Storing and Querying DICOM Data with HYTORMO

Danh Nguyen-Cong[1(✉)], Laurent d'Orazio[1], Nga Tran[2], and Mohand-Said Hacid[3]

[1] LIMOS Laboratory, UMR 6158 CNRS, Blaise Pascal University Clermont-Ferrand II, 63173 Aubière, France
{nguyenda,dorazio}@isima.fr

[2] HPE Vertica, Cambridge, MA, USA
nga.tran@hpe.com

[3] LIRIS – University of Claude Bernard Lyon 1, 43, boulevard du 11 Novembre 1918, 69622 Villeurbanne, France
mshacid@liris.univ-lyon1.fr

Abstract. In the health care industry, DICOM (Digital Imaging and Communication in Medicine) standard has become very popular for storage and transmission of digital medical images and reports. The ever-increasing size, high velocity and variety of the DICOM data collections make them more and more inefficient to be stored and queried them using a single data storage technique, e.g., a row store or a column store. In this study, we first highlight challenges in DICOM data management. We then describe HYTORMO, a new model to store and query the DICOM data. HYTORMO uses a hybrid data storage strategy that is aimed not only to leverage the advantage of both row and column stores, but also to attempt to keep a trade-off among reducing disk I/O cost, reducing tuple construction cost and reducing storage space. In addition, Bloom filters are applied to reduce network I/O cost during query processing. We prototyped our model on the top of Spark. Our preliminary experiments validate the proposed model in real DICOM datasets and show the effectiveness of our method.

Keywords: DICOM · Medical image data · Hybrid store · Bloom filter

1 Introduction

In the health care industry, the management of ever-increasing volumes of medical image data becomes a real challenge. The development of imaging technologies, the long-term retention of medical data imposed by medical laws and the increase of image resolution are all causing a tremendous grow in data volume. In addition, the different acquisition systems (Philips, Olympus, etc.) to be used, the distinct specialties (gastroenterology, gynaecology, etc.) to be considered as well as preferences of physicians, nurses or other health-care professionals lead to a high variety, even if data follow the widely adopted DICOM (Digital Imaging and Communication in Medicine) standard [1]. The huge volume, high velocity and variety of the medical image data make this

DOI: 10.1007/978-3-319-57741-8_4

domain a concrete example of Big Data [2]. In this paper, we focus on storing and querying medical image data that follow the DICOM standard.

With the widely use of the DICOM standard nowadays, there have been some studies on DICOM data management [3–6]. However, complex characteristics of DICOM data make efficient storing and querying non-trivial tasks. The proposed solutions limited themselves to query types and could have negative impacts on performance and scalability. Most state-of-the-art DICOM data management systems employ traditional row-oriented databases ("row-RDBMS/row stores"). Using a row-RDBMS typically requires a query processor to read the entire database table into memory. This causes a lot of unnecessary disk I/O even when only a few attributes are used. In addition, the current studies have not introduced solutions to reduce a large amount of useless intermediate results created during query processing even if the final result is very small. As a result, in large-scale distributed database systems, running a computing cluster, these useless intermediate results can generate a lot of unnecessary network I/O to exchange data between cluster nodes.

In recent years, some studies have already proposed read-optimized databases to avoid reading unnecessary data from query processing. The read-optimized databases can include either column-oriented databases ("column-RDBMS/column stores"), such as MonetDB [7] and C-Store [8], or hybrid row/column-oriented databases, such as Fractured Mirrors [9], HYRISE [10], and SAP HANA [30]. The advantage of these databases is to reduce disk I/O cost. However, their tuple reconstruction cost is high and thus cannot cope with the high heterogeneity and evolution of DICOM data.

On the other side, cloud-based systems have provided solutions of high performance computing together with reliable and scalable storage to facilitate growth and innovation at lower operational costs. Hadoop [23] has become one of the de facto industry standards in this area. MapReduce has also shown a very good scalability for batch-oriented data processing. However, since they are designed for general-purpose, the major challenge is how to build a specialized system to effectively store and query DICOM data.

In this paper, we introduce a novel model, called HYTORMO, to store and query DICOM data. It provides a novel hybrid data storage strategy using both row and column stores to avoid reading unnecessary data as well as to reduce tuple construction cost and storage space. In addition, Bloom filters [11] are integrated into query processing to reduce intermediate results in join sequences.

Our major contributions can be summarized as follows: (1) We determine the characteristics of DICOM data that cause challenges in data management. (2) We propose a hybrid data storage strategy using both row and column stores to reduce disk I/O cost, tuple reconstruction cost and storage space. (3) We provide a query processing strategy with Bloom filters to reduce network I/O cost. (4) We finally present preliminary experiments with real DICOM data to show the effectiveness of our approaches.

The rest of this paper is organized as follows. Section 2 highlights problem definition. Section 3 describes the architecture of HYTORMO and the details of its components. Section 4 presents preliminary experimental results. Section 5 discusses related works. Finally, we conclude the paper and give an outlook on future works in Sect. 6.

2 Problem Definition

In this Section, we introduce DICOM standard, its challenges in data management, current database techniques, and problem formulation of our study.

2.1 DICOM Standard and Its Challenges

The DICOM standard was initially developed in 1983 by a joint committee of American College of Radiology and the National Electrical Manufacturers Association [1]. After many changes, in 1993 DICOM Version 3.0 was published to be widely used. The primary objective of this standard is to define data layouts and exchange protocols for storage and transmission of digital medical images and reports between medical imaging systems. Here we are mainly interested in data in DICOM files. The structure of a DICOM file is divided into three portions: (i) *a header (to recognize if it is a DICOM file)*, (ii) *metadata (to store information related to the image),* and (iii) *pixel data (to store the actual image pixels).* The metadata contains attributes which encode attributes of real-world entities (Patients, Studies, Series, etc.) related to the image. For instance, information about Patient is stored in attributes such as Name, Identity Number, Date of Birth, and Ethnic Group.

The following characteristics of DICOM data mainly cause challenges in data management: **Heterogeneous Schema.** The number of attributes in a DICOM file is very large, about 3000 attributes. Some of them are mandatory while others are optional. However, the number of attributes that are really used at a time varies dramatically depending on the availability of information acquired through performing a particular examination modality (CT, MRI, etc.) using a certain DICOM device (CT scanner, MRI scanner, etc.). **Evolutive Schema.** For instance, modalities or image acquisition devices are modified or added newly. **Variety.** Images and metadata. **Voluminous Data.** The storage space requirements of image databases are very large (e.g., terabyte) and ever-increasing tremendously. For instance, in France, information and tests results of a patient should be stored for up to 30 years [13].

So far, current solutions have provided limited supports to handle the above-mentioned characteristics. We present current database techniques in next Subsection.

2.2 Row- vs. Column-Oriented Databases

Most traditional databases (Oracle, SQL Server, etc.) are row-oriented databases that employ a row-oriented layout, moving horizontally across the table and storing attributes of each tuple consecutively on disk. This architecture is optimized for write-intensive online transaction processing (OLTP) because it is easily to add a new tuple and to read all attributes from a tuple. Their disadvantage is that if only a few attributes are accessed per query at once, the entire tuple still needs to be read into memory from disk before projecting. This wastes the I/O bandwidth [27]. Therefore row-oriented databases are not efficient in the case of highly heterogeneous data. In contrast, column-oriented databases (MonetDB, C-Store, etc.) are optimized for read-intensive workloads (OLAP). By storing data in columns rather than rows, only necessary attributes are read per a query. This saves I/O bandwidth [28]. However, their tuple reconstruction cost is higher than that cost of row-oriented databases.

2.3 MapReduce vs. Spark

Most of current row- and column-oriented databases have been developed to be used for structured data in relational database systems. They do not scale well and are ineffective to process semi/unstructured data. In contrast, MapReduce is originally developed to process extremely large amounts of semi/un/structured data. It provides a scalability and elasticity solution for Big Data. Unfortunately, batch processing data model of MapReduce is suitable for long running queries [29]. It does not support efficiently for users to execute interactive applications (e.g., ad hoc queries to explore data) because these applications have to share data (between parallel operations) across multiple steps of MapReduce and thus need overhead costs in both data replication and disk I/O. In contrast, Spark is an in-memory cluster computing system which can run on Hadoop [14]. Spark improves upon MapReduce by removing the need to write data to disk between steps. We are justified to use Spark due to its high performance for interactive queries and scalability.

2.4 Problem Formulation

Storing and querying DICOM data have been challenged by the complex characteristics of DICOM data (i.e., huge/ever-growing data size, variety, and heterogeneous/evolutive schema). We transform these problems into optimization problems of query performance and storage space. In this way, our study focuses on reducing I/O costs and storage space. We determine four main technical challenges: disk I/O cost, network I/O cost, tuple reconstruction cost, and storage space. We propose the HYTORMO model for efficient storing and querying DICOM data. The following design rules are considered to build this model:

– Reduce disk I/O, tuple reconstruction cost, and storage space by a hybrid data storage strategy using both row and column stores.
– Transparently rewrite user queries to access to data in row and column stores, without the need for any user intervention.
– Reduce network I/O cost by minimizing the intermediate results during query processing. An application of Bloom filters will help us to achieve this goal.

3 HYTORMO Architecture

HYTORMO is aimed to cope with heterogeneity, evolution, variety and huge volume of DICOM data. Its architecture consists of two components: *Centralized System* and *Distributed Nodes*, as shown in Fig. 1. The query processing is tightly integrated in

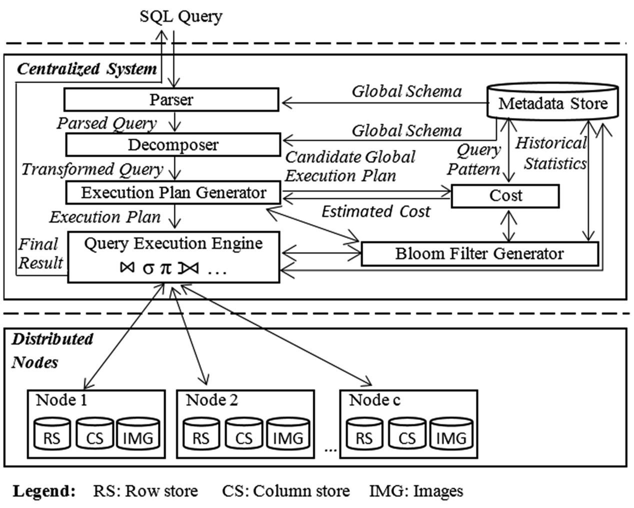


Fig. 1. The overall architecture of HYTORMO.

both Centralized System and Distributed Nodes. DICOM data are distributedly stored in nodes of Distributed Nodes (using both row and column stores).

3.1 Data Storage Strategy

The main goals of data storage strategy are to reduce disk I/O, tuple reconstruction cost, and storage space. *Metadata* and *pixel data* of DICOM files are extracted and stored in Hadoop distributed file system (HDFS) in a manner to achieve these goals. However, in the scope of this paper, we mainly concern about storing the metadata portion as the full-content images can be easily accessed from the metadata via links.

We propose a hybrid data storage strategy using both row and column stores for the metadata. Due to the complexity characteristics of DICOM data, identifying which attributes should be put in a particular store is a challenge work. We present a novel vertical data partitioning schema that is based on attribute classification.

First, we create entities and use an entity-relationship (ER) model to represent the logical relationships between the entities such as Patient, Study, Series, GeneralInfoTable, SequenceAttributes, and Image. Each entity is described by a set of attributes, e.g., *Patient(UID, Patient Name, Patient ID, Patient Birth Date, Patient Sex, Ethnic Group, Issuer Of Patient ID, Patient Birth Time, Patient Insurance Plan Code Sequence, Patient Primary Language Code Sequence, Patient Primary Language Modifier Code Sequence, Other Patient IDs, Other Patient Names, Patient Birth Name, Patient Telephone Numbers, Smoking Status, Pregnancy, Last Menstrual Date, Patient Religious Preference, Patient Comments, Patient Address, Patient Mother Birth Name, Insurance Plan Identification)*, where *UID* is an unique identifier. The ER model then is converted into a relational data model that consists of relations.

Second, attributes of each relation will be classified to fall into one of three categories: *(1) Mandatory:* Attributes are not allowed to get *null* values and are frequently-accessed-together. *(2) Frequently-accessed-together:* Attributes are allowed to get *null* values and frequently accessed together. *(3) Optional/Private/Seldom-accessed:* Attributes are allowed to get *null* values and not frequently accessed together (for short, we sometimes call them *"Optional"*).

Finally, attributes of the same category will be stored in the same table as below:

– Attributes of the first two categories are stored in tables of row stores, called "row tables". The aim is to reduce tuple reconstruction cost. For instance, *Patient Name, Patient ID, Patient Birth Date, Patient Sex,* and *Ethnic Group* are classified as "Mandatory" and stored in a row table. *Pregnancy* and *Last Menstrual Date* are classified as "Frequently-accessed-together" and stored in another row table.
– Attributes of the last category are stored in tables of column stores, called "column tables". The aim is to save the I/O bandwidth if only a few attributes are accessed per query at once. For instance, the rest of attributes of the Patient entity are classified as "Optional" and stored in a column table.

The above vertical data partitioning schema is non-overlapping, that is an attribute only belongs to a table except *UID*. In addition, to reduce storage space, we do not store rows with only *null* values.

3.2 Query Processing Strategy

The goal of query processing strategy can be briefly described as follows: It is given that DICOM data are distributedly stored across row and column stores. Find a query processing strategy to minimize the intermediate results.

Global Description of the Strategy. The actual query processing includes query parsing, query decomposition, query optimization, and query execution. These phases are shown in Fig. 1. The query is parsed by the Parser. It then is decomposed into sub-queries by Decomposer. The query decomposition increases the efficiency of the query by directing sub-queries only to the corresponding row and column tables that contain the required data, leading to a significant reduction of query input size. This also allows HYTORMO to utilize benefits of both row and column stores. The query optimization is performed by Execution Plan Generator that evaluates possible execution plans (i.e., different join strategies for combing results of sub-queries) and chooses the one with minimum cost. Since a given query could have a large number of 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 Steinbrunn et al. [15]. In this plan, a join that yields a smaller intermediate result will be computed first. Metadata Store keeps metadata of database tables (schemas, cardinality of tables, etc.) that can be used during query processing. Finally, Query Execution Engine processes the query execution plan. It sends sub-queries to be executed on Distributed Nodes and retrieves intermediate results. In the end, it returns the final result to the front end. *Distributed Nodes* is mainly responsible for storing DICOM data and executing tasks that are assigned by the Centralized System. Bloom Filter Generator generates Bloom filters to remove irrelevant data out of inputs of joins if their benefits are found.

Query Decomposition. Our study focus on *Select-Project-Join (SPJ)* queries that involve selection conditions followed by equi-joins on surrogate attributes (UIDs) of row and column tables. In order to avoid loss of generality, we use a general form of SQL query to present a user query Q as given below.

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**
$\{T_J.att_x^{Rm}\,\theta\ value_x^{Rm}\}$ **AND** $\{T_K.att_z^{C}\,\theta\ value_z^{C}\}$

where:

- T_I, T_J, T_K: entity tables
- $T_I(UID^{RC}, att\_^{Rm}, \ldots, att\_^{Rf}, \ldots, att\_^{C}, \ldots)$: schema of T_I
- $T_J(UID^{RC}, att\_^{Rm}, \ldots, att\_^{Rf}, \ldots, att\_^{C}, \ldots)$: schema of T_J
- $T_K(UID^{RC}, att\_^{Rm}, \ldots, att\_^{Rf}, \ldots, att\_^{C}, \ldots)$: 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
- θ: one of $\{<, \leq, =, >, \geq,$ **LIKE**, **NOT LIKE**$\}$

We use superscripts *Rm*, *Rf*, and *C* to indicate that the corresponding attribute will be stored in a row table of mandatory attributes, a row table of frequently-accessed-together attributes, or a column table of optional/private/seldom-accessed attributes, respectively. A superscript *RC* is to indicate that the corresponding attribute is stored in both row and column tables. However, these superscripts are not shown to the user.

The plan tree of query *Q* is given in Fig. 2(a). Here, T_I, T_J, and T_K are entity tables whose names, *e.g.*, *Patient, Study, Series, etc.*, are used in *Q* by the user. We assume that each of these tables, has been vertically partitioned into several "child" tables, i.e., row and column tables, by applying the data storage strategy presented in Sect. 3.1. However, only some of the child tables are required by *Q*. We also assume that *Q* is decomposed into sub-queries Q_I, Q_J, and Q_K that are further decomposed into smaller sub-queries $Q_{I,1}$, $Q_{I,2}$, $Q_{J,1}$, $Q_{J,2}$, and $Q_{K,1}$ to able to directly map to child tables containing required attributes. As presented in Fig. 2(b), $Q_{I,1}$ and $Q_{I,2}$ access to T_1 and T_2, respectively, that are child tables of T_I. Similarly, $Q_{J,1}$ and $Q_{J,2}$ access to T_3 and T_4, respectively, that are child tables of T_J. $Q_{K,1}$ only accesses to T_N, a child table of T_K.

HYTORMO uses a *left-deep sequential tree plan* for joining intermediate results of sub-queries. It will automatically determine a join as an inner or a left outer join. Because *Q* is a user query, entity tables are used in *Q*. Thus the type of a join between two entity tables is explicitly identified by the user. For instance, *Q* in Fig. 2(a) can be

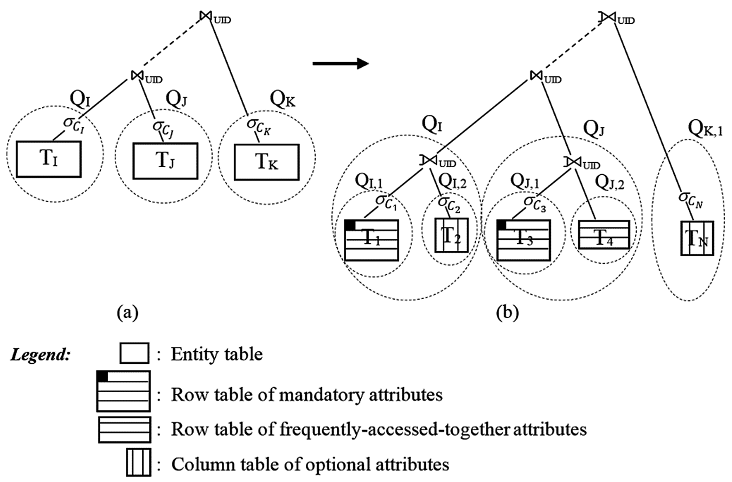

⋈UID
⋈UID
QI
QJ
QK
σCI
σCJ
σCK
TI
TJ
TK
(a)
⋈UID
⋈UID
QI
QJ
QK,1
⋈UID
⋈UID
QI,1
QI,2
QJ,1
QJ,2
σC1
σC2
σC3
σCN
T1
T2
T3
T4
TN
(b)
Legend:
: Entity table
: Row table of mandatory attributes
: Row table of frequently-accessed-together attributes
: Column table of optional attributes

Fig. 2. Plan tree of the SQL query

written as Q = $Q_I \bowtie_{UID} Q_J \bowtie_{UID} Q_K$, only using inner joins. However, it is necessary to evaluate some joins between sub-queries as left outer joins to prevent data loss caused by the tuples discarded by inner joins.

We consider two cases in which a left outer join should be used. First, in a join between two child tables of the same entity table, if the left table is a row table of mandatory attributes while the right table is either a column table of optional attributes or a row table of frequently-accessed-together attributes, this join should be evaluated as a left outer join. For instance, in Fig. 2(b), both sub-queries $Q_{I,1} ⟕_{UID} Q_{I,2}$ and $Q_{J,1} ⟕_{UID} Q_{J,2}$ will evaluated as left outer joins. This is because $Q_{I,1}$ and $Q_{J,1}$ are mapped to row tables of mandatory attributes T_1 and T_3, respectively, while $Q_{I,2}$ and $Q_{J,2}$ are mapped to a column table of optional attributes T_2 and a row table of frequently-accessed-together attributes T_4, respectively. Second, in a join between two entity tables, if the right table has been changed to either a column table of optional attributes or a row table of frequently-accessed-together attributes, this join should be evaluated as a left outer join. For instance, in Fig. 2(b), $Q_{K,1}$ is mapped to column table of optional attributes, thus the join using the result of $Q_{K,1}$ is rewritten to a left outer join.

In the scope of this paper, we concern on inner joins and the two above-mentioned cases of left outer joins. To improve performance of a query, we need to reduce the number of left outer joins and to apply Bloom filters.

Reducing the Number of Left Outer Joins. We use the following heuristic rule for deciding whether or not a left outer join should be rewritten as an inner join: *Given a left outer join $T_1 ⟕_{UID} T_2$, if the right table T_2 does not contain non-null constraints on its attributes, the left outer join is kept no change. In contrast, if the right table T_2 contains non-null constraints on its attributes, the left outer join should be rewritten as an inner join that might improve query performance.*

The above heuristic rule is based on the fact that, in the left outer join $T_1 ⟕_{UID} T_2$, if T_2 does not contain non-null constraints on its attributes, the left outer join returns all matching tuples between T_1 and T_2, like an inner join. The unmatched tuples are also preserved from T_1 and are supplied with *nulls* from T_2. Thus, in this case, the left outer join is kept no change. However, if T_2 contains non-null constraints on its attributes, these constraints must be evaluate to TRUE to form a tuple in the result. They also eliminate any *nulls* of attributes from T_2. In this case, a left outer join is unnecessary, the left outer join thus should be rewritten as an inner join.

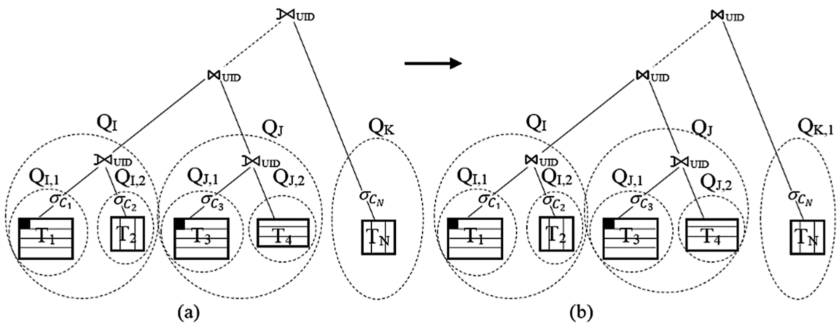


Fig. 3. Rebuilding the plan tree after reducing the number of left outer joins

Figure 3(a) presents a plan tree that has been introduced in Fig. 2. To apply the above heuristic rule to this plan tree, we look at right tables of left outer joins. Assume that σ_{C_2} and σ_{C_N} are non-null constraints on attributes of tables T_2 and T_N, respectively. Then left outer joins in $Q_{I,1} ⟕_{UID} Q_{I,2}$ and $(Q_I \bowtie_{UID} Q_J) ⟕_{UID} Q_{K,1}$ are rewritten as inner joins $Q_{I,1} \bowtie_{UID} Q_{I,2}$ and $(Q_I \bowtie_{UID} Q_J) \bowtie_{UID} Q_{K,1}$, respectively, as given in Fig. 3(b).

Application of Bloom Filters. Bloom filter (BF) is a space-efficient probabilistic data structure with little error allowable when used to test whether an element is a member of a set [11]. In our case, we consider to apply an intersection Bloom filter (IBF) rather than BFs because of its benefit in removing irrelevant data, as presented in [12]. The way how to apply an *IBF* to HYTORMO is given below.

We consider a general form of queries in HYTORMO. Assume that a query Q can be decomposed into a set of sub-queries Q_1, Q_2, …, Q_K, each of which can be further decomposed into smaller sub-queries to able to map to input tables, i.e., row- and column-oriented tables T_1, T_2, …, T_N. Q is in form of a multi-way join on common join attributes. Because HYTORMO uses a *left-deep sequential tree plan,* we focus on the application of the *IBF* for this plan.

Although input tables T_1, T_2,…, T_N might have some common join attributes, in the scope of this paper, we assume these tables only share the common join attribute *UID*. In this case, we can build a common *IBF* on the join attribute *UID* of a subset of the input tables. After built, the *IBF* can be probed to filter these input tables.

The build and probe phases of the *IBF* are illustrated in Fig. 4(a) and (b), respectively. We assume that the heuristic rule has been applied to reduce the number of left outer joins to obtain a plan tree (see Fig. 4(a)). In the build phase, we first compute a set of *BFs* of the same size and the same hash functions on the join attribute *UID* for intermediate result tables D_1, D_2, …, D_N that have been created as results of sub-queries $Q_{I,1}$, $Q_{I,2}$, $Q_{J,1}$, $Q_{J,2}$, and $Q_{K,I}$. We use DataFrames [14] of Spark to store these intermediate result tables. The *IBF* then is computed by bitwise *ANDing* the *BFs*. It is worthy to note that we do not compute a *BF* for the right table of a left outer join if it does not contain non-null constraints on its attributes. For instance, we do not compute a *BF* for D_4 (intermediate result table of $Q_{J,2}$) because there do not exist non-null constraints in $Q_{J,2}$ (i.e., $Q_{J,1} ⟕_{UID} Q_{J,2}$ is not equivalent to $Q_{J,1} \bowtie_{UID} Q_{J,2}$). Thus, building a *BF* for D_4 can cause data loss caused by tuples discarded by *ANDing* this *BF* with others. The *IBF* is probed to filter input tables before a join occurs (see Fig. 4(b)).

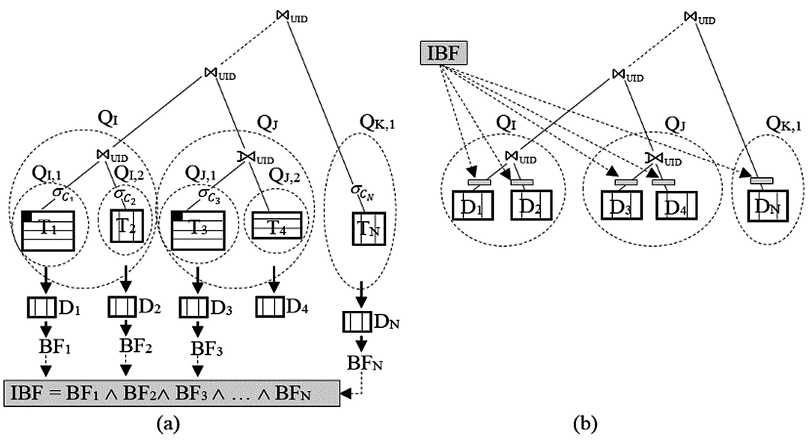


Fig. 4. Build (a) and Probe (b) phases of an IBF.

Cost Effectiveness of IBF. Our goal is to evaluate the effect of using or not using an *IBF* to query performance.

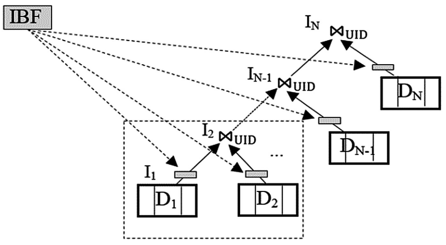


Fig. 5. The approximate left-deep sequential tree plan with the *IBF*.

For the sake of simplicity, we present an approximate cost estimate for the *IBF* on a sequential join sequence of *N* tables. For this approximation, we assume *BFs* are computed for all input tables D_1, D_2, …, D_N. The *IBF* is computed from these *BFs* and probed to filter all of these input tables. We also assume that the sequential join sequence only includes inner joins, as illustrated in Fig. 5. Let that the multi-way join operation be $Q = D_1 \bowtie_{UID} D_2 \bowtie_{UID} \ldots \bowtie_{UID} D_N$, where $|D_i| \leq |D_{i+1}|$, for every $i \in [1, N-1]$. The sequential join sequence for the left-deep sequential tree plan is:

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

Tables D_1, D_2, …, D_N and intermediate result tables I_1, I_2, …, I_{N-1} are used as inputs of joins. Here, we are setting $I_1 = D_1$ and I_N = *final query result*.

The performance of a multi-way join in a cluster is usually determined by *network I/O cost* and *disk I/O cost*. We thus use these costs to analysis the effectiveness of the *IBF*. We start this work by giving definitions and basic mathematical concepts. Assume that *BFs* and *IBF* have been built from tables D_1, D_2, …, D_{N-1} with the same configuration: using a vector of *m* bits and *k* hash functions $h_1(v)$, $h_2(v)$, …, $h_k(v)$, where *v* is a value of join attribute. Table 1 shows notations used in cost models.

Table 1. Notations of cost models.

Notation	*Description*
D_i	Table is used as either a build or a probe table
I_i	Intermediate result table of sequential join sequence
BF_i	Bloom filter is built on table D_i
IBF	Intersection Bloom filter
ρ_{D_i}	Selectivity of table D_i
ρ_{BF_i}	Selectivity of Bloom filter BF_i that is associated to table D_i
ρ_{IBF}	Selectivity of *IBF* that is built on tables D_1, D_2, …, D_N
P_{BF_i}	False positive of BF_i of table D_i due to hash collisions
P_{IBF}	False positive of *IBF* that is built on tables D_1, D_2, …, D_N

The *probability of a false positive* of a Bloom filter BF_i due to hash collisions is calculate by (1) [16], where BF_i is representing a set of n_i values of the join attribute *UID* of table D_i in a vector of *m* bits and using *k* independent hash functions.

$$P_{BF_i} = \left(1 - \left(1 - m^{-1}\right)^{kn_i}\right)^k \approx \left(1 - e^{kn_i/m}\right)^k. \tag{1}$$

We define the *selectivity of Bloom filter* BF_i of table D_i in (2).

$$\rho_{BF_i} = \rho_{D_i} + \left(1 - \rho_{D_i}\right).P_{BF_i} \tag{2}$$

where $\left(1 - \rho_{D_i}\right).P_{BF_i}$ is the fraction of tuples from the probe table D_i that are not discarded by BF_i and do not join with any tuples in the build table.

Given the selectivity of Bloom filters of tables D_1, D_2, …, D_N that have been calculated by (2), the *selectivity of the IBF* is determined by (3).

$$\rho_{IBF} = \prod\nolimits_{i=1}^{N} \rho_{BF_i}. \tag{3}$$

The *false positive of the IBF* can be calculated by (4).

$$P_{IBF} = \prod\nolimits_{i=1}^{N} P_{BF_i} = \prod\nolimits_{i=1}^{N} \left(1 - \left(1 - m^{-1}\right)^{kn_i}\right)^{k}. \tag{4}$$

where N is the number of *BFs* with assumption that there exists a *BF* for each table D_i.

A comparison between (1) and (4) shows that value of P_{IBF} is less than value of P_{BF_i}. This means that applying an *IBF* will give a lower amount of false positive errors than only applying a single *BF*. The larger value of N, the smaller value of P_{IBF}.

In order to estimate *network I/O cost* and *disk I/O cost*, we depend on build and probe phases of the *IBF*, as given in Fig. 4(a) and (b), that include the following steps:

1. *Execute sub-queries to create intermediate result tables D_1, D_2, …, D_N.*
2. *Compute BF_1, BF_2, …, BF_N on values of UIDs of D_1, D_2, …, D_N, respectively.*
3. *Compute the IBF = $BF_1 \wedge BF_2 \wedge \ldots \wedge BF_N$.*
4. *Broadcast the IBF to all slave nodes of the cluster.*
5. *Apply the IBF to input tables D_1, D_2, …, D_N to obtain results $D_{1(filtered)}$, …, $D_{N(filtered)}$.*
6. *Execute the sequential join sequence using tables $D_{1(filtered)}$, …, $D_{N(filtered)}$ as inputs.*

The first three steps are in the build phase while the rest are in the probe phase. Assume that the first step has been done. We start to estimate costs from step 2.

Network I/O Cost. Since each join operation in the sequential join sequence will join an intermediate result table (created by the previous join) with an input table D_i. The *network I/O cost* when the *IBF* is not used, C_{Net}^{NoIBF}, can be calculated by (5).

$$C_{Net}^{NoIBF} = \sum\nolimits_{i=1}^{N} size(D_i) + \sum\nolimits_{i=1}^{N-1} size(I_i) \times size(D_{i+1}) \times \rho_{D_{i+1},I_i}. \tag{5}$$

where $size(D_i)$ and $size(I_i)$ are size of input table D_i and intermediate result table I_i, respectively. ρ_{D_{i+1},I_i} is selectivity factor (ratio of the joined tuples of D_{i+1} with I_i).

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

The *network I/O cost* when the *IBF* is used, C_{Net}^{IBF}, can be computed by (6).

$$\begin{aligned} C_{Net}^{IBF} = c * size(IBF) + \sum\nolimits_{i=1}^{N} size\left(D_{i(filtered)}\right) \\ + \sum\nolimits_{i=1}^{N-1} size(I_i) \times size\left(D_{i+1(filtered)}\right) \times \rho_{D_{i+1(filtered)},I_i}. \end{aligned} \tag{6}$$

where c is the number of slave nodes of the cluster.

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

A comparison between (5) and (6) shows that $c * size(IBF)$ is usually small and $size\left(D_{i(filtered)}\right) \ll size(D_i)$. Therefore C_{Net}^{IBF} is less than C_{Net}^{NoIBF}.

Disk I/O Cost. The *disk I/O cost* without using *IBF*, $C_{I/O}^{NoIBF}$, can be calculated by (7).

$$C_{I/O}^{NoIBF} = \sum\nolimits_{i=1}^{N-1} [size(I_i) + size(D_{i+1})] + \sum\nolimits_{i=2}^{N} size(I_i). \tag{7}$$

where:

- $\sum\limits_{i=1}^{N-1} [size(I_i) + size(D_{i+1})]$: reading intermediate results and input tables for joins.
- $\sum\limits_{i=2}^{N} size(I_i)$: writing intermediate results to disk (here we are setting $I_1 = D_1$).

When the *IBF* is used, the *disk I/O cost*, $C_{I/O}^{IBF}$, can be calculated by (8).

$$\begin{aligned} C_{I/O}^{IBF} = 2 \times \sum_{i=1}^{N} size(D_i) + \sum_{i=1}^{N} size\left(D_{i(filtered)}\right) \\ + \sum_{i=1}^{N-1} \left[size(I_i) + size\left(D_{i+1(filtered)}\right)\right] + \sum_{i=2}^{N} size(I_i). \end{aligned} \tag{8}$$

where:

- $2 \times \sum_{i=1}^{N} size(D_i)$: reading the input tables two times (to build and to apply the *IBF*).
- $\sum_{i=1}^{N} size\left(D_{i(filtered)}\right)$: writing filtered input tables to disk after applying the *IBF*.
- $\sum_{i=1}^{N-1} \left[size(I_i) + size\left(D_{i+1(filtered)}\right)\right]$: reading intermediate results and filtered input tables to be used as inputs of joins.
- $\sum_{i=2}^{N} size(I_i)$: writing intermediate results to disk (here we are setting $I_1 = D_1$).

We assume that the *BFs* and the *IBF* are small enough to be stored in internal memories of slave nodes so that no disk I/O cost is needed for them. A comparison between (7) and (8) shows that $C_{I/O}^{IBF}$ includes extra costs to read and to write input tables during build and probe phases. However, then join operations will use filtered tables as their inputs. Therefore, if $size(D_{i(filtered)}) \approx size(D_i)$, there is no benefit when applying the *IBF*. However, if $size(D_{i(filtered)}) \ll size(D_i)$, we can achieve $C_{I/O}^{IBF} \approx C_{I/O}^{NoIBF}$.

4 Preliminary Experimental Results

This Section presents preliminary experimental results. We first describe experimental environment, datasets and experimental query. We then compare query performance against various storage strategies and effectiveness of *IBF*.

4.1 Experimental Environment

We have used Hadoop 2.7.1, Hive 1.2.1 and Spark 1.6.0 to create a cluster of seven different nodes (one master node and six slave nodes). The hardware of each node is the same and has the following configuration: Intel(R) core(TM) i7-3770 CPU @ 3.40 GHz, 16 GB RAM and 500 GB hard disk. We use the standard configuration with a modification: we change the replication factor of HDFS from 3 to 2 in order to save space. We implement the execution plan for the experimental query using Spark [14].

4.2 Datasets

We have used a mixed DICOM dataset of [17–21]. The metadata and pixel data are extracted from DICOM files using the library dcm4che-2.0.29 [22]. The attributes of metadata are classified and stored in a fashion as discussed in Sect. 3.1. We use sequence files and ORC files in Hive [23] to store row and column tables, respectively. A statistic of the DICOM datasets are given in Table 2.

Table 2. DICOM datasets used in the experiment.

Datasets	Number of files	Number of extracted attributes	Size of extracted metadata	Total size of files
CTColonography [17]	98,737	86	7.76 GB	48.6 GB
Dclunie [18]	541	86	86.0 MB	45.7 GB
Idoimaging [19]	1,111	86	53.9 MB	369 MB
LungCancer [20]	174,316	86	1.17 GB	76.0 GB
MIDAS [21]	2,454	86	63.4 MB	620 MB

Although there are many entities for DICOM data, below we only present schemas of entities required in the experimental query. The superscripts *Rm*, *Rf*, *C* and *RC* are the same as mentioned in Sect. 3.2.

– **Patient**(UID^{RC}, PatientName^{Rm}, PatientID^{Rm}, PatientBirthDate^{Rm}, PatientSex^{Rm}, EthnicGroup^{Rm}, IssuerOfPatientID^{C}, PatientBirthTime^{C}, PatientInsurancePlanCodeSequence^{C}, PatientPrimaryLanguageCodeSequence^{C}, PatientPrimaryLanguageModifierCodeSequence^{C}, OtherPatientIDs^{C}, OtherPatientNames^{C}, PatientBirthName^{C}, PatientTelephoneNumbers^{C}, SmokingStatus^{C}, Pregnancy^{Rf}, LastMenstrualDate^{Rf}, PatientReligiousPreference^{C}, PatientComments^{C}, PatientAddress^{C}, Patient-MotherBirthName^{C}, InsurancePlanIdentification^{C})

Table 3. Row and column tables are used by hybrid data storage strategy.

Entity	Row table of “Rm” attributes	Row table of “Rf” attributes	Column table of “C” attributes
Patient	RowPatient	RowPregnancy	ColPatient
Study	RowStudy	–	ColStudy
GeneralInfoTable	–	–	ColGeneralInfoTable
SequenceAttributes	RowSequenceAttributes	–	–

- **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})

Table 3 shows the corresponding row and column tables that are used to store the above schemas. Because our experiment will compare the query performance against various storage strategies, these schemas also need to be stored in only row tables and only column tables.

4.3 Experimental Query

We measure the execution time of the query using three different storage strategies: row store only, column store only, and hybrid store (hybrid data storage strategy). We also validate the effect of using or not using the *IBF* to query performance.

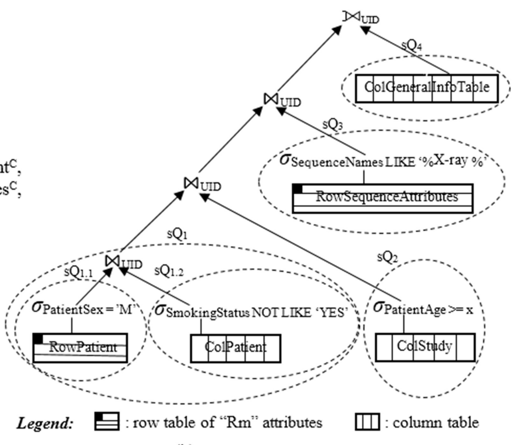
The experimental query is given in Fig. 6(a). It is to retrieve the information stored in *X-ray* DICOM files of *men* who are *non-smoking*, greater than or equal to *x years old*. The query is based on TPC-H query 3 and 4 [24], but here we only focus on *SPJ* queries. The attributes used in *SELECT* and *WHERE* clauses are also marked by superscripts to indicate that they are being stored in row or a column tables. Five tables *RowPatient*, *ColPatient*, *ColStudy*, *ColGeneralInfoTable*, and *RowSequenceAttribute* are required by the query. The query is decomposed into four subqueries *sQ1*, *sQ2*, *sQ3*, and sQ4, as given in Fig. 6(b). The query processing strategy presented in Sect. 3.2 is applied to build a left-deep processing tree step by step while trying to keep intermediate results as small as possible.

4.4 Preliminary Query Performance

We ran the query for (i) *storing all tables in row stores (RS),* (ii) *storing all tables in column stores (CS),* and (iii) *storing all tables in the proposed hybrid store (HS).*

SELECT p.UID^{RC}, p.PatientID^{Rm},
p.PatientName^{Rm}, p.PatientBirthDate^{Rm},
p.PatientSex^{Rm}, p.EthnicGroup^{Rm},
p.SmokingStatus^{C}, s.PatientAge^{C}, s.PatientWeight^{C},
s.PatientSize^{C}, i.GeneralNames^{C}, i.GeneralValues^{C},
q.UID^{RC}, q.SequenceTags^{Rm}, q.SequenceVRs^{Rm},
q.SequenceNames^{Rm}, q.SequenceValues^{Rm}
FROM Patient p, Study s, GeneralInfoTable i,
SequenceAttributes q
WHERE p.UID^{RC} = s.UID^{RC}
AND p.UID^{RC} = i.UID^{RC}
AND p.UID^{RC} = q.UID^{RC}
AND p.PatientSex^{Rm} = 'M'
AND p.SmokingStatus^{C} **NOT LIKE** 'YES'
AND s.PatientAge^{C} >= x
AND q.SequenceNames^{Rm} **LIKE** '%X-ray%'

(a)



(b)

Fig. 6. The experimental query (a) and its execution plan tree (b)

The selectivity of the query vary depending on the predicates on attributes *PatientAge, PatientSex, SmokingStatus,* and *SequenceNames*. In our experiment, we vary the selectivity of the predicate on *PatientAge* to be *0.06 (PatientAge* >= *90)*, *0.42 (PatientAge* >= *60)*, and *0.88 (PatientAge* >= *10)* but fix the selectivity of the others.

The chart in Fig. 7 shows that, for all cases of the selectivity, storing all schemas in only row stores or only column stores leads to higher execution time than that in a hybrid store. The rationale behind the query performance is in the own benefit of each data storage strategy. With the use of the hybrid store, storing mandatory attributes in row tables, e.g., *RowPatient* and *RowSequenceAttribute*, helped to reduce tuple reconstruct cost because most of these attributes are accessed together by the query. In contrast, only a few of optional/private/seldom-used attributes are required by the query. They thus should be stored in column tables, e.g., *ColPatient*, *ColStudy*, and *ColGeneralInfoTable* to save I/O bandwidth because only relevant attributes need to be read. If we store these attributes in row tables, the entire rows still have to be read from disk no matter how many attributes are accessed per query at once. This causes a waste of I/O bandwidth. Therefore, depending on a good understanding about the workload of regular queries, we can choose a right store for each attribute extracted from DICOM files to improve the query performance.

To evaluate the effect of using an *IBF*, we build an *IBF* from *BFs* that are computed for all tables except *ColGeneralInfoTable* since it is the right table of a left outer join. The accuracy of a *BF* is decided by ratio *m/n* where *m* is length of bit vector and *n* is size of set (i.e., cardinality of *UID* list of an input table). *m* = *8n* has been considered a good balance between accuracy and space usage [24]. We thus apply this setting with *n* is the biggest cardinality value among tables (*RowPatient* in our case).

The chart in Fig. 8 gives a comparison between HYTORMO with (HS + IBF) and without (HS) using the *IBF*. In the best case, where the query is very highly selective (*PatientAge* >= *90*), the IBF helped to reduce 80% of execution time, whereas in the worst case, where the query is lowly selective (*PatientAge* >= *10*), the *IBF* increased

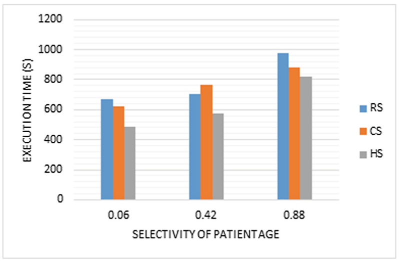


Fig. 7. Comparison of different storage strategies

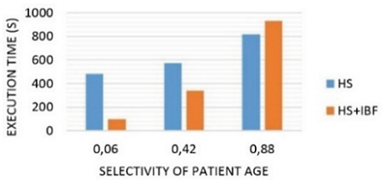


Fig. 8. A comparison of the effect of the IBF.

14% of the execution time. When *PatientAge >= 60*, the *IBF* helped to reduce *41%* of the execution time. This is because in the best case the *IBF* removed a large amount of irrelevant tuples from joins. However, in the worst case, most of tuples of input tables are required in the final result and thus there are not much useless data to be removed by the *IBF*. The overhead costs incurred by build and probe phases of the *IBF* decrease query performance.

5 Related Works

There already exist several solutions to implement a DICOM data storage system. PACSs [25] mostly use row-RDBMSs to store, retrieve, and distribute medical image data. These systems are expensive but only support queries with predefined attributes and thus do not cope with heterogeneous schemas. eDiaMoND [2] stores DICOM data using a Grid-enabled medical image database that is built from row-RDBMSs. The system aims to provide inter-operability, scalability and flexibility. However, the development of query optimization techniques have not been introduced. Some commercial row-RDBMSs [4] have provided features to store and manage large-scale repositories of DICOM files. They add a new data type that enables any column of this type to store a DICOM content in their database table. Since a new separate object is created for each DICOM file, the storage space is quickly increased and thus decreases the overall performance of system. DCMDSM [5] is based on the original DSM [26] to vertically partition DICOM metadata into multiple small tables. The method is able to cope with the evolutive/heterogeneous schemas of DICOM data and saves I/O bandwidth. Unfortunately, the method already uses a centralized database approach developed on the top of a row-RDBMS and has not been designed to operate in a clustering environment. NoSQL document-based storage system [6] shares the schema-free non-relational design of standard key-value stores. It thus can handle the evolution of metadata. However, unlike traditional row-RDBMSs, there is no standardized query language for the proposed system.

6 Conclusion and Future Work

The high-performance DICOM data management becomes a real challenge. The current solutions still exist limitations to cope with the high heterogeneity, evolution, variety, and high volume of DICOM data. In this paper, we propose HYTORMO, using a hybrid data storage strategy and query processing strategy with *BFs*. Our preliminary experimental results have showed that it is necessary to carefully choose the right stores for attributes extracted from DICOM files. The use of both row and column stores results in lower execution time because it helps to reduce disk I/O, tuple reconstruction cost, and storage space. The application of the *IBF* helped to reduce network I/O cost because it removed irrelevant tuples out of inputs of joins. Our query performance is promising.

The next steps of our work is to reduce the overhead cost of *BFs*, a new cost model needs to be built to specify a threshold for selectivity factor of input tables so that *BFs* are only computed for tables that can be reduced large enough. Our future work also will include a comparison of HYTORMO to other methods such as commercial row-RDBMS [4] that use tables of a row-RDBMS to store schemas of metadata and use a single column of Object type in a table to store image content. SDSS SkyServer [31] also proposed a similar method, but to manage astronomy data. Furthermore, we consider to generate a bushy execution plan, instead of a left-deep tree plan.

References

1. Pianykh, O.S.: Digital Imaging and Communications in Medicine (DICOM): A Practical Introduction and Survival Guide. Springer, Heidelberg (2008)
2. Merelli, I., et al.: Managing, analysing, and integrating big data in medical bioinformatics: open problems and future perspectives. BioMed. Res. Int. 1–13 (2014)
3. Power, D., Politou, E., Slaymaker, M., Harris, S., et al.: A relational approach to the capture of DICOM files for grid-enabled medical imaging databases. In: SAC, pp. 272–279 (2004)
4. Annamalai, M., Guo, D., Susan, M., Steiner, J.: An oracle white paper: oracle database 11 g DICOM medical image support (2009)
5. Savaris, A., Härder, T., von Wangenheim, A.: DCMDSM: a DICOM decomposed storage model. J. Am. Med. Inform. Assoc. **21**, 917–924 (2014)
6. Rascovsky, S.J., et al.: Informatics in radiology: use of CouchDB for document-based storage of DICOM objects. Radiographics **32**, 913–927 (2012)
7. Boncz, P., et al.: MonetDB/X100: hyper-pipelining query execution. In: CIDR (2005)
8. Stonebraker, M., et al.: C-store: a column-oriented DBMS. In: VLDB, pp. 553–564 (2005)

9. Ramamurthy, R., DeWitt, D.: A case for fractured mirrors. VLDB **12**, 89–101 (2003)
10. Grund, M., et al.: HYRISE: a main memory hybrid storage engine. VLDB **4**, 105–116 (2010)
11. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**, 422–426 (1970)
12. Phan, T.C., Orazio, L.D., Rigaux, P.: Toward intersection filter-based optimization for joins in MapReduce. In: Workshop Proceedings of the Cloud-I (2013)
13. OECD: Genetic Testing: A Survey of Quality Assurance and Proficiency Standards. OECD Publishing, Paris (2007)
14. Armbrust, M., et al.: Spark SQL: relational data processing in spark. In: SIGMOD (2015)
15. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and randomized optimization for the join ordering problem. VLDB J. **6**, 191–208 (1997)
16. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: a survey. Internet Math. **1**(4), 485–509 (2004)
17. CT Colonography. https://idash.ucsd.edu. Accessed 11 Oct 2015
18. David Clunie's Medical Image Format Site. http://www.dclunie.com. Accessed Oct 2015
19. Sample Data. http://idoimaging.com/wiki/. Accessed 12 Oct 2015
20. Lung Cancer Datasets. http://giveascan.org. Accessed 11 Oct 2015
21. MIDAS Datasets. http://www.insight-journal.org. Accessed 12 Oct 2015
22. Open Source Clinical Image and Object Management. http://www.dcm4che.org
23. White, T.: Hadoop: The Definitive Guide. 4th edn. O'Reilly Media, Inc., California (2015)
24. TPC-H specification 2.8.0. http://www.tpc.org/tpch/
25. Möller, M., Mukherjee, S.: Context-driven ontological annotations in DICOM images: towards semantic PACS. In: Proceedings of HEALTHINF (2009)
26. Copeland, G., Khoshafian, S.: A decomposed storage model. In: SIGMOD (1985)
27. Harizopoulos, S., et al.: Performance tradeoffs in read-optimized databases. In: VLDB (2006)
28. Floratou, A., Minhas, U.F., Özcan, F.: SQL-on-Hadoop: full circle back to shared-nothing database architectures. VLDB **7**, 1295–1306 (2014)
29. Popescu, A.D., Dash, D., Kantere, V., Ailamaki, A.: Adaptive query execution for data management in the cloud. In: CloudDB, pp. 17–24 (2010)
30. Rösch, P., Dannecker, L., Färber, F., Hackenbroich, G.: A storage advisor for hybrid-store databases. Proc. VLDB **5**(12), 1748–1758 (2012)
31. Szalay, A.S., et al.: The SDSS Skyserver: public access to the sloan digital sky server data. In: Proceedings of SIGMOD, pp. 570–581. ACM (2002)