ENFrame: A Platform for Processing Probabilistic Data

Sebastiaan J. van Schaik\textsuperscript{1,2}  Dan Olteanu\textsuperscript{1}  Robert Fink\textsuperscript{1}

\{Robert.Fink, Dan.Olteanu, Sebastiaan.van.Schaik\}@cs.ox.ac.uk
\textsuperscript{1}Department of Computer Science, University of Oxford, United Kingdom
\textsuperscript{2}Oxford e-Research Centre, University of Oxford, United Kingdom

ABSTRACT
This paper introduces ENFrame, a unified data processing platform for querying and mining probabilistic data. Using ENFrame, users can write programs in a fragment of Python with constructs such as bounded-range loops, list comprehension, aggregate operations on lists, and calls to external database engines. The program is then interpreted probabilistically by ENFrame.

The realisation of ENFrame required novel contributions along several directions. We propose an event language that is expressive enough to succinctly encode arbitrary correlations, trace the computation of user programs, and allow for computation of discrete probability distributions of program variables. We exemplify ENFrame on three clustering algorithms: \(k\)-means, \(k\)-medoids, and Markov Clustering. We introduce sequential and distributed algorithms for computing the probability of interconnected events exactly or approximately with error guarantees.

Experiments with \(k\)-medoids clustering of sensor readings from energy networks show orders-of-magnitude improvements of exact clustering using ENFrame over naïve clustering in each possible world, of approximate over exact, and of distributed over sequential algorithms.

1. INTRODUCTION
Recent years have witnessed a solid body of work in probabilistic databases with sustained systems building effort and extensive analysis of computational problems for rich classes of queries and probabilistic data models of varying expressivity \cite{Schaik2013}. In contrast, most state-of-the-art probabilistic data mining approaches so far consider the restricted model of probabilistically independent input and produce hard, deterministic output \cite{Fink2013}. This technology gap hinders the development of data processing systems that integrate techniques for both probabilistic databases and data mining.

The ENFrame data processing platform aims at closing this gap by allowing users to specify iterative programs to query and mine probabilistic data. The semantics of ENFrame programs is based on a unified probabilistic interpretation of the entire processing pipeline from the input data to the program result. It features an expressive set of programming constructs, such as assignments, bounded-range loops, list comprehension, aggregate operations on lists, and calls to external database engines, coupled with aspects of probabilistic databases, such as possible worlds semantics, arbitrary data correlations, and exact and approximate probability computation with error guarantees. The existing probabilistic data mining algorithms do not share these latter aspects.

Under the possible worlds semantics, the input is a probability distribution over a finite set of possible worlds, whereby each world defines a standard database or a set of input data points. The result of a user program is equivalent to executing it within each world and is thus a probability distribution over possible outcomes (e.g., partitionings). ENFrame exploits the fact that many of the possible worlds are alike, and avoids iterating over the exponentially many worlds.

Correlations occur naturally in query results \cite{Olteanu2013}, after conditioning probabilistic databases using constraints \cite{Schaik2014}, and are supported by virtually all mainstream probabilistic models. If correlations are ignored, the output can be arbitrarily off from the expected result \cite{Fink2013}. For instance, consider two similar, but contradicting sensor readings (mutually exclusive data points) in a clustering setting. There is no possible world and thus no cluster containing both points, yet by ignoring their negative correlation, we would assign them to the same cluster.

The user is oblivious to the probabilistic nature of the input data, and can write programs as if the input data were deterministic. It is the task of ENFrame to interpret the program probabilistically. The approach taken here is to trace the user computation using fine-grained provenance, which we call events. The event language is a non-trivial extension of provenance semirings \cite{Fink2012} and semimodules \cite{Schaik2012} that are used to trace the evaluation of positive relational algebra queries with aggregates and to compute probabilities of query results \cite{Fink2013}. It features events with negation, aggregates, loops, and definitions. It is expressive enough to succinctly encode arbitrary correlations occurring in the input data (e.g., modelled on Bayesian networks and pc-tables) and in the result of the user program (e.g., co-occurrence of data points in the same cluster), and trace the program state at any time. By annotating each computation in the program with events, we effectively translate it into an event program: variables become random variables whose possible outcomes are conditioned on events. Selected events represent the probabilistic program output, e.g. in case of clustering: the probability that a data point is a medoid, or the probability that two data points are assigned to the same cluster. Besides probability computation, events can
be used for sensitivity analysis and explanation of the program result.

The most expensive task supported by ENFrame is probability computation for event programs, which is \#P-hard in general. We developed sequential and distributed algorithms for both exact and approximate probability computation with error guarantees. The algorithms operate on a graph representation of the event programs called event networks. Expressions common to several events are only represented once in such graphs. Event networks for data mining tasks are very repetitive and highly interconnected due to the combinatorial nature of the algorithms: the events at each iteration are expressions over the events at the previous iteration and have the same structure at each iteration. Moreover, the event networks can be cyclic, so as to account for program loops. While it is possible to unfold bounded-range loops, this can lead to prohibitively large event networks.

The key challenge faced by ENFrame is to compute the probabilities of a large number of interconnected events that are defined over several iterations. Rather than computing the probability of each event separately, ENFrame’s algorithms employ a novel bulk-compilation technique, using Shannon expansion to depth-first explore the decision tree induced by the input random variables and the events in the program. The approximation algorithms use an error budget to prune large tree fragments of this decision tree that only account for a small probability mass. We introduce three approximation approaches (eager, lazy, and hybrid), each with a different strategy for spending the error budget. The distributed versions of these algorithms divide the exploration space into fragments to be explored concurrently by distinct workers.

While the computation time can grow exponentially in the number of input random variables in worst case, the structure of correlations can reduce it dramatically. As shown experimentally, ENFrame’s algorithm for exact probability computation is orders of magnitude faster than executing the user program in each possible world.

To sum up, the contributions of this paper are as follows:

- We propose the ENFrame platform for processing probabilistic data. ENFrame can evaluate user programs on probabilistic data with arbitrary correlations following the possible worlds semantics.
- User programs are written in a fragment of Python that supports bounded-range loops, list comprehension, aggregates, and calls to external database engines. We illustrate ENFrame’s features by giving programs for three clustering algorithms (k-means, k-medoids, and Markov clustering) and provide a formal specification of ENFrame’s user language which can be used to write arbitrary programs for the platform.
- User programs are annotated by ENFrame with events that are expressive enough to capture the correlations of the input, trace the program computation, and allow for probability computation.
- ENFrame uses novel sequential and distributed algorithms for exact and approximate probability computation of event programs.
- We implemented ENFrame’s probability computation algorithms in C++.
- We report on experiments with k-medoids clustering of readings from partial discharge sensors in energy networks [28]. We show orders-of-magnitude performance improvements of ENFrame’s exact algorithm over the naive approach of clustering in each possible world, of approximate over exact clustering, and of distributed over sequential algorithms.

The paper is organised as follows. Section 2 introduces the Python fragment supported by ENFrame along with encodings of clustering algorithms. Section 3 introduces our event language and shows how user programs are annotated with events. Our probability computation algorithms are introduced in Section 4 and experimentally evaluated in Section 5. Section 6 overviews recent related work.

2. ENFRAME’S USER LANGUAGE

This section introduces the user language supported by ENFrame. Its design is grounded in three main desiderata:
1. (O, n) = loadData()  # list and number of objects
2. (k, iter) = loadParams()  # number of clusters and iterations
3. M = init()  # initialize centroids
4. for it in range({0,iter}):
   # clustering iterations
5. InCl = [None] * k  # assignment phase
6. for i in range({0,n}):
   7. InCl[i] = [None] * n
   8. for l in range({0,n}):
      9. InCl[i][l] = reduce_and([d for (d, j) in range({0,k})])
10. InCl = breakTies2(InCl)  # each object is in exactly one cluster
11. M = [None] * k  # update phase
12. for i in range({0,n}):
   13. M[i] = scalar_mult(invert((O[i] for l in range({0,n}) if InCl[i][l])))
14. reduce_sum([1 for l in range({0,n}) if InCl[i][l]])
15. reduce_count([1 for l in range({0,n}) if InCl[i][l]])
16. M[it] = W· \sum_{l=0}^{n} O[l]· \text{for } k \text{ in range({0,n}) if InCl[i][l]])

Figure 2: K-means clustering specified as user program (left) and simplified event program (right).

1. (O, n, M) = loadData()  # M is a stochastic n\times n matrix of
2. # edge weights between the n nodes, O is list of nodes
3. (r, iter) = loadParams()  # Hadamard power, number of iterations
4. for it in range({0,iter}):
5. N = [None] * n  # expansion phase
6. for i in range({0,n}):
   7. N[i] = [None] * n
   8. for j in range({0,n}):
      9. N[i][j] = reduce_sum([M[i][k]·M[k][j] for k in range({0,n})])
10. M = [None] * n  # inflation phase
11. for i in range({0,n}):
12. M[i] = [None] * n
13. for j in range({0,n}):
14. M[i][j] = pow(N[i][j], r)*invert(
15. reduce_sum([pow(N[i][k], r) for k in range({0,n})]))

Figure 3: Markov clustering specified as user program (left) and simplified event program (right).

1. It should naturally express common mining algorithms, allow to issue queries and manipulate their results.
2. User programs must be oblivious to the deterministic or probabilistic nature of the input data and to the probabilistic formalism considered.
3. It should be simple enough to allow for an intuitive and straightforward probabilistic interpretation.

We settled on a subset of Python that can express, among others, k-means, k-medoids, and Markov Clustering. In line with query languages for probabilistic databases, where a Boolean query \(Q\) is a map \(Q : D \rightarrow \{\text{true, false}\}\) for deterministic databases and a Boolean random variable for probabilistic databases, every user program has a sound semantics for both deterministic and probabilistic input data: in the former case, the result of a clustering algorithm is a deterministic clustering, in the latter case it is a probability distribution over possible clusterings.

The user language comprises the following constructs:

**Variables and arrays.** Variables can be of scalar types (real, integer, or Boolean) or arrays. Examples of variable assignments: \(V = 2, W = V, M[2] = \text{True}, \) or \(M[1] = W\). Arrays must be initialised, e.g., for array \(M\) of cardinality \(k\): \(M = [\text{None}] * k\). Additionally, the expression \(\text{range}(O, n)\) specifies the array \([0, \ldots, n-1]\).

**Functions.** Scalar variables can be multiplied, exponentiated \((\text{pow}(B, r))\) for \(B^r\), and inverted \((\text{invert}(B))\) for \(1/B\). The function \(\text{dist}(A, B)\) is a distance measure on the feature space between the vectors specified by arrays \(A, B\) of reals; \(\text{scalar_mult}\) is component-wise multiplication of an array with a scalar.

**Reduce.** Given a one-dimensional array \(M\) of some scalar type, it can be reduced to a scalar value by applying one of the functions \(\text{reduce_or}, \text{reduce_sum}, \text{reduce_count}\). For instance, for an array \(B\) of Booleans, the expression \(\text{reduce_and}(B)\) computes the Boolean conjunction of the truth values in \(B\), and the expression \(\text{reduce_count}(B)\) computes the number of elements in \(B\). For a two-dimensional array of reals or integers, i.e., an array of vectors, \(\text{reduce_sum}\) computes the component-wise sum of the vectors.

**List comprehension.** Inside a \(\text{reduce-function}\), anonymous arrays may be defined using list comprehension. For example, given an array \(B\) of Booleans of size \(n\), the expression \(\text{reduce_sum([1 for i in range(n) if B[i]])}\) counts the number of \(\text{True}\) values in \(B\).

**Loops.** We only allow bounded-range loops; for any fixed integer \(n\) and counter variable \(i\), for-loops can be defined by: \(\text{for } i \text{ in range}(n)\). This allows us to know the size of each constructed array at compile time.

**Input data.** The special abstract primitive \(\text{loadData()}\) is used to specify input data for algorithms. This function can be implemented to statically specify the objects to be clustered, to load them from disk, or to issue queries to a database. ENFrame supports positive relational algebra queries with aggregates via the SPROUT query engine for probabilistic data [14]. The abstract methods \(\text{loadParams()}\) and \(\text{init()}\) are used to set algorithm parameters such as the number of iterations and clusters of a clustering algorithm.

### 2.1 Clustering Algorithms in ENFrame

We illustrate ENFrame’s user language with three example data mining algorithms: k-means, k-medoids, and Markov Clustering. Figures 1 and 2 list user programs for these algorithms; we next discuss each of them.

**k-means clustering.** The \(k\)-means algorithm partitions a set of \(n\) data points \(a_1, \ldots, a_n\) into \(k\) groups of similar data points. We initially choose a centroid \(M^0\) for each cluster, i.e., representing the cluster centre (initialisation phase). In successive iterations, each data point is
assigned to the cluster with the closest centroid (assignment phase), after which the centroid is recomputed for each cluster (update phase). The algorithm terminates after a given number of iterations or after reaching convergence. Note that our user language does not support fixpoint computation, and hence checking convergence.

Figure 2 implements k-means. The set O of n input objects is retrieved using a loadData call. Each object is represented by a feature vector (i.e., array) of reals. We then load the parameters k, the number iter of iterations, and initialise cluster centroids M (line 3). The initialisation phase has a significant influence on the clustering outcome and convergence. We assume that initial centroids have been chosen, for example by using a heuristic [31]. Subsequently, an array InCl of Booleans is computed such that InCl[i][l] is True if and only if M[i] is the closest centroid to object O[l] (lines 5–10); every object is then assigned to its closest cluster. Since two clusters may be equidistant to an object, ties are broken using the breakTies2 call (line 11); it fixes an order of the clusters and enforces that each object is only assigned to the first of its potentially multiple closest clusters. Next, the new cluster centroids M[i] are computed as the centroids of each cluster (lines 12–16). The assignment and update phases are repeated iter times (line 4).

k-medoids clustering. The k-medoids algorithm is almost identical to k-means, but elects k cluster medoids rather than centroids: these are cluster members that minimise the sum of distances to all other objects in the cluster. The assignment phase is the same as for k-means (lines 5–11), while the update phase is more involved: We first compute an array DistSum of sums of distances between each cluster medoid and all other objects in its cluster (lines 12–17), then find one object in each cluster that minimises this sum (lines 18–24), and finally elect these objects as the new cluster medoids M (lines 25–27). The last step uses reduce_sum to select exactly one of the objects in a cluster as the new medoid, since for each fixed i only one value in Centre[i] is True due to the tie-breaker in line 24.

Markov clustering (MCL). MCL is a fast and scalable unsupervised clustering algorithm for graphs based on simulation of stochastic flow in graphs [36]. Natural clusters in a graph are characterised by the presence of many edges within a cluster and few edges across clusters. MCL simulates random walks within a graph by alternating two operations: expansion and inflation. Expansion corresponds to computing random walks of higher length. It associates new probabilities with all pairs of nodes, where one node is the point of departure and the other is the destination. Since higher length paths are more common within clusters than between different clusters, the probabilities associated with node pairs lying in the same cluster will, in general, be relatively large as there are many ways of going from one to the other. Inflation has the effect of boosting the probabilities of intrACLuster walks and denoting inter-cluster walks. This is achieved without a priori knowledge of cluster structure; it is the result of cluster structure being present.

Figure 3 gives the MCL user program. Expansion coincides with taking the power of a stochastic matrix M using the normal matrix product (i.e. matrix squaring). Inflation corresponds to taking the Hadamard power of a matrix (taking powers entry-wise). It is followed by a scaling step to maintain the stochastic property, i.e. the matrix elements correspond to probabilities that sum up to 1 in each column.

Section 3 discusses the probabilistic interpretation of the computation of the above three clustering algorithms.

LOOP := { {DECL} for ID in RANGE: {LOOP} } DECL := (ID = EXPR) | (‘{ID, ID ′ = EXT) EXPR := LIT | ID | [None] * EXPR | (EXPR COMP EXPR) | (REDUCE ‘{ LCOMP ‘} ) | (pow’ (EXPR, EXPR’)) | (invert’ (EXPR’)) | (EXPR ‘*’ EXPR) | (EXPR ‘+’ EXPR) | (scalar_mult’ (EXPR, EXPR’)) | (breakTies’ (EXPR’)) LCOMP := [EXPR for ID in RANGE if EXPR] REDUCE := reduce_and | reduce_or | reduce_sum | reduce_mult | reduce_count RANGE := range(EXPR, EXPR) COMP := ‘<’ | ‘>’ | ‘==’ | ‘<=’ | ‘>=’ EXT := loadData() | loadParams() | init() ID := An identifier LIT := A (Boolean, integer, float) literal

Figure 4: The grammar of the user language.

2.2 Syntax of the User Language

Figure 4 specifies the formal grammar for the language of user programs. A program is a sequence of declarations (DECL) and loop blocks (LOOP), each of which may again contain declarations and nested loops. The language allows to assign expressions (EXPR) to variable identifiers (ID). An expression may be a Boolean, integer, or float constant (LIT), an identifier, an array declaration, the result of a Boolean comparison between expressions, or the result of such operations as sum, product, inversion, or exponentiation. The result of a reduce operation on an anonymous array created through list comprehension (LCOMP), and the result of breaking ties in a Boolean array give rise to expressions; we elaborate on these two constructions below.

In addition to the syntactic structure as defined by the grammar, programs have to satisfy the following constraints: Bounded-range loops. The parameters to the range construct must be integer constants (or immutable integer-valued variables). This restriction ensures that for-loops (LOOP) and list comprehensions (LCOMP) are of bounded size that is known at compile time.

Anonymous arrays via list comprehension. List comprehension may only be used to construct one-dimensional arrays of base types, i.e., arrays of integers, floats, or Booleans. Breaking ties. Clustering algorithms require explicit handling of ties: For instance, if two objects are equidistant to two distinct cluster centroids in k-means, the algorithm has to decide which cluster the object will be assigned to. In ENFrame programs, the membership of objects to clusters can be encoded by a Boolean array like InCl such that InCl[i][l] is true if and only if object l is in cluster i. We explicitly break such ties using the function breakTies2(M). For each fixed value i of the second dimension (hence the 2 in the function name) of the 2-dimensional array M, it iterates over the first dimension of M and sets all but the first true value of M[i][l] to False. Symmetrically, the function breakTies1(M) fixes the dimension and breaks ties in the second dimension of M, and breakTies(M) breaks ties in a one-dimensional array.

3. TRACING COMputation BY EVENTS

The central concept for representing user programs in ENFrame is that of events. Each event is a concise syntactic encoding of a random variable and its probability
distribution. This section describes the syntax and semantics of events and event programs, and finally explains how ENFrame programs written in the user language from Section 2 can be translated to event programs.

The key features of events and event programs are:

- Events can encode arbitrarily correlated, discrete probability distributions over input objects. In particular, they can succinctly encode instances of such formalisms as Bayesian networks and pc-tables. The input objects and their correlations can be explicitly provided, or imported via a positive relational algebra query with aggregates over pc-tables \[1\].
- By allowing non-Boolean events, our encoding is exponentially more succinct than an equivalent purely Boolean description.
- Each event has a well-defined probabilistic semantics that allows to interpret it as a random variable.
- The iterative nature of many clustering algorithms carries over to event programs, in which events can be defined by means of nested loops. This construction together with the ability to reuse existing, named events in the definition of new, more complex events leads to a concise encoding of a large number of distinct events.

Example 1. Clustering in possible worlds. We start by presenting an instructive example of k-medoids clustering under possible worlds semantics. Let \( o_0, \ldots, o_3 \) be objects in the feature space as shown below. They can be clustered into two clusters using k-medoids with medoids \( o_1 \) and \( o_2 \).

In order to go from deterministic to uncertain objects, we associate each object \( o_i \) with a Boolean propositional formula \( \Phi(o_i) \) – the event – over a set of independent Boolean random variables \( X \). The possible valuations \( v : X \rightarrow \{ true, false \} \) define the possible worlds of the input objects; for each valuation \( v \) there exists on world that contains exactly those objects \( o_i \) for which \( \Phi(o_i) \) is true under \( v \). The probability of a world is the product of the probabilities of the variables \( x \in X \) taking a truth value \( v(x) \).

Let us assume that the objects have the following events:

\[
\Phi(o_0) = x_1 \lor x_3, \quad \Phi(o_1) = x_2, \quad \Phi(o_2) = x_4, \quad \Phi(o_3) = \neg x_2 \land x_4.
\]

Distinct worlds can have different clustering results, as exemplified next. The world defined by \( \{ x_1 \mapsto \top, x_2 \mapsto \bot, x_3 \mapsto \top, x_4 \mapsto \top \} \) consists of objects \( o_0, o_2, \) and \( o_4 \), for which k-medoids clustering yields:

Similarly, the worlds defined by \( \{ x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top \} \) and any assignment for \( x_4 \), yields:

The probability of a query “Are \( o_1 \) and \( o_2 \) in the same cluster?” is the sum of the worlds in which \( o_1 \) and \( o_2 \) are in the same cluster.

Events do not only encode the correlations and probabilities of input objects, but can symbolically encode the entire clustering process. We illustrate this in the next example.

Example 2. Symbolic encoding of k-means by events. We again assume four input objects \( o_0, \ldots, o_3 \) with their respective events \( \Phi(o_i) \). This example introduces conditional values (c-values) which are expressions of the form \( \Phi \otimes v \), where \( \Phi \) is a Boolean formula and \( v \) is a vector from the feature space. Intuitively, this c-value takes the value \( v \) whenever \( \Phi \) evaluates to true, and a special undefined value when \( \Phi \) is false. C-values can be added and multiplied; for example, the expression \( \Phi \otimes v + \Psi \otimes w \) evaluates to \( v + w \) if \( \Phi \) and \( \Psi \) are true, or to \( v \) if \( \Phi \) is true and \( \Psi \) is false, etc.

Equipped with c-values, an initialisation of k-means with \( k = 2 \) can for instance be written in terms of two expressions \( M^0 = \Phi(o_0) \otimes o_0 + \Phi(o_3) \otimes o_3 \) and \( M^0 = \top \otimes 0.5 \cdot (o_1 + o_3) \): Centroid \( M^0 \) is set to object \( o_0 \) if \( \Phi(o_0) \) is true and to \( o_3 \) if \( \Phi(o_3) \) is false; centroid \( M^1 \) is always set to the geometric centre of \( o_1 \) and \( o_3 \).

In the assignment phase, each object is assigned to its nearest cluster centroid. The condition InCl\(i,d\) for object \( o_i \) being closest to \( M^d \) can be written as the Boolean event \( \text{InCl}^{i,d} \equiv \bigwedge_{j=0}^{3} [\text{dist}(\Phi(o_i) \otimes o_j, M^d) \leq \text{dist}(\Phi(o_i) \otimes o_j, M^d)] \), which encodes that the distance from \( o_i \) to centroid \( M^d \) is smaller than the distance to centroid \( M^d \).

Given the Boolean events \( \text{InCl}^{i,d} \), we can represent the centroid of cluster \( i \) for the next iteration by the expression \( \left( \sum_{d=0}^{3} \text{InCl}^{i,d} \otimes 1 \right)^{-1} \cdot \left( \sum_{d=0}^{3} \text{InCl}^{i,d} \otimes o_i \right) \), which specifies a random variable over possible cluster centroids conditioned on the assignments of objects to clusters as encoded by \( \text{InCl}^{i,d} \). This expression is exponentially more succinct than an equivalent purely Boolean encoding of cluster centroids, since the later would require one Boolean expression for each subset of the four input objects.

The event programs corresponding to the three user programs for k-means, k-medoids, and MCL are given on the right side of Figures \ref{fig:example1}\ref{fig:example2}\ref{fig:example3}. In addition to the constructs introduced in Example 2 they use event declarations that assign identifiers to event expressions, and \( \forall \)-loops that specify sets of events parametrised by \( i \). The remainder of this section specifies the formal syntax and semantics of event programs, and gives a translation from user to event programs.

3.1 Syntax of Event Expressions

The grammar for event expressions is as follows:

- \( \text{VAL} ::= \text{A scalar or feature vector} \)
- \( \text{INT} ::= \text{Any integer} \)
- \( \text{CVAL} ::= \text{EVENT} \otimes \text{VAL} | \text{CVAL}^{-1} | \text{CVAL} \cdot \text{CVAL} | \text{CVAL} \text{INT} | \text{CVAL} \cdot \text{INT} | \text{dist} \text{(CVAL,CVAL)} | \text{EVENT} \land \text{CVAL} \)
- \( \text{COMP} ::= \leq | \geq | = | < | > \)
- \( \text{ATOM} ::= \text{[CVAL COMP CVAL]} \)
- \( \text{EID} ::= \text{Elements of a set of event identifiers} \)
- \( \text{EVENT} ::= \text{Propositional formula over X, EID, ATOM} \)

The main constructs are:

- **Conditional values.** Reals and feature vectors are denoted by VAL. Together with a propositional formula, they give rise to a conditional value (CVAL), c-value for short.
- **Functions of conditional values.** Very much like scalars and feature vectors, c-values can be added, multiplied, and exponentiated. Additionally, the distance between two c-values yields another c-value. In addition to the binary operations specified in the grammar (e.g., \( \text{CVAL} \cdot \text{CVAL} \)), we allow \( \sum \) and \( \prod \) expressions (see Figure 2).

**Event expressions.** Event expressions (EVENT) are propositional formulas over constants \( \top \) (true), \( \bot \) (false), a set \( X \) of Boolean random variables, event identifiers, and
propositions defined by ATOM: [CVAL COMP CVAL] represents the truth value obtained by comparing two c-values.

### 3.2 Semantics of Event Expressions

The semantics of event expressions is defined by extending a Boolean valuation \( \nu : X \to \{\text{true, false}\} \) to a valuation of c-values and event expressions. We define in the sequel how \( \nu \) acts on each of the expression types in the grammar. The base cases of this mapping are the standard algebraic operations on scalars and the feature space, extended by special undefined elements as follows.

We extend the reals (and their operations +, −, ·, (−)−1) by a special element \( u \) (for undefined) such that \( 0^{-1} = u \). Operators +, ·, propagate \( u \) as \( u + x = x \) and \( u \cdot x = u \) for any real \( x \). For any other reals \( x, y, + \) and \( \cdot \) are as usual. For example, \( 5 \cdot (3 - 3)^{-1} = 5 \cdot u = u \).

Similarly, we extend the feature space by an element \( \mathbf{u} \) for vector-valued c-values; the recursively constructed \( \nu(CVAL) \) is only defined for vector-valued variable symbols \( x, y \).

**CVAL.** Conditional values of the form \( \text{EVENT} \odot \text{VAL} \) have an if-then-else semantics: if \( \text{EVENT} \) evaluates to \( \text{true} \), then \( \text{EVENT} \odot \text{VAL} \) evaluates to \( \text{VAL} \), else it evaluates to \( \text{u} \) (or \( \text{u} \) for vector-valued c-values); the recursively constructed CVAL expressions have the natural recursive semantics that ultimately defaults to \(+\) and \(\cdot\) for scalars and feature vectors.

\[
\nu(\text{EVENT} \odot \text{VAL}) = \begin{cases} 
\text{VAL}, & \text{if } \nu(\text{EVENT}) = \text{true} \\
\text{u}, & \text{if } \nu(\text{EVENT}) = \text{false} \\
\nu(\text{CVAL1} \odot \text{CVAL2}), & \text{otherwise}
\end{cases}
\]

\[
\nu(\text{CVAL1} \odot \text{CVAL2}) = \nu(\text{CVAL1}) + \nu(\text{CVAL2})
\]

\[
\nu(\text{CVAL1} \odot \text{CVAL2}) = \nu(\text{CVAL1}) \cdot \nu(\text{CVAL2})
\]

\[
\nu(\text{CVAL}^{-1}) = \nu(\text{CVAL})^{-1}
\]

\[
\nu(\text{dist}(\text{CVAL1}, \text{CVAL2})) = \begin{cases} 
\text{u}, & \text{if } \nu(\text{CVAL1}) = \text{u} \\
\nu(\text{CVAL2}), & \text{if } \nu(\text{CVAL1}) \neq \text{u} \\
\nu(\text{dist}(\nu(\text{CVAL1}), \nu(\text{CVAL2}))), & \text{otherwise}
\end{cases}
\]

\[
\nu(\text{CVAL}^\text{INT}) = \nu(\text{CVAL})^\text{INT}
\]

\[
\nu(\text{EVENT} \land \text{VAL}) = \begin{cases} 
\nu(\text{CVAL}), & \text{if } \nu(\text{EVENT}) = \text{true} \\
\text{u}, & \text{if } \nu(\text{EVENT}) = \text{false} \\
\nu(\text{EVENT1} \land \text{EVENT2}), & \text{otherwise}
\end{cases}
\]

**ATOM, EVENT.** Comparisons \( \nu([\text{CVAL1} \theta \text{CVAL2}]) \) for \( \theta \in \{\leq, \geq, =, <, >\} \) between two c-values evaluate to \( \text{false} \) if they are both defined \( \nu(\text{CVAL1}) \neq \text{u} \) and \( \nu(\text{CVAL2}) \neq \text{u} \) and the comparison does not hold; otherwise (i.e. at least one of the c-values is undefined, or if the comparison holds), it evaluates to \( \text{true} \). The semantics of the Boolean propositional EVENT expressions is standard, i.e. by propagating \( \nu \) through the propositional operators \( \land, \lor, \neg \). For instance \( \nu(\text{EVENT1} \land \text{EVENT2}) \) evaluates to \( \text{true} \) if \( \nu(\text{EVENT1}) = \nu(\text{EVENT2}) = \text{true} \), and to \( \text{false} \) otherwise.

### 3.3 Probabilistic Semantics of Events

We next give a probabilistic interpretation of event expressions that explains how they can be understood as random variables: Boolean event expressions (EVENT) give rise to Boolean random variables, and conditional values (CVAL) give rise to random variables over their respective domain.

For every random variable \( x \in X \), we denote by \( P_x[\text{true}] \) and \( P_x[\text{false}] \) the probability that \( x \) is \( \text{true} \) or \( \text{false} \), respectively; we also simply write \( P_x \) for \( P_x[\text{true}] \).

Let \( \Omega = \{\nu : X \to \{\text{true, false}\}\} \) be the set of mappings from the random variables \( X \) to \( \text{true} \) and \( \text{false} \).

**Definition 1 (Induced Probability Space).** Together, the probability mass function \( \text{Pr}(\nu) = \prod_{x \in X} P_x[\nu(x)] \) for every sample \( \nu \in \Omega \), and the probability measure \( \text{Pr}(E) = \sum_{\nu \in E} \text{Pr}(\nu) \) for \( E \subseteq \Omega \) define a probability space \( (\Omega, 2^\Omega, \text{Pr}) \) that we call the probability space induced by \( X \).

An event expression \( E \) is a random variable over the probability space induced by \( X \) with probability distribution \( P_E[s] = \text{Pr}\{\nu \in \Omega : (\nu(E) = s)\} = \sum_{\nu \in \Omega : \nu(E) = s} \text{Pr}(\nu) \).

By virtue of this definition, every Boolean event expression becomes a Boolean random variable, and real-valued (vector-valued) c-values become random variables over the reals (the feature space).

### 3.4 Event Programs

Event programs are imperative specifications that define a finite set of named c-values and event expressions. The grammar for event programs is as follows:

\[
\text{INT} ::= \text{A positive integer} \\
\text{VAR} ::= \text{A variable symbol} \\
\text{LOOP} ::= \{\text{DECL}\} \ (\forall \text{VAR} \in \text{INT}..\text{INT}: \{\text{LOOP}\} \} \\
\text{DECL} ::= \text{EID} \equiv \text{EVENT}
\]

Event programs consist of a sequence of event declarations (DECL) and nested loops (LOOP) of event declarations.

A central concept is that of event identifiers (EID); it is required that event declarations are immutable, i.e. each distinct EID may only be assigned once to an event expression. Inside a \( \forall \)-loop, identifiers can be parametrised by \( i \) to create a distinct identifier in each iteration of the loop.

The meaning of an event program is simply the set of all named and grounded c-value and event expressions defined by the program; grounded here means that all identifiers in expressions are recursively resolved and replaced by the referenced expressions. For declarations outside of loops, this is clear; each declaration inside of (nested loops) is instantiated for each value of the loop counter variables.

### 3.5 From User Programs to Event Programs

The translation of user to event programs has two main challenges: (i) Translating immutable variables and arrays to immutable events, and (ii) translating function calls such as \text{reduce}. We cover these two issues separately.

**From mutable variables to immutable events.** It is natural to reassign variables in user language programs, for example when updating \( k \)-means centroids in each iteration based on the cluster configuration of the previous iteration. In contrast, events in event programs are immutable, i.e., can be assigned only once. The translation from the user language to the event language utilises a function \text{getLabel} that generates for each user language variable \( M \) a sequence of unique event identifiers whose lexicographic order reflects the sequence of assignments of \( M \).

The basic idea of \text{getLabel} is to first identify the nested loop blocks of the given user language program, and then to establish a counter for each distinct variable symbol \( M \) and each block. An assignment of a variable within \( k \) nested blocks corresponds to an event identifier of the form \( M_{c_1, \ldots, c_k} \) where \( c_1, \ldots, c_k \) are the \( k \) counters for the \( k \) blocks. Within each block, its corresponding counter is incremented for every assignment of its variable symbol. When going from one block into a nested inner block, the counters for the outer blocks are kept constant while the counter for the inner block is incremented as \( M \) is reassigned in the inner block.
Example 3. Consider the following user language program (left) and its translation to an event program (right).

Translation of arrays. Since arrays in a user language program have a known fixed size, their translation is straightforward: A k-dimensional array \( M[n_1]...[n_k] \) translates to \( \bigoplus_{i=1}^{n_1} \bigoplus_{i=2}^{n_2} \cdots \bigoplus_{i=k}^{n_k} M_{i_1...i_k} \). Translation of reduce_* calls. According to the grammar in Figure 4, reduce-operations can only be applied to anonymous arrays created by list comprehension. The expression \( \text{reduce and}(\text{EXPR} \text{ for ID in range(FROM, TO) if COND}) \) is translated to the Boolean event \( T_{\text{ID=FROM}} \land \text{COND} \land \text{EXPR} \). Symmetrically, \( \text{reduce or} \) translates to \( \lor \) \( \bigoplus \) \( \text{reduce_sum} \) to \( \bigoplus \) \( \bigoplus \) \( \sum \) \( \text{reduce_mult} \) to \( \bigcup \). A call to \( \text{reduce count}(\text{EXPR} \text{ for ID in range(FROM, TO) if COND}) \) translates to the event \( \sum_{\text{ID=FROM}}^{\text{TO}} \text{COND} \land \text{ID} \). The goal is to compute probabilities for the top nodes in the network, which are referred to as compilation targets. These nodes represent events such as “object detection” or “motion”. For practical reasons, we do not construct a full tree, but limit the number of branches to a fixed number. We keep lower and upper bounds for the probability of each target. Initially, these bounds are \([0,1]\) and they eventually converge during computation.

# 4. PROBABILITY COMPUTATION

The probability computation problem is known to be #P-hard already for simple events representing propositional formulas such as positive bipartite formulas in disjunctive normal form. In ENFrame, we need to compute probabilities of a large number of interconnected complex events. Although the worst-case complexity remains hard, we attack the problem with three complementary techniques: (1) bulk-compile all events into one decision tree while exploiting the structure of the events to obtain smaller trees, (2) employ approximation techniques to prune significant parts of the decision tree, and ultimately (3) distribute the compilation by assigning distinct distributed workers to explore disjoint parts of the tree.

We next introduce the bulk-compilation technique, look at three approximation approaches, and discuss how to distribute the probability computation.

4.1 Compilation of event programs

The event programs consist of interconnected events, which are represented in an event network: a graph representation of the event programs, in which nodes are, e.g., Boolean connectives, comparisons, aggregates, and \(\varepsilon\)-values. An example of such a network is depicted in Figure 5.

The bulk-compilation procedure is based on Shannon expansion: select an input random variable \( x \) and partially evaluate each compilation target \( \Phi_{i_x} \) for \( x \) being set to \( x \) \( \{T\} \) and \( \Phi_{\neg i_x} \) for \( x \) being set to \( \{F\} \). Then, the probability of \( \Phi \) is defined by \( \text{Pr}(\Phi) = \text{Pr}(x) \cdot \text{Pr}([x] \lor \text{Pr}(\Phi | x) - \text{Pr}([-x] \lor \text{Pr}(\Phi | \neg x)) \cdot \text{Pr}(\Phi | x) \). We are now left with two simpler events \( \Phi_1 \) and \( \Phi_{\neg 1} \). By repeating this procedure, we eventually resolve all variables in the events to the constants \( \text{true} \) or \( \text{false} \). The trace of this repeated expansion is the decision tree. We need not materialise the tree. Instead, we just explore it depth-first and collect the probabilities of all visited branches as well as record for each event \( \Phi \) the sums \( L(\Phi) \) and \( N(\Phi) \) of probabilities of those branches that satisfied and respectively did not satisfy the event. At any time, \( L(\Phi) \) and \( 1 - N(\Phi) \) represent a lower bound and respectively an upper bound on the probability of \( \Phi \).

This compilation procedure needs time polynomial in the network size and in the size of the input data set, yet in worst case (unavoidably) exponential in the number of variables used by the events.

For practical reasons, we do not construct \( \Phi_1 \) and \( \Phi_{\neg 1} \) explicitly, but keep minimal information that, in addition to the network, can uniquely define them. The process of computing this minimal information is called masking. We achieve this by traversing the network bottom-up and re-numbering the nodes that become \( \text{true} \) or \( \text{false} \) given the values of their children. When a compilation target \( t \) is
eventually masked by a variable assignment \( \nu \), the probability \( \Pr(\nu) \) is added to its lower bound if \( \nu(t) = \top \), or subtracted from its upper bound if \( \nu(t) = \bot \). If one or more targets are left unmasked, a next variable \( x' \) is chosen and the process is repeated with \( \nu' = \nu \cup \{x' \mapsto c\} \), where \( c \) is either \( \top \) or \( \bot \). The algorithm chooses a next variable \( x' \) such that it influences as many events as possible.

Once all compilation targets are masked by an assignment \( \nu \), the compilation backtracks and selects a different assignment for the most recently chosen variable whose assignments are not exhausted. When all branches of the decision tree have been investigated, the probability bounds of the targets have necessarily converged and the algorithm terminates.

**Example 4.** Figure 5 shows a simplified event network under the assignment \( \{x_0 \mapsto \top, x_1 \mapsto \top\} \). The masks of \( x_0 \) and \( x_1 \) are propagated to event nodes \( \Phi(o_0), \Phi(o_1), \Phi(o_3) \), which are now also masked. The red nodes are masked for \( \nu \) and the blue lines represent that \( \nu \) is chosen and \( \nu \) is added to its lower bound if \( \nu(t) = \top \) or \( \nu(t) = \bot \). The green nodes are masked for \( \bot \), whereas the green nodes are masked for \( \top \). \( \Box \)

Algorithm 1 gives the pseudocode for the DFS-traversal of the decision tree. The blue lines are necessary for approximate probability computation and will be explained later. Compilation starts with an empty branch (variable assignment) \( \nu \); the mask values \( M[\nu] \) and probability bounds for all nodes in the event network are initialised. The error budgets \( E[t_i] \) for the targets are set to 0 for exact computation. After the initialisation, the DFS procedure is called using \( \nu = \{\} \). The procedure selects the first variable \( x \) and recursively calls itself using two newly created branches of the decision tree: one for \( \nu = \{x \mapsto \top\} \) and one with \( \nu = \{x \mapsto \bot\} \). These branches are propagated into the event network using the \( \text{Mask} \) procedure. If every target is reached, DFS returns. Otherwise, it selects a next variable \( x \) and recursively calls DFS on the two new tree branches.

Algorithm 2 performs mask propagation: a mask (assignment) for a variable \( x \) is inserted into the network, and the variable node propagates the mask to its parent nodes. Depending on the event node, its node mask is either updated and propagated further, or propagation is stopped in case a mask cannot be established.

Convergence of the algorithm (e.g., clustering) can be detected by comparing the mask values at network nodes corresponding to iteration \( t \) with the masks of nodes for iteration \( t + 1 \). If none of the mask assignments has changed between iterations, then the algorithm has converged.

### 4.2 Bounded-range loops in event networks

Event programs can contain bounded-range loops for iterative algorithms. ENFrame offers two ways of encoding such loops in an event network: *unfolded*, in which case the events at any loop iteration are explicitly stored as distinct nodes in the network, or a more efficient *folded* approach in which all iterations are captured into a single set of nodes. The compilation of the network then involves looping. The pseudocode in Algorithms 1 and 2 assumes unfolded event networks. They need minor modifications to work on folded networks: the mask data structure \( M \) becomes two-dimensional to be able to store the mask for a node \( v \) at any iteration \( t \) \((M[t][v])\) the DFS procedure needs an additional parameter \( t \) for the current iteration, and the network requires an additional node to perform the transition from iteration \( t \) to iteration \( t + 1 \). The extra logic required for the \( \text{Mask} \) function is:

- loop nodes \( M[t+1][\nu] \rightarrow M[t][\nu] \) carry over mask to next iteration
- increase iteration counter

### 4.3 Approximation with error guarantees

The compilation procedure can be extended to achieve an anytime absolute \( \varepsilon \)-approximation with error guarantees. The idea is to stop the probability computation as soon as the bounds of all compilation targets are sufficiently tight.

**Definition 2.** Given a fixed error \( 0 \leq \varepsilon \leq 1 \) and events \( (t_0, \ldots, t_{n-1}) \) with probabilities \( (p_0, \ldots, p_{n-1}) \). An absolute \( \varepsilon \)-approximation for these events is defined as a tuple \((p_0, \ldots, p_{n-1})\) such that \( \forall 0 \leq i < n: p_i - \varepsilon \leq p_i \leq p_i + \varepsilon \).

The compilation of the network yields probability bounds \((L_0, U_0, \ldots, L_{n-1}, U_{n-1})\) for the targets \( (t_0, \ldots, t_{n-1}) \). It can easily seen that an absolute \( \varepsilon \)-approximation can be defined by any tuple \((p_0, \ldots, p_{n-1})\) such that \( \forall 0 \leq i < n: U_i - \varepsilon \leq L_i \leq U_i + \varepsilon \). We thus need to run the algorithm until \( U_i - L_i \leq 2\varepsilon \) for each target \( t_i \).

There exist multiple strategies for investing this \( 2\varepsilon \) error budget for every target. We next discuss three such strategies. The lazy scheme follows the exact probability computation approach and stops as soon as the bounds become tight enough. Effectively, this results in investing the entire error budget into the rightmost branches of the decision tree. The eager scheme spends the entire error budget as soon as possible, i.e., on the leftmost branches of the decision tree, and then continues as in the case of exact computation. At each node in the decision tree, the hybrid scheme divides the current error budget equally over the two branches. Any residual, unused error budget is transferred to the next branch.

The blue lines in Algorithm 1 show how the DFS procedure can be extended to support anytime absolute \( \varepsilon \)-approximation with error guarantees using the hybrid scheme. The DFS procedure is called using a non-zero error budget \( 2\varepsilon \), and it assigns half of the budget to the newly created left branch of the decision tree. The recursive DFS call returns the residual error budget of each target, which is then added to the budget for the right branch.

### 4.4 Distributed probability computation

By splitting the task of exploring the decision tree in a number of jobs, the compilation can be performed concurrently by multiple threads or machines. A worker explores a tree fragment of a given maximum size. For simplicity, we define the size of a job to be the depth \( d \) of the sub-tree to explore. The computation then proceeds as follows. One worker explores the tree from the root and every time it reaches depth \( d \), it forks a new job that continues from that node as its root. Given that the maximum depth of the tree is the number of variables \( m \), the number of jobs created is
Our goal is to show that ENFrame can deal with common correlation patterns that occur in probabilistic data. We use a data set describing network load and occurrences of partial discharge sensors installed on switchgear in energy distribution networks, partitioned in mutex sets of cardinality (at most) m: any two points are mutually exclusive within a mutex set and independent across the sets. The conditional correlations scheme expresses uncertainty as a Markov chain, using one node per data point. Let \( \Phi_i \) be the event that the data point \( o_i \) exists. The event \( \Phi_{i+1} \) becomes \( (\Phi_i \lor x_{i+1}) \lor (\neg \Phi_i \land x_{i+1}) \); it is a disjunction of two events, for the cases that \( o_i \) exists or not. We thus introduce two new Boolean random variables \( x_{i+1} \) and \( x_{i+1}' \) per data point \( o_{i+1} \). For every correlation scheme, a group size of 4 has been used, i.e. data points were divided in groups with identical lineage. This is realistic for uncertain time-series sensor data; readings from a small time window have identical correlations and uncertainty. Additionally, we show experiments with a varying fraction of certain data points.

### 5. EXPERIMENTAL EVALUATION

This section describes an experimental evaluation of clustering probabilistic data using ENFrame. The focus of this evaluation is a preliminary benchmark of the performance of the probability computation algorithms introduced in Section 4. At the end of this section, we comment on further experimental considerations that could not be included in full due to space limitations.

#### Data

We use a data set describing network load and occurrences of partial discharge in energy distribution networks. This data is gathered from two different types of sensors: partial discharge sensors installed on switchgear and cables in substations of the distribution network, and network load sensors in substations. We aggregate the number of partial discharge occurrences over the duration of an hour and subsequently pair this value with the average network load during that hour. Clustering can assist in detecting anomalies and predicting failures in the energy networks.

#### Uncertainty

Our goal is to show that ENFrame can deal with common correlation patterns that occur in probabilistic data. Each data point is associated with an event described by Boolean random variables, whose probabilities for true are chosen at random from the range \([0.5, 0.8] \). Different values would make the probabilities of clustering events too close to 0 or 1 which are then easily approximated. The experiments were carried out using three types of correlations to illustrate ENFrame’s capability to process arbitrarily correlated data.

The positive correlations scheme yields events such that two data points are either positively correlated or independent. Each event is a disjunction of \( l \) distinct positive literals. In the mutex correlations scheme, the data points are partitioned in mutex sets of cardinality (at most) \( m \): any two points are mutually exclusive within a mutex set and independent across the sets. The conditional correlations scheme expresses uncertainty as a Markov chain, using one node per data point. Let \( \Phi_i \) be the event that the data point \( o_i \) exists. The event \( \Phi_{i+1} \) becomes \( (\Phi_i \lor x_{i+1}) \lor (\neg \Phi_i \land x_{i+1}) \); it is a disjunction of two events, for the cases that \( o_i \) exists or not. We thus introduce two new Boolean random variables \( x_{i+1} \) and \( x_{i+1}' \) per data point \( o_{i+1} \). For every correlation scheme, a group size of 4 has been used, i.e. data points were divided in groups with identical lineage. This is realistic for uncertain time-series sensor data; readings from a small time window have identical correlations and uncertainty. Additionally, we show experiments with a varying fraction of certain data points.

#### Algorithms

We report on performance benchmarks for k-means clustering on the energy network data set, comparing ENFrame to naive clustering. The naive approach computes an equivalent clustering by explicitly iterating over all possible worlds. We show the performance of multiple probability computation algorithms of ENFrame: the sequential exact approach, three sequential approximation schemes (eager, lazy, hybrid), and distributed hybrid approximation (hybrid-d). All approximation algorithms are set to compute probabilities with an (absolute) error of at most \( \varepsilon = 0.1 \), the compilation targets are the events that represent medoid selection.

Algorithms described in the literature (see Section 9) simplify the clustering problem by ignoring correlations, using expected distances, and producing a deterministic output. They might outperform our sequential algorithms, at the cost of producing an output that can be arbitrarily off from the golden standard of clustering in each world. Unfortunately, none of the reported prototypes was available for testing at the time of writing.

#### Setup

The experiments were carried out on an Intel Xeon X5660/2.80GHz machine with 4GB of RAM, running Ubuntu with Linux kernel 3.5. The timings reported for hybrid-d were obtained by simulating distributed computation on a single machine. The algorithms are implemented in C++ (GCC 4.7.2). Each plot in Figures 6 and 7 depicts average performance with min/max ranges of five runs with randomly generated event expressions, different probabilities, and three clustering iterations (using Euclidian distance).

#### Sequential algorithms

Figures 6 and 7 show that all of ENFrame’s probability computation algorithms outperform the naive algorithm by up to six orders of magnitude for each data set with more than 10 variables. Furthermore, the hybrid approximation can be up to four orders of magnitude faster than exact computation.

Indeed, for a very small number of possible worlds (i.e., a small number of variables), it pays off to cluster individually in each world and avoid the overhead of the event networks. For a larger number of worlds, our exact and approximate approaches quickly become up to six orders of magnitude faster. The naive method times out for over 25 variables in every correlation scheme.

The reason why our approximation schemes outperform exact is as follows. For a given depth \( d \), there are up to \( 2^d \)
nodes in the decision tree that contribute to the probability mass of a node in the event network. The contributed mass decreases exponentially with an increase in depth, which suggests that most nodes in the decision tree only contribute a small fraction of the overall probability mass. Depending on the desired error bound, a shallow exploration of the decision tree could be enough to achieve a sufficiently large probability mass.

Among the approximation algorithms, hybrid performs best; it outperforms exact by up to four orders of magnitude since it does only need to traverse a shallow prefix of the decision tree. The algorithm invests the error budget over the entire width of the decision tree, cutting branches of the tree after a certain depth. The other two methods (eager and lazy) use the budget to respectively cut the first and last branches, while exploring other branches in full depth.

For positive correlations, lazy performs very well, because the decision tree is very unbalanced under this scheme. The left branches of the tree correspond to variables being set to true, which quickly satisfy the (disjunctive) input events and allow for compilation targets to be reached. Further to the right, branches correspond to variables being set to false. More variables need to be set to (un)satisfy the disjunctive input event, and allow for compilation targets to be reached. This means that the lazy algorithm saves the error budget until the very last moment and can therefore prune the deep branches whilst maintaining the \( \epsilon \)-approximation. The decision trees for the mutex and conditional correlation schemes are more balanced, resulting in both lazy and eager to perform almost identically to exact. Hence, they are not shown in Figure 7.

**Distributed algorithms.** By distributing the probability computation task, we can significantly increase ENFrame’s performance. Figures 6 and 7 show the timings for hybrid using \( w = 16 \) workers and job size \( d = 3 \) (as detailed below). For all correlation schemes, hybrid gets increasingly faster than hybrid as we increase the number of variables. For small numbers of variables, the overhead of distribution does not pay off. The benefits are best seen for mutex correlations and over 100 objects (over 60 variables), where hybrid becomes more than one order of magnitude faster than hybrid. For readability reasons, the performance of hybrid is not depicted in Figure 6 (right); its performance is up to one order of magnitude better than hybrid, as can be seen in Figure 6. For ten variables, there is only a small performance gain when compared to the single-threaded hybrid approximation: the decision tree remains small, as is the number of jobs that can be generated. However, for 30 and 50 variables, hybrid yields a performance improvement of more than one order of magnitude when compared to hybrid.

Figure 8 shows the influence of the number of workers on hybrid’s performance for varying job sizes. A job is the work unit allocated to a worker at any one time; a size of \( d \) means that the worker has to explore a fragment of the decision tree of depth at most \( d \) and would need to traverse the event network at most \( 2^d \) times. For large job sizes, the overall number of jobs decreases; in the case of positive correlations, the number of jobs of size 9 is small since the decision tree is very unbalanced and only a few branches on the right-hand side of the tree grow deeper than nine variables. Therefore, increasing the number of workers would not help; indeed, there is no improvement for more than four workers for job sizes larger than 5. However, for a job size of 3, up to 16 workers can still be beneficial. In our experiment, smaller job sizes led to a performance gain of up to one order of magnitude, since they allow for a more equal distribution of the work over the available workers. Synchronisation did not play a significant role in our setup.

**Certain data points.** Figure 8 shows that the performance improves as the number of certain data points (i.e., objects that occur in all possible worlds) increases. The speedup in such cases is explained by the fact that the distance sums of possible medoids to data points in a cluster become less complex and can be initialised using the distances to objects that certainly exist. Consequently, fewer var-
Further findings. We have investigated the influence of the number of dimensions, data point coordinates, the error budget, the numbers of iterations, and alternative clustering compilation targets on the performance of ENFrame, as well as its total memory usage. As is the case with traditional \(k\)-medoids on certain data, the number of dimensions has no influence on the computation time. The reported performance gap between exact and hybrid shows that performance is highly sensitive to the error budget. The number of iterations has a linear effect on the running time of the algorithm. The number of targets (including targets representing co-occurrence queries) has a minor influence on performance; due to the combinatorial nature of \(k\)-medoids, clustering events are mostly satisfied in bulk and it is thus very rare that one event alone is satisfied at any one time. This also explains why experiments with other types of compilation targets (e.g., object-cluster assignment, pairwise object-cluster assignment) show very similar performance. In our experiments, the size of the event networks grows linearly in the number of objects and clusters and the memory usage of ENFrame is under 1GB.

Comments on clustering quality. The research effort described in this paper is mainly concerned with expressing a rich class of algorithms for data analysis in ENFrame, and scalability of the probability computation task; \(k\)-medoids clustering is merely a use-case to show ENFrame’s flexibility and scalability. The adaptation of \(k\)-medoids to ENFrame has the exact same quality as the “golden standard”: \(k\)-medoids applied in each possible world, yet without actually explicitly iterating over all possible worlds. This is not the case for prior work that does not support correlated uncertain input and uncertain output. An extensive quality comparison is out of scope of this paper, but is part of future work. A common approach to assess the quality measure of a probabilistic method like ours, is to assume a notion of ground truth. It is unclear how to deal with probabilistic data which represents inherently contradictory information for which no ground truth exists or is known.

6. RELATED WORK

Our work is at the confluence of several active research areas: probabilistic data management, data analytics platforms, and provenance data management.

Probabilistic data mining and querying. Our work adds to a wealth of literature on this topic [1, 2, 5] along two directions: distributed probability computation techniques and a unified formalisation of several clustering algorithms in line with work on probabilistic databases.
A key aspect that differentiates ENFrame from the above platforms is the probabilistic nature of input data and of the whole computation process. This calls for specifically tailored algorithms. So far, ENFrame lacks the scalability achievable by the above platforms, since it only distributes the probability computation task, while the actual mining task is performed on one machine. The next step is to consider a fully distributed computational approach.

**Provenance in database and workflow systems.** To enable probability computation, we trace fine-grained provenance of the user computation. This is in line with a wealth of work in probabilistic databases [34]. Our event language is influenced by work on provenance semirings [16] and semimodules [4][25] that capture provenance for positive queries with aggregates in relational databases. The construct $\Phi \otimes v$ resembles the algebraic structure of a semimodule that is a tensor product of the Boolean semiring $\mathbb{B}[X]$ freely generated by the variable set $X$ and of the SUM monoid over the real numbers $\mathbb{R}$. There are two differences between our construct and these structures. Firstly, we allow negation in events, which is not captured by the Boolean semiring. Secondly, even for positive events, $\mathbb{B}[X] \otimes \mathbb{R}$ is not a semimodule since it violates the following law: $(\Phi_1 \otimes \Phi_2) \otimes v = \Phi_1 \otimes v + \Phi_2 \otimes v$. Indeed, under an assignment that maps both $\Phi_1$ and $\Phi_2$ to $\top$, the left side of the equality evaluates to $v$, whereas the right side becomes $v + v$. Furthermore, our event language allows to define events via iterations, as needed to succinctly trace data mining computation.

Workflows employ a much wider variety of programming constructs than databases. Workflow provenance aims to capture a complete description of evaluation of a workflow [12], though it sees tasks as black-boxes, and therefore, consider all outputs of a task to depend on all of its inputs. This provenance model is too coarse to support exact derivations of output as needed in our case for probability computation.

A distinct line of work is on such relational provenance systems as Perm [15], DBNotes [6], Orchestra [23] that trace provenance using query rewriting or modified query operators. Panda [22] enables provenance-aware querying.

**Acknowledgements.** This research was supported by EPSRC grant agreement ADEPT (EP/I00194/1).

7. REFERENCES

[1] C. Aggarwal. Managing and Mining Uncertain Data. Springer, 2009.
[2] C. Aggarwal and C. Reddy. Data Clustering: Algorithms and Applications, chapter A Survey of Uncertain Data Clustering Algorithms. Chapman and Hall, 2013.
[3] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Suguha, and J. Widom. Trio: A system for data, uncertainty, and lineage. In VLDB, 2006.
[4] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In PODS, 2011.
[5] Apache Software Foundation. The Mahout machine learning library. http://mahout.apache.org/v0.7.
[6] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. VLDB Journal, 2005.
[7] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. Data Eng. Bull., 35(2), 2012.
[8] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. The halop approach to large-scale iterative data analysis. VLDB J., 2012.
[9] Z. Cai, Z. Vagenka, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using SimSQL. In SIGMOD, 2013.
[10] L. Cartey, R. Lyngso, and O. de Moor. Synthesising graphics card programs from DSLs. In PLDI, 2012.
[11] M. Chau, R. Cheng, B. Kao, and J. Ng. Uncertain data mining: An example in clustering location data. In PAKDD, 2006.
[12] S. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludäscher, T. McPhillips, S. Bowers, and J. Freire. Provenance in scientific workflow systems. Data Eng. Bull., 32(4), 2007.
[13] X. Feng, A. Kumar, R. Recht, and C. Ré. Towards a unified architecture for in-RDBMS analytics. In SIGMOD, 2012.
[14] R. Fink, L. Han, and D. Oteel. Aggregation in probabilistic databases via knowledge compilation. PVLDB, 5(5), 2012.
[15] B. Glavic and A. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In ICDE, 2009.
[16] T. J. Green, G. Kavounarakis, and V. Tannen. Provenance semirings. In PODS, 2007.
[17] F. Gullo, G. Ponti, and A. Tagarelli. Clustering uncertain data via k-medoids. In SUM, 2008.
[18] F. Gullo, G. Ponti, and A. Tagarelli. Minimizing the variance of cluster mixture models for clustering uncertain objects. In ICDE, 2010.
[19] F. Gullo, G. Ponti, A. Tagarelli, and S. Greco. A hierarchical algorithm for clustering uncertain data via an information-theoretic approach. In ICDM, 2008.
[20] F. Gullo and A. Tagarelli. Uncertain centroid based partitional clustering of uncertain data. PVLDB, 2012.
[21] J. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library or MAD skills, the SQL. PVLDB, 5(12), 2012.
[22] R. Ieda and J. Widom. Panda: A system for provenance and data. Data Eng. Bull., 33(3), 2010.
[23] Z. Ives, T. Green, G. Kavounarakis, N. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira. The Orchestra collaborative data sharing system. SIGMOD Rec., 2008.
[24] R. Jamparsi, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. The Monte Carlo Database System: Stochastic analysis close to the data. ACM TODS, 36(3), 2011.
[25] B. Kao, S. Lee, F. Lee, D. Cheung, and W. Ho. Clustering uncertain data using Voronoi diagrams and R-Tree index. TKDE, 2010.
[26] C. Koch and D. Oteel. Conditioning probabilistic databases. In VLDB, 2008.
[27] H. Kriegel and M. Pfeifle. Density-based clustering of uncertain data. In SIGKDD, 2005.
[28] M. Michel and C. Eastham. Improving the management of MV underground cable circuits using automated on-line cable partial discharge mapping. In CIRED, 2011.
[29] S. Mihaylov, Z. Ives, and S. Guha. REX: Recursive, delta-based data-centric computation. PVLDB, 5(11), 2012.
[30] W. Ngai, B. Kao, C. Chui, R. Cheng, M. Chau, and K. Yip. Efficient clustering of uncertain data. In ICDM, 2006.
[31] S. Oumur and N. Duru. Decreasing iteration number of k-medoids algorithm with IFART. In ELECO, 2011.
[32] J. Provan and M. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. SIAM Journal on Computing, 12(4), 1983.
[33] F. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In ICDE, 2007.
[34] D. Suciu, D. Oteel, C. Ré, and C. Koch. Probabilistic Databases. Morgan & Claypool, 2011.
[35] L. Sun, R. Cheng, D. W. Cheung, and J. Cheng. Mining uncertain data with probabilistic guarantees. In KDD, 2010.
[36] S. van Dongen. Graph clustering by flow simulation. PhD thesis, University of Utrecht, 2000.