Modular Data Clustering - Algorithm Design beyond MapReduce

Martin Hahmann, Dirk Habich, Wolfgang Lehner

Technische Universität Dresden
Department of Computer Science
Database Technology Group
01062 Dresden
{martin.hahmann; dirk.habich; wolfgang.lehner} @tu-dresden.de

ABSTRACT

In the context of Big Data, flexible and adjustable data analytics become more and more important, whereas an efficient, scalable and fault-tolerant execution is required as well. To fulfill the flexibility as well as the execution requirements, the specification of the analysis methods have to be in an appropriate and easy adjustable manner. The MapReduce approach has demonstrated that such flexible specification as well as scalable execution is possible and applicable. However, the MapReduce programming model is too generic and complicates the specification from a data analysis point of view. Therefore, we propose a novel programming approach using well-defined modular building blocks for a specific and highly utilized data analysis domain named data clustering in this paper. Our approach offers many advantages: (i) a unified and specific instruction set for data clustering which eases understanding and algorithm adaptation in an abstract way, and (ii) enables an efficient and scalable execution of all data clustering algorithms based on an efficient mapping of the unified instruction set to a specific target environment is possible.

1. INTRODUCTION

In order to efficiently process and analyze massive data, highly scalable parallel data processing platforms have been developed [16]. In this area, MapReduce [7] is a well-establish programming and execution framework. A MapReduce cluster is able to scale to thousands of commodity computer nodes in a fault-tolerant manner. Furthermore, the programmers can parallelize their applications in an easy way by implementing map and reduce functions to transform and aggregate data, whereas the underlying data structure consists of *(key,value)* pairs. As shown in different application domains, many algorithms fit perfectly into the MapReduce model, such as word counting in information retrieval or equi-join queries in databases.

Generally, the success of MapReduce is based on the simple and flexible programming model with the ability to execute the application in a highly parallel and fault-tolerant fashion [7, 16]. Nevertheless, the MapReduce model has several shortcomings. For example, one drawback of MapReduce is the missing support for iterative computations. However, many data analysis techniques require those iterative computations, therefore Bu et al. [4] have proposed an extension to MapReduce. A second drawback is that MapReduce itself does not support any high-level language like SQL in database systems. Users always have to code their operations in map and reduce functions, whereas they have to consider the *(key, value)* data structure. Mapping of individual data structures to *(key, value)* pairs is not always trivial [2]. To overcome this issue, a variety of projects aim at providing higher-level interfaces. An example is Jaql[1] as a declarative query interface with rich data processing features such as transformation, filtering, join processing, grouping and aggregation. The Jaql scripts are automatically compiled and executed as MapReduce jobs.

However, the available higher-level interfaces focus on data transformation and processing tasks. To support deeper analytics as necessary for Big Data, Das et al. [6] integrate the statistical analysis system R[2] in the MapReduce system Hadoop. This integration is done on the language level by combining Jaql with R scripts as well as on the execution level using a R-Jaql bridge between R and Hadoop. The advantage of this approach is the efficient utilization of the rich functionalities of R for analytical task. From a usability point of view, this integration fits perfectly, because users can immediately start deep analytics on their data using standard functions. The disadvantage of this approach is availability of two different runtime infrastructures which have to interact to determine a result. Furthermore, this language approach of Jaql combined with R is not well suited for the specification of new analysis methods in a MapReduce style, so that these methods are finally scalable on a highly parallel platform.

To tackle the flexible specification or engineering of analytical methods for large scale analytics, we propose an alternative approach with regard to modular algorithm design inspired by MapReduce. We illustrate our approach using algorithms from data mining, in particular from the data clustering domain. Fundamentally, data clustering is a highly applicable analysis method that is used to reduce the amount of data or to gain understanding and acquire novel, previously unknown knowledge. The task of data clustering is to partition a set of objects into groups—so called clusters—in a way that similar objects are put in the same cluster, while dissimilar objects are located in different clusters. To determine such cluster-

[1]https://code.google.com/p/jaql/
[2]http://www.r-project.org

ing result, a large algorithmic landscape has been established. To get an idea of this landscape, in terms of size, we looked through the proceedings of the SIGKDD and ICDM data-mining conference from 2005 and 2012 and counted more than 120 papers introducing new algorithms or variants resp. optimization of existing techniques. This multitude of algorithms exists because it is impossible to design an algorithm that automatically produces optimal results for any data set or application, thus a lot of techniques are highly specialized and custom-made for specific scenarios or types of data sets.

Our Contribution

In this paper, we present our modular language approach for the description and specification of clustering algorithms. Our design principles are (i) to establish a unified view on clustering algorithms using a compact set of instructions and (ii) to hide details of parallel execution on the programming level as successfully practiced by MapReduce. Both design principles enable users to focus only on the clustering analytical part, without considering the efficient execution on a scalable data processing platform. Generally, our approach is based on a mathematical formulation and utilizes a *matrix* concept for the unified representation of all data aspects. As a result, clustering specific operations are expressed as functions over matrices. From a language perspective, this eases understanding and engineering of clustering algorithms and allows their comparison. Furthermore, our modular approach offers an efficient way to adapt clustering algorithms. From a execution perspective, several different execution and optimization strategies are possible.

Outline

In Section 2, we review essentials of data clustering by decomposing clustering algorithms into conceptual components. These conceptual components are concretized in Section 3 and 4 with a data model, building blocks in the form of matrix functions and control-flow structures. In Section 5, we demonstrate how clustering algorithms are transcribed using our approach. For this, we choose algorithms from different clustering classes to emphasize the wide-range applicability of our approach. The contributions of our approach and its potentials are described in detail in Section 6. The future development of our concepts is described in Section 7, before we review existing related work in Section 8. The paper closes with a short summary in Section 9.

2. ESSENTIALS OF DATA CLUSTERING

To reach our goal of a unified and specific instruction set for clustering algorithms, we first need to decompose the corresponding algorithms into their conceptual components. This decomposition concentrates only on the core clustering procedure and does not consider pre- and post-processing tasks like feature selection, data cleansing and so on. As starting point, we assume the a general definition of data clustering [12]: *"Data clustering is the partitioning of a set of points into groups—so called clusters—in a way that similar points are put in the same cluster, while dissimilar points are located in different clusters."*

From this definition, we can derive certain fundamental tasks that have to be performed in order to generate a clustering. The first fundamental task an algorithm needs to fulfill, is to measure the similarity of points. This is a prerequisite for the second task, which is to explicitly choose the points that are similar and should be grouped together. The actual grouping of points, forms the third and final task that must be executed in order to create a clustering.

2.1 Basic Elements

The three identified tasks, which are observable in all clustering algorithms, are independent and have to be executed in sequence. As result of this abstract consideration, we define the *phases* of a clustering algorithm, that form the frame for our algorithm representation. This general frame needs to be fitted with additional building blocks to complete the description of an algorithm. In the following, we introduce each phase and investigate its basic elements, in order to find these blocks.

Evaluation Phase

During this first phase, the similarity between all points or between all points and some set of references is measured. For the determination of similarity between two objects, a dedicated function is necessary. In data clustering, there are two general approaches: (i) similarity functions and (ii) distance functions [12]. While the former express the degree of equality, the latter point out the amount of disagreement between objects. As both options are analogous, we assume that similarity is expressed through distances, without losing generality. Based on this, *distance measure* becomes the first basic element of the evaluation phase.

A distance measure takes at least two values as input and outputs one value. Obviously, in data clustering one input are the *points* which are to be clustered. The second input offers some kind of variability. On the one hand, there exist algorithms like DBSCAN [8] that calculate all point-to-point distances and thus use *points* also as second input. On the other hand, approaches like k-means [10] employ a special set of representatives/centroids as second input for distance computation. To combine both alternatives, we introduce the term *references* for the second input of the *distance measure*. The *references* can be (i) equal to *points*, (ii) a subset of *points* or (iii) a set of objects that are not part of *points* but share its feature-space. Following this, we add these two inputs to the set of basic elements for the evaluation phase.

The output of the distance measure consists of *distances* which is the next basic element. With this, we identified the four basic elements of the evaluation phase *points, references, distances, and distance measure* that allow a more specific definition of its task. In essence, during evaluation the distance measure is used to create a relation between points and references that represents their similarity and is explicitly expressed in the form of distances. It is important to mention that the distance itself is not the only result of evaluation, but the relation-triple *point-distance-reference*.

Selection Phase

In this phase, the points that are eligible to be grouped together are selected according to the algorithms specification. For this, the *point-distance-reference* triples generated by evaluation are taken as an input. Referring to our initial clustering definition, the goal is that only points which are similar should overcome the selection process in order to be clustered together. Therefore, it is necessary to define the constraints to acquire the status "similar" and to test all points whether they fulfill these or not. For this, we propose *filters*, which represent the basic element of this phase. Utilizing a set of *filters*, the selection phase tests each *point-distance-reference* triple coming from evaluation and only passes on those that comply.

Association Phase

During this phase, the previously chosen points are associated with a cluster. The input of the association phase is a set of *point-distance-reference* triples that passed the selection phase and have to be grouped together to create a clustering. This *clustering* forms the output of this phase and qualifies as basic element. Like the

Phase	Input	Processing	Output
Evaluation	Points, References	Distance Measure	Point-Distance-Reference Triples
Selection	Point-Distance-Reference Triples	Filters	Point-Distance-Reference Triples
Association	Point-Distance-Reference Triples	Association Function	Adjacencies (Point-Reference Tuples)
Optimization	-	-	-

Figure 1: Overview Clustering Algorithm Phases and Basic Elements.

similarities from evaluation, a clustering is made up of relations i.e. the affiliation between points and clusters. That means during the association phase *point-distance-reference* triples have to be transformed into *point-cluster* tuples. To perform this task, we define an *association function* as basic element.

For some algorithms, this is already enough to create a clustering e.g., the *association function* of k-means effectively takes a *point-distance-reference* triple, removes the distance and adopts the reference as cluster. But not all clustering techniques work in that way. For example, DBSCAN [8] first associates a core-object with its neighborhood–by creating *point-reference* tuples–before the the actual clusters are formed on the basis of overlapping neighborhoods. Additionally, the direct creation of point-cluster tuples in DBSCAN is prevented by the fact that clusters are not known in advance. This issue necessitates a stopover between association function and clustering, which forms our next basic element: *adjacencies*. With it, we describe the association phase as follows: incoming *point-distance-reference* triples are transformed by the *association function* into *adjacencies* from which the clustering is derived. The transition from adjacencies to clustering is a part that can be done in a variety of ways, which is why we do not appoint basic elements for it on this conceptual level. Doing so would result in a substantial set of basic elements, that would be contradictory to our goal of finding only fundamental components. However, we solve this problem in a later section.

Optimization Phase

This fourth phase originates from analyzing existing algorithms that often feature parameter adjustments and target function maximization leading to multiple iterations of the first three phases. As the name optimization implies, this phase is mainly concerned with improving the result generated by the preceding phases. Therefore, we assume that it is optional in contrast to the other three phases, that are mandatory for each clustering algorithm. The problem of variety we explained in the association phase, holds for this phase as a whole. As optimization can involve tasks like parameter adjustment, updates to points or references, iteration etc. the derivation of a minimal set of basic elements is not feasible at the moment. As stated before, we will solve this problem in a later section by moving to a different level of abstraction.

2.2 Summary

Figure 1 summarizes our conceptual decomposition of clustering algorithms. We identified three core phases which have to be executed and one additional phase for several optimizations. Furthermore, we defined the basic elements for each phase. In the following sections, we concretize our approach and iron out the flaws still existing at this point.

3. DATA MODEL

To realize our concept of a unified and clustering-specific instruction set, we have to define a data model for our presented input and output elements: *points, references, point-distance-reference triples (distances), adjacencies* and *clustering* at first. In our approach, we propose to use *matrices* as a unified formal representation for all input and output basic elements.

Points P and References R

When it comes to the formal definition of a dataset for clustering, existing literature generally uses multi-dimensional vectors to represent the location of data points inside a feature-space. Following this procedure, we define our basic element *points* as a set P of f-dimensional vectors $\vec{p} = \{p_0, \ldots, p_f\}$ where $(0 \leq j \leq f)$ and with $n = |P|$. This set can be easily converted into a matrix P by interpreting each vector $\vec{p_i}$ as a row $p_{i,*}$ of said matrix, thus giving P the dimensions of n rows and f columns. We stated earlier, that the set of *references* can either be a subset of P or just be located in the same feature space. This allows us to define it similar to P, as set of vectors R, containing $\vec{r} = \{r_0, \ldots, p_f\}$ where $(0 \leq j \leq f)$ and $k = |R|$, which forms a matrix $R^{k \times f}$.

Distances D, Adjacencies A, Clustering C

While P and R basically express the values of features per point, the remaining actors are instantiations of relations between objects e.g. *point-distance-reference* triples from evaluation or point-cluster tuples from clustering. For the formal description of these actors, matrices are especially convenient as the objects involved in the relation correspond to a row and column pair which addresses the matrix element holding the value of the actual relation. As an illustration, we define the basic element *distances* as a matrix D with n rows and k columns, where $n = |P|$ and $k = |R|$. Each element d_{ij} of D relates to a point/row $p_{i,*}$ of P and a reference $r_{j,*}$ of R. Thus, d_{ij} contains the distance between $p_{i,*}$ and $r_{j,*}$. Besides triples, this description can be translated smoothly to tuples like point-cluster from *clustering*. Such a tuple point-cluster expresses an existing relation in a binary fashion i.e. a relation between a point and a cluster only exists, if the corresponding tuple exists. This can be described with a binary matrix, where a value of 1 at position (i, j) indicates an existing relation between the objects referenced by i and j, while 0 states the opposite. Accordingly, we define *adjacencies* as binary matrix A having the same dimensions as D. To complete our description, we also define *clustering* as binary matrix C with n rows and a number of columns determined by the number of clusters found.

Notation

Fundamentally, the notation for our reduced instruction set corresponds to a pseudocode notation. Matrices are denoted with a single capital letter e.g. D for the distances. Additional designation is done in the sub- and superscript of the letter. To distinguish different matrix versions we use the superscript: D^{I} and D^{II} are versions of D after 1 resp. 2 function applications, while D^x and D^{x+1} designate the versions of D that are in effect for the current resp. next iteration of the algorithm. With the subscript, matrices can be described in more detail e.g. D_R denotes the distances between all references R.

4. BUILDING BLOCKS

So far, we defined the *matrix* as unified data model for our reduced instruction set in the previous section. Next, we have to find a fitting formal representation for the remaining basic elements like distance measure, filters and association function as well as a necessary set of control flow structures.

4.1 Functions

We describe the remaining basic elements as functions over matrices. Whereas, we use infix notation for elementary matrix functions: addition, subtraction, multiplication and entry wise multiplication, while prefix notation is used for all other functions.

```
    $A \circ B \rightarrow C$                 ▷ infix example: entry wise product
    $function(A) \rightarrow A^I$       ▷ prefix example: single input function
    $function(A, B) \rightarrow C^I$         ▷ prefix example: double input function
```

In the next step, we formally define the functions for the proposed basic elements distance measure, filters and association function.

Distance Function

We start with the distance measure $dist$, which takes a pair of rows $(p_{i,*}, r_{j,*})$ from P and R and assigns a scalar value to it, that represents the distance between the corresponding objects. The abstract function *dist* can be defined as:

$$\begin{aligned} dist : \mathbb{M}^{n\times f} \times \mathbb{M}^{k\times f} &\rightarrow \mathbb{M}^{n\times k} \\ (P, R) &\mapsto D \\ d_{ij} &= f(p_{i,*}, r_{j,*}) \end{aligned}$$

Filter Functions

The task of a filter is to check whether a matrix or one of its elements fulfills certain conditions and to pass them on or sort them out accordingly. Thus a filter resembles an *if-then* statement. Describing this behavior by using mathematical functions requires the breakdown of the task and the establishment of some conventions. The defining part of each filter is its *condition*, which can be described in mathematical terms as function with the co-domain $0, 1$, representing the results false and true. A simple threshold condition, that is satisfied by all numbers smaller 10 could be defined as:

$$threshold : \mathbb{R} \rightarrow \{0, 1\}, X \mapsto X^I$$
$$x^I = \begin{cases} 1, & \text{if} \quad x < 10 \\ 0, & \text{otherwise.} \end{cases}$$

Adopting this notation for each condition would be pretty extensive, so we settle for a minimized version and only denote the condition leading to true resp 1 as the function name. Thus, notation of the preceding definition is reduced to $\langle x < 10 \rangle$.

With this convention, we have to look into the 'then' part of a filter. While elements that fulfill the provided condition are left untouched, those who fail have to be sorted out or rather deleted. Actual deletion of elements or matrices cannot be modeled as mathematic function, therefore we need a workaround for this issue. Let us regard k-means as an example, where the minimal point-cluster distance is evaluated. Assume a row $d_{2,*} = (d_{21}, d_{22}, d_{23}, d_{24})$, from D whose components show the distances between point $p_{2,ast}$ and the four centroids of R. The filter necessary for k-means requires to sort out all components that are not minimal. Without the capability for removal, it is necessary to define a neutral element to which all inputs that fail the condition are mapped. For our scenario, we state this neutral element as 0, which allows us to define the minimum filter as:

$$\begin{aligned} minFilter : \mathbb{M}^{1\times k} &\rightarrow \mathbb{M}^{1\times k} \\ (D, \langle x = min(D)\rangle) &\mapsto D^I \\ d^I_{ij} &= \langle x = min(D)\rangle\,(d_{ij}) \cdot d_{ij} \end{aligned}$$

Assuming d_{23} as minimum of $d_{2,*}$, the filtered row becomes $d^I_{i,*} = (0, 0, d_{23}, 0)$. Using this approach, the subsequent functions in a clustering algorithms have to be aware of 0 as neutral element. Our filter description is a composite of a variable condition and a fixed function that maps to the neutral element. In the context of clustering algorithms, filters cannot exist without conditions. However, conditions can exist by themselves and are necessary to describe branching and conditional execution of functions. Thus, standalone conditions are a way to realize control flow. The following pseudocode shows the notation for both cases:

```
    filter$(M, \langle cond \rangle)$                  ▷ input for filter
    if cond then function           ▷ standalone use
```

In both applications, the condition itself is denoted as boolean expression $\langle cond \rangle$, which is sufficient for its utilization as part of a filter. In standalone use a condition affects the control flow i.e. some actions are performed only if the condition is met. To illustrate this, we embed $\langle cond \rangle$ in an if-then block, where the then part contains the action to be executed.

Association Function

By executing filters, a modified version of the input is created. For the selection phase, this is the modified distance matrix D^I, which is passed on to the association-function. The goal of this function is the transformation of distances into adjacencies i.e. the *point-distance-reference* triples that made it past the selection phase, must be converted into *point-reference* tuples. Basically D^I is converted into a binary matrix, where a value of 1 represents an existing adjacency. Due to the filtering, non-existent adjacencies have already been mapped to 0 which leaves the task of mapping every non-zero value to 1. Based on this, we define the binary association function *assoc* as

$$\begin{aligned} assoc : \mathbb{M}^{m\times n} &\rightarrow \mathbb{M}^{m\times n} \\ D^I &\mapsto A \\ a_{ij} &= sgn(d^I_{ij}) \end{aligned}$$

where *sgn()* is the sign function. This function is quite convenient as it keeps the neutral element 0 and maps all positive values to 1. Although *sgn()* can yield -1 for negative inputs, this does not need to be considered in our setting as distances are always positive.

4.2 Control-Flow Structures

Until now we have not discussed one basic element which is crucial for almost every clustering algorithm but whose necessity is not evident. This additional basic element is the loop, which is a part of the control flow that we have not considered so far, with standalone conditions being the exception. The incorporation of loops in our scenario is tricky as they cannot be mathematically modeled, thus they are defined outside the mathematical domain. To describe clustering algorithms, we basically need two loop types: a *for-each* loop for element-wise traversal of datasets or clusterings and a *repeat-until* loop for conditioned iterations. These two loops are denoted with the following pseudocode:

```
for each $element$ of $M$ do
    $\langle body \rangle$
end for $\rightarrow M^I$

repeat with $A$
    $\langle body \rangle$
until $cond$ output $\rightarrow B$
```

At the top, we find the block for the for-each loop, which is generally used to traverse datasets or clusterings by element. The opening statement of the loop specifies the traversed matrix M and the element/granularity of traversal: row, column or component. In our scenario, element-wise traversal is done by splitting up the source matrix into element-matrices—rows, columns or components at the beginning of the loop. After the split, the elements are processed individually according to the instructions of the $\langle body \rangle$. As we want a single matrix as output again, the processed elements have to be re-assembled at the end of the loop e.g. row-matrices are appended. This re-assembly is implicitly assumed and not specifically noted in pseudocode. The loop output is denoted with the assignment after *end for*.

In addition to the described functionality, we use for-each loops for the actual removal of rows and columns from matrices. This is sometimes necessary when clustering algorithms delete references or clusters during optimization. With filters, we introduced mapping to the neutral element 0 as a means to tackle deletion. This works well and is also necessary for the clear definition of functions. However, the handling of whole rows and columns of zeros can become challenging e.g. it can lead to empty clusters in C or cause problems during the selection phase. Some of this issues could be tackled by introducing constraints to each function to ignore all-zero rows/columns. But this would be complex and not an overall solution. Our described for-each loop offers an elegant way to solve this problem. By inserting an appropriate condition before matrix reassembly at the end of the loop we prevent zero element-matrices from entering the output matrix. Since loops are outside the mathematic formalism anyway, adding row/column removal here provides us with a convenient tool without compromising the formal description of the remaining building blocks.

The *repeat-until* loop is used to represent conditioned loops. This kind of loops is normally used to control algorithm iterations, often during minimization/maximization of target functions like the sum of squared errors in k-means [10]. The stopping condition for the loop is always specified after the closing *until* statement. A repeat-until loop has one or more input matrices—denoted in the opening statement—which are continuously processed from iteration to iteration and an output matrix, obtained when the loop finishes. This output matrix can be either a processed version of the input or an assembly of element-matrices generated during the loop. The particular output type can be derived from the $\langle body \rangle$ of the loop.

4.3 Summary

In Section 2, we introduced the *core* of a clustering as a sequence of the phases *evaluation*, *selection* and *association* that acts as a general frame. Now that we finished the description of our building blocks and defined their syntax, we are able to concretize these phases and flesh out the mandatory algorithm *core*:

- **evaluation** - This phase requires at least 4 building blocks: one function playing the role of distance measure and three matrices acting as points, references and distances.
- **selection** - This phase consists of at least one filter or condition.
- **association** - For this phase 3 building blocks are mandatory: two matrices acting as adjacencies resp. clustering and the association function.

Beyond this essential structure, arbitrary clustering functionality can be added to each of these phases—including optimization—by utilizing the existing building blocks. We move on to the next section, where we demonstrate how clustering algorithms are transcribed using our proposed approach.

5. TRANSCRIPTION OF ALGORITHMS

This section demonstrates how clustering algorithms are transcribed using our approach, whereas we utilize two prominent clustering algorithms *k-means*[10] and DBSCAN[8]. While k-means is a representative of the clustering partitioning alorithm class, DBSCAN is from a completely different classed named density-based clustering.

5.1 k-means

Our representation of k-means is shown in Algorithm 1 and begins with the evaluation phase in which four building blocks take part. Three of these are actors: two matrices P and R that contain the points of the dataset and the k initial centroids as rows, as well as the distance matrix D. The fourth is the distance measure used to generate D from P and R. For k-means, this role is taken by the euclidean distance, denoted by the function L_2 which we define for our matrix setting as:

$$L_2 : \mathbb{M}^{n\times f} \times \mathbb{M}^{k\times f} \rightarrow \mathbb{M}^{n\times k}$$

$$(P, R) \mapsto D \quad \text{with} \quad d_{ij} = \sqrt{\sum_{l=1}^{f} (p_{il} - r_{jl})^2}$$

where $p_{i,*}$ and $r_{j,*}$ are rows of their respective matrices. The resulting matrix D provides the input for the following selection phase, which starts with a for-each loop for row-wise traversal of D (6). Due to the evaluation phase, each row $d_{i,*}$ contains all distances between point $p_{i,*}$ and R that we need for the selection. The following filter function selects the minimum element d_{ij} from each row, which reflects the target function of k-means. At the end of the loop, the processed rows are assembled into the filtered matrix D^I, that is passed on to the association phase. There, our *assoc* function is deployed to generate the binary adjacency matrix A. Due to the character of k-means, A basically contains the final cluster assignments. As centroids resp. references represent the clusters, A is simply adopted as C (12).

With the core phases finished and a clustering result generated, k-means enters its optimization phase which updates the centroids (references) for the next iteration. Each centroid is recalculated

Algorithm 1 k-means

```
 1: repeat with $R^x$
 2:     phase EVALUATION
 3:         $dist.L_2(P, R^x) \rightarrow D$
 4:     end phase
 5:     phase SELECTION
 6:         for each $d_{i,*}$ of $D$ do
 7:             $filter(d_{i,*}, \langle x = min(d_{i,*})\rangle)$
 8:         end for $\rightarrow D^I$
 9:     end phase
10:     phase ASSOCIATION
11:         $assoc(D^I) \rightarrow A$
12:         $A \rightarrow C$
13:     end phase
14:     phase OPTIMIZATION
15:         $updt(C^T, P) \rightarrow R^{x+1}$
16:     end phase
17: until $R^x = R^{x+1}$ output $\rightarrow C$
```

as the arithmetic average of all points that were assigned to it in the current iteration. In our matrix-based setting, we realize this by using the matrix-multiplication as a template with C and P as input. The first input is matrix C that contains the point-cluster assignments and has the dimensions $n \times k$ with k being the number of references/centroids. The second input P has the dimension $n \times f$ with n being the number of points and f being the number of features of the dataset. By multiplying C with P we want to create an updated version of R having the dimension of $k \times f$. For this, the number of columns of C has to match the number of rows in P, which is not the case as $k \neq p$. Therefore, we transpose C to C^T which leads to the required column-row-match and results in a $k \times f$ matrix R^{x+1} that contains the updated centroids for the next iteration. The function used for calculation of the update is defined as:

$$updt : \mathbb{M}^{k \times n} \times \mathbb{M}^{n \times f} \rightarrow \mathbb{M}^{k \times f} \tag{1}$$

$$(C^T, P) \mapsto R_{x+1} \tag{2}$$

$$r_{ij} = \frac{\sum_{l=1}^{n} c_{il} \cdot p_{lj}}{\sum_{l=1}^{n} c_{il}} \tag{3}$$

with $c_{i,*}$ being a row of C^T and $p_{*,j}$ being a *column* of P. Basically, this function uses each cluster represented by a binary row of C^T to select those values from the feature represented by $p_{*,j}$ that belong to its members. This selection is summed up and normalized with the number of cluster members obtained by summing up all elements of binary $c_{i,*}$.

The whole algorithm is surrounded by a *repeat-until* loop that describes the iteration of k-means using the updated references/centroids of R_{x+1}. The stopping criterion, shown after *until* (17) is evaluated before a new iteration is started. For our depiction we choose $R^x = R^{x+1}$ as stopping condition and quit the algorithm if the references/centroids no longer change, which indicates stabilized clusters. Of course other stopping conditions can be used e.g. reaching a fixed number of iterations.

5.2 DBSCAN

DBSCAN [8] is a density-based clustering algorithm that defines clusters as dense regions separated by regions of lower density. The algorithm uses two parameters ε and $minPts$ to define a density threshold. With ε a neighborhood is defined around each point p. If this neighborhood contains at least $minPts$ additional points, p is considered as member of a dense area i.e. a cluster and is named *core-object*. Sets of core-objects with overlapping ε-neighborhoods are merged in order to create clusters. This is done recursively i.e. if p is a core-object each member of its ε-neighborhood is checked for the density condition.

Algorithm 2 DBSCAN

```
 1: phase EVALUATION
 2:     $dist.L_2(P, R^x) \rightarrow D$
 3: end phase
 4: phase SELECTION
 5:     for each $d_{i,*}$ of $D$ do
 6:         $filter(d_{i,*}, \langle x < \varepsilon\rangle)$
 7:         $sgn(d^I_{i,*})$
 8:         $filter(d^I_{i,*}, \left\langle \sum_{j=0}^{n} x \leq minPts \right\rangle)$
 9:     end for $\rightarrow D^I$
10: end phase
11: phase ASSOCIATION
12:     $assoc(D^I) \rightarrow A$
13:     $merge(A) \rightarrow C$
14:     $distinct(C) \rightarrow C_{distinct}$
15: end phase
```

The fully transcribed version of DBSCAN using our approach is shown in Algorithm 2. Although the evaluation phase may look the same as with the previously described algorithms, DBSCAN is different as it calculates the distances between all points, which means P and R are actually identical. The selection phase uses a for-each loop for row-wise traversal and contains three steps. First, a filter is employed to remove all distances that are bigger than the ε-neighborhood. Next $sgn()$ is applied in preparation of the following filter, that tests if the neighborhood contains the necessary number of objects by checking the sum of components of the binary row-matrix. With the selection phase done, association starts with the known application of $assoc()$ (12). After that, we face a challenge as $assoc()$ effectively creates a cluster for each core-object and its ε-neighborhood.

Now, to determine the final clusters, overlapping ε-neighborhoods have to be merged. Utilizing recursion as proposed in [8] is not a valid approach in our matrix based setting. Therefore, we use a repeat-until loop to connect overlapping ε-neighborhoods as specified in Algorithm 3. For this, the adjacencies–labeled here as M^x–are multiplied with itself and the result is transformed into the binary M^{x+1}, which is the input for the next loop. With this, indirect/transitive cluster assignments are resolved. The loop ends if M^{x+1} does not change anymore, which means that all direct adjacencies have been found and the resulting clustering C is delivered.

Algorithm 3 Transitive Merging Function: $merge(M^x)$

```
1: repeat with $M^x$
2:     $M^x \cdot M^x \rightarrow M^I$
3:     $sgn(M^I) \rightarrow M^{x+1}$
4: until $M^x = M^{x+1}$ output $\rightarrow M^{x+1}$
```

Example matrices for this association are shown in Figure 2, where the first three columns of M^x show the indirect cluster assignment of p_1, p_2 and p_3. The matrix $A = M^x$ is the result the selection phase and therefore, the input of the association function. Although the points p_1,p_2 and p_3 form a cluster, the adjacency of p_1 and p_3 is indirect via p_2. After multiplication, all adjacencies are explicit in M^{x+1}. While this solves the problem of merg-

$$\begin{array}{c} A = M^x \\ \begin{matrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{matrix} \end{array} \rightarrow \begin{array}{c} M^I \\ \begin{matrix} 2 & 2 & 1 & 0 & 0 & 0 \\ 2 & 3 & 2 & 0 & 0 & 0 \\ 1 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 2 \end{matrix} \end{array} \rightarrow$$

$$\begin{array}{c} M^{x+1} \\ \begin{matrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{matrix} \end{array}$$

Figure 2: DBSCAN association.

ing overlapping ε-neighborhoods into clusters, it leads to duplicate columns/clusters. Complete explicit adjacencies mean that each member of a cluster is associated with each remaining member i.e. a cluster with 4 members manifests in 4 identical rows in C.

To get rid of the duplicates, we apply a duplicate elimination function *distinct* (14) on the output association matrix. The algorithm of this function is depicted in Algorithm 4. The function takes M^x as input and starts by aggregating it into a row matrix of cluster sizes H_{select}. Then, a maximum filter is deployed to select the biggest cluster and $sgn()$ is applied to make the resulting row-matrix H^I_{select} binary. This is only done to implement a processing sequence for the rows of M^x. Multiplication of H^{II}_{select} with M^x extracts a particular row matrix from M^x. This row matrix becomes the first row $m_{i,*}$ of $M_{distinct}$. Example matrices for this are shown in Figure 3. After getting the first row, we have to get rid of all its duplicates in M^x. For this we apply *sgn()* to $m_{i,*}$ and create binary $(m_{i,*})^I$, whose transpose is multiplied with $m_{i,*}$ to get a square filter matrix H_{filter}. By subtracting it from M^x, the processed row and its duplicates are set to zero, effectively removing them from further processing in the loop. The resulting M^{x+1} enters the next iteration, where another unique row is extracted. The loop ends when M^{x+1} becomes a zero matrix, which means all unique rows have been extracted. Examples of H_{filter}, M^{x+1} and final $M^{distinct}$ can be found in Figure 3, where the example output of the association phase in Figure 2 is continued.

Algorithm 4 Distinct Function: *distinct(M^x)*

```
1: repeat with $M^x$
2:     $agg(M^x) \rightarrow H_{select}$
3:     $filter(H_{select}, \langle x = max(H_{select}) \rangle) \rightarrow H^I_{select}$
4:     $sgn(H^I_{select}) \rightarrow H^{II}_{select}$
5:     $H^{II}_{select} \cdot M^x \rightarrow m_{i,*}$ of $M_{distinct}$
6:     $sgn(m_{i,*}) \rightarrow m^I_{i,*}$
7:     $(m^I_{i,*})^T \cdot m_{i,*} \rightarrow H_{filter}$
8:     $M^x - H_{filter} \rightarrow M^{x+1}$
9: until $\sum M^{x+1} = 0$ output $\rightarrow M_{distinct}$
10: output $M^T_{distinct}$
```

At the end of the *distinct* function, the result is transposed as we want clusters to be represented in columns. Afterwards, DBSCAN finishes as it has no optimization phase or global loop.

5.3 Summary

Due to the large number of approaches to clustering, we cannot transcribe each method or even a representative of each larger

$$\begin{array}{c} C = M^x \\ \begin{matrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{matrix} \\ \text{(a)} \end{array} \qquad \begin{array}{c} H_{select} \\ \begin{matrix} 3 & 3 & 3 & 2 & 1 & 2 \end{matrix} \\ \text{(b)} \end{array}$$

$$\begin{array}{c} H^{II}_{select} \\ \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 \end{matrix} \\ \text{(c)} \end{array} \qquad \begin{array}{c} H_{filter} \\ \begin{matrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \\ \text{(d)} \end{array}$$

$$\begin{array}{c} M^{x+1} \\ \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{matrix} \\ \text{(e)} \end{array} \qquad \begin{array}{c} M_{distinct} \\ \begin{matrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{matrix} \\ \text{(f)} \end{array}$$

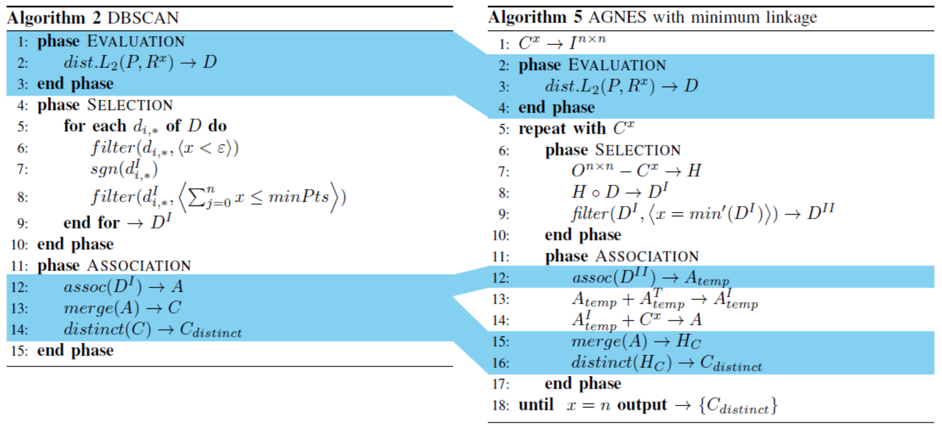
Figure 3: Duplicate Elimination.

family of algorithms in this paper. However, we are sure that our approach is general enough to cope with this variety. Often it seems that a particular algorithm cannot be transcribed at first, but after reconsidering a way can be found to adapt to our setting. Sometimes, little modification are enough, but in other cases the problem must be rethought thoroughly to find an equivalent building block representation. Examples include but are not limited to:

- Graph-clustering, where graphs must be transformed into a matrix for adaptation.
- Evolutionary approaches e.g. artificial immune systems [18], model centroids or proto-clusters as a population of cells that is influenced by a fitness function. This can be modeled in our approach by modifying references/clusters between iterations. Creation, deletion, re-calculation as well as splitting and merging, can be reproduced with modified matrix multiplications, filters and custom mathematical functions.
- Spectral clustering approaches, work with the eigenvalues of a similarity matrix. Although it seems that this resembles our distances *D*, this is not the case. As partitioning is based on the correspondence between eigenvalues, this measure must be considered during evaluation. In this case the initial similarity matrix must be seen as an additional input.

This should only point our the versatility of our approach. We are pretty sure that there are still a lot of further clustering approaches, that we did not consider in detail. But we are also sure that we could describe them with our approach after giving them some thought. To summarize our approach offers the following advantages:

1. Easy adaptation and specification of clustering algorithms in an abstract and implementation-independent way. In general, our approach is designed for the data scientist.
2. The comparison of algorithms and the easy identification of common functionalities.



Algorithm 2 DBSCAN
```
phase EVALUATION
    $dist.L_2(P, R^x) \rightarrow D$
end phase
phase SELECTION
    for each $d_{i,*}$ of $D$ do
        $filter(d_{i,*}, \langle x < \varepsilon \rangle)$
        $sgn(d^I_{i,*})$
        $filter(d^I_{i,*}, \left\langle \sum_{j=0}^{n} x \leq minPts \right\rangle)$
    end for $\rightarrow D^I$
end phase
phase ASSOCIATION
    $assoc(D^I) \rightarrow A$
    $merge(A) \rightarrow C$
    $distinct(C) \rightarrow C_{distinct}$
end phase
```

Algorithm 5 AGNES with minimum linkage
```
$C^x \rightarrow I^{n \times n}$
phase EVALUATION
    $dist.L_2(P, R^x) \rightarrow D$
end phase
repeat with $C^x$
    phase SELECTION
        $O^{n \times n} - C^x \rightarrow H$
        $H \circ D \rightarrow D^I$
        $filter(D^I, \langle x = min'(D^I) \rangle) \rightarrow D^{II}$
    end phase
    phase ASSOCIATION
        $assoc(D^{II}) \rightarrow A_{temp}$
        $A_{temp} + A^T_{temp} \rightarrow A^I_{temp}$
        $A^I_{temp} + C^x \rightarrow A$
        $merge(A) \rightarrow H_C$
        $distinct(H_C) \rightarrow C_{distinct}$
    end phase
until $x = n$ output $\rightarrow \{C_{distinct}\}$
```

Figure 4: Similarities of DBSCAN and AGNES.

6. CONTRIBUTIONS

Besides providing a way to specify clustering algorithms, our building blocks approach offers several novel application possibilities. Due to the use of consistent components, we can evaluate the similarity between algorithms to a certain degree. During the transcription of our example algorithms, we uncovered the existence of several description blocks that are used by multiple algorithms e.g. removal of duplicates and resolution of transitivity. Fig. 4 shows a comparison of the already described DBSCAN and AGNES[12, 13] as an example from a further hierarchical clustering algorithm class, where identical building blocks are highlighted in blue. It is easy to see, that both methods are very similar. Evaluation phases are identical, while the association phases show only minor differences in two lines that are necessary because each C generated by AGNES is also considered a hierarchy level. The main differences between both methods are located in the selection phase and during optimization. While this allows the easy identification of an algorithms characteristic parts, it can also be used to classify algorithms. Families of algorithms that share certain traits could be identified on the basis of commonalities. In order too find common patterns, frequent itemset mining could be applied to a repository of algorithms. For this, each building block is considered as an item and each phase resp. algorithm as transaction.

Each of our phases and building blocks encapsulates a defined functionality. Furthermore, we observed that certain blocks can be used in different algorithms. Therefore, we do not limit our concept to the description/translation of existing methods but also use it for the creation of new ones. The modular character of our approach enables algorithm creation in a novel and easy way. Basically, there are three levels of modularity that can be used to build new algorithms and are depicted in Fig. 5.

1st Level: Phase-Swap Phases realize the basic tasks necessary for clustering in an algorithm-specific way. Although phases are implemented individually, they share a defined interface. This means, evaluation always produces a distance matrix D, selection always uses D and creates D^I, and association always creates C from D^I. With this, phases become interchangeable and form the largest modules of our approach. The optimization phase is more individual, but can still be swapped if only the mandatory elements *P,R,D,D',A,C* are accessed. All this allows users to easily create new algorithms by recombining phases of available stock algorithms.

2nd Level: Custom Phase This level works on the finer granularity of building blocks and enables intermediate users to create new evaluation, selection, association, and optimization phases from scratch. For this, existing blocks from a repository are combined and fitted into our introduced phase-templates. Examples for such blocks are *updt()* and *distinct()* from our example algorithms section. Newly created phases can be added to the existing repository and thus provide new options for level 1.

3rd Level: Custom Block On this level, experienced user can freely define new building blocks e.g a new distance function or a scenario specific variant of *assoc()*. In addition, block sequences or subroutines that occur very often, can be integrated as higher-order building blocks to ease description. Like before, new building blocks are added to a repository, where they are available for other users. The creation of new building blocks also makes the previous levels more versatile.

The first and second level implement creation exclusively by combination of modules/blocks from a repository. This makes it possible to realize this task with interactive interfaces instead of IDE's normally used for software development. In Fig. 6, we depict a prototypical interface for modular algorithm design that shows k-means by using our building blocks as basis for the visual elements. Each phase is marked by a different color: blue for evaluation, green for selection and so on. This prototype was designed for smart devices, and allows users to swap phases by swiping and switch building blocks by touching. If one of these actions is performed, the system provides a list of alternatives, available in the repository, from which the user can choose.

7. FUTURE WORK

Looking back at our motivation, we argued that the ever increasing diversity of clustering algorithms is necessary to cope with the individual characteristics of various data sets. Because, this diversity complicates the application of clustering in the context of Big Data, a building block approach would be desirable and was proposed in this paper. Aside from supporting the specification, our approach facilitates the adaptation of core clustering principles for specific applications.

Generally, our modular approach consists of a few functions like *assoc*, *merge*, *updt* and *filter* over a single data structure of type *matrix*. Our next research step focuses on a general mapping of these functions to a MapReduce infrastructure. In this step, the

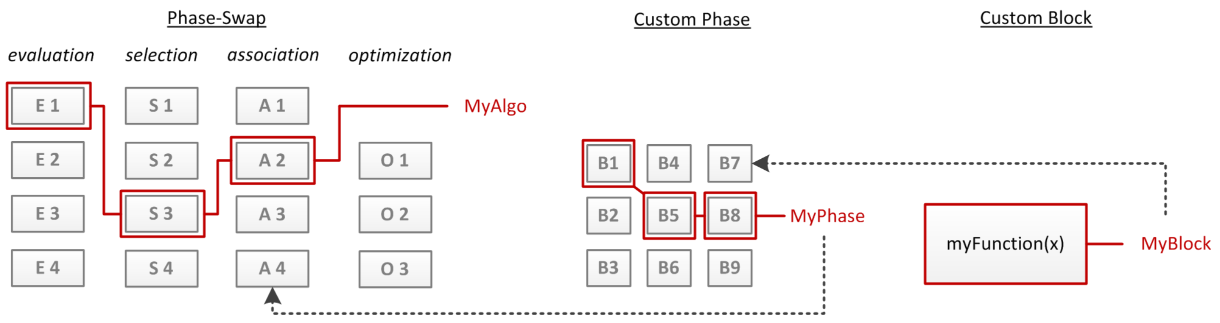


Figure 5: Levels of modular algorithm design.

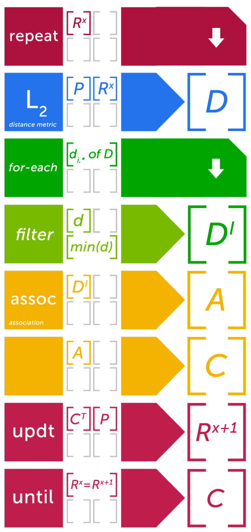


Figure 6: Prototype UI for modular algorithm design.

following challenges arise:

- As described in Section 4, the *repeat-until* loops are integral part of our building blocks. Therefore, we require a MapReduce infrastructure with support for iterative computations as proposed in [4].
- The most challenging issue is the mapping of our matrices to *(key, value)* pairs in MapReduce. This mapping has to be done, so that an efficient and scalable partitioning is possible in order to parallelize necessary function evaluations. In [14], we investigated this aspect in MapReduce and presented several approaches, whereas there exists no best fitting approach for all cases. The best-fitting approach depends on several properties like matrix size or number of available nodes.
- Aside from mapping of our matrix construct, we also have to specify physical operators for our limited set of logical functions.

As a result of this work, we are able to transform any arbitrary clustering algorithm specified in our building blocks into an efficient and scalable execution form. Furthermore, we can optimize the transformation by the utilization of different mapping strategies of matrices to *(key, value)* pairs as well as selection of the best-fitting physical operator for a logical building block function. For this optimization, we have to define various optimization strategies depending e.g., on matrix sizes or sequence of logical functions. Furthermore, our iterations can be integrated in the optimization, since our loops iterate either row- or column-wise over the matrix. In this way, we establish a similar approach as conducted in database systems for over 30 years with SQL as logical interface and for each logical operator different physical operators exist. The transformation between the logical and physical layer is done using an optimizer component, which is responsible for efficient transformation using roles and a cost-model. In our next research step, we want to establish such an approach for data clustering algorithms in highly scalable parallel data processing platforms.

A second major approach to execution is direct integration into the database. Already, approaches like SciDB [5] consider matrices as first-order citizens in database management systems. This way of integration is especially compelling as it allows a tight coupling of data management and data analysis in the scope of a single platform. Besides the deployment to different software systems, also hardware specifics can be considered. By developing platform-specific compilers for e.g. NUMA or GPU-centric systems, users could create optimized versions of their algorithm libraries in an on-demand fashion. In this case, a lot of related research is and has been done in different domains, that can be used in our future work. For example, efficient large matrix computation has a long tradition as area of research in high performance computing [11, 15]. Furthermore, graphic cards and CUDA are strongly geared to matrix processing [1, 9, 17] and seem to be an ideal target architecture our approach. In this domain, dense [1, 19] as well as sparse [3] matrix operations are well-investigated.

8. RELATED WORK

On the one hand, our work is motivated by SciDB [5], which only considers the storage and the processing of a natural nested multi-dimensional array data model. One the other hand, our approach originates from the ongoing *key-value* hype of MapReduce [7], because from our point of view the *key-value* model is not appropriate for the data clustering domain. As shown in [20, 2], MapReduce and an enhanced paradigm based on a *key-value* data model can be used for the *k-means* clustering algorithm. However, the implementation of essential clustering functionality is complicated by the restricted data model. Several approaches are proposed to map the necessary matrix data to a *key-value* model. One possibility is to encrypt row and column information in the key forming a super-key. Nevertheless, a pure matrix model as proposed in our approach is more direct and eases the specification of clustering algorithms.

In a derived implementation out of our modular algorithm specification, we are able to use the *key-value* data model for scalable execution.

In the context of highly scalable parallel data processing platforms, the Mahout[3] project has to be considered, which directly implements various machine-learning algorithms in Hadoop. The implementations currently neither exploit high-level data processing languages built on top of Hadoop nor do they make use of any statistical software. With more and more analysis methods are added, leveraging existing functionality adds to the stability and simplicity of the implementation. Instead of implementing and optimizing each single analysis method separately, our approach introduces a novel abstraction layer on top to specify methods in an implementation-independent way using building blocks. The building blocks have to be mapped to the execution unit, e.g. Hadoop once and each algorithm can directly benefit. The mapping of our building blocks is our next step as described in the previous section.

9. SUMMARY AND CONCLUSION

In this paper, we proposed our modular building blocks approach as a unified construction kit for clustering algorithms. We decomposed clustering methods and derived the core of every algorithm in the form of the three phases: evaluation, selection and association. In addition with the optional optimization phase, this provides a general frame for algorithm description. To fill this frame, we identified the basic elements of each phase and transformed them into a set of building blocks. In our data model, all necessary objects for clustering i.e. points, references, distances, adjacencies and clustering are formally represented as matrices. Matrices are the exclusive and universally valid way for data modeling in our approach and naturally match the concept of clustering. Based on this, all necessary operations like distance measurement, filtering and association are modeled as mathematical functions on matrices. To complete our set of building blocks, we introduced conditions and loops to represent the control-flow of a clustering algorithm.

All this was put to use during the transcription of k-means and DBSCAN which are well-known members of the two major classes of clustering algorithms. Our transcription proved that different methods can be easily represent with our unified description. Furthermore, our approach allows the comparison of algorithms and the easy identification of common functionalities. Besides this benefits for understanding, adaptation and construction of clustering algorithms, our descriptions can be used as a starting point for platform-specific implementation. From our point of view, this offers considerable potential for the efficient execution of any clustering algorithm.

10. REFERENCES

[1] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí. Exploiting the capabilities of modern gpus for dense matrix computations. *Concurrency and Computation: Practice and Experience*, 21(18):2457–2477, 2009.

[2] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*, 2009.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[5] P. Cudré-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. J. DeWitt, B. Heath, D. Maier, S. Madden, J. M. Patel, M. Stonebraker, and S. B. Zdonik. A demonstration of scidb: A science-oriented dbms. *PVLDB*, 2(2):1534–1537, 2009.

[6] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating r and hadoop. In *SIGMOD Conference*, pages 987–998, 2010.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231, 1996.

[9] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2004, Grenoble, France, August 29-30, 2004*, pages 133–137, 2004.

[10] E. W. Forgy. Cluster analysis of multivariate data: Efficiency versus interpretability of classification. *Biometrics*, 21, 1965.

[11] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[12] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[13] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[14] T. Kiefer, P. B. Volk, and W. Lehner. Pairwise element computation with mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 826–833, 2010.

[15] M. Krishnan and J. Nieplocha. Memory efficient parallel matrix multiplication operation for irregular problems. In *Conf. Computing Frontiers*, pages 229–240, 2006.

[16] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *SIGMOD Record*, 40(4):11–20, 2011.

[17] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba. Parallel processing of matrix multiplication in a cpu and gpu heterogeneous environment. In *High Performance Computing for Computational Science*, pages 305–318, 2006.

[18] J. Timmis, M. Neal, and J. Hunt. An artificial immune system for data analysis. *Biosystems*, 55(1âĂŞ3):143 – 150, 2000.

[19] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, 2008.

[20] W. Zhao, H. Ma, and Q. He. Parallel k-means clustering based on mapreduce. In *Proceedings of the 1st International Conference on Cloud Computing*, pages 674–679, 2009.

[3]http://mahout.apache.org