Noname manuscript No.
(will be inserted by the editor)

Concept Acquisition and Improved In-Database Similarity Analysis for Medical Data

Ingmar Wiese · Nicole Sarna · Lena Wiese · Araek Tashkandi · Ulrich Sax

Received: date / Accepted: date

Abstract Efficient identification of cohorts of similar patients is a major precondition for personalized medicine. In order to train prediction models on a given medical data set, similarities have to be calculated for every pair of patients – which results in a roughly quadratic data blowup. In this paper we discuss the topic of in-database patient similarity analysis ranging from data extraction to implementing and optimizing the similarity calculations in SQL. In particular, we introduce the notion of chunking that uniformly distributes the workload among the individual similarity calculations. Our benchmark comprises the application of one similarity measures (Cosine similariy) and one distance metric (Euclidean distance) on two real-world data sets; it compares the performance of a column store (MonetDB) and a row store (PostgreSQL) with two external data mining tools (ELKI and Apache Mahout).

Keywords Patient similarity · Row store · Column store · Cosine similarity · Euclidean distance

I. Wiese / N. Sarna / L. Wiese / A. Tashkandi
Institute of Computer Science, University of Goettingen
Goldschmidtstraße 7, 37077 Göttingen, Germany
(A.T. is also affiliated to King Abdulaziz University, Faculty of Computing and Information Technology, 21589 Jeddah, Kingdom of Saudi Arabia)
E-mail: ingmar.wiese@stud.uni-goettingen.de
E-mail: n.sarna@stud.uni-goettingen.de
E-mail: wiese@cs.uni-goettingen.de
E-mail: araek.tashkandi@cs.uni-goettingen.de

U. Sax
Department of Medical Informatics,
University Medical Center Goettingen,
University of Goettingen
Von-Siebold-Straße 3, 37075 Göttingen, Germany
E-mail: ulrich.sax@med.uni-goettingen.de

1 Introduction

The increasing amount of Electronic Health Records (EHRs) has led to an enormous wealth of medical data in recent years. Consequently, personalized medicine tries to exploit the opportunities coming from this phenomenon. Similarity searches spark the interest of many clinicians, as for example in internal medicine or in oncology not all parameters can be inspected visually or measured directly. Thus interdisciplinary therapy boards collect data and expertise for proposing therapy options for patients. As this so-called precision medicine could lead to the perception that all patient cases are individual, functions like "show me similar patients" and "show me their treatment and the outcome" are of paramount interest in the current precision medicine scene. Given the fact of very often incomplete and unstructured documentation in clinical routine, these algorithms have to be able to scale with the data quality and data size.

However, there are two sides to the coin. On the one hand, the accumulated data can be used to provide individualized patient care as opposed to models based on average patients. Two of the most common models based on the average patient are SAPS [17] and SOFA [32]. These models perform well in predicting various clinical outcomes [8, 11] but it has been shown that personalized prediction models can lead to even better results [18]. Personalized models use an index patient's data and then predict future health conditions (or recommend treatments) based on similar patients and not the average patient. On the other hand, these individualized prediction models come at an increased computational complexity and therefore entail longer run-times because large patient data sets from EHR databases are used for each individual index patient.

A patient similarity metric (PSM) is the basis for personalized health predictions. The PSM could be one of different algorithms such as neighborhood-based algorithms or distance metrics. This PSM defines a cohort of patients similar to a given index patient. Subsequently only data of similar patients are used to predict the future health of the index patient or recommend a personalized treatment. In order to train prediction or recommender models appropriately, *pairwise* similarity computations between any two individual patients are necessary. With n patients the amount of $\binom{n}{2}$ similarity calculations is required. By increasing the data size, the computational burden of this analysis increases.

Our use case applies to a scenario where a large data set (exceeding the random access memory capacities) is already stored in a database system. Hence we need a technology that is independent of RAM size and has support for built-in hard disk support. We also assume that users of our system are not skilled programmers and are just familiar with basic SQL statements. Hence we want to avoid excessive programming as well as any installation, configuration and execution of external data mining (DM) tools. Most DM tools score badly with respect to these requirements. In particular, most DM tools just work on in-memory data and cannot scale for larger data sizes. Hence we develop and test a solution that purely focuses on in-database calculations.

Similarity lies at the basis of many data mining and machine learning tasks (for example, k-nearest neighbor or clustering). Hence we believe that precomputing and storing similarities in a table for quick lookup is beneficial for further analysis. In general, in-DB calculations are not appropriate for tasks more complex than the similarity calculations considered in our work. Recent developments show how to integrate Python machine learning tools with MonetDB [26] to reap the best of both worlds.

1.1 Contributions

This paper focuses on increasing the performance of calculating the $\binom{n}{2}$ pairwise similarity values. We make the following contributions:

– We present a concept acquisition module that helps find relevant data values in a set of diverse item descriptions.
– We show that the similarity calculations can be achieved solely by **in-database analytics**, without the need of external software. When the data are reliably stored inside a database system and the majority of the workload is processed directly within the database, the cost of external data mining tools (like R, ELKI or Apache Mahout) is saved. Relying on SQL as the data manipulation language and taking advantage of optimizations of the internal database engine, data transfer latency and data conversion problems (in particular, conversion into vectorized data representations of the data mining tools) can be eliminated.
– We analyze two data models (a **column-based** and a **row-based** data representation) and provide the appropriate similarity calculation expression in SQL. We compare the performance of each data model on one **column store** and one **row store** database system.
– We investigate optimization techniques and quantify their impact on patient similarity calculations. In addition to **multi-threading** and **batch processing** we introduce **chunking** as a further optimization.
– For several optimized settings, we compare the in-database approach to two external data mining tools (ELKI and Apache Mahout).
– For several optimized settings, we compare the performance of one similarity measure and a distance metric (Cosine similarity and Euclidean distance).
– We develop our method with a real-world dataset containing intensive care unit (ICU) data. We verify our in-database approach with a second larger real-world dataset containing data of diabetes patients.

1.2 Outline

This article is structured as follows. Section 2 presents several related approaches in the area of patient similarity analysis. Section 3 introduces the real-world data set and discusses the process of data extraction. Section 4 defines the applied similarity measure (Cosine similarity) and the applied distance metric (Euclidean distance). Section 5 describes the basic difference between patient analysis with row-oriented versus column-oriented data formats. Section 6 proposes three optimization methods. Section 7 presents several benchmark results. Section 8 concludes the article.

2 Related Work

Patient similarity analysis is a relatively new field of study, nevertheless the increased interest in the field has led to numerous studies being conducted. The next two sections will give a twofold overview of the subject by first presenting the literature on several application areas of the approach and, secondly, exploring whether performance issues have already been addressed.

A lot of research efforts have been made applying patient similarity analysis with different predictive approaches in mind. These approaches include discharge diagnosis prediction by Wang *et al.* [34], future health prediction by Sun *et al.* [31] and mortality prediction as found in Morid *et al.* [20], Lee *et al.* [18], and Hoogendoorn *et al.* [15]. All of these patient similarity approaches can also be found in the comprehensive survey by Sharafoddini *et al.* [29]. There are several validation techniques and algorithms that can be employed in the field of patient similarity analysis. Morid *et al.* [20] applied a similarity-based classification approach with a k-nearest neighbor algorithm for Intensive Care Unit (ICU) patient similarity analysis. Similarly, Hoogendoorn *et al.* [15] use Euclidean distance with a k-nearest neighbor algorithm. The related and also quite commonly used approach of cosine similarity as a metric is used by Lee *et al.* [18] whose hypothesis revolves around personalized mortality prediction. They were able to show that their approaches outperform traditionally used scores like SAPS in prediction capabilities.

All of the above mentioned research papers on patient similarity analysis lay the focus on the evaluation of the accuracy of the prediction models. Therefore, most researchers do not particularly pay attention to the performance of their methods and only a few mention the limitation induced by the increased computational complexity of their analysis algorithms (for example, Lee *et al.* [18], and Brown *et al.* [3]). Certainly, accuracy is the justifying factor when the predictive models are presented. However, in the age of big data where the EHRs get massively larger, the performance of analysis methods is critical. High-dimensional data (i.e. data with a wide array of feature variables like medical measurements) and a large data set naturally lead to an increased computational burden. This can become a major issue, particularly for training prediction methods since these training sets rely on the calculation of all pairwise similarity values between each patient in the training set. The pairwise patient similarity calculations insofar intensify the challenge of handling big EHR data.

Despite the above mentioned remarks about the computational complexity generated by patient similarity analysis, there are few current efforts that address performance optimization of such methods. In general, the analytics engine is located outside the database or data warehouse, because the methods require advanced tools that for example are capable of conducting sophisticated statistical analysis. This is not the primary application field of database systems but rather software such as *R* and *SPSS*. We can therefore observe that in related work on patient similarity analysis the database systems are only used as mere repositories for patient data and not taken into consideration for use on even part of the workload.

In other application areas however, it was demonstrated that in-database analytics outperforms ex-database applications like R while still maintaining fully fledged transaction support. One example for this is the work by Passing et al. [24] who extend SQL with data analysis operators for processing (in particular, k-Means, Naive Bayes, and PageRank) in a main-memory database system. In a similar vein, [21] survey in-database evaluation of data with an amount of features varying between 4 and 64 by implementing user-defined functions for of several statistical models (like linear regression, PCA, clustering and Naive Bayes). The article compares implementations for horizontal (row-wise) and vertical (column-wise) data layouts. Focusing on a non-relational graph data model, [4] compare graph algorithms (like reachability, shortest path, weakly connected components, and PageRank) in a column store, an array database and an external graph processing framework. Similarly, [22] focus on evaluating recursive queries on graphs by using each a columnar, a row and an array database. The general gist in these approaches is that the huge advantage of in-database data analysis lies in the avoidance of maintenance of external data analysis tools as well as any data extraction and loading overheads.

3 Data Sets and Data Extraction

We describe the two data sets we used for testing our approach as well as the process of extracting the test data. Note that the final data sets used for testing contain purely numeric values and they are normalized to avoid any influence of the different scales of the features on the resulting similarity values.

3.1 Data Set I (MIMIC)

The majority of our tests were executed with the data set MIMIC-III [16] which is freely accessible for researchers worldwide. It contains patient data that were collected in an Intensive Care Unit (ICU) between 2001 and 2012. All personal information of the 46520 distinct patients in the MIMIC-III database was removed or deidentified to comply with the *Health Insurance Portability and Accountability Act* (HIPAA).

The ICU constitutes a particularly intriguing case for clinical data analysis [1]: the breadth and scale of the data that is collected on a daily and hourly basis allows for extensive data analysis. MIMIC-III comprises data ranging over a wide variety of domains:

– **Descriptive items** like demographic details and dates of death
– **Laboratory values**, for example blood chemistry and urine analysis
– **Medication records** of intravenous and oral administration
– **Physiological values** like vital signs
– **Free text notes and reports** such as discharge summaries and electrocardiogram studies
– **Billing information** of, among others, ICD codes and Diagnosis-Related Group (DRG) codes

The MIMIC-III utilizes a snowflake schema in data warehouse terms [5]. This implies that all of the above mentioned domains of patient data are realized in individual fact tables which are connected to a hierarchy of dimension tables. Figure 1 presents an example using a snippet of the database, namely the fact table *chartevents*. *Chartevents* mainly stores physiological records that are being collected on roughly an hourly basis. The dimension tables that are connected to it are *patients*, *admissions*, *icustays*, and *d_items*. The former three constitute data on a single patient with the following hierarchy: Each individual patient and her/his basic data is stored in *patients*. Since naturally every patient can have multiple admissions to a hospital, the relation *admissions* stores data on when a patient was admitted, released, or died as well as additional information that is susceptible to change for each admission like insurance coverage. During each admission a patient can be transferred in and out of the ICU which is tracked in *Icustays*. In total, MIMIC-III contains 61532 distinct ICU-stays comprised of 58976 admissions by 46520 individual patients.

Lastly, *d_items* is one of several dictionary tables in the database. An item in MIMIC-III refers to measurements such as 'heart rate' in *chartevents* or a specific type of drug whose administration is captured in an *inputevents* table.

3.2 Data Extraction

Patient similarity computations are in essence pairwise vector comparisons. The idea is to select specific **predictors** (that is, features used in prediction models) that are available for all patients in the EHR and place them in a vector, which we call **patient vector**. These features can be extremely diverse: from vital signs like heart rate and blood pressure, over lab results like white blood cell count and serum potassium, to whether the patient received mechanical ventilation. Of course, elementary features such as weight, age and gender can also be taken into consideration. All in all, in theory medical professionals can potentially create the patient vector of their choice and adjust it to their specific needs.

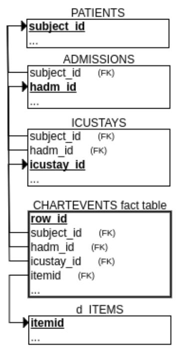

Fig. 1 MIMIC-III chartevents table in snowflake schema

```
SELECT item_id, label
FROM d_items
WHERE (lower(label) LIKE %blood pressure%
  OR lower(label) LIKE %bp%)
  AND lower(label) LIKE %mean%;
```

Fig. 2 SQL query for mean blood pressure

The selected features have to be extracted from the EHR. Usually, a patient's interesting data is scattered all over a hospital's data warehouse, or even across multiple institutions in case of for example a transfer between two hospitals. As a result, complex queries have to be made in order to compile all relevant data points. This also applies to the MIMIC-III data set. The MIMIC database in its version II [27] only consisted of one main data source, the *Philips CareVue* system which was used between 2001 and 2008. Version III retained all the data of version II but added data that was collected with a new system, *Metavision*, between 2008 and 2012. As mentioned above, every piece of information or measurement is an item in MIMIC's *d_items* table. Now as a consequence of the merging of two data sources, MIMIC-III contains multiple *item ids* referencing the same type of measurement or item. As an example let us look at the heart rate item which has *item id* 211 in version II. In version III, however, *item id* 211 *and* 220045 refer to heart rate respectively. This conceptual redundancy applies virtually to all measurements in the database. For data extraction this means that multiple *item ids* have to be grouped in order to excerpt data for one concept.

The merging of data systems mentioned in the previous paragraph is not the only aspect of MIMIC-III's design that forces us to group several items together. This is best illustrated when looking at an example. Let us assume we want to include a patient's mean blood pressure as a predictor in our vector. When querying the database for 'BP', 'Blood Pressure', and 'Mean' with the SQL query shown in Figure 2 we get the result shown in Table 1.

itemid	*label*
52	Arterial BP Mean
224	IABP Mean
443	Manual BP Mean(calc)
456	NBP Mean
2732	Femoral ABP (Mean)
3312	BP Cuff [Mean]
3314	BP Left Arm [Mean]
3316	BP Left Leg [Mean]
3318	BP PAL [Mean]
3320	BP Right Arm [Mean]
3322	BP Right Leg [Mean]
3324	BP UAC [Mean]
5731	FEMORAL ABP MEAN
6653	femoral abp mean
6702	Arterial BP Mean #2
7618	BP Lt. leg Mean
7620	BP Rt. Arm Mean
7622	BP Rt. Leg Mean
220052	Arterial Blood Pressure mean
220181	Non Invasive Blood Pressure mean
224322	IABP Mean
225312	ART BP mean

Table 1 Query Result for mean blood pressure from *d_items*

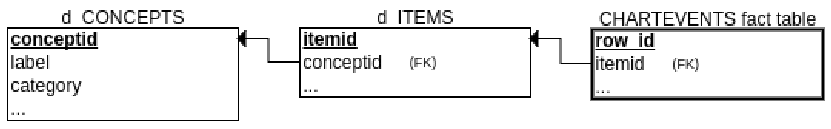

Fig. 3 Added concept dimension

This overabundance of possible items, that refer to some kind of measure related to the concept 'mean blood pressure', clarifies that there is the need for a *mapping* step in the process of data extraction. As a matter of fact, the creators of MIMIC-III seem to be aware of this issue, as there is a dedicated column called *conceptid* in the *d_items* table, though the value is null for each entry. Therefore, the first step for extraction is to devise a mapping from items to concepts.

In order to introduce these concepts, we build on what what is already present in MIMIC-III and utilize the *utilize* column in *d_items*. This will require the introduction of another dimension table, which shall for logical reasons be called *concepts*. In this relation, we store all concepts, and assign them an ID. Figure 3 illustrates the added dimension to the relevant segment of Figure 1 (MIMIC-III snowflake architecture).

Once the mapping has been established the actual extraction can be undertaken whose first step is to gather data on every ICU-stay of a certain time frame. This time frame is then subdivided into intervals, eg. we extract patient data from the first two days of an ICU-stay and further partition this into 12 hourly intervals.

Lastly, it has to be decided how null values should be treated. A null value in the context we are now in, is a missing value for a defined interval. Sharafoddini *et al.* [29] found that most approaches have decided to remove patients with missing values from the pool of similar patients altogether; our system therefore follows the same approach. In order to delete all tuples belonging to an ICU-stay that contains null values, the distinct intervals for each *conceptid*-grouping have to be counted. Should this number be smaller than the maximum interval, all tuples comprised of the same *icustay*-identification are deleted.

We want to reinforce the point that data extraction is the major precondition for further analysis and often is one of the most time-consuming steps in the data analysis workflows. In particular for large distributed data sets implementing these data filtering steps inside the database system can offer benefits due to platform-independence of the SQL language and built-in indexing support.

3.3 Data Set II (Diabetes)

To validate our approach, we used the diabetes dataset from the Health Facts database [30] as a second dataset. It is publicly available via the UCI machine learning repository [7]. This validation dataset was originally provided by the Center for Clinical and Translational Research, Virginia Commonwealth University [30]. The data set covers patient data from 130 US hospitals collected over a period of 10 years (1999–2008). It was extracted to study the relationship between the measurement of Hemoglobin A1c (HbA1c) and early hospital readmission. The predictor selection by [30] was based on medication and blood measurements associated with diabetes. In addition, demographic data were extracted like race, gender and age. Moreover, they defined a readmission attribute. We applied some preprocessing to obtain a dataset without null values. We also converted several categorical terms into integer numbers to obtain a purely numerical data set. As a result our test data set contained 101,766 rows with 43 columns.

3.4 Normalization

As mentioned before, data have to be normalized to equalize the influence of the different scales of the features on the resulting similarity values. Normalization to the range 0 to 1 is executed and the normalized data set is stored in a separate table. Storing both the original data set and the normalized data set is not a problem because we rely on sufficient disk space. This is also a benefit of using the database system itself for the calculation because we can access both data tables in parallel. In contrast, executing the normalization with external data mining tools relying only on RAM capacity also leads to this kind of data duplication but for larger data sets is bound to lead to exceeded memory.

4 Similarity Measures

Once the data extraction is complete, a metric that is used to establish a similarity measure between two patients has to be chosen. This metric has to be applied to all possible combinations of the n patients in the data set which will result in $\binom{n}{2} = \frac{1}{2}(n-1)n$ comparisons which is $\in \mathcal{O}(n^2)$ in terms of complexity. This quadratic complexity can undoubtedly become an issue when dealing with a sizable amount of patients.

4.1 Cosine Similarity

In analogy to Lee *et al.* [18] we apply the cosine similarity in our investigations. Cosine-similarity-based metrics measure the distance between two patient vectors by means of the angle between them. More precisely, cosine similarity returns the cosine of the angle between the two vectors. The cosine is determined by dividing the dot product of the two patient vectors by the product of their respective norms:

$$\cos(\theta) = \frac{p \cdot q}{\|p\| \|q\|} = \frac{\sum_i p_i \cdot q_i}{\sqrt{\sum_i (p_i)^2}\sqrt{\sum_i (q_i)^2}} \tag{1}$$

where θ is the angle between two patient vectors, p and q, and i ranges over all features. Because cosine is bounded between -1 and 1, so will be the cosine similarity between patients. However, with the restriction that the components of a patient vector can never be a negative value, the resulting cosine similarity will be in the range of 0 to 1. There are many implementations of patient similarity analysis applying cosine similarity in different health prediction areas like for example [18,12,19].

4.2 Euclidean Distance

A similarity measure can usually be derived from a distance metric. Several options arise for the conversion of a distance value into a similarity value [6].

Hence, as an alternative to the cosine similarity, we also tested the Euclidean distance. The Euclidean distance is one of the most common distance measures. The Euclidean Distance describes the shortest distance between two data points in a line. In other words, for any two patient vectors it takes the square root of the sum of squared differences in each dimension. In an n-dimensional feature space for two vectors p and q the Euclidean distance between these two vectors is shown in the following Equation 2 (where i ranges over all the features).

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{2}$$

The Euclidean distance is also often used for patient similarity analysis, for instance in [23,33,13].

5 In-Database Similarity Calculations

Optimized similarity computation inside the database system promises great performance enhancements. In order to avoid data extraction into text files and analyzing them with an external data mining tool, in-database analytics performs the similarity calculations directly within the database and also stores results there for use in prediction models. As we show in the following sections, the calculations themselves offer a lot of room for improvement concerning how many vectors are computed at the same time and which vectors are compared to which.

5.1 Data Models

The 'natural' form in which the patient vectors exist after the data extraction described in Section 3.2 can be regarded as column-oriented schema. As shown in Table 2, it consists of three columns, *VectorID*, *PredictorID*, and the *value*. Therefore, each *VectorID* occurs in m rows (when there are m selected predictors). The columnar orientation is due to the fact that rows in MIMIC-III's fact tables consist of single measurements described by patient and item identification (*subject_id*, *hadm_id*, *icustay_id*, and *itemid*, respectively). However, there is also an advantage to this way of representing the vectors when it comes to using aggregation functions – which will become clear in the next section when we take a look at how to calculate the similarities in-database.

Before delving into details, we discuss an alternative schema. Each predictor will be represented by its own column and as a result, every row in this schema constitutes one patient vector as illustrated in Table 3.

The downside of this approach is, however, that it requires an extra step after data extraction. Since all vectors are already available in the column-oriented format, a reorganization has to take place. Data have to be grouped or partitioned (depending on the functionality offered by the chosen DBMS) by *VectorID*. Next, for each predictor the value has to be obtained and placed in the corresponding column.

Our system implements both approaches in order to determine whether the row-wise approach has advantages over its column-wise counterpart. This will be especially interesting when comparing both schemas on DBMSs that follow two different paradigms, namely *column store* vs *row store* systems.

$VectorID$	$PredictorID$	$Value$
1	$Predictor_1$	$value_1$
⋮	⋮	⋮
1	$Predictor_m$	$value_m$
2	$Predictor_1$	$value_1$
⋮	⋮	⋮
2	$Predictor_m$	$value_m$
⋮	⋮	⋮
n	$Predictor_1$	$value_1$
⋮	⋮	⋮
n	$Predictor_m$	$value_m$

Table 2 Column-wise vector schema

$VectorID$	$Predictor_1$	$Predictor_2$	$\cdots$	$Predictor_m$
1	$value_1$	$value_2$	$\cdots$	$value_m$
⋮	⋮	⋮	⋮	⋮
n	$value_1$	$value_2$	$\cdots$	$value_m$

Table 3 Row-wise vector schema

5.2 Cosine Similarity in SQL

The first step of assessing patient similarity calculations in-database, is to find out whether *SQL* offers the functions to implement the cosine similarity metrics. When looking at Equation 1, we notice that cosine similarity relies on square root, summation and multiplication. All of these mathematical operators/functions are part of the *SQL* standard, so that cosine similarity can be implemented within the *SELECT*-statement of an *SQL* query.

Moreover, the data table has to be self-joined to receive all pairs of vectors which can then be assigned the similarity value by the metric. For this purpose, it is important to note that all distance functions are **symmetric**, that is, $distance(v_1, v_2) = distance(v_2, v_1)$, for all patient vectors v_i. Thus, merely self-joining on a different *VectorID* would result in the Cartesian product and hence n^2 rows, given n patient vectors. What we really want is all possible combinations of patient vectors which means $\binom{n}{2} = \frac{1}{2}n(n-1)$ – roughly less than half of the Cartesian product. The simple solution to this issue is to join each patient vector with just vectors of a higher ID: the upper part over the diagonal can be ignored for the distance calculation such that we obtain a triangular similarity matrix as shown in Table 4. Depending on whether the column-wise or the row-wise vector schema is used, the SQL statements differ in particular regarding the amount of self-joining.

In the *column-wise* format (Table 2), groupings of tuples by *VectorID* make up one patient vector. In this case, the join cannot only be on higher *VectorIDs* but must also incorporate each predictor. The columnar schema hence introduces a significant overhead. In fact, it will require $2 \cdot n \cdot m$ join operations,

```
SELECT   v1.vid, v2.vid,
         --The dot product of two vectors
         SUM(v1.value*v2.value)
            /* Divided by the product of
             * the respective norms*/
            /(SQRT(SUM(v1.value*v1.value))
            *SQRT(SUM(v2.value*v2.value)))
            AS CosineSim
FROM     vectors v1 JOIN vectors v2
         ON v1.pid = v2.pid AND v1.vid < v2.vid
GROUP BY v1.vid,v2.vid;
```

Fig. 4 SQL code for Cosine similarity with column-wise schema

```
SELECT p1.id, p2.id
       (p1.pid_1 * p2.pid_1 +
        ...
        p1.pid_m * p2.pi_m)
       / (p1.norm * p2.norm)
FROM   patients p1
       JOIN patients p2
       ON p1.vid < p2.vid;
--norms calculated beforehand and
  added as a column to patient vector table
```

Fig. 5 SQL code for Cosine similarity with row-wise schema

that is, two join operations per row (*VectorID* and *PredictorID*) times n vector groupings of m predictors. Figure 4 shows the similarity calculation when utilizing the columnar schema. On the positive side, this version allows for the usage of the built-in aggregate function (the SUM function) for the summation involved in calculating the dot product as well as the respective vector norms. Nonetheless, it also displays the negative impact of the column schema, namely the required joining on *PredictorID and* bigger *VectorID* (here pid and vid respectively). We also tested a variant where the norm values ($||p||$ and $||q||$ in Equation 1) of each patient vector were **precomputed** and stored in a separate table. This avoids re-executing the squaring, summing and square root computations several times for the same patient vector. However this precomputation only had a negligible impact on the runtime.

In the *row-wise* representation (Table 3) of the vectors each tuple constitutes a patient vector. Hence, this case requires just n join operations. In this case, the precomputed norm values were added as an additional column to the table. Figure 5 shows the calculation query for the row-wise data model. As a downside, this approach cannot utilize aggregate functions (that is, SUM) and the user must explicitly specify every single predictor (that is, pid_i) of the patient vector in the part in which the dot product is determined. However, the advantage when it comes to joining lies in the fact that only one join condition is required.

```
SELECT p1.vid, p2.vid, sqrt(
       power(p1.pid_1 - p2.pid_1) +
        ...
       power(p1.pid_m - p2.pi_m)
       )
FROM   patients p1
       JOIN patients p2
       ON p1.vid < p2.vid;
```

Fig. 6 SQL code for Euclidean distance with row-wise schema

5.3 Euclidean Distance in SQL

Figure 6 shows the Euclidean distance calculation in SQL for the row-wise data schema; we refrain from testing the column-wise version due to its suboptimal performance when tested with the Cosine similarity. In analogy to the SQL code for Cosine similarity only one join condition on the *VectorID* is required. Again exploiting symmetry of the distance, we only compute a triangular distance matrix to avoid unnecessary computations; this is ensured by the inequality on the *VectorID*. The SQL code applies the built-in square root and power functions of the database systems.

6 Optimizations

The following sections will concern themselves with performance optimization of the similarity calculations. To this end the concepts of *batching*, *chunking* and *multithreading* will be introduced.

6.1 Batching

For our tests we used 32638 patient vectors extracted from the MIMIC-III dataset – each consisting of 73 predictor values; hence $\binom{32638}{2} = 532,603,203$ pairwise similarities have to be calculated. Naturally, the RAM in a computer is limited and relatively small when compared to mass storage like HDD or SSD. If we assume that our similarities are stored as double-precision floating-point decimals (64-bit), then these alone would take up:

$$\frac{\binom{32638}{2} \cdot 8}{2^{30}} \approx 3.97 \text{ GiB}.$$

To this we have to add the raw vector data which – assuming that 135 predictors are chosen – amounts to at least

$$\frac{\binom{32638}{2} \cdot 73 \cdot 8}{2^{30}} \approx 289.68 \text{ GiB}.$$

Therefore, we cannot expect that we will be able to obtain every similarity in one query since these numbers would exceed standard RAM capacities. This is

ID	1	2	3	$\cdots$	n-1	n
1	1	s	s	s	s	s
2	·	1	s	s	s	s
3	·	·	1	s	s	s
·	·	·	$\vdots$	1	s	s
n-1	·	·	·	·	1	s
n	·	·	·	·	·	1

Table 4 Triangular Similarity Matrix

where a technique we call *batching* comes into play: we divide the whole patient similarity calculation into equally sized bundles of index vectors that are then compared to all vectors of higher ID. Every intermediate result produced by a single batch is stored in the result table. Afterwards, the system can clear the RAM and load all necessary data for the next batch. This gets repeated until all similarities have been obtained.

The whole concept of batching can be regarded as a *for-loop* that starts at 1 and is increased by the batch size until the final ID has been reached, iterating over the similarity calculations. The declarative nature of *SQL* does not provide looping without substantial effort. We employ an external tool to manage the iteration in a *JDBC* application; the overhead induced by a *Java* program is negligible since it will more less only count up IDs and send commands to the database system. The major portion of the workload is still handled in-database.

The batch size can be adjusted to according to the system, as the presence of more RAM leaves more room for vectors and hence a higher batch size. However, since each patient vector is only compared to vectors of higher ID, there is a discrepancy in how many values are produced by a single batch. This discrepancy gets bigger, the more we get away from the mean ID. While the vector with ID 1 is paired with all other $n-1$ vectors, ID $n-1$ is only paired with the last ID n; ID n itself is then need not be paired at all.

6.2 Chunking

We now address the issue of unequally distributed load caused by batching. If we were to put all similarities in a matrix, the resulting matrix would be symmetric as a consequence of the symmetry in the distance function mentioned in Section 5.2. Therefore, we do not need the lower left triangle of the matrix as well as the main diagonal because there is no use in comparing vectors with themselves in our context. This fact is illustrated in Table 4. With increasing ID, the proportion of a row that were are interested in gets shorter, namely $n-i$ for ID_i and n patients. Wrapping this in a sum, we receive our total number of similarities: $\sum_{i=1}^{n} n-i = \binom{n}{2}$.

In order to prevent this skewed assignment and obtain a balanced distribution of similarity calculations per patient vector, we introduce a new concept called *chunking*. If we recall that $\binom{n}{2} = \frac{1}{2}n(n-1)$ is another way of writing

ID	1	2	3	...	$1+\lceil\frac{(n-1)}{2}\rceil$	...	n-1	n
1	1	s	s	s	s	·	·	·
2	·	1	s	s	s	s	·	·
3	·	·	·	·	1	s	s	s
⋮	·	·	·	·	·	1	s	s
n-1	s	s	·	·	·	·	1	s
n	s	s	s	s	·	·	·	1

Table 5 Balanced Similarity Matrix with Chunking

our total amount of similarities needed, it becomes apparent that to achieve a balanced distribution among n patient vectors we need $\frac{n-1}{2}$ comparisons per vector.

Since IDs are integers, we have to take care of odd numbers when dividing by two. Therefore, we can round up, $\lceil\frac{(n-1)}{2}\rceil$, if an ID is odd, and round down, $\lfloor\frac{(n-1)}{2}\rfloor$, if an is ID even. For the next steps, we split the n IDs into their corresponding n similarity lists, that is, the list of IDs they are going to be compared to:

– for odd ID i, compute similarities with IDs $j \in \{i+1,\ldots,i+\lceil\frac{(n-1)}{2}\rceil\} \cup \{1,\ldots,i+\lceil\frac{(n-1)}{2}\rceil-n\}$
– for even ID i, compute similarities with IDs $j \in \{i+1,\ldots,i+\lfloor\frac{(n-1)}{2}\rfloor\} \cup \{1,\ldots,i+\lfloor\frac{(n-1)}{2}\rfloor-n\}$

The result can be seen in Table 5 showing a uniform assignment of similarity calculations per ID.

6.3 Multithreading

Many renowned database systems (including *MySQL*, *Oracle*, and *PostgreSQL*) only utilize one thread per database connection. Yet, the average CPU consists of $2-8$ individual cores; it would hence be desirable to make use of them all, in order to achieve the highest possible performance. However, multithreading can also be a twofold affair. On the one hand programs can benefit a lot from parallel execution of certain execution threads while on the other hand it induces significant coordination overhead; moreover, not all processes are geared for parallelization [14]. In theory, our similarity computations should be highly parallelizable since the distance between two vectors does not depend on any other value that is gained throughout the whole process. Nevertheless, we have to fetch all vector data from disk and this I/O barrier cannot be broken by multithreading.

In order to achieve multithreading on DBMS, we exploit connection pooling. Establishing a database connection is time- and resource-consuming. Connection pools help by providing a cache of database connections that can be accesses and released in a short amount of time. Thus, our Java program creates a connection pool and then allocates as many connections, and therefore threads, as the system includes cores to our similarity calculations. Since *chunking* will equally distribute the workload, it is safe to assume that dividing n vectors into n/k groups of vectors, where k is the amount of cores/threads available in the system, will yield the best result. For example, let n be 40000 and k be 8, then each vector will be compared with $(40000 - 1)/2 = 19999.5$ other patients on average. These comparisons get distributed in eight threads, each handling the similarity calculations of 5000 vectors.

7 Results

In this section the results achieved by the settings described in the previous sections are presented and discussed. The performance optimizations are presented in the order they were conceived. As a unit of measurement milliseconds per patient (ms/p) is introduced, that is, the time it takes to perform all calculations assigned to one individual vector in the data set. Milliseconds per patient are calculated by dividing the total time needed to run the similarity computation over the whole data set by the amount of vectors in it.

7.1 Database Systems and Data Mining Tools

All computations were executed on a machine with 64 GB of RAM, an *Intel i7-7700k*, and an SSD of 500 GB. *PostgreSQL* version 10.1 as an open source row store and *MonetDB* version 11.27.11 as an open source column store were used. Batching, Chunking and multithreading were managed by a Java program setting up the database connection and issuing the SQL queries as shown in Section 5. In the first tests, we focus on data set I (the ICU data set MIMIC-III). The data set II (the diabetes data set) is used for validation in Section 7.5.

In order to compare our in-database approach, we tested several external data mining tools that offer comprehensive distance libraries. Unfortunately, R showed extremely poor performance when executed on our data sets when using the package *philentropy* provided by CRAN [9]. That is why we looked for alternatives. We chose the two fastest Java-based data mining tools from our tests (ELKI [10,28] and Apache Mahout [2]). Setting up these two tools required us to acquire an in-depth understanding to the internal workings of their data models and hand-coding the transformation into the in-memory representation.

Mahout and Elki are data mining frameworks that can be included as a dependency in every Java project with Apache Maven. The implementation for both tools is similar. Firstly, the program connects via JDBC to MonetDB fetching all patients' data in a result set. In order to calculate the Euclidean distance and Cosine similarity, we need to create patient vectors. This is why we iterate over the result-set to put each feature in a vector which will be

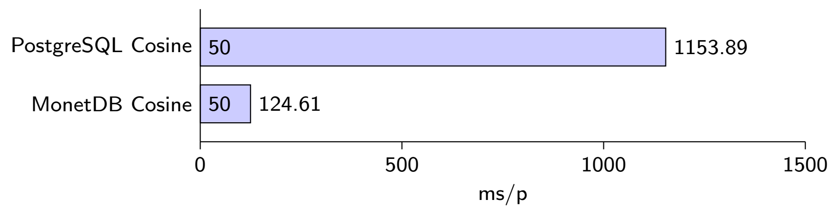

Fig. 7 Column Format

stored in an in-memory hashmap holding the row ID as key and the patient vector as value.

Further, we iterate over the hashmap, create eight threads, and assign each of the eight threads the patient keys to handle. In more detail, each thread creates its own CSV file, calculates the distances between the assigned patients and their chunks, and writes the results into a CSV file with the format

```
(uniqueid1, uniqueid2, distance/similarity).
```

To obtain the Cosine similarity, Mahout and ELKI both calculate cosine similarity as $1 - \mathit{CosineDistance}$; hence we implicitly call the appropriate distance implementations. For the Euclidean distances the corresponding implementations are used. The number of patients each thread is handling depends on the previously set batching size (which in our case is 50 or 100). Meanwhile, another thread is started by the main process to connect via JDBC to MonetDB; it copies all finished CSV-files into the appropriate result table. This procedure is repeated until all patients in the hashmap are processed by one of the eight threads.

7.2 Column vs Row Format

We first analyzed the question whether the extra step of converting column-oriented vector groupings into their row-wise counterparts is worthwhile. Figure 7 provides the baseline results for both, *PostgreSQL* and *MonetDB*. The bar chart (and all following) have to be read as follows: the number in the bar itself indicates the **batch size** (either 50 or 100 patients at a time), the x-axis is the milliseconds per patient scale and the y-axis states the respective database systems.

As expected the columnar schema works better with the column store system. In fact, MonetDB is actually faster by an order of magnitude compared to *PostgreSQL*, or to be more precise, it exhibits an eightfold advantage in speed. This can be explained by the great amount of joining that is demanded by the columnar approach (see Section 5.2). A look at *PostgreSQL*'s query plan revealed that it utilizes a costly hash-join. The *PredictorID* for each row is hashed into a map and then used to determine where to join. The total cost is stated in terms of disk pages that have to be fetched, here 4245993.31. Unfortunately, *MonetDB*'s query plan does not provide a cost estimate but

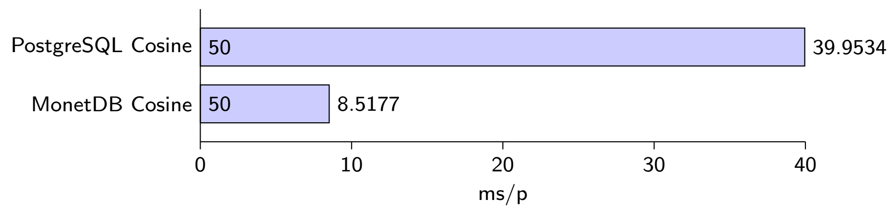

Fig. 8 Row Format

from the result it is apparent that the internal columnar storage helps to leverage performance when the data is also stored in a columnar fashion.

The more intriguing question is how *MonetDB* will fare when confronted with the row-wise representation of patient vectors. The results for the row schema are presented in Figure 8.

Most noticeably, *MonetDB* is still ahead of *PostgreSQL*. However, the milliseconds per patient for both system have improved drastically: 98.95% for *PostgreSQL* and 98.15% for *MonetDB* (compared to Figure 7). Thus, when the column-wise computations took in total 4.26 *hours* in *MonetDB* and 34.63 *hours* in *PostgreSQL*, in row-wise format they now require 4.63 *minutes* in *MonetDB* and 21.73 *minutes* in *PostgreSQL*, respectively. This can also be seen in the cost that the *PostgreSQL* query planner now estimates to be 40687.33 pages to fetch; a decrease of 99%. This is mainly due to the fact that *PostgreSQL* can now utilize indices on the vector IDs that only require a sequential scan when joining. As a consequence, we can infer that the computation run-times correspond to the estimated cost by the query planner. Furthermore, we can conclude that the similarity calculations themselves have very little weight in the whole process compared to the execution of the join.

At this point, it is reasonable to abandon the column-wise schema due to its obvious speed limitations. The benefits we expect from multithreading and *chunking* are not going to be able to make up for the high deficit created by the column format. Therefore, only the row-wise data model was considered for all following performance optimizations.

7.3 Multithreading and Chunking

We already raised the question whether multithreading pays off – because our computations require a lot of disk I/O. Especially, after the first tests established that the join part of the computations takes up most of the processing time, we might assume that, due to its reliance on disk data, multithreading might not give us a lot of improvements. Our test results in Figure 8 present a different picture, though. Note that for *PostgreSQL* the multithreading was implemented by connection pooling as described in Section 6.3.

When utilizing a batch size of 50 patients at a time, *PostgreSQL* again improves by 15.72% (compared to Figure 8). A batch size of 100 was also tested to see if this positive effect could be utilized to an even higher degree –

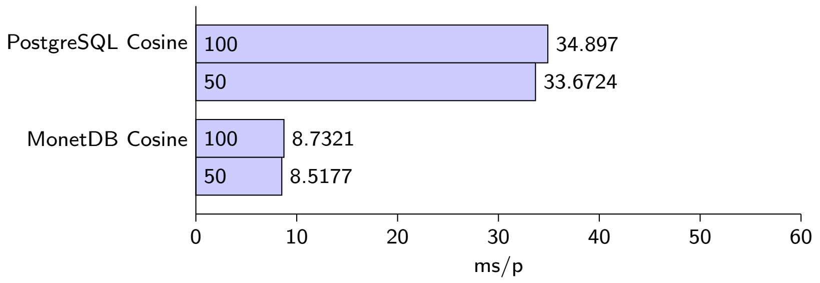

Fig. 9 Row Format Multi-Threaded

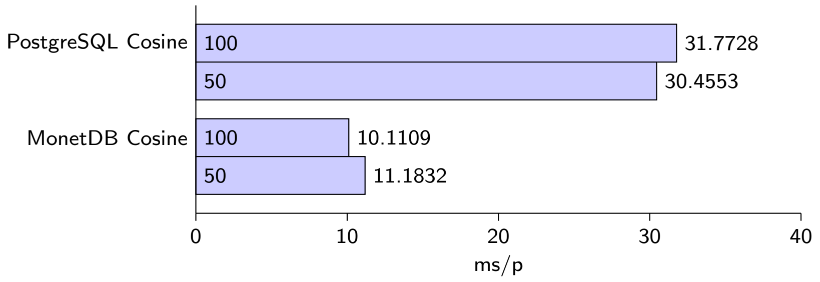

Fig. 10 Row Format Multi-Threaded w/ Chunking

but again a batch size 50 proves to be the sweet spot. In contrast, *MonetDB* does not display any improvements or degradations at all, even at a different batch size. Most likely, having all threads access the disk in close succession reduces the amount of time disk was waiting for write and read instructions. Specifically, having the benefit of loading a lot of data into RAM at the same time takes off load from the disk and the complex joins which are independent of each other and therefore greatly parallelizable.

MonetDB was created with data analytics in mind. That is why it already performs well without any user-induced multithreading and hence already utilized multiple threads on its own for various tasks. *Chunking* on the contrary is independent from any parallelization efforts and could potentially also show benefits on MonetDB-based system. However, in Figure 10 we show another contrasting picture: Performance has degraded by 23.83% in *MonetDB* compared to only the multithreaded approach in Figure 9. In contrast, *PostgreSQL* has improved by 9.55%; yet, still taking about three times as much time as *MonetDB* needs. The control logic required to take advantage of *chunking*, naturally, produces some slight overhead. In the case of *MonetDB* it seems that this overhead is actually interfering with the computation process while *PostgreSQL* can benefit from the more balanced workload by chunking.

Chunking	Cosine total (min.)	Cosine ms/patient	Euclidean total (min.)	Euclidean ms/patient
PostgreSQL 100	17.2834	31.7728	19.0	34.9286
PostgreSQL 50	16.5667	30.4553	18.0833	33.2435
MonetDB 100	5.5	10.1109	6.9334	12.7458
MonetDB 50	6.08334	11.1832	6.2834	11.5509
Mahout 100	4.1605	7.6484	4.1348	7.6012
Mahout 50	4.2573	7.8264	4.2449	7.8036
ELKI 100	4.3376	7.9740	4.2974	7.9001
ELKI 50	4.5007	8.2739	4.3535	8.0032

Table 6 Runtime Comparison of Database Systems and Data Mining Tools with Chunking

No Chunking (triangular)	Cosine total (min.)	Cosine ms/patient	Euclidean total (min.)	Euclidean ms/patient
MonetDB 100	4.75	8.7321	5.6	10.2947
MonetDB 50	4.6334	8.5177	5.0167	9.2338
Mahout 100	4.49645	8.26604	4.3578	8.01115
Mahout 50	4.44634	8.17392	4.39976	8.08829
ELKI 100	4.43433	8.15184	4.39637	8.08206
ELKI 50	4.558	8.37918	4.52528	7.81812

Table 7 Runtime Comparison of Database Systems and Data Mining Tools without Chunking

7.4 Comparison of Database Systems and Data Mining Tools

In this section we compare the best setting for each of the database systems with the two data mining tools ELKI and Apache Mahout. In addition we also computed the Euclidean distance with each system as an alternative to the Cosine similarity. Tables 6 and 7 show the exact runtime measurements (averaged over several runs). Figure 11 visualizes these measurements by comparing the best setting for each system (chunking for Postgres, ELKI and Mahout, no Chunking for MonetDB). We make the following observations:

– Chunking also improves the runtime of both data mining tools. Our multi-threaded implementation using ELKI and Mahout benefits from the more balanced workload due to chunking.
– MonetDB performance is competitive with the performance of the two data mining tools.
– The Euclidean distance calculation in the database systems is more time-consuming than the Cosine similarity, while this is the other way round for the data mining tools. Using the built-in power and square root functions seems to be the reason for this slower execution. When the exact Euclidean distance values are not required, using the squared Euclidean distance might improve this situation because the square root computation can be avoided.

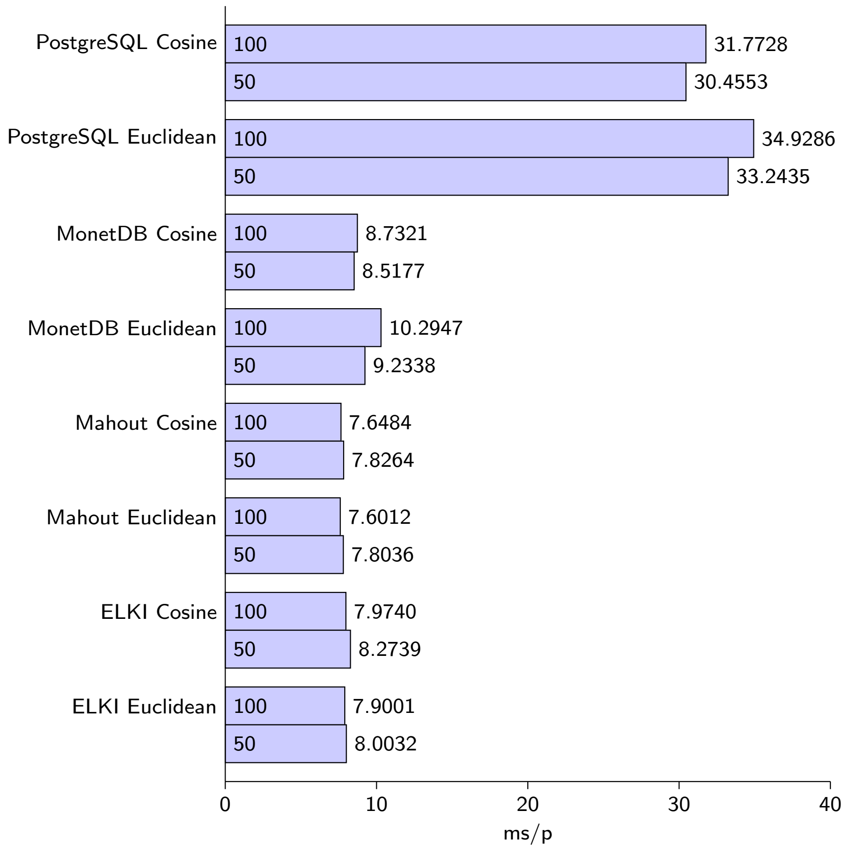

Fig. 11 Row Format Multi-Threaded w/ Chunking for Postgres, ELKI and Mahout, w/o Chunking for MonetDB

7.5 Validation with Larger Data Set

In order to validate our approach we executed the same tests with the second data set containing diabetes-related patient data.

We repeated the test runs for the row format with this validation data set. The column format was not considered for the validation because it showed significantly less performance. The test runs with the larger validation data set verify our previous results. Figure 12 shows the influence of different batch sizes (150, 100 and 50 patients at a time). Comparing to Figure 9, *PostgreSQL* needed significantly more time per patient when processing the larger validation data set. In contrast, *MonetDB*'s performance per patient remained nearly uninfluenced by the size of the data set.

Figure 13 shows the performance with additional chunking applied. For *PostgreSQL*, chunking again improved the runtime per patient – although comparing to Figure 10 the larger data size still shows its impact. Batch size 50 gave optimal run time performance when using *PostgreSQL* with chunking.

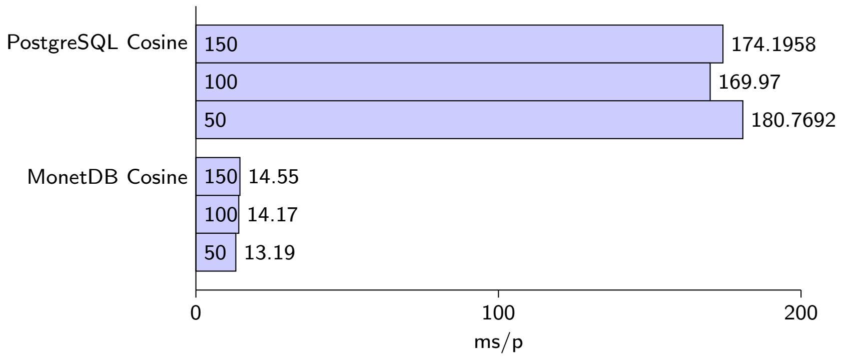

Fig. 12 Row Format Multi-Threaded Diabetes

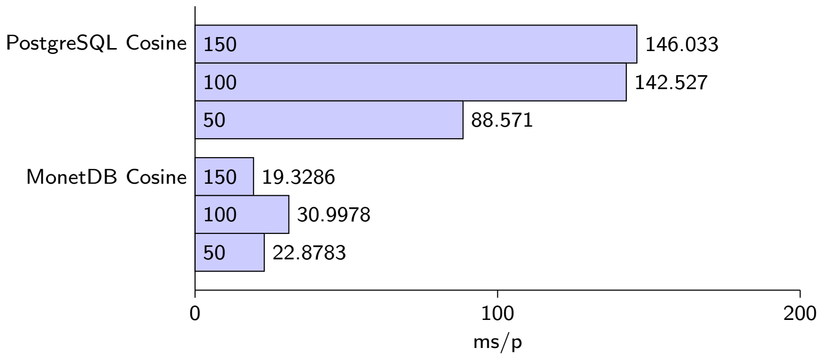

Fig. 13 Row Format Chunking Diabetes

Indeed decreasing the batch size further (to 25 patients at a time) led again to an increase of the runtime. Most notably chunking for batch size 50 resulted in a performance improvement of roughly 50% (comparing Figures 12 and 13). Due to our chunking approach is able to scale much better with the larger data set size. Yet again, for *MonetDB* chunking did not prove to be beneficial and slightly increased the runtime. *MonetDB*'s internal task scheduling seems to work best when only applying batching.

8 Conclusion

In this article, we provided an in-depth investigation of in-database patient similarity analysis on two real-world data sets. We introduced and tested several optimizations. We compared one similarity measure and one distance metric with a column store, a row store and two data mining tools. As stated in the related work section, current investigations on patient similarity analysis mainly focus on prediction accuracy rather than computational performance. Applying our optimization approaches will speed up the pairwise patient similarity calculations which is the prerequisite of measuring the prediction accuracy. Furthermore, different patient similarity metrics could be used for different prediction approaches; our method can be used as a good basement for testing the effect of similarity metrics on different prediction models. Specifically, the predictive models that require the use of high-dimensional predictors and probably a big dataset will benefit when using our method.

To summarize our results, for the row store database system *PostgreSQL* a performance gain (in the best case of around 50%) could be achieved by several optimizations. The column store *MonetDB*, in general performed faster in comparison, no matter which optimization method was applied to it. The column store performance was competitive with the two tested data mining tools. While systems like *MonetDB* provide good out-of-the-box performance when it comes to analytical tasks, they lack multi-user and transaction support. These features are, however, crucial in real-world applications where multiple users input and change data on a regular basis. Concerning these practical requirements, a DBMS like *PostgreSQL* still be might be more suited than *MonetDB*. Therefore, a reduction of the time required for patient similarity calculations in such a DBMS can be regarded a helpful contribution to the overall process of patient similarity analysis. Moreover, we observed that the similarity calculations are dominated to the greatest extent by the amount of self-joins; we hence conjectured that the selected similarity metrics has little impact on the overall performance of the analysis. We confirmed this conjecture by comparing Cosine similarity and Euclidean distance.

Benefits of in-database similarity analytics can be summarized as follows:

– SQL is a platform-independent language that can be executed unchanged with numerous database systems. The simplicity of similarity calculations in SQL reduce the risk of unwanted coding errors. In contrast, there is no standardized way to interact with the data mining tools: programming skills for each of the tools have to be acquired before being able to use them; this may lead to a kind of lock-in effect that hinders switching and comparing different tools.
– Similarity calculations in SQL can easily be adjusted for each of the features (for example giving different weights to each feature or also considering categorical values in combination with numerical values). In contrast, using the fixed interfaces of the data mining tools do not allow a flexible adaptation of similarity calculations.
– Databases offer a reliable storage engine that can easily access data stored on disk. In contrast, with the data mining tools extraction of data and transformation into an in-memory representation is needed.
– Multithreading is offered by MonetDB as a built-in feature. In contrast, thread handling for the data mining tools must be implemented by hand.

In ongoing work we currently use the different similarity and distance values precomputed in the database systems to assess the effect of the choice of similarity/distance on the accuracy of disease predictions. We are also applying feature selection and dimensionality reduction on both data sets to filter out the features relevant for the predictions.

Future investigations might for example include further optimizations of the join process – potentially following the line of Qin and Rusu [25] who developed a dedicated dot-product join operator for relational database systems that can improve the processing of the Cosine similarity. Furthermore, it might be worthwhile to contrast the presented relational approach with patient similarity analysis in several non-relational data models [35].

References

1. Anthony Celi, L., Mark, R.G., Stone, D.J., Montgomery, R.A.: "Big data" in the intensive care unit. Closing the data loop. American Journal of Respiratory and Critical Care Medicine **187**(11) (2013)
2. Apache Mahout Committers: Apache Mahout. https://mahout.apache.org
3. Brown, S.A.: Patient similarity: Emerging concepts in systems and precision medicine. Frontiers in physiology **7** (2016)
4. Cabrera, W., Ordonez, C.: Scalable parallel graph algorithms with matrix–vector multiplication evaluated with queries. Distributed and Parallel Databases **35**(3-4), 335–362 (2017)
5. Chaudhuri, S., Dayal, U.: An overview of data warehousing and olap technology. ACM Sigmod record **26**(1), 65–74 (1997)
6. Deza, M.M., Deza, E.: Encyclopedia of distances. Springer Science & Business Media (2012)
7. Dheeru, D., Karra Taniskidou, E.: UCI machine learning repository (2017). URL http://archive.ics.uci.edu/ml
8. Domínguez-Muñoz, J.E., Carballo, F., Garcia, M.J., de Diego, J.M., Campos, R., Yangúela, J., de la Morena, J.: Evaluation of the clinical usefulness of apache II and saps systems in the initial prognostic classification of acute pancreatitis: a multicenter study. Pancreas **8**(6), 682–686 (1993)
9. Drost, H.G.: R philentropy package. https://cran.r-project.org/web/packages/philentropy/philentropy.pdf
10. Elki development team: ELKI: Environment for Developing KDD-Applications Supported by Index-Structures. https://elki-project.github.io/
11. F, F., D, B., A, B., C, M., J, V.: Serial evaluation of the SOFA score to predict outcome in critically ill patients. JAMA **286**(14), 1754–1758 (2001)
12. Garcelon, N., Neuraz, A., Benoit, V., Salomon, R., Kracker, S., Suarez, F., Bahi-Buisson, N., Hadj-Rabia, S., Fischer, A., Munnich, A., et al.: Finding patients using similarity measures in a rare diseases-oriented clinical data warehouse: Dr. warehouse and the needle in the needle stack. Journal of biomedical informatics **73**, 51–61 (2017)
13. Gottlieb, A., Stein, G.Y., Ruppin, E., Altman, R.B., Sharan, R.: A method for inferring medical diagnoses from patient similarities. BMC medicine **11**(1), 194 (2013)
14. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**(7) (2008)
15. Hoogendoorn, M., el Hassouni, A., Mok, K., Ghassemi, M., Szolovits, P.: Prediction using patient comparison vs. modeling: A case study for mortality prediction. In: Engineering in Medicine and Biology Society (EMBC), 2016 IEEE 38th Annual International Conference of the, pp. 2464–2467. IEEE (2016)
16. Johnson, A.E., Pollard, T.J., Shen, L., Lehman, L.w.H., Feng, M., Ghassemi, M., Moody, B., Szolovits, P., Celi, L.A., Mark, R.G.: MIMIC-III, a freely accessible critical care database. Scientific data **3** (2016)
17. Le Gall, J.R., Lemeshow, S., Saulnier, F.: A new simplified acute physiology score (SAPS II) based on a european/north american multicenter study. Jama **270**(24), 2957–2963 (1993)
18. Lee, J., Maslove, D.M., Dubin, J.A.: Personalized mortality prediction driven by electronic medical data and a patient similarity metric. PloS one **10**(5), e0127428 (2015)

19. Li, L., Cheng, W.Y., Glicksberg, B.S., Gottesman, O., Tamler, R., Chen, R., Bottinger, E.P., Dudley, J.T.: Identification of type 2 diabetes subgroups through topological analysis of patient similarity. Science translational medicine **7**(311), 311ra174–311ra174 (2015)
20. Morid, M.A., Sheng, O.R.L., Abdelrahman, S.: PPMF: A patient-based predictive modeling framework for early ICU mortality prediction. arXiv preprint arXiv:1704.07499 (2017)
21. Ordonez, C.: Statistical model computation with udfs. IEEE Transactions on Knowledge and Data Engineering **22**(12), 1752–1765 (2010)
22. Ordonez, C., Cabrera, W., Gurram, A.: Comparing columnar, row and array dbmss to process recursive queries on graphs. Information Systems **63**, 66–79 (2017)
23. Park, Y.J., Kim, B.C., Chun, S.H.: New knowledge extraction technique using probability for case-based reasoning: application to medical diagnosis. Expert Systems **23**(1), 2–20 (2006)
24. Passing, L., Then, M., Hubig, N., H. Lang, M.S., Günnemann, S., Kemper, A., Neumann, T.: Sql-and operator-centric data analytics in relational main-memory databases. In: EDBT, pp. 84–95 (2017)
25. Qin, C., Rusu, F.: Dot-product join: Scalable in-database linear algebra for big model analytics. In: Proceedings of the 29th International Conference on Scientific and Statistical Database Management, p. 8. ACM (2017)
26. Raasveldt, M., Holanda, P., Mühleisen, H., Manegold, S.: Deep integration of machine learning into column stores. In: EDBT, pp. 473–476. OpenProceedings.org (2018)
27. Saeed, M., Villarroel, M., Reisner, A.T., Clifford, G., Lehman, L.W., Moody, G., Heldt, T., Kyaw, T.H., Moody, B., Mark, R.G.: Multiparameter intelligent monitoring in intensive care II (MIMIC-II): a public-access intensive care unit database. Critical care medicine **39**(5), 952 (2011)
28. Schubert, E., Koos, A., Emrich, T., Züfle, A., Schmid, K.A., Zimek, A.: A framework for clustering uncertain data. Proceedings of the VLDB Endowment **8**(12), 1976–1979 (2015)
29. Sharafoddini, A., Dubin, J.A., Lee, J.: Patient similarity in prediction models based on health data: a scoping review. JMIR medical informatics **5**(1) (2017)
30. Strack, B., DeShazo, J.P., Gennings, C., Olmo, J.L., Ventura, S., Cios, K.J., Clore, J.N.: Impact of hba1c measurement on hospital readmission rates: analysis of 70,000 clinical database patient records. BioMed research international **2014** (2014)
31. Sun, J., Sow, D., Hu, J., Ebadollahi, S.: A system for mining temporal physiological data streams for advanced prognostic decision support. In: Data Mining (ICDM), 2010 IEEE 10th International Conference on, pp. 1061–1066. IEEE (2010)
32. Vincent, J.L., Moreno, R., Takala, J., Willatts, S., De Mendonça, A., Bruining, H., Reinhart, C., Suter, P., Thijs, L.: The SOFA (sepsis-related organ failure assessment) score to describe organ dysfunction/failure. Intensive care medicine **22**(7), 707–710 (1996)
33. Wang, F., Hu, J., Sun, J.: Medical prognosis based on patient similarity and expert feedback. In: Pattern Recognition (ICPR), 2012 21st International Conference on, pp. 1799–1802. IEEE (2012)
34. Wang, S., Li, X., Yao, L., Sheng, Q.Z., Long, G., et al.: Learning multiple diagnosis codes for ICU patients with local disease correlation mining. ACM Transactions on Knowledge Discovery from Data (TKDD) **11**(3), 31 (2017)
35. Wiese, L.: Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases. DeGruyter/Oldenbourg (2015)