0.0017	smokingstatus='NO'	0.0061	PatientAge >= 90	0.0019	SequenceNames LIKE '%X-Ray%';

Dans le tableau 6.6, nous présentons une comparaison du temps d'exécution de la requête avec et sans IBF. Ce résultat montre que les performances de la requête sont significativement améliorées pour tous les ensembles de prédicats. Le temps d'exécution de la requête avec est réduit de 10 à 38% par rapport au temps sans IBF.

Table 6.6: Comparaison du temps d'exécution avec l'utilisation de l'IBF

Ens. Pré.	Temps d'exécution lorsque vous n'utilisez pas l'IBF		Temps d'exécution lors de l'utilisation de l'IBF		Rapport de temps réduit (%)
	Moyenne (s)	Std. dev.	Moyenne (s)	Std. dev.	
1	1264.80	389.20	1007.20	176.89	20%
2	1209.20	234.63	748.00	92.29	38%
3	1068.40	438.10	962.80	197.97	10%
4	1122.80	330.83	908.80	202.48	19%
5	1215.80	407.01	964.80	189.23	21%
6	1452.40	421.58	930.40	127.05	36%

Le tableau 6.7 représente une comparaison du temps d'exécution de la requête entre l'utilisation de l'IBF et de l'IBF incrémental. Il montre que, dans tous les cas d'ensembles de prédicats, le temps d'exécution de la requête est réduit lors de l'utilisation d'un IBF incrémental.

Table 6.7: Comparaison entre l'IBF et l'IBF incrémental

Ens. Pré.	Temps d'exécution lors de l'utilisation de l'IBF		Temps d'exécution lors de l'utilisation d'un IBF incrémental		Rapport de temps réduit (%)
	Moyenne (s)	Std. dev.	Moyenne (s)	Std. dev.	
1	1007.20	176.89	862.60	242.25	14%
2	748.00	92.29	925.40	198.97	-23%
3	962.80	197.97	995.40	167.60	-3%
4	908.80	202.48	901.80	216.55	1%
5	964.80	189.23	779.00	98.02	19%
6	930.40	127.05	729.80	202.91	22%

En tant que tel, pour cette requête, l'IBF incrémental donne de meilleures performances d'interrogation que l'IBF pour la majorité des ensembles de prédicats. Plus particulièrement, pour les premiers et derniers des trois ensembles de prédicats, les rapports temporels réduits sont respectivement de 14%, 1%, 19% et 22% lorsque l'IBF incrémental est appliqué. Cependant, pour les deuxièmes et troisièmes ensembles de prédicats, l'IBF surpasse l'IBF incrémental. Cela est probablement dû au fait que le coût élevé de la construction et de l'examen de l'IBF a été compensé de manière significative par la quantité de données filtrées.

6.8 Analyse et interprétation

Cette section évalue les résultats des expériences et les hypothèses.

6.4.4 Résultats de l'hypothèse H1

Hypothèse H1 : *Le modèle de données hybride, c'est-à-dire HYTORMO, associé à la stratégie de stockage de données proposée, donne un temps d'exécution de charge de travail plus rapide que l'utilisation d'un stockage en lignes ou d'un stockage en colonnes.*

Les résultats de l'expérience 1 montrent que les stockages en lignes et en colonnes doivent être utilisés pour les données DICOM car chacun d'entre eux a ses propres avantages :

- Un stockage en colonnes permet un traitement des requêtes plus rapide et plus efficace pour les charges de travail OLAP qu'un stockage en lignes. Par exemple, *GeneralInfoTable* est utilisé pour une charge de travail OLAP (W1) et donc suggéré d'être stocké dans un stockage en colonnes.
- Un stockage en lignes offre des performances supérieures aux charges de travail OLTP qu'un stockage en colonnes. Par exemple, *SequenceAttribute* est utilisé pour une charge de travail OLTP (W2) et suggère d'être stocké dans un stockage en lignes.

Les résultats de l'expérience 2 montrent que, dans une charge de travail mixte OLAP et OLTP, une utilisation mixte des stockages en lignes et en colonnes donnera un temps d'exécution de charge de travail plus rapide qu'une seule utilisation d'un stockage en lignes ou en colonnes. Par exemple, la configuration G * qui utilise à la fois les tables en lignes et en colonnes est plus rapide que la configuration G1 (en utilisant des tables en lignes) et G2 (en utilisant des tables en colonnes).

Les résultats ci-dessus indiquent qu'il est avantageux d'utiliser le modèle de stockage de données hybride pour stocker des données DICOM. Par conséquent, l'hypothèse H1 est validée.

6.4.5 Résultats de l'hypothèse H2

Hypothèse H2a : *La prise en compte de l'impact combiné des informations spécifiques à la charge de travail et aux données peut aider HADF à produire de meilleures configurations de stockage de données que d'utiliser uniquement des informations spécifiques aux données ou uniquement des informations spécifiques à la charge de travail.*

Les expériences 1, 2 et 3 montrent ce qui suit : Pour les tables d'entités denses, telles que *GeneralInfoTable* et *SequenceAttribute*, l'utilisation d'informations spécifiques aux données n'a pas aidé à réduire l'espace de stockage. En revanche, l'utilisation d'informations spécifiques à la charge de travail est utile pour améliorer les performances de la charge de travail car elle a une incidence sur le résultat du partitionnement vertical et sur la sélection de dispositions de stockage de données appropriées. Pour les tables larges, telles que *Patient* et *Study*, l'utilisation d'informations spécifiques aux données a un effet important sur le résultat du partitionnement vertical qui contribue à réduire la demande d'espace de stockage.

Par conséquent, l'utilisation combinée d'informations spécifiques à la charge de travail et aux données est utile. A partir de ce résultat, l'hypothèse H2a est validée.

Hypothèse H2b : *HADF est capable de générer une configuration de stockage de données qui peut réduire la demande d'espace de stockage et le temps d'exécution de la charge de travail en même temps.*

Les résultats des expériences 1, 2 et 3 montrent que cet objectif a été atteint. Pour les tables larges très éparpillées, par exemple *Patient*, HADF les décompose en tables partitionnées verticalement à partir desquelles les lignes nulles sont supprimées (c'est-à-dire que l'espace de stockage est réduit). En outre, les entrées/sorties réduites accélèrent l'exécution de la charge de travail. L'hypothèse H2b est validée.

6.4.6 Résultats de l'hypothèse H3

Hypothèse H3 : *La stratégie de traitement de requête avec l'intégration d'un IBF conduit à de meilleures performances que de ne pas utiliser un IBF.*

Les résultats de l'expérience 4 montrent que l'IBF et l'IBF incrémental ont accéléré significativement le traitement des requêtes. Par conséquent, l'hypothèse H3 est validée.

Chapitre 7 Conclusion et Travaux Futurs

Ce chapitre résume et conclut la thèse. Il présente également des recherches futures.

7.3 Résumé et conclusion

Six contributions principales ont émergé de notre travail : Premièrement, nous avons effectué une évaluation des systèmes de gestion de données DICOM existants en mettant l'accent sur les caractéristiques pour traiter les caractéristiques et les charges de travail des données DICOM. Deuxièmement, nous avons fourni une comparaison de des systèmes de gestion de données actuels. Troisièmement, nous avons proposé un modèle de stockage hybride, appelé HYTORMO, associé à une stratégie de stockage de données. Quatrièmement, nous avons proposé un cadre de conception automatisée hybride, appelé HADF. Cinquièmement, nous introduisons une stratégie de traitement des requêtes adaptée et efficace, basée sur HYTORMO. Enfin, nous validons les méthodes proposées.

Les résultats expérimentaux montrent que le modèle de stockage de données hybride offre de meilleures performances de charge de travail que l'utilisation d'un stockage en lignes pur ou d'un stockage en colonnes pur. L'utilisation combinée d'informations spécifiques à la charge de travail et aux données est nécessaire pour générer des configurations de stockage de données pouvant réduire à la fois l'utilisation de l'espace de stockage et le temps d'exécution de la charge de travail. De plus, l'utilisation d'IBF améliore considérablement les performances de la requête.

7.4 Travaux futurs

Il existe des axes de recherche ouverts que nous pouvons étudier et étendre à l'avenir.

Modèle de stockage hybride : Au lieu de simplement utiliser des stockages en lignes et en colonnes, nous prévoyons d'étendre HYTORMO pour prendre en charge plusieurs stockages, y compris les stockages en lignes, les stockages en colonnes, les stockages en valeurs-clés, etc., afin qu'il puisse être utilisés pour de nombreuses applications Big Data.

Cadre heuristique pour la conception automatisée : Nous nous sommes, dans un premier temps, basés sur les expériences et les avis d'experts pour sélectionner des valeurs appropriées pour les paramètres d'entrée de HADF (c'est à dire, β , θ et λ), donc il faudrait développer une méthode pour déterminer ces valeurs. Nous étudierons l'application de techniques d'optimisation qui pourraient donner de meilleurs résultats que notre approche. Deuxièmement, HADF sera étendu pour sélectionner des configurations de stockage de données pour les tables horizontales dont les colonnes ont des largeurs différentes. Troisièmement, l'effet de la compression est également envisagé. Enfin, nous prévoyons de rechercher comment de nouveaux attributs DICOM sont ajoutés au stockage de données existant.

Stratégie de traitement de requête : Nous allons explorer un plan d'exécution de requête avec l'utilisation de jointures internes et de jointures externes, au lieu de

seulement des jointures internes et des jointures externes gauches. En effet, si les jointures externes complètes sont utilisées, les n-uplets résultants d'une requête peuvent être reconstruits en joignant plusieurs tables partitionnées verticalement dans n'importe quel ordre de jointure. En outre, nous envisagerons d'appliquer bushy plans avec jointures n-aires pour augmenter le parallélisme dans le traitement des requêtes.

BF non précalculés et précalculés : Nous pensons qu'il serait utile d'avoir deux types de BF: (1) Les BF non précalculés sont calculés à partir des tables d'entrée lors du traitement de la requête tels qu'utilisé dans notre thèse. (2) Les BF précalculés sont précalculés afin d'éviter des étapes de calcul supplémentaires requises lors du traitement de la requête.