# Benchmarking a Distributed Database Design that Supports Patient Cohort Identification

Jero Mario Schäfer
Institute of Computer Science
Department of Mathematics and Computer Science
University of Göttingen
Göttingen, Germany
jeromario.schaefer@stud.uni-goettingen.de

Ulrich Sax
Department of Medical Informatics
University Medical Center Göttingen
Göttingen, Germany
ulrich.sax@med.uni-goettingen.de

Lena Wiese
Research Group Bioinformatics
Fraunhofer Institute for Toxikology and Experimental Medicine (ITEM)
Hannover, Germany
lena.wiese@item.fraunhofer.de

## ABSTRACT

In this article we present the implementation and benchmarking of a medical information system on top of a distributed relational database system. We enhanced a distributed database system with the implementation of a clustering (based on similarity of disease terms) that induces a primary horizontal fragmentation of a data table and derived fragmentations of secondary tables. With our clustering-based fragmentation, data locality for similarity-based query answering is ensured so that data do not have to be sent unnecessarily over the network. In our benchmark we show that we achieve a significant efficiency gain when retrieving all relevant related answers.

## CCS CONCEPTS

• **Information systems** → *Query optimization*; *Relational parallel and distributed DBMSs.*

## KEYWORDS

Distributed database system, relational databases, similarity-based query answering

**ACM Reference Format:**
Jero Mario Schäfer, Ulrich Sax, and Lena Wiese. 2020. Benchmarking a Distributed Database Design that Supports Patient Cohort Identification. In *24th International Database Engineering & Applications Symposium (IDEAS 2020), August 12–14, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3410566.3410608

## 1 INTRODUCTION

For the ever-growing amount of data in our world, *distributed databases* (DDBs) gained significantly more importance over the past years because they provide physically distributed storage [13, 16]. Especially in fields where the amount of data to be hosted in a database is challenging, a fragmentation of such big data as well as allocation to multiple servers in a cloud storage infrastructure can pay off. The data fragmentation overcomes the limitation of storage capacity of a single database server because the total amount of data can be split into smaller parts, so-called fragments. Moreover, fragmentation can increase the performance of query processing: queries of different users addressing different fragments that are stored on different servers can be answered independently and in parallel. Furthermore, the replication of the data fragments inside the network yields a tolerance to failures by compensating data loss with recovery, e.g. in case of a malfunction or total failure of a single server, to guarantee the desired availability and reliability of the data accessed by the users of the distributed database system (DDBS).

As a use case, the trend towards personalized medicine or the increased usage of novel biomedical technology – for example, next-generation sequencing [17] – produce vast amounts of data. In this article, our particular application scenario is a *medical information system* that uses a distributed database system as a storage backend. In our example system, patients' personal information as well as the diseases they suffer from are contained in a distributed database. A possible usage scenario is that researchers make use of these patient data, for example to identify a cohort of patients [1] that are similar to a current "target patient" – hence instead of exact query answering, a notion of similarity-based query answering is needed.

Our proposed data management system is supposed to support medical staff in identifying a relevant subset of patient data from partitioned tables in an efficient way. We make use of a taxonomy-induced data fragmentation and data distribution in a DDBS to achieve this. In order to support the use case of similarity-based query answering in an efficient way, we present here an implementation that enhances the basic distributed data management with an automatic clustering-based fragmentation that does not require any user interaction other than posing standard SQL queries.

A similarity measure defined between the disease terms from the MeSH taxonomy [9] yields the similarity values needed for the clustering. Our system ensures that data records holding information about patients suffering from similar illnesses are stored on the same site in a distributed database system; this enables the system to preserve data locality with respect to the semantics of the data as defined by the underlying taxonomy. In this way our system is able to answer similarity-based queries efficiently (without the need to access multiple servers).

Our specific contributions in this article are that we (1) obtain similarity values of disease terms by applying a shortest path algorithm in the Neo4J graph database; (2) implement a clustering


*IDEAS 2020, August 12–14, 2020, Seoul, Republic of Korea*

ACM ISBN 978-1-4503-7503-0/20/06…$15.00
https://doi.org/10.1145/3410566.3410608

procedure on top of a relational database system that uses these similarity values; (3) compare the runtime behavior of similarity-based query answering with a round-robin approach and a clustering-based approach.

The remainder of this article is organized as follows. Section 2 surveys related work on flexible query answering and DDBs. Section 3 provides the necessary background on the employed database system. Section 4 introduces our notion of clustering-based fragmentation. Section 5 describes the implementation details and presents a comparative evaluation. Section 6 concludes the article.

## 2 RELATED WORK

### 2.1 Flexible Query Answering

Conventional database systems merely support exact query answering and are not supportive to the user when some query conditions cannot be satisfied. Upon query failure, that is, when the database cannot give an exact answer to a query formulated by a user, an empty result is returned. Assuming the query was formulated correctly, this empty result indicates to the user that the information he or she was looking for is not present in the current database instance. In this case of a lack of any exact answers, the empty database result is non-informative for the user in the general case. In contrast, real-life information systems should provide users with supportive mechanisms when they want to retrieve a particular information from the system. To wit, *flexible query answering* can overcome this lack of information when the database system uses techniques like *query relaxation* and *query generalization* to provide the user with a non-empty result. While this result does not match the query exactly, it nevertheless contains information instead that may be relevant to the user. This relevance property must be supported by an appropriate notion of similarity. A real-world use case is to find "patients like mine" in electronic health records [1].

Several approaches are based on different theoretical backgrounds for intelligent flexible query answering. A recent comprehensive survey of query relaxation in graph-structured data can be found in [12]. Opposed to this, [7] give an in-depth analysis of including taxonomic information on the relational algebra level. Several approaches consider similarity on the syntactical query level [5] or analyse combinations of so-called query generalization operators [3]. Furthermore, in contrast to the related work we explicitly consider a specific similarity in a taxonomy and propose a practical approach that considers query answering based on a semantic clustering. In addition, we devise a bridge between high-level similarity-based query answering and low-level distributed data management.

### 2.2 Distributed Databases

A *distributed database management system* (DDBMS) manages several underlying database instances and provides the access to the data spread across a computer network [11]. A partitioning of data tables can be achieved by using a certain fragmentation strategy and the resulting fragments, which contain parts of the whole data set, can then be dispersed across the network by mapping the fragments to database instances that possibly reside at different physical locations. A relational DDBS has to be able to answer a query in the same manner as a non-distributed relational database. The result of a query is a set of tuples fulfilling the query. To obtain the result, the database system uses relational algebra operations for its computation, e.g. a join of two tables or the selection on an attribute with a certain condition. As the data is physically distributed, queries have to be processed and rewritten according to the underlying distribution to allow for an appropriate answer to the given query. The performance of computing answers to the query can be a big issue due to increased network communication inferred by data transfer between the database sites. This can even occur for simple queries that only scan a certain relation and project to some subset of the attributes. Fragmentation and replication influence the query execution strategy as they require for a distribution of the query itself to possibly multiple servers, too, in order to get the complete and correct result set. In our system we apply appropriate rewriting techniques to support the intended similarity-based query answering.

### 2.3 Horizontal Fragmentation

One strategy to obtain a fragmentation of the data is *horizontal fragmentation* that divides a relation in a row-wise manner into smaller portions of tuples. More formally, a relation $R$ is divided into fragments $F_1, F_2, ..., F_n$ by assigning each tuple $\mu$ of the relation $R$ to at least one fragment The result of this is that for all $i \in \{1, \ldots, n\},\ F_i \subseteq R$. Additionally, in order to avoid redundancy of data, we can require that each tuple is only assigned to exactly one fragment; more formally, the fragments are pairwise disjoint: $\forall \mu \in F_i$ it holds that $\mu \notin F_j, i \neq j$ for $i, j \in \{1, \ldots, n\}$. In relational algebra such a *primary horizontal fragmentation* can be described by a selection operation $\sigma$ on the relation $R$, where the selection condition defines the desired mapping of tuples to fragments. Based on this primary horizontal fragmentation, a further fragmentation of another relation $S$ can be *derived* by computing the semi-join of the relation $S$ with fragments $F_i,\ i \in \{1, \ldots, n\}$ of the primary relation $R$, i.e. the derived fragments $G_i$ of the relation $S$ are computed as $G_i = S \ltimes F_i$ for $i \in \{1, \ldots, n\}$. The *derived horizontal fragmentation* depends on the underlying primary horizontal fragmentation, and, to prevent tuples in $S$ from getting lost during the semi-join with fragments of $R$, it is necessary to have for each tuple $y \in S$ matching tuples in $R$ in order to let the tuples from $S$ "survive" the semi-join. An integrity constraint in form of a foreign key reference of the relation $S$ to the relation $R$ can be used to enforce this condition for the sake of completeness of the derived fragmentation.

Several horizontal partitioning approaches in distributed database systems focus on numerical data. As an example for numerical data partitioning, the AdaptDB system [6] builds up a binary search tree where the median of a range of numerical values is chosen as the pivot element in each level of the search tree. Their storage model is based on low-level blocks of equal size in distributed file systems. Another approach [18] is predicate-based referenced partitioning (PREF) addressing relational DB systems. They start with a partitioning of a seed table and then co-partition all tables with incoming or outgoing foreign keys which could be potentially joined. This way of co-partitioning can lead to redundancy in the sense that tuples of the co-partioned tables can occur in more than one fragment. The co-partitionings are then further cascaded by

also co-partitioning tables that could be joined with the existing fragments.

Our approach differs from these two by supporting similarity-based query answering (which has a wide applicability in cohort identification on medical data) as well as enabling partitioning on categorical attributes (by clustering based on a similarity measure between terms). We visualize these differences in Table 1.

# 3 BACKGROUND

## 3.1 System Design

We will make use of horizontal partitioning to support similarity-based query answering. We postpone the formal definition until Section 4.3. We assume that one table is chosen as the primary table and one attribute of it is determined on which similarity-based query answering should be performed. Other tables that join to the primary table will be co-partitioned by derived fragmentation. Inherently with the definition of the derived horizontal fragmentation on a semi-join, redundancy of tuples of $S$ in the derived fragments may occur if tuples in $S$ match multiple tuples that belong to different fragments of $R$. This causes the fragments $G_i$ of $S$ to be non-disjoint in general – an approach also followed in [18]. We will make use of this property to ensure data locality of primary and derived fragments: while we require the primary fragmentation to be disjoint (and hence non-redundant) the derived fragmentation might contain fragments that have some tuples in common. These derived fragments are however stored on different sites together with their matching primary fragment.

## 3.2 Default Hash Partitioning

Our specific implementation reported on in this paper is based on in-memory storage inside a network of servers each running an Apache Ignite instance. The Ignite server nodes store data depending on the fragmentation (called partitioning with Ignite) and replication. The fragmentation and replication is defined per relation. The default Ignite partitioning is a horizontal fragmentation of the data defined with hash functions; thus, there are no explicit selection conditions on a certain attribute of the relation that represent a horizontal fragment as we would need for our semantic clustering-based fragmentation – and the assignment of tuples to the partitions/fragments is done according to the hash function and hence rather arbitrarily.

## 3.3 Ignite's Affinity Collocation

One important concept of Ignite is the collocation of data – which corresponds to the concept of a derived horizontal fragmentation: data that are accessed together, e.g. because they are joined via a common attribute or a foreign key reference, are also stored together on the same server. This affinity collocation can be defined in Ignite by so-called *affinity keys*: An affinity key can be identical to the primary key of a relation or an attribute of a composite primary key of a relation. Ignite ensures that all tuples where the affinity keys match are stored on the same server. The usage is restricted to a single affinity key definition per relation. With this restriction, there cannot be a collocation of three or more relations that could be joined via a chain of join conditions where the join attributes would form possible affinity keys of the different relations. Note that this notion of affinity for horizontal fragmentation is different from the notion of attribute affinity which has long since been used for vertical fragmentation [10].

If all the joined relations in the SQL query are collocated, the query can be evaluated locally by each node, because all the data they need to compute a correct result set regarding their portion of the whole data in the cluster is available, i.e. stored by themselves. Non-collocated joins must be enabled explicitly in Ignite to enforce the distributed answering with data transfer across the network if necessary. If not enabled explicitly, the query will be executed only in a collocated (local) manner which, in general, leads to incomplete result sets due to required data not being available locally.

# 4 CLUSTERING-BASED FRAGMENTATION

To improve data collocation from a semantic point of view, we replace the hash-based horizontal partitioning by our proposed clustering-based fragmentation. More precisely, the clustering-based fragmentation is a horizontal fragmentation strategy that, on the one hand, enables fragmentation regarding a similarity measure which allows for a semantic partitioning of the data set, and, on the other hand, supports similarity-based query answering.

## 4.1 Similarity Computation

A specific measure of similarity has to be defined on terms connected in a taxonomy (or more generally any ontology on which we can define a notion of similarity). The pairwise similarity values between any two terms in the taxonomy form the basis for a clustering on the terms contained in one of the attributes in the provided database tables. More formally, we assume that in our primary relation $R$ there is a specific attribute $A$ which will be used for clustering and hence similarity-based query answering. In order to capture semantic closeness, we need a similarity relationship $sim(a, b)$ for pairs of elements $a, b$ from the *active domain* of the chosen attribute $A$: the projection $\pi_A(R)$ of $R$ to the values of $A$. More formally, $sim : \pi_A(R) \times \pi_A(R) \rightarrow \mathbb{R}$. It is customary to restrict the range of the similarity to the interval $[0, 1]$ such that a similarity of 1 denotes that the elements $a$ and $b$ have highest similarity, whereas the closer the similarity value gets to 0, the more dissimilar the two elements are. In cases where the terms occurring in the queries are not contained in the active domain, we have to obtain the similarity for each such query term to the terms of the active domain, too. Hence we assume that all terms in the active domain of the selected attribute as well as in any value for the attribute required in a query are contained in the taxonomy (or at least can be mapped to a term in the taxonomy).

In our application of a medical information system, a similarity is defined on the disease information of patients; we assume that the disease terms conform to the vocabulary provided by the *Medical Subject Headings* taxonomy (MeSH) of the U.S. National Library of Medicine [9]. In MeSH the same disease term can be located under different subtrees (corresponding to different disease classifications). The levels in the classification tree are represented by numbers; so each term can be uniquely identified by its "tree number".

We imported the MeSH taxonomy into the graph database Neo4J. We used Neo4J because it has a convenient query language called Cypher as well as provides a graph algorithms library that can be

| | primary partitioning method | secondary partitioning method | query method | storage |
|---|---|---|---|---|
| AdaptDB [6] | median of the attribute in root of partitioning tree | medians of other attributes in the partitioning tree | exact | block level (HDFS) |
| PREF [18] | hash-based | co-partitioning on join attributes | exact | relational (XDB) |
| Our approach | clustering on active domain of an attribute | co-partitioning on join attributes | similarity-based | relational (Apache Ignite) |

**Table 1: Comparison of approaches**

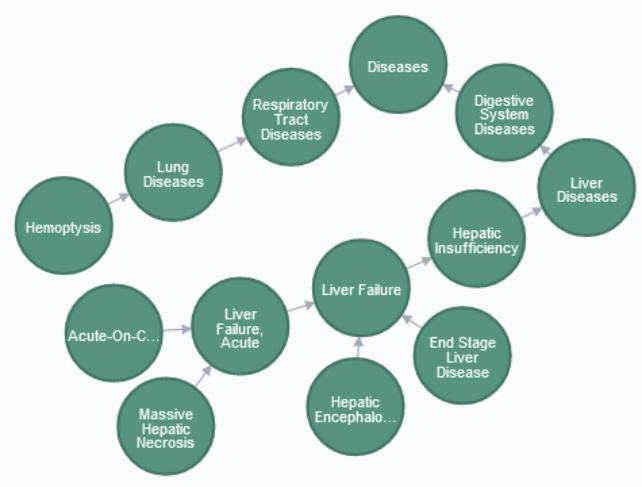


**Figure 1: A snippet of the MeSH taxonomy in Neo4J**

used to compute shortest paths. As shown in Table 2 we read in the MeSH entries from a CSV file where the first entry in each line is the tree number and second entry is the disease term. First of all we create a node for every term in the file resulting in 11,649 nodes. Next we create edges between a term and its parent term by identifying the parent term by its tree number (which is the tree number of the term without the last 4-digit level) resulting in 11,648 edges. Figure 1 shows a snippet of the obtained tree where the Diseases category is the root node.

Next, we obtain the minimal length shortest path between two disease terms by a Cypher query that looks up any two disease terms, using the Neo4J graph algorithms library to obtain the shortest paths and returning the length of the shortest one (that is, the amount of edges on the path). Lastly, we obtain a similarity value by dividing 1 by the shortest path length plus 1 (that is, the amount of nodes on the shortest path):

$$sim(a,b) = \frac{1}{minlength(shortestpath(a,b)) + 1} \quad (1)$$

This approach of similarity defined by shortest path length can generally be applied to any graph-shaped ontology.

We could of course compute any similarity value on demand by issuing a query to Neo4J. However we need all pairwise similarities for the clustering up front. Hence, for sake of performance we precompute all similarity values and store them in a similarity table in the Ignite system that and similarity values are read from this table while performing the clustering.

## 4.2 Clustering

When loading the data into the distributed database, the underlying clustering is computed with an approximation algorithm [2] on all values that occur in the active domain of a chosen attribute. The pseudocode of the clustering procedure is described in Listing 1. The clustering starts with a single cluster (Line 1) containing the whole active domain and an arbitrarily chosen representative "head" element from the cluster and then identifies the minimal similarity inside the cluster between all elements from the active domain and the cluster head (Line 6). Subsequently, new clusters are created with new head elements based on the minimal similarity of a term to the head of a cluster as long as the similarity threshold is not exceeded; all elements are reassigned if they are more similar to the head of the newly created cluster (Lines 7 and 8). The procedure iterates as long as there are still elements inside one of the clusters that have a similarity to the corresponding head element that is lower than a user-defined similarity threshold $\alpha$ (while-condition in Line 5). Hence, the iteration proceeds until each element of the active domain is clustered such that the minimal similarity according to threshold $\alpha$ can be ensured.

**Listing 1** Clustering procedure

**Input:** Set $\pi_A(F)$ of values for attribute $A$, similarity threshold $\alpha$
**Output:** A set of clusters $c_1, \ldots, c_f$

1: Let $c_1 = \pi_A(F)$
2: Choose arbitrary $head_1 \in c_1$
3: $sim_{min} = min\{sim(a, head_1) \mid a \in c_1; a \neq head_1\}$
4: $i = 1$
5: **while** $sim_{min} < \alpha$ **do**
6: Choose $head_{i+1} \in \{b \mid b \in c_j; b \neq head_j; sim(b, head_j) = sim_{min}; 1 \leq j \leq i\}$
7: $c_{i+1} = \{head_{i+1}\} \cup \{c \mid c \in c_j; c \neq head_j; sim(c, head_j) \leq sim(c, head_{i+1}); 1 \leq j \leq i\}$
8: $c_i = c_i \setminus \{c \mid c \in c_j; c \neq head_j; sim(c, head_j) \leq sim(c, head_{i+1}); 1 \leq j \leq i\}$
9: $i = i + 1$
10: $sim_{min} = min\{sim(d, head_j) \mid d \in c_j; d \neq head_j; 1 \leq j \leq i\}$
11: **end while**

As an example, consider the sample disease terms *Hemoptysis*, *Acute-On-Chronic Liver Failure*, *Massive Hepatic Necrosis*, *Hepatic Encephalopathy*, and *Ulna Fracture* from Figure 1 as the active domain of the diagnosis attribute. We see that all liver diseases are similar to one another (either $\frac{1}{3}$ or $\frac{1}{4}$) – but are less similar to lung diseases (either $\frac{1}{9}$ or $\frac{1}{10}$), and vice versa. By, for example, setting $\alpha$

| | |
|---|---|
| Create nodes | `USING PERIODIC COMMIT LOAD CSV FROM "file:///ctree2019MeSH.csv" AS line`<br>`CREATE (d:Disease {TREE_NUMBER:toString(line[0]),DESCRIPTOR:toString(line[1])});` |
| Create edges | `USING PERIODIC COMMIT LOAD CSV FROM "file:///ctree2019MeSH.csv" AS line`<br>`MATCH (d:Disease {TREE_NUMBER: toString(line[0])}) WHERE size(d.TREE_NUMBER)>4`<br>`MATCH (d2:Disease {TREE_NUMBER: substring(d.TREE_NUMBER,0,(size(d.TREE_NUMBER)-4))})`<br>`MERGE (d)-[:PARENT]->(d2)` |
| Return shortest path length | `MATCH (n:Disease {DESCRIPTOR: "Massive Hepatic Necrosis"})`<br>`WITH n MATCH (n2:Disease {DESCRIPTOR: "Hemoptysis"})`<br>`WITH n,n2 MATCH p = shortestPath((n)-[*]-(n2)) RETURN min(length(p))` |

**Table 2: Loading MeSH into Neo4J and finding shortest paths**

between $\frac{1}{5}$ and $\frac{1}{8}$ we can separate liver diseases and lung diseases into two separate clusters.

The different clusters contained in the clustering are then used to obtain a horizontal fragmentation of the entire relation instance $R$ by partitioning the relation along the disjunctive subsets of elements from the active domain with respect to the chosen attribute $A$. Each cluster induces one horizontal fragment: any two tuples are mapped to the same fragment if their values for clustering attribute $A$ belong to the same cluster.

Completeness of the clustering provides coverage for all elements of the active domain $\pi_A(R)$ regarding the attribute $A$. Furthermore, the mapping of any element of the active domain to one of the clusters is functional. This makes the clusters pairwise disjoint, and hence, all fragments induced by the clustering are non-redundant; each tuple is assigned to only fragment according to the maximum similarity to one of the cluster heads. Thus, the relation instance $R$ can be reconstructed from the horizontal fragments.

### 4.3 Query Answering

As the data is distributed according to the clustering into horizontal fragments, similarity-based queries can now be executed in a distributed manner according to the clustering. Finding the relevant data fragment is done based on the selection condition on the attribute $A$ chosen for the clustering: for each term in the selection condition the term's similarity to the head elements of the different clusters is obtained. The maximal similarity of the comparison element and all cluster heads determines the matching cluster as well as the induced horizontal fragment. More formally, we can define a similarity-based answer for each selection (sub-)query on attribute $A$ as follows. Given a clustering-based fragmentation of relation $R$, for a selection query $\sigma_{A=\text{“}s\text{”}}R$ on the clustering attribute $A$ the similarity-based answer is

$$\{F_i \mid head_i = argmax_{j=1,\ldots,n} sim(s, head_j)\}$$

If there is more than one cluster head with maximal similarity, we choose one of them at random. In this case it is sufficient to execute the query only locally at a single server which hosts the single relevant data fragment. An alternative to this approach would be to return the union of all fragments with most similar head elements.

## 5 IMPLEMENTATION AND EVALUATION

To evaluate the proposed clustering-based fragmentation, we compare its implementation to the default hash-based partitioning. The source code is available in a Github repository[1].

### 5.1 Data Set

We generated a synthetic data set to comparatively investigate the behavior of the implementation variants. We analyzed scalability of our approach by varying both the data set size and the size of the term set (that is, active domain of the clustering attribute) on which similarities are calculated. Our test data set is modelled according to three tables: A table “Ill” (containing attributes for the patient ID and the diagnosis), a table “Treat” (containing attributes for the patient ID and the medication) and a table “Info” (containing attributes for the patient ID and additional administrative data like address and age). In this way we simulate a simplified database schema found in medical datasets like MIMIC [4] or platforms like i2b2 [8].

The Ill table is our primary table; the patient IDs are generated in decreasing order; the diagnosis attribute contains disease terms extracted randomly from the MeSH data set. The Treat table contains the patient IDs and randomly generated string data as the prescription. The Info table contains one random address string for each patient as well as a randomly generated age.

Scaling of the data set was obtained by a default number of tuples for each of the tables multiplied by a scaling factor. The default size of the Ill table is 100 tuples and the default size of both Treat and Info table is 50 tuples – that is, an average of two disease entries per person plus one sample prescription. For a given scaling factor $s$ the dataset is expanded to a total of $s \cdot (100+50+50) = 200 \cdot s$ tuples. The MeSH term set was divided into smaller, randomly chosen subsets ranging from a minimum of 100 terms up to all 4798 terms from which the Diagnosis column of the Ill table is filled.

[1] https://github.com/l-wiese/SiFAMIS

| | |
|---|---|
| $Q_1$ | `SELECT p.name, p.age, p.address FROM ILL i, INFO p WHERE i.id = p.id AND i.disease='Hepatic Encephalopathy'` |
| $Q_2$ | `SELECT p.name, p.age, p.address FROM ILL i1, ILL i2, INFO p WHERE i1.id = p.id AND i2.id = p.id AND i1.disease='Hepatic Encephalopathy' AND i2.disease='Hemoptysis'` |
| $Q_3$ | `SELECT t.prescription FROM ILL i,TREAT t WHERE t.id = i.id AND i.disease = 'Hepatic Encephalopathy'` |
| $Q_4$ | `SELECT p.name, p.age, t.prescription FROM ILL i,TREAT t, INFO p WHERE t.id = i.id AND i.id = p.id AND i.disease = 'Hepatic Encephalopathy'` |
| $Q_5$ | `SELECT t.prescription FROM ILL i, ILL i2, TREAT t WHERE i.id = i2.id AND i.id = t.id AND i.disease = 'Massive Hepatic Necrosis' AND i2.disease = 'Hepatic Encephalopathy'` |

**Table 3: Benchmark queries (disease terms are chosen randomly)**

## 5.2 Clustering-Based Implementation

Before inserting data, we have a certain one-time overhead in terms of initialising the similarity table inside Ignite and clustering the active domain of the selected attribute:

- For clustering the disease attribute (implemented as an external Java program) we measured runtimes of 0.96 seconds for 500 MeSH terms, 2.05 seconds for 1000 MeSH terms, 13.60 seconds for 2500 MeSH terms, 75.61 seconds for all 4798 MeSH terms.
- For the initialisation of the similarity table we measured runtimes of 8.21 seconds for 500 MeSH terms, 28.82 seconds for 1000 MeSH terms, 109.00 seconds for 2500 MeSH terms, 228.03 seconds for all 4798 MeSH terms.
- For the batch loading of our data set for 200000 tuples (that is, scale factor 100) which includes a lookup on the similarity table we measured runtimes of 88.57 seconds for 500 MeSH terms, 115.45 seconds for 1000 MeSH terms, 162.48 seconds for 2500 MeSH terms, 258.66 seconds for all 4798 MeSH terms.

Yet we assume that this kind of batch loading of a large data set only occurs once before actually using the system. Hence the performance gains during querying (as shown later in Section 5.5) pay off when querying the system. As discussed in Section 6 modifications of the data set are up to future work.

In order to deploy the clustering-based approach with Ignite, we assign different partition numbers (each corresponding to one cluster) to the different semantic fragments of the relations via Ignite's affinity function API. The fragments are then distributed and mapped to the servers. Collocation of derived partitions is also achieved with the help of the affinity function and the identification of the correct partition number that is inferred from the clustering.

In our data set we can join both the *Treat* as well as the *Info* table with the *Ill* table based on the patient ID. For any fragment of *Ill*, we obtain one derived fragment of each of *Treat* and *Info*: we fragment the *Treat* as well as the *Info* table according the patient IDs contained in the fragments of the primary table *Ill*. For example, if a patient has disease $x$ that belongs to cluster $c_j$, then partition $j$ of the *Ill* relation stores the tuple stating that this patient has disease $x$ and additionally partition $j$ of the *Info* relation is responsible for the tuple with the patient's personal information.

In order to implement similarity-based query answering, the selection condition on the clustering attribute is omitted and the query is adapted by restricting it to the fragment that belongs to the cluster with the relevant diseases, such that then all answers only need to be obtained from this fragment; the partition number has to be identified and set via the appropriate class method of the SqlQuery or SqlFieldsQuery of Ignite's SQL API.

## 5.3 Default Implementation

On the other hand, the default hash-based partitioning is implemented by creating partitioned tables by the standard Ignite partitioning methodology. The primary table (in our example, *Ill*) is partitioned horizontally based on a hash function applied to its affinity key (that is, attribute patient ID); The other tables are collocated via their shared attribute, the patient ID, such that personal information and the diseases and treatments of a patient are stored together to ensure the collocation of the data. That is, only collocation via the attribute patient ID is guaranteed – but no similar disease terms are collocated. In order to implement similarity-based query answering, the selection condition on the clustering attribute is replaced by a more general expression: the original SQL query is translated into one with a SQL `IN`-clause containing the similar disease terms from the appropriate cluster.

## 5.4 Queries

Our benchmark queries $Q_1$ to $Q_5$ (see Table 3) consist of executing joins (between primary and secondary fragments) with a selection condition on the diagnosis attribute. Similarity-based query execution includes finding the fragment that is closest to the query condition (in terms of similarity to the cluster head that is contained in the diagnosis attribute of the fragment). In other words, query execution extracts the selection condition, applies the query rewriting and returns the obtained fragment (or joins of fragments, respectively) as the set of related answers. Note that queries Q2 and Q5 both have two selection conditions on which similarity-based query-answering is applied. The difference between the two queries is that in Q5 both selection conditions come from the same fragment (and the tuples are hence collocated); whereas in Q2 the selection conditions are not in the same cluster and data have to be retrieved from different fragments before joining them.

## 5.5 Results

Our distributed system is evaluated in a network of three Apache Ignite nodes where each of the nodes runs in a JVM and is hosted by one of three servers. A total of 24 GB memory for the cloud is split equally among all machines and each server has 4 processors.

Table 4 shows the results of executing our five benchmark queries (Table 3) when scaling the amount of tuples in the database from 20000 over 200000 to 2000000. For the clustering-based approach we tested in addition different amounts of underlying disease terms

| Query | # Tuples | Default | Partitions 100 terms | Speed up | Partitions 500 terms | Speed up | Partitions 1000 terms | Speed up | Partitions 2500 terms | Speed up | Partitions all terms | Speed up |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 20000 | 171.57 | 80.56 | 2.13 | 59.15 | 2.90 | 71.09 | 2.41 | 59.11 | 2.90 | 61.70 | 2.78 |
| 2 | 20000 | 269.27 | 112.30 | 2.40 | 58.30 | 4.62 | 170.10 | 1.58 | 121.51 | 2.22 | 102.83 | 2.62 |
| 3 | 20000 | 180.99 | 91.21 | 1.98 | 55.20 | 3.28 | 108.45 | 1.67 | 82.77 | 2.19 | 86.56 | 2.09 |
| 4 | 20000 | 108.41 | 113.47 | 0.96 | 56.56 | 1.92 | 89.41 | 1.21 | 76.68 | 1.41 | 64.20 | 1.69 |
| 5 | 20000 | 188.60 | 92.11 | 2.05 | 55.80 | 3.38 | 136.02 | 1.39 | 87.87 | 2.15 | 89.94 | 2.10 |
| 1 | 200000 | 424.51 | 148.24 | 2.86 | 196.29 | 2.16 | 135.11 | 3.14 | 107.74 | 3.94 | 125.16 | 3.39 |
| 2 | 200000 | 4000.00 | 199.12 | 20.09 | 157.96 | 25.32 | 146.65 | 27.28 | 382.47 | 10.46 | 327.74 | 12.20 |
| 3 | 200000 | 357.09 | 138.97 | 2.57 | 147.80 | 2.42 | 260.87 | 1.37 | 534.77 | 0.67 | 385.51 | 0.93 |
| 4 | 200000 | 1362.86 | 146.70 | 9.29 | 132.54 | 10.28 | 262.20 | 5.20 | 250.47 | 5.44 | 309.57 | 4.40 |
| 5 | 200000 | 4000.00 | 181.84 | 22.00 | 181.10 | 22.09 | 275.14 | 14.54 | 256.14 | 15.62 | 289.77 | 13.80 |
| 1 | 2000000 | 4000.00 | 1168.24 | 3.42 | 1314.28 | 3.04 | 1150.21 | 3.48 | 1257.63 | 3.18 | 1160.64 | 3.45 |
| 2 | 2000000 | 4000.00 | 2410.04 | 1.66 | 2009.41 | 1.99 | 2301.47 | 1.74 | 2391.26 | 1.67 | 2271.32 | 1.76 |
| 3 | 2000000 | 4000.00 | 1120.11 | 3.57 | 1350.73 | 2.96 | 1540.83 | 2.60 | 1747.32 | 2.29 | 1670.13 | 2.40 |
| 4 | 2000000 | 4000.00 | 1296.70 | 3.08 | 1396.45 | 2.86 | 1320.13 | 3.03 | 1520.87 | 2.63 | 1309.04 | 3.06 |
| 5 | 2000000 | 4000.00 | 2411.80 | 1.66 | 2566.07 | 1.56 | 2700.19 | 1.48 | 2619.55 | 1.53 | 2729.25 | 1.47 |

**Table 4: Runtime measurements for queries 1 to 5 in milliseconds for varying amount of tuples in the database instance and varying amount of MeSH terms in the active domain of the clustering attribute; the default approach is stopped when exceeding 4 seconds; speedup is relative to the default approach; overall average speedup of our approach is** 4.79

from MeSH that are used in the Diagnosis column: we tested 100, 500, 1000, and 2500 randomly chosen as well as all (4798) MeSH terms. We ran the five queries three times and averaged the runtimes. Whenever the default Ignite approach runtime significantly exceeded the runtime of the other approaches, we cut off its measurements after 4000 milliseconds. Our measurements show that the clustering-based approach scales better for the similarity-based query answering use case. For the largest scaling factor (2000000 tuples) the default approach always faced a timeout whereas the average query execution time of our similarity-based approaches was 1530 milliseconds. For most of the cases the similarity-based approach also performs better for smaller data sets (smaller scaling factors); only for some term set sizes in Queries 3 and 4, the similarity-based query answering introduces a slight overhead – evidenced by a speed up $< 1$ compared to the default Ignite approach. Averaged over all measurements we achieve a speedup of 4.79

## 6 CONCLUSION

The presented distributed database design demonstrates the capabilities of our novel similarity-based query answering where data fragmentation is based on a clustering with respect to a given similarity. The query execution runtimes show that the clustering-based fragmentation improves the execution time of queries against the DDB significantly when comparing to the basic implementation that provides only an arbitrary, hash-based horizontal fragmentation of the data.

In future work, other taxonomies and other disease similarities could be used – depending on the analyzed medical use case. In addition, several notions of similarity (semantic as well as numeric as in [14, 15]) can be combined in order to identify not only patients that suffer from similar diseases but also whole patient profiles based on the similarity of their personal characteristics (e.g. their age or weight) and some other recorded measurements (e.g. body temperature or blood parameters). Modifications in the data set are crucial for a real-world applications. In future work we plan to support adaptivity to changing attributes values (by either insertions, deletions or updates) and hence changing clusters. Further practical advancements include analysis of other clustering methods and their influence on the resulting distributed data management behavior.

## ACKNOWLEDGEMENTS

This work was partially supported by the Fraunhofer Internal Programs under Grant No. Attract 042-601000.

## REFERENCES

[1] S. Gombar, A. Callahan, R. Califf, R. Harrington, and N. H. Shah. It is time to learn from patients like mine. *npj Digital Medicine*, 2(1):1–3, 2019.
[2] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293 – 306, 1985.
[3] K. Inoue and L. Wiese. Generalizing conjunctive queries for informative answers. In *International Conference on Flexible Query Answering Systems*, pages 1–12. Springer, 2011.
[4] A. E. Johnson, T. J. Pollard, L. Shen, H. L. Li-Wei, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. A. Celi, and R. G. Mark. MIMIC-III, a freely accessible critical care database. *Scientific data*, 3(1):1–9, 2016.
[5] V. Kantere. Query similarity for approximate query answering. In *International Conference on Database and Expert Systems Applications*, pages 355–367. Springer, 2016.
[6] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. Adaptdb: adaptive partitioning for distributed joins. *Proceedings of the VLDB Endowment*, 10(5):589–600, 2017.
[7] D. Martinenghi and R. Torlone. Taxonomy-based relaxation of query answering in relational databases. *The VLDB Journal*, 23(5):747–769, 2014.
[8] S. Murphy and A. Wilcox. Mission and sustainability of informatics for integrating biology and the bedside (i2b2). *eGEMs*, 2(2), 2014.
[9] National Library of Medicine. Medical subject headings, Nov 2019.
[10] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710, 1984.
[11] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
[12] A. Poulovassilis. Applications of flexible querying to graph data. In *Graph Data Management*, Data-Centric Systems and Applications, pages 97–142. Springer, 2018.

[13] K. Tan. Distributed database systems. In *Encyclopedia of Database Systems (2nd ed.)*. Springer, 2018.

[14] A. Tashkandi, I. Wiese, and L. Wiese. Efficient in-database patient similarity analysis for personalized medical decision support systems. *Big data research*, 13:52–64, 2018.

[15] I. Wiese, N. Sarna, L. Wiese, A. Tashkandi, and U. Sax. Concept acquisition and improved in-database similarity analysis for medical data. *Distributed and Parallel Databases*, pages 1–25, 2018.

[16] L. Wiese. *Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases*. DeGruyter/Oldenbourg, 2015.

[17] L. Wiese, A. O. Schmitt, and M. Gültas. Big data technologies for DNA sequencing. In *Encyclopedia of Big Data Technologies*. Springer, 2019.

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