open proceedings

ENFrame: A Platform for Processing Probabilistic Data

Sebastiaan J. van Schaik[1,2] Dan Olteanu[1] Robert Fink[1]

{Sebastiaan.van.Schaik,Dan.Olteanu,Robert.Fink}@cs.ox.ac.uk

[1] *Department of Computer Science, University of Oxford, United Kingdom*

[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 [29]. 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 [1]. 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 to close 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 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. 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 [29], after conditioning probabilistic databases using constraints [19], and are supported by virtually all mainstream probabilistic models. If correlations are ignored, the output can be arbitrarily off from the expected result [32, 2]. 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 [11] and semimodules [4] that are used to trace the evaluation of positive relational algebra queries with aggregates and to compute probabilities of query results [10]. It features events with negation, aggregates, loops, and definitions. The language 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. Events are also essential 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

(c) 2014, Copyright is with the authors. Published in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece: ISBN 978-3-89318065-3, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

10.5441/002/edbt.2014.33

```
1:  (O, n) = loadData()         # list and number of objects
2:  (k, iter) = loadParams()    # number of clusters and iterations
3:  M = init()                  # initialise medoids

4:  for it in range(0,iter):    # clustering iterations
5:   InCl = [None] * k          # assignment phase
6:   for i in range(0,k):
7:    InCl[i] = [None] * n
8:    for l in range(0,n):
9:     InCl[i][l] = reduce_and(
10:       [(dist(O[l],M[i]) <= dist(O[l],M[j])) for j in range(0,k)])
11:  InCl = breakTies2(InCl)    # each object is in exactly one cluster

12:  DistSum = [None] * k       # update phase
13:  for i in range(0,k):
14:   DistSum[i] = [None] * n
15:   for l in range(0,n):
16:    DistSum[i][l] = reduce_sum(
17:      [dist(O[l],O[p]) for p in range(0,n) if InCl[i][p]])

18:  Centre = [None] * k
19:  for i in range(0,k):
20:   Centre[i] = [None] * n
21:   for l in range(0,n):
22:    Centre[i][l] = reduce_and(
23:      [DistSum[i][l] <= DistSum[i][p] for p in range(0,n)])
24:  Centre = breakTies1(Centre)  # enforce one Centre per cluster

25:  M = [None] * k
26:  for i in range(0,k):
27:   M[i] = reduce_sum([O[l] for l in range(0,n) if Centre[i][l]])
```

$$\forall i \text{ in } 0..n-1 : O^i \equiv \Phi(o_i) \otimes \mathbf{o}_i$$
$$M^0_{-1} \equiv \Phi(o_{\pi(0)}) \otimes \mathbf{o}_{\pi(0)}; \ldots; M^{k-1}_{-1} \equiv \Phi(o_{\pi(k-1)}) \otimes \mathbf{o}_{\pi(k-1)}$$

$$\forall \text{it in } 0..\text{iter}-1 : \forall i \text{ in } 0..k-1 : \forall l \text{ in } 0..n-1 : \text{InCl}^{i,l}_{\text{it}} \equiv \bigwedge_{j=0}^{k-1} \left[\text{dist}(O^l, M^i_{\text{it}-1}) \leq \text{dist}(O^l, M^j_{\text{it}-1})\right]$$

\# Encoding of `breakTies2` omitted

$$\forall i \text{ in } 0..k-1 : \forall l \text{ in } 0..n-1 : \text{DistSum}^{i,l}_{\text{it}} \equiv \sum_{p=0}^{n-1} \text{InCl}^{i,p}_{\text{it}} \wedge \top \otimes \text{dist}(O^l, O^p)$$

$$\forall i \text{ in } 0..k-1 : \forall l \text{ in } 0..n-1 : \text{Centre}^{i,l}_{\text{it}} \equiv \bigwedge_{p=0}^{n-1} \left[\text{DistSum}^{i,l}_{\text{it}} \leq \text{DistSum}^{i,p}_{\text{it}}\right]$$

\# Encoding of `breakTies2` omitted

$$\forall i \text{ in } 0..k-1 : M^i_{\text{it}} = \sum_{l=0}^{n-1} \text{Centre}^{i,l}_{\text{it}} \wedge O^l$$

Figure 1: K-medoids clustering specified as user program (left) and simplified event program (right).

in general. We developed sequential and distributed algorithms for both exact and approximate probability computation with error guarantees. The algorithms operate on graph representations 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, exhibit deep nesting structures, and use novel expressive constructs. This contrasts with earlier work on probability computation for propositional events, *e.g.* [5, 10], which only considers one event in disjunctive normal form at a time. Clustering events are many, highly interconnected, and have deep nesting structures. Rather than separately computing the probability of each event, 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 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 for concurrent exploration.

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 correlations in the input data, 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 [22]. We show orders-of-magnitude performance improvements of ENFrame's exact algorithm over the naïve approach of clustering in each possible world, of approximate over exact clustering, and of distributed over sequential algorithms. Experiments also confirm that ENFrame's clustering accuracy is identical to that of the naïve approach and that this is not the case for clustering in the top-k most probable worlds.

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.

```
1:  (O, n) = loadData()        # list and number of objects
2:  (k, iter) = loadParams()   # number of clusters and iterations
3:  M = init()                 # initialise centroids

4:  for it in range(0,iter):   # clustering iterations
5:   InCl = [None] * k         # assignment phase
6:   for i in range(0,k):
7:    InCl[i] = [None] * n
8:    for l in range(0,n):
9:     InCl[i][l] = reduce_and(
10:       [dist(O[l],M[i]) <= dist(O[l],M[j]) for j in range(0,k)])
11:  InCl = breakTies2(InCl)   # each object is in exactly one cluster

12:  M = [None] * k            # update phase
13:  for i in range(0,k):
14:   M[i] = scalar_mult(invert(
15:     reduce_count([1 for l in range(0,n) if InCl[i][l]])),
16:     reduce_sum([O[l] for l in range(0,n) if InCl[i][l]]))
```

$\forall i$ in $0..n-1 : O^i \equiv \Phi(o_i) \otimes \mathbf{o}_i$
$M^0_{-1} \equiv \Phi(o_{\pi(0)}) \otimes \mathbf{o}_{\pi(0)}; \ldots ; M^{k-1}_{-1} \equiv \Phi(o_{\pi(k-1)}) \otimes \mathbf{o}_{\pi(k-1)}$

$\forall$it in $0..\text{iter}-1$:
$\quad\forall i$ in $0..k-1$:
$\quad\quad\forall l$ in $0..n-1$:
$\quad\quad\quad \text{InCl}^{i,l}_{\text{it}} \equiv \bigwedge_{j=0}^{k-1} \left[\text{dist}(O^l, M^i_{\text{it}-1}) \leq \text{dist}(O^l, M^j_{\text{it}-1})\right]$

Encoding of `breakTies2` omitted

$\forall i$ in $0..k-1$:
$\quad M^i_{\text{it}} \equiv \left(\sum_{l=0}^{n-1} \text{InCl}^{i,l}_{\text{it}} \wedge \top \otimes 1\right)^{-1} \cdot \left(\sum_{l=0}^{n-1} \text{InCl}^{i,l}_{\text{it}} \wedge O^l\right)$

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

2. ENFRAME'S USER LANGUAGE

This section introduces the user language supported by ENFrame. Its design is grounded in three main desiderata:

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] = True`, or `M[i] = W`. Arrays must be initialised, *e.g.*, for array `M` of cardinality `k`: `M = [None] * k`. Additionally, the expression `range(0, n)` specifies the array `[0,...,n-1]`.

Functions. Scalar variables can be multiplied, exponentiated (`pow(B, r)` for B^r), and inverted (`invert(B)` for $1/B$). The function `dist(A,B)` is a distance measure on the feature space between the arrays `A`,`B` of reals; `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 using one of the functions `reduce_or`, `reduce_and`, `reduce_sum`, `reduce_mult`, `reduce_count`. For instance, for an array `B` of Booleans, the expression `reduce_and(B)` computes the conjunction of the truth values in `B`, and the expression `reduce_count(B)` computes the number of elements in `B`. For a two-dimensional array of reals or integers (an array of vectors), `reduce_sum` computes the component-wise sum of the vectors.

List comprehension. Inside a `reduce`-function, anonymous arrays may be defined using list comprehension. For example, given an array `B` of Booleans of size `n`, the expression `reduce_sum([1 for i in range(0,n) if B[i]])` counts the number of `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: `for i in range(0,n)`. This allows us to know the size of each constructed array at compile time.

Input data. The abstract primitive `loadData()` is used to specify input data for algorithms. This function can be implemented to load the objects 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 [10]. The abstract methods `loadParams()` and `init()` are used to set algorithm parameters such as the number of iterations and clusters of a clustering algorithm.

Output. All program variables have a probabilistic interpretation and thus define a probability density function (PDF) that can be presented to the user. For instance, the user can define a variable stating that two objects belong to the same cluster, or are likely to co-occur together in clusters. PDFs of program variables can be further processed, *e.g.* by a probabilistic DBMS.

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, 2, and 3 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 $o_1, \ldots, o_n$ into k groups of similar data points. We initially choose a centroid M^i for each cluster, *i.e.*, a data point 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. 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 clus-

```
1:  (O, n, M) = loadData()     # M is a stochastic n*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)]))
```

$\forall i$ in $0..n-1 : O^i \equiv \Phi(o_i) \otimes \mathbf{o}_i$

$\forall$it in 0..iter $- 1$:
  $\forall i$ in $0..n-1$:
    $\forall j$ in $0..n-1$:
      $N_{\text{it}}^{i,j} = \sum_{k=0}^{n-1} M_{\text{it-1}}^{i,k} \cdot M_{\text{it-1}}^{k,j}$

  $\forall i$ in $0..n-1$:
    $\forall j$ in $0..n-1$:
      $M_{\text{it}}^{i,j} = \left(\sum_{k=0}^{n-1} (N_{\text{it}}^{i,k})^r\right)^{-1} \cdot (N_{\text{it}}^{i,j})^r$

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

ters. 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][l]` is `True` due to the tie-breaker in line 24.

Markov clustering (MCL). MCL is a fast and scalable unsupervised cluster algorithm for graphs based on simulation of stochastic flow [31]. 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 intra-cluster walks and demoting 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.

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).

```
LOOP ::= { {DECL}{ for ID in RANGE: {LOOP} } }
DECL ::= (ID = EXPR) | (‘(’{ID, } ID ‘)’ = EXT)
EXPR ::= LIT | ID | [None] ‘*’ EXPR) | (EXPR COMP EXPR) |
        (REDUCE ‘(’ LCOMPR ‘)’) | (pow‘(’EXPR, EXPR‘)’) |
        (invert‘(’EXPR‘)’) | (EXPR ‘*’ EXPR) | (EXPR ‘+’ EXPR) |
        (scalar_mult‘(’EXPR, EXPR‘)’) | (breakTies‘(’EXPR‘)’)
LCOMPR ::= [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.

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 (LCOMPR), 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 (LCOMPR) 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.*, 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`. In this context, a tie is a configuration of `InCl` in which for a fixed object `l`, `InCl[i][l]` is `True` for more than one 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`. Similarly, the

function `breakTies1(M)` fixes the first 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

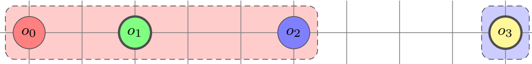
The central concept for representing user programs in ENFrame is that of *events*. Each event is a concise syntactic encoding of a random variable and its probability distribution. This section describes the syntax and semantics of events and event programs, and finally explains how ENFrame programs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

The key features of events and event programs are:

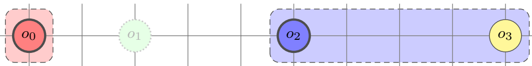
- Events can encode arbitrarily correlated, discrete probability distributions over input objects. In particular, they can succinctly encode instances of such formalisms as Bayesian networks and pc-tables. The input objects and their correlations can be explicitly provided, or imported via a positive relational algebra query with aggregates over pc-tables [10].
- 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.

The events generated by user programs are compositions of events associated with objects (data points) in the input data set: each object o_l is annotated with an event over a set $\mathbf{X}$ of independent Boolean random variables.

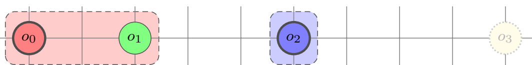
Example 1. Clustering in possible worlds. We start by presenting an instructive example of k-medoids clustering under possible worlds semantics. Let $o_0, \ldots, o_3$ be objects in the feature space as shown below. They can be clustered into two clusters using k-medoids with medoids o_1 and o_3.



The possible valuations $\nu : \mathbf{X} \to \{true, false\}$ define the the *possible worlds* of the input objects: for each valuation ν there exists one world that contains exactly those objects o_l for which $\Phi(o_l)$ is *true* under ν. The probability of a world is the product of the probabilities of the variables $x \in \mathbf{X}$ taking a truth value $\nu(x)$.
Assume that the input data specifies the following events: $\Phi(o_0) = x_1 \vee x_3,\ \Phi(o_1) = x_2,\ \Phi(o_2) = x_3,\ \Phi(o_3) = \neg x_2 \wedge x_4$.
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_3, 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 $\Phi(o_l)$. This example introduces *conditional values* (c-values) which are expressions of the form $\Phi \otimes v$, where Φ is a Boolean formula and v is a vector from the feature space. Intuitively, this c-value takes the value v whenever Φ evaluates to *true*, and a special *undefined* value when Φ is *false*. C-values can be added and multiplied; for example, the expression $\Phi \otimes v + \Psi \otimes w$ evaluates to $v + w$ if Φ and Ψ are *true*, or to v if Φ is *true* and Ψ 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 $M^0 = \Phi(o_0) \otimes o_0 + \neg\Phi(o_0) \otimes o_2$ and $M^1 = \top \otimes 0.5 \cdot (o_1 + o_3)$: centroid M^0 is set to object o_0 if $\Phi(o_0)$ is *true* and to o_2 if $\Phi(o_0)$ is *false*; M^1 is the geometric centre of o_1 and o_3.

In the assignment phase (lines 5–11 of Figure 2), each object is assigned to its nearest centroid. The condition $\mathrm{InCl}^{i,l}$ (o_l being closest to M^i) can be written as the Boolean event

$$\mathrm{InCl}^{i,l} \equiv \bigwedge_{j=0}^{1} \left[\mathrm{dist}(\Phi(o_l) \otimes o_l, M^i) \le \mathrm{dist}(\Phi(o_l) \otimes o_l, M^j)\right],$$

which encodes that the distance from o_l to centroid M^0 is smaller than the distance to centroid M^1.

Given the Boolean events $\mathrm{InCl}^{i,l}$, we can represent the centroid of cluster i for the next iteration by the expression

$$\left(\sum_{l=0}^{3} \mathrm{InCl}^{i,l} \otimes 1\right)^{-1} \cdot \left(\sum_{l=0}^{3} \mathrm{InCl}^{i,l} \otimes o_l\right),$$

which specifies a random variable over possible cluster centroids conditioned on the assignments of objects to clusters as encoded by $\mathrm{InCl}^{i,l}$ (lines 12–16). This expression is exponentially more succinct than an equivalent purely Boolean encoding of centroids, since the latter would require one Boolean expression for each subset of the 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 1–3. In addition to the constructs introduced in Example 2, they use event declarations that assign identifiers to event expressions, and $\forall i$-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:

VAL ::= A scalar or feature vector
INT ::= Any integer
CVAL ::= EVENT⊗VAL | CVAL^{-1} | CVAL+CVAL | $\mathrm{CVAL}^{\mathrm{INT}}$ | CVAL · CVAL | dist(CVAL, CVAL) | EVENT ∧ CVAL
COMP ::= $\le \mid \ge \mid = \mid < \mid >$
ATOM ::= [CVAL COMP CVAL]
EID ::= Elements of a set of event identifiers
EVENT ::= Propositional formula over $\mathbf{X}$, EID, ATOM

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. Like scalars and feature vectors, c-values can be added, multiplied, and exponentiated. Additionally, the distance between c-values yields another c-value, and $\sum$- and $\prod$-expressions are supported.

Event expressions. Event expressions (EVENT) are propositional formulas over constants $\top$ (*true*), $\bot$ (*false*), a set $\mathbf{X}$ 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 $\nu : \mathbf{X} \to \{\mathit{true}, \mathit{false}\}$ to a valuation of c-values and event expressions. We define in the sequel how ν 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 $+$, $\cdot$, $()^{-1}$) by a special element u (for *undefined*) such that $0^{-1} = u$. Operators $+, \cdot$ 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 any real a and feature vector, u and $\mathbf{u}$ are propagated as $u \cdot \mathbf{x} = \mathbf{u}$, $\mathbf{u} + \mathbf{x} = \mathbf{x}$, $a \cdot \mathbf{u} = \mathbf{u}$, and $\mathbf{u} \cdot \mathbf{x} = u$.

The grammar for event programs does not distinguish between scalars and feature vectors for the sake of notational clarity. The following description implicitly assumes that the expressions are well-typed; *e.g.*, the expression $\mathrm{dist}(x, y)$ is only defined for vector-valued variable symbols x, y.

CVAL. Conditional values of the form EVENT$\otimes$VAL have an if-then-else semantics: if EVENT evaluates to *true*, then EVENT$\otimes$VAL evaluates to VAL, else it evaluates to u (or $\mathbf{u}$ 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.

$$\nu(\mathrm{EVENT} \otimes \mathrm{VAL}) = \begin{cases} \mathrm{VAL}, & \text{if } \nu(\mathrm{EVENT}) = \mathit{true} \\ u\ (\mathbf{u}, \text{resp.}) & \text{otherwise} \end{cases}$$

$$\nu(\mathrm{CVAL}_1 + \mathrm{CVAL}_2) = \nu(\mathrm{CVAL}_1) + \nu(\mathrm{CVAL}_2)$$

$$\nu(\mathrm{CVAL}_1 \cdot \mathrm{CVAL}_2) = \nu(\mathrm{CVAL}_1) \cdot \nu(\mathrm{CVAL}_2)$$

$$\nu(\mathrm{CVAL}^{-1}) = \nu(\mathrm{CVAL})^{-1}$$

$$\nu(\mathrm{dist}(\mathrm{CVAL}_1, \mathrm{CVAL}_2)) = \begin{cases} u, \text{ if } \nu(\mathrm{CVAL}_1) = u \text{ or} \\ \quad \nu(\mathrm{CVAL}_2) = u \\ \mathrm{dist}(\nu(\mathrm{CVAL}_1), \nu(\mathrm{CVAL}_2)), \text{ else} \end{cases}$$

$$\nu(\mathrm{CVAL}^{\mathrm{INT}}) = \nu(\mathrm{CVAL})^{\mathrm{INT}}$$

$$\nu(\mathrm{EVENT} \wedge \mathrm{CVAL}) = \begin{cases} \nu(\mathrm{CVAL}), & \text{if } \nu(\mathrm{EVENT}) = \mathit{true} \\ u\ (\mathbf{u}, \text{resp.}) & \text{otherwise} \end{cases}$$

ATOM, EVENT. Comparisons $\nu([\mathrm{CVAL}_1\ \theta\ \mathrm{CVAL}_2])$ for $\theta \in \{\leq, \geq, =, <, >\}$ between two c-values evaluate to *false* if they are both defined ($\nu(\mathrm{CVAL}_1) \neq u$ and $\nu(\mathrm{CVAL}_2) \neq u$) and the comparison does not hold; otherwise (*i.e.*, if at least one of the c-values is undefined, or if the comparison holds), it evaluates to *true*. The semantics of the Boolean propositional EVENT expressions is standard, *i.e.*, by propagating ν through the propositional operators $\wedge, \vee, \neg$. For instance $\nu(\mathrm{EVENT}_1 \wedge \mathrm{EVENT}_2)$ evaluates to *true* if $\nu(\mathrm{EVENT}_1) = \nu(\mathrm{EVENT}_2) = \mathit{true}$, and to *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 $x \in \mathbf{X}$, we denote by $P_x[\mathit{true}]$ and $P_x[\mathit{false}]$ the probability that x is *true* or *false*, respectively; we also simply write P_x for $P_x[\mathit{true}]$. Let $\Omega = \{\nu : \mathbf{X} \to \{\mathit{true}, \mathit{false}\}\}$ be the set of mappings from the random variables $\mathbf{X}$ to *true* and *false*.

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

An event expression E is a random variable over the probability space induced by $\mathbf{X}$ with probability distribution

$$P_E[s] = \Pr\big(\{\nu \in \Omega \mid \nu(E){=}s\}\big) = \sum_{\nu \in \Omega : \nu(E) = s} \Pr(\nu).$$

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:

INT ::= A positive integer
VAR ::= A variable symbol
LOOP ::= { {DECL} { ∀ VAR in INT..INT: {LOOP} } }
DECL ::= EID ≡ 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 $\forall i$-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; here, grounded 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: (1) Translating mutable variables and arrays to immutable events, and (2) translating function calls such as `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 `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 idea of `getLabel` is to first identify the nested loop blocks of the 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.

Special attention must be paid to the encoding of entering and leaving a block: in order to carry over the reference to a variable $M_{c_1.\ldots.c_k}$ to the next block at level $k+1$, we establish a copy $M_{c_1.\ldots.c_k.(-1)} \equiv M_{c_1.\ldots.c_k}$, such that the first access to M in the block may access its last assignment of M via $M_{c_1.\ldots.c_k.(-1)}$. Similarly, the last assignment of a variable in the inner block is passed back to the outer block by copying the last identifier of an inner block to the next identifier of the outer block.

Example 3. Consider the following user language program (left) and its translation to an event program (right).

User program	Event program
`1: M = 7`	A: $M_0 \equiv 7$
`2: M = M+2`	B: $M_1 \equiv M_0 + 2$
`3: for i in range(0,2):`	C: $M_{1.-1} \equiv M_1$
	D: $\forall i$ in 0..1 :
`4:  M = M+i`	E: $M_{1.(2i)} \equiv M_{1.(2i-1)} + i$
`5:  for j in range(0,3):`	F: $M_{1.(2i).-1} \equiv M_{1.(2i)}$
	G: $\forall j$ in 0..2 :
`6:   M = M+1`	H: $M_{1.(2i).j} \equiv M_{1.(2i).(j-1)} + 1$
	I: $M_{1.(2i+1)} \equiv M_{1.(2i).2}$
	J: $M_2 \equiv M_{1.(2\cdot 1+1)}$
`7: M = M+1`	K: $M_3 \equiv M_2 + 1$

The program has three nested blocks. Within each block, the respective counter is incremented for each assignment of M: $M_0, \ldots, M_3$ for the outermost block, $M_{1.0}, \ldots, M_{1.3}$ in the second block, and $M_{1.(2i).0}, \ldots, M_{1.(2i).2}$ for the innermost block. The encodings for entering and leaving a block are in lines C and F, and lines I and J, respectively. □

Translation of arrays. Since arrays in user language programs have a known fixed size, their translation is straightforward: A k-dimensional array $M[n_1], \ldots, [n_k]$ translates to $\prod_i n_i$ distinct identifiers $M^{0,\ldots,0}, \ldots, M^{n_1-1,\ldots,n_k-1}$.

Translation of `reduce_*`. According to the grammar in Figure 4, reduce operations can only be applied to anonymous arrays created by list comprehension. The expression `reduce_and([EXPR for ID in range(FROM, TO) if COND]` is translated to the Boolean event $\bigwedge_{\text{ID}=\text{FROM}}^{\text{TO}-1} \text{COND} \wedge \text{EXPR}$. Symmetrically, `reduce_or` translates to $\bigvee$, `reduce_sum` to $\sum$, and `reduce_mult` to $\prod$. A call to `reduce_count([EXPR for ID in range(FROM, TO) if COND])` translates to the event $\sum_{\text{ID}=\text{FROM}}^{\text{TO}-1} \text{COND} \otimes 1$.

4. PROBABILITY COMPUTATION

The probability computation problem is #P-hard already for simple events representing propositional formulas such as positive bipartite formulas in disjunctive normal form [26]. 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.

Algorithm 1: Exact and approx. compilation of network

```
▷ Blue comments and pseudocode are related to ε-approx.
COMPILE(NETWORK D, ABSOLUTE ERROR ε)
  ▷ Initialise initial (empty) masks for nodes in the network
  foreach v_i ∈ D do M[v_i] ← unknown
  foreach t_i ∈ targets(D) do
    t_i.problower ← 0            ▷ initial probability lower bound: 0
    t_i.probupper ← 1            ▷ initial probability lower bound: 1
    B[t_i] ← 2ε                  ▷ error budget (for exact, ε = 0)

  DFS(D, M, { }, B)              ▷ empty DFS branch ν = { }, Pr(ν) = 1

DFS(NETWORK D, MASKS M, BRANCH ν, ERROR BUDGETS B)
  if ∀t_i ∈ targets(D) : B[t_i] ≥ Pr(ν) then      ▷ sufficient budget
    foreach t_i ∈ T do B'[t_i] ← B[t_i] − Pr(ν); ▷ to trim branch ν
    return B'
  if ν ≠ ∅ then                  ▷ propagate variable mask into DAG
    M[top(ν).var] ← top(ν).val
    M ← MASK(D, M, top(ν).var, NULL, Pr(ν))
  if ∀t_i ∈ targets(D) : (t_i.probupper − t_i.problower ≤ 2ε or
    M[t_i] ≠ unknown) then return B ;      ▷ all reached/approx.
  x ← nextVariable(ν)
  ▷ error budget for left DFS-branch
  foreach t_i ∈ targets(D) do B_left[t_i] ← B[t_i]/2
  ▷ DFS left branch, storing the residual error budget B'_left
  B'_left ← DFS (D, M_left, [ν, x ↦ true], B_left)
  ▷ compute error budget for right DFS-branch B'_left
  foreach t_i ∈ targets(D) do B_right[t_i] ← B[t_i]/2 + B'_left[t_i] if
  ∃t_i ∈ targets(D) : t_i.probupper − t_i.problower > 2ε then
    ▷ right branch (pass error budget, returns residual budget)
    return DFS(D, M_right, [ν, x ↦ false], B_right)
  else
    ▷ all probability bounds reached ε-approx., no right DFS
    return B_right
```

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 c-values.

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 o_i is assigned to cluster C_j in iteration t". We keep lower and upper bounds for the probability of each target. Initially, these bounds are $[0, 1]$ and they eventually converge during computation.

The bulk-compilation procedure is based on Shannon expansion: select an input random variable x and partially evaluate each compilation target Φ to $\Phi|_x$ for x being set to *true* ($\top$) and $\Phi|_{\neg x}$ for x being set to *false* ($\bot$). Then, the probability of Φ is defined by $\Pr[\Phi] = \Pr[x] \cdot \Pr[\Phi|x] + \Pr[\neg x] \cdot \Pr[\Phi|\neg x]$. We are now left with two simpler events $\Phi|_x$ and $\Phi|_{\neg x}$. By repeating this procedure, we eventually resolve all variables in the events to the constants *true* or *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 Φ 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, respectively, a lower and upper bound on the probability of Φ. 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 $\Phi|_x$ and $\Phi|_{\neg x}$

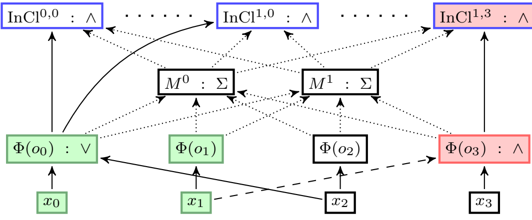


Figure 5: Simplified example of an event network.

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 recording the nodes that become *true* or *false* given the values of their children. When a compilation target t is eventually masked by a variable assignment ν, 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 ν, the compilation backtracks and selects a different assignment for the most recently chosen variable whose assignments are not exhausted. If 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 have propagated to event nodes $\Phi(o_0)$, $\Phi(o_1)$, $\Phi(o_3)$, which are also masked. The red nodes are masked for $\bot$, whereas the green nodes are masked $\top$. □

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) ν; the mask values $M[v_i]$ and probability bounds for all nodes in the event network are initialised. The error budgets $B[t_i]$ 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 with $\nu = \{x \mapsto \top\}$ and one with $\nu = \{x \mapsto \bot\}$. These branches are propagated into the event network using the 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 variable assignment (mask) M is inserted into the network, and the variable node propagates the mask to its parents. Depending on the node, its mask is either updated and propagated, or propagation is stopped in case the node cannot be masked.

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 compilation iteration, and the network requires an additional node c to perform the transition from iteration t to iteration $t+1$. The extra logic required for the MASK function is:

case ∇ ▷ loop node
  $M[t+1][c] \leftarrow M[t][v]$ ▷ carry over mask to next iteration
  $t \leftarrow t+1$ ▷ increase iteration counter

Additionally, probability bounds of compilation targets should only be updated if t is the last iteration, and propagation should only take place if t is *not* the last iteration.

4.3 Approximation with error guarantees

The compilation procedure can be extended to achieve an anytime absolute ε-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 ε-approximation for these events is defined as a tuple $(\hat{p}_0, \ldots, \hat{p}_{n-1})$ such that $\forall 0 \leq i < n : p_i - \varepsilon \leq \hat{p}_i \leq p_i + \varepsilon$. □

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 be easily seen that an absolute ε-approximation can be defined by any tuple $(\hat{p}_0, \ldots, \hat{p}_{n-1})$ such that $\forall 0 \leq i < n$: $U_i - \varepsilon \leq \hat{p}_i \leq L_i + \varepsilon$. 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 2ε 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 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 ε-approximation with error guarantees using the `hybrid` scheme. The DFS procedure is called using a non-zero error budget 2ε, 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.

Other approximation techniques can be added to ENFrame, *e.g.*, the randomised ε-approximation Oracle average, and probability computation using top-k most probable worlds.

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

Algorithm 2: Masking of nodes in an event network

```
Mask(network D, masks M, node v, child c, prob p)
switch v.nodetype do
    case ¬ : M[v] ← ¬M[c];                          ▷ M[c] ∈ 𝔹, M[v] ∈ 𝔹
    case ∧                                           ▷ M[c] ∈ 𝔹, M[v] ∈ 𝔹
        if M[c] = false then M[v] ← false else if
        ∀c_i ∈ v.children : M[c_i] = true then M[v] ← true
    case ∨                                           ▷ M[c] ∈ 𝔹, M[v] ∈ 𝔹
        if M[c] = true then M[v] ← true else if
        ∀c_i ∈ v.children : M[c_i] = false then M[v] ← false
    case ⊗                              ▷ c-value: M[c] ∈ 𝔹, M[v] ∈ ℝ
        ▷ update lower or upper bound of c-value (ℝ)
        if M[c] = true then M[v].lower ← v.val
        M[v].upper ← M[v].lower
    case Σ                     ▷ sum of c-values: M[c] ∈ ℝ, M[v] ∈ ℝ
        ▷ update lower or upper bound of c-values (ℝ)
        M[v].lower ← M[v].lower + M[c].lower
        M[v].upper ← M[v].upper − (c.val − M[c].lower)
    case <                                           ▷ M[c] ∈ ℝ, M[v] ∈ 𝔹
        if v.left.upper < v.right.lower then M[v] ← true else
        v.left.lower ≥ v.right.upper M[v] ← false

if (M[v] ∈ 𝔹 and M[v] ≠ unknown) or M[v] ∈ ℝ then
    if v ∈ targets(D) then          ▷ update prob. bounds of target
        if M[v] = true then v.problower ← v.problower + p else
        v.probupper ← v.probupper − p
    ▷ propagate mask to parents of v
    foreach p_i ∈ v.parents do
        ▷ check whether p_i is already fully masked
        if (M[p_i] ∈ 𝔹 and M[p_i] = unknown) or
          (M[p_i] ∈ ℝ and M[p_i].upper ≠ M[p_i].lower) then
            M ←mask(D, M, p_i, v, p)
return M
```

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 at most $\sum_{i=0}^{\lceil \frac{m}{d} \rceil - 1} 2^{i \cdot d}$, where the cost of each job would propagate at most 2^d variable valuations ν into the event network. Each job incurs the cost of communicating the mask at job creation and the probability bounds for each target at the end of the job. In case of approximation, the error budgets need to be synchronised both at the start and end of a job.

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 [22]. 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 correlations patterns that occur in probabilistic data [3, 28, 29]. 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 approximable. 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 Φ_i be the event that the data point o_i exists. The event Φ_{i+1} becomes $(\Phi_i \wedge x^{\mathrm{t}}_{i+1}) \vee (\neg\Phi_i \wedge x^{\mathrm{f}}_{i+1})$; 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^{t}_{i+1} and x^{f}_{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.

ENFrame algorithms. We report on performance benchmarks for k-medoids clustering on the energy network data set, comparing ENFrame to `naïve` clustering and clustering in the top-k worlds. The `naïve` approach computes a 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.

The literature describes both existing platforms for processing of uncertain data and various algorithms for clustering of uncertain data. Only the newest version of SPROUT supports a form of conditional values, yet only for query evaluation; no other probabilistic DBMS (including MCDB, MayBMS) provides this functionality, which makes it impossible to compare ENFrame to any of the platforms described in Section 6. In addition, work on probabilistic clustering only considers basic uncertainty models based on independence. Those algorithms might outperform our sequential algorithms, at the cost of producing an output that can be arbitrarily off from the golden standard of clustering in every possible world. None of the 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++ and compiled using 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 Euclidean distance).

Sequential algorithms. Figures 6 and 7 show that all of ENFrame's probability computation algorithms outperform the `naïve` algorithm by up to six orders of magnitude for each data set with more than 10 variables. Furthermore, the

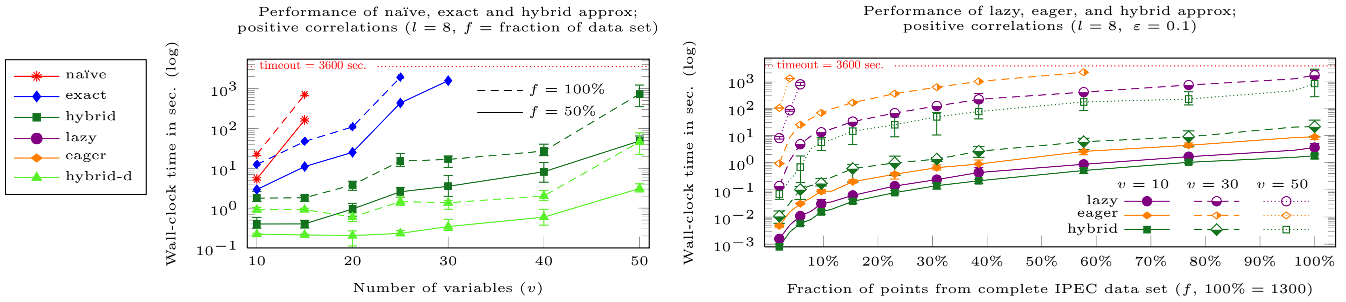


Figure 6: Positively correlated data. On the left: scalability in terms of variables, on the right: scalability of approximations in terms of size of the data set (`hybrid-d` not shown for visibility).

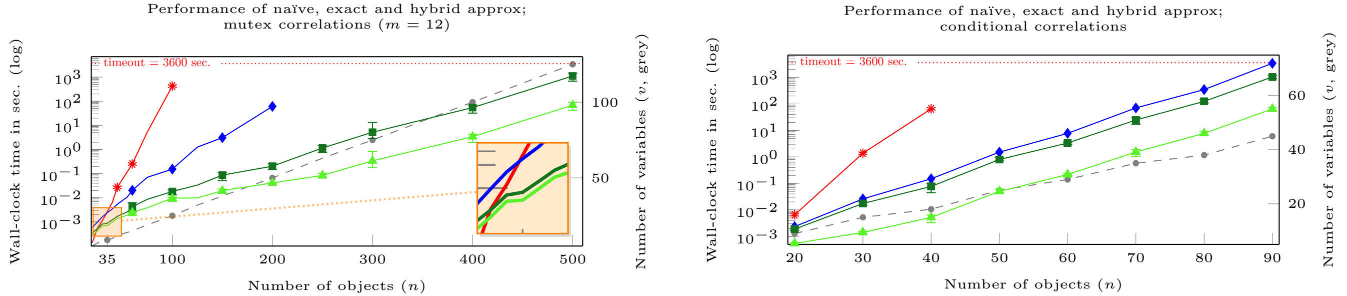


Figure 7: Mutex and conditionally correlated data (legend: see Fig. 6). Algorithms `eager` and `lazy` overlap with `exact`, and are not shown. Grey dashed line indicates number of variables.

`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 are up to six orders of magnitude faster. The `naïve` 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, suggesting that most nodes in the decision tree only contribute a small fraction of the total mass. Depending on the desired error bound, a shallow exploration of the decision tree could be enough to obtain 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, thus leading to longer branches. The `lazy` algorithm saves the error budget until the very last moment and can therefore prune the deep branches whilst maintaining the ε-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-d` using $w = 16$ workers and job size $d = 3$. For all correlation schemes, `hybrid-d` 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-d` becomes more than one order of magnitude faster than `hybrid`. For readability reasons, the performance of `hybrid-d` 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-d` yields a performance improvement of more than one order of magnitude over `hybrid`.

Figure 10 shows the influence of the number of workers on `hybrid-d`'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

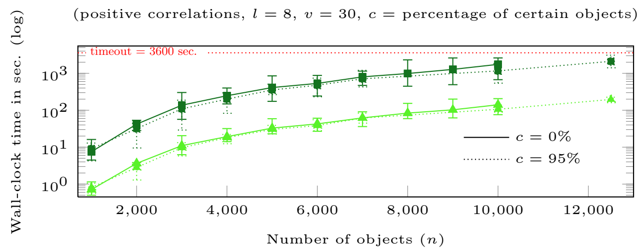


Figure 8: Performance of `hybrid(-d)` on large-scale generated data sets with certain data points.

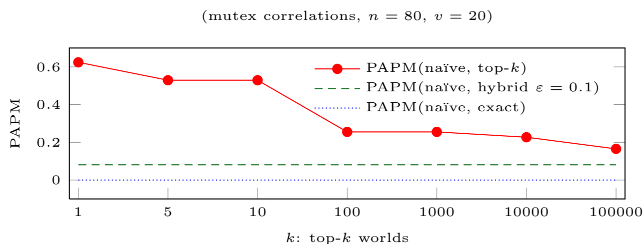


Figure 9: Accuracy experiment: comparing `naïve` (golden standard) to `exact`, `hybrid`, and top-k worlds.

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 performance improves as the number of certain data points (objects that occur in all possible worlds) increases. The speed-up in such cases is explained by the fact that the distance sums of medoids to data points in a cluster become less complex and can be initialised using the distances to objects that certainly exist. Consequently, fewer variables assignments are needed to decide on a cluster medoid, resulting in a shallower decision tree and a speedup in the compilation time.

Clustering accuracy. We investigated the accuracy of ENFrame versus clustering in the top-k most probable worlds and `naïve` clustering. We consider the latter as golden standard, since it respects correlations and does not introduce errors due to approximations of probabilities. To this end, we developed an external evaluator for clustering of uncertain data: the pairwise assignment probability metric (PAPM). Cluster evaluation using pairwise object assignments is a well-described technique for deterministic clusterings, and is used for the Rand measure [27]. PAPM($\mathcal{C}_1, \mathcal{C}_2$) compares two clustering results $\mathcal{C}_1, \mathcal{C}_2$ using the probability that pairs of distinct objects o_a, o_b are assigned to the same cluster (pairwise assignment probability: $\Pr[o_a \sim o_b]$). PAPM reports the maximum difference in this probability between $\mathcal{C}_1$ and $\mathcal{C}_2$ over all pairs of objects:

$$\text{PAPM}(C_1, C_2) = \max_{o_a, o_b} \left| \Pr_{\mathcal{C}_1}[o_a \sim o_b] - \Pr_{\mathcal{C}_2}[o_a \sim o_b] \right|$$

Figure 9 experimentally confirms that the accuracy of ENFrame's `exact` clustering is equal to the golden standard of clustering in all possible worlds (`naïve`). The accuracy of clustering in the top-k most probable worlds if far off from the golden standard, even for large k. Already for $k > 1000$, ENFrame outperforms top-k clustering (not shown in this figure). As expected, the difference between `hybrid` and `naïve` never exceeds 0.1. The data set for this experiment was restricted due to the limited scalability of `naïve`.

A more extensive accuracy comparison is out of scope of this paper, but is part of future work. A common approach to assess the accuracy 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

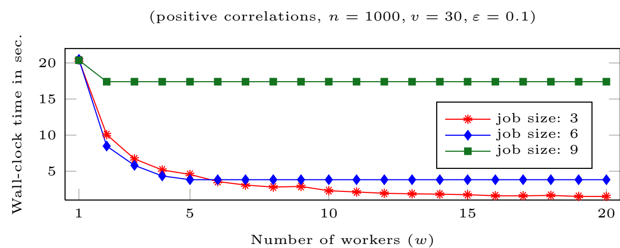


Figure 10: Performance of distributed probability computation as function of number of workers.

contradictory information for which no ground truth exists or is known. In turn, we proposed PAPM, which takes correlations and probabilities into account.

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 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 those representing co-occurrence queries) has a minor influence on performance; due to the combinatorial nature of k-medoids, 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.

6. RELATED WORK

Our work is at the confluence of several research areas: probabilistic data management, data analytics platforms, and provenance data management. A key aspect that differentiates ENFrame from the algorithms and platforms described in this section, is the probabilistic and correlated nature of input data and the entire processing pipeline. This calls for tailored algorithms. As a platform for expressing algorithms to process uncertain data, ENFrame goes beyond any of the single data mining algorithms described here.

Probabilistic data mining and querying. Our work adds to a wealth of literature on this topic [1, 29] along two directions: distributed probability computation techniques and a unified formalisation of several clustering algorithms in line with work on probabilistic databases.

Distributed probability computation has been approached only in the context of the SimSQL/MCDB system, where approximate query results are computed by Monte Carlo simulations [17, 7]. This contrasts with our approach in that MCDB was not designed for exact and approximate computation with error guarantees and does not exploit correlations allowed by pc-tables and ENFrame.

Early approaches to mining uncertain data are based on imprecise (fuzzy) data, for example using intervals, and produce fuzzy (soft) and hard output. Follow-up work shifted to representation of uncertainty by (independent) probability density functions per data point. In contrast, we allow for arbitrarily correlated discrete probability distributions. The importance of correlations has been previously acknowl-

edged for clustering [32] and frequent pattern mining [30]. A further key aspect of our approach that is not shared by existing uncertain data mining approaches is that we follow the possible worlds semantics throughout the whole mining process. This allows for exact and approximate computation with error guarantees and sound semantics of the mining process that is compatible with probabilistic databases. This cannot be achieved by existing work; for instance, most existing k-means clustering approaches for uncertain data define cluster centroids using *expected distances* between data points [8, 25, 12, 20, 14, 18] or the *expected variance* of all data points in the same cluster [13]; they also compute hard clustering where the centroids are deterministic. The recently introduced UCPC approach to k-means clustering [15] is the first work to acknowledge the importance of probabilistic cluster centroids. However, it assumes independence in the input and does not support correlations.

Data analytics platforms. Support for iterative programs is essential in many applications including data mining, web ranking, graph analysis, and model fitting. This has recently led to a surge in data-intensive computing platforms with built-in iteration capability. REX supports iterative distributed computation along database operations in which changes are propagated between iterations [23]. MADlib is an open-source library for in-database analytics [16]. Similarly, Bismarck is an architecture for in-database analytics [9]. GraphLab [21] uses graph representations for scalable parallel programming. The Iterative Map-Reduce-Update programming abstraction for machine learning compiles programs into declarative Datalog code [6]. Infer.NET [24] originates from the programming languages community, but has a closed nature with restricted availability.

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 [29]. Our event language is influenced by work on provenance semirings [11] and semimodules [4, 10] 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 $\mathbb{B}[\mathbf{X}]$ freely generated by the variable set $\mathbf{X}$ 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, $\mathbb{B}[\mathbf{X}] \otimes \mathbb{R}$ is not a semimodule since it violates the following law: $(\Phi_1 \vee \Phi_2) \otimes v = \Phi_1 \otimes v + \Phi_2 \otimes v$. Indeed, under an assignment that maps both Φ_1 and Φ_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.

7. REFERENCES

[1] C. Aggarwal. *Managing and Mining Uncertain Data.* Kluwer, 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. Sugihara, 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] N. Bakibayev, T. Kocisky, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 2013.

[6] 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.*, 2012.

[7] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using SimSQL. In *SIGMOD*, 2013.

[8] M. Chau, R. Cheng, B. Kao, and J. Ng. Uncertain data mining: An example in clustering location data. In *PAKDD*, 2006.

[9] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-RDBMS analytics. In *SIGMOD*, 2012.

[10] R. Fink, L. Han, and D. Olteanu. Aggregation in probabilistic databases via knowledge compilation. *PVLDB*, 5(5), 2012.

[11] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[12] F. Gullo, G. Ponti, and A. Tagarelli. Clustering uncertain data via k-medoids. In *SUM*, 2008.

[13] F. Gullo, G. Ponti, and A. Tagarelli. Minimizing the variance of cluster mixture models for clustering uncertain objects. In *ICDM*, 2010.

[14] F. Gullo, G. Ponti, A. Tagarelli, and S. Greco. A hierarchical algorithm for clustering uncertain data via an information-theoretic approach. In *ICDM*, 2008.

[15] F. Gullo and A. Tagarelli. Uncertain centroid based partitional clustering of uncertain data. *PVLDB*, 2012.

[16] 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.

[17] R. Jampani, 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.

[18] B. Kao, S. Lee, F. Lee, D. Cheung, and W. Ho. Clustering uncertain data using Voronoi diagrams and R-Tree index. *TKDE*, 2010.

[19] C. Koch and D. Olteanu. Conditioning probabilistic databases. In *VLDB*, 2008.

[20] H. Kriegel and M. Pfeifle. Density-based clustering of uncertain data. In *SIGKDD*, 2005.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, July 2010.

[22] M. Michel and C. Eastham. Improving the management of MV underground cable circuits using automated on-line cable partial discharge mapping. In *CIRED*, 2011.

[23] S. Mihaylov, Z. Ives, and S. Guha. REX: Recursive, delta-based data-centric computation. *PVLDB*, 5(11), 2012.

[24] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge.

[25] W. Ngai, B. Kao, C. Chui, R. Cheng, M. Chau, and K. Yip. Efficient clustering of uncertain data. In *ICDM*, 2006.

[26] 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.

[27] W. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of ASA*, 1971.

[28] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.

[29] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases.* Morgan & Claypool, 2011.

[30] L. Sun, R. Cheng, D. W. Cheung, and J. Cheng. Mining uncertain data with probabilistic guarantees. In *KDD*, 2010.

[31] S. van Dongen. *Graph clustering by flow simulation.* PhD thesis, University of Utrecht, 2000.

[32] P. B. Volk, F. Rosenthal, M. Hahmann, D. Habich, and W. Lehner. Clustering uncertain data with possible worlds. In *ICDE*, 2009.