JudgeD: a Probabilistic Datalog with Dependencies

Brend Wanders and **Maurice van Keulen** and **Jan Flokstra**
Department of Electrical Engineering, Math and Computer Science
University of Twente
{b.wanders, m.vankeulen, jan.flokstra}@utwente.nl

Abstract

We present JudgeD, a probabilistic datalog. A JudgeD program defines a distribution over a set of traditional datalog programs by attaching logical sentences to clauses to implicitly specify traditional data programs. Through the logical sentences, JudgeD provides a novel method for the expression of complex dependencies between both rules and facts. JudgeD is implemented as a proof-of-concept in the language Python. The implementation allows connection to external data sources, and features both a Monte Carlo probability approximation as well as an exact solver supported by BDDs. Several directions for future work are discussed and the implementation is released under the MIT license.

1 Introduction

Several probabilistic logics have been developed over the past three decades. Prominent examples include Probabilistic Horn Abduction [Poole, 1993]; PRISM [Sato and Kameya, 2001]; Stochastic Logic Programs [Muggleton, 1996]; Markov Logic Networks [Richardson and Domingos, 2006]; constraint logic programming for probabilistic knowledge, know as CLP($\mathcal{BN}$), [Costa et al., 2002]; probabilistic Datalog, known as pD, [Fuhr, 2000]; and ProbLog [De Raedt, Kimmig, and Toivonen, 2007]. In these logics probabilities can be attached to logical formulas, under the imposition of various constraints. In SLPs clauses defining the same predicate are assumed to be mutually exclusive; PRISM and PHA only allow probabilities on factual data and under constraints that effectively enforce mutual exclusivity.

During the same period, several relational probabilistic databases have been developed. Relational probabilistic database systems that, to a certain degree, have outgrown the laboratory bench include: MayBMS [Koch, 2009; Antova, Koch, and Olteanu, 2009], Trio [Widom, 2004], and MCDB [Jampani et al., 2008] as a prominent example of a Monte Carlo approach. MayBMS and Trio focus on tuple-level uncertainty, that is, probabilities are attached to tuples, and mutually exclusive sets of tuples are defined. MCDB focuses on attribute-level uncertainty where a probabilistic distribution captures the possible values for the attribute. As with the probabilistic logics, certain constraints are imposed. In Trio probabilities are attached to tuples in exclusive sets, that is, a set of mutually exclusive tuples, of which at most one is selected. MCDB supports expressing correlation between attributes through correlated sample functions. MayBMS allows the expression of mutual exclusivity and mutual dependency. In all cases, probabilities can only be attached to factual data.

In this paper we present JudgeD, a probabilistic datalog in which probabilities can be attached to both factual data and rules. Furthermore, complex dependencies can be expressed between clauses. JudgeD has been motivated by our ongoing work on maritime evidence combination, where we want to reason with uncertain facts and rules expressing heuristics. We present a proof-of-concept implementation of both a Monte Carlo based answer probability approximation and an exact solver supported by binary decision diagrams (BDDs).

The key contributions of this paper are:

- The expression of dependencies between arbitrary clauses, both facts and rules (e.g., mutual exclusivity, independence, mutual dependence, implication and more complex dependency relations),
- The proof-of-concept implementation of both a Monte Carlo based approximation as well as an exact solver.

In the next section we will introduce the motivating example for JudgeD. Sections 3 and 4 summarize the formalism on which we base JudgeD and present the syntax, respectively. Section 5 discusses the implementation of the system. Section 6 discusses JudgeD in relation to other work, section 7 presents avenues for future work, and we conclude in section 8.

2 Example: Maritime Evidence Combination

A motivating example for the development of JudgeD is its use as reasoning system for the combination of uncertain evidence about maritime data. The case described in [Habib et al., 2015] has as ultimate goal the automatic determination of the chance that an observed vessel is engaged in smuggling based on a observations about these vessels. A simplified example of such an observation would be the following: `seen("ZANDER", "AMSTERDAM").` The `seen/2`

predicate expresses that a vessel, ZANDER, is seen in a port, AMSTERDAM.

Reasoning about the observations is supported by a knowledge base of vessels and their attributes. The knowledge base consists of factual knowledge about the vessels expressed through `vessel/1`, `vessel_name/2` and `vessel_imo/2` predicates. A sample fishing vessel called ZANDER and identified with IMO number 7712767 (the International Maritime Organization number is a unique identifier for the vessel) is described as:

```
vessel(v0).
vessel_name(v0, "ZANDER").
vessel_imo(v0, 7712767).
vessel_type(v0, stern_trawler).
```

Additional attributes in the knowledge base are described as additional predicates matching the `vessel_???/2` pattern.

The goal is to answer the query `smuggling(V)?` with a set of vessels, each associated with the probability that, given the observations, they are engaged in smuggling.

Uncertain facts Facts, both observations and vessel information in the knowledge base, can be uncertain. An example of an uncertain observation is the interpretation of a verbally reported observation: uncertainty about the observed vessel is easily possible due to a low-quality radio communication. The two interpretations of the spoken report are: `seen("ZANDER", "AMSTERDAM")` and `seen("XANDER", "AMSTERDAM")`. These two observations are mutually exclusive with each other.

Another example would be uncertainty about two observations. For example, if a harbourmaster receives two reports from different sources about a ship sighted of the coast, there can be doubt about whether these are two ships, or if this is one vessel sighted twice. When he receives a radioed report about a sighting of the vessel `XANDER` and at the same time gets a report about the just sighted `ZANDER`, there are three different ways to report the situation:

```
report(r1, "XANDER").
```

He makes a single report stating that the vessel `XANDER` was sighted, assuming that the second sighting was actually the same ship, but with an unclearly pronounced name.

```
report(r1, "ZANDER").
```

He makes a single report stating that the `ZANDER` was sighted, confident that the other report was simply a report for the same ship.

```
report(r1, "XANDER").
report(r2, "ZANDER").
```

Alternatively, the harbourmaster can make two reports. If both names were heard correctly, there are two ships of the coast. In this situation, there is uncertainty about what facts are true.

Uncertain rules Probabilities attached to rules can be interpreted as a form of heuristic. By stating that a rule does not always hold, any answers derived through that rule will take the probability that the rule holds into account. For example, if domain expertise holds that any vessel caught smuggling is likely to be engaged in smuggling again, this can be expressed by the rule: `smuggling(V) :- caught_smuggling(V).` By attaching a probability to this rule, it becomes a heuristic for determining if a vessel is engaged in smuggling.

Dependencies between rules are necessary to express such heuristics with disjunctions in them: if there is a 0.45 chance that ship is smuggling if it is "blue or has an unreadable name" — a purely fictitious heuristic — this is expressed in datalog through two separate rules `smuggling(V) :- vessel_blue(V)` and `smuggling(V) :- vessel_name_unreadable(V)`, and these two rules need to be in or out together.

3 Formal Basis

The semantics of JudgeD programs are similar to those of ProbLog, in that JudgeD programs specify multiple traditional datalog programs. In this section we try to give an intuitive understanding of JudgeD semantics, a more in-depth discussion of the formal underpinnings of JudgeD can be found in [Wanders and van Keulen, 2015].

A JudgeD program J specifies a multitude of traditional datalog programs, albeit in a more compact representation. Let W_J be the set of all traditional datalog programs specified by the JudgeD program. Each partitioning ω^n divides W into n covering disjoint partitions. Each program is labeled with a label $\omega{=}v$, with ω the partitioning, and v the partition into which the program is placed. This way, every program in W has a set of associated labels, with exactly one label from each partitioning. In other words, labels from the same partitioning are mutually exclusive, and exactly one of them is true for any given datalog program. For example, given partitionings x^2 and y^2 we can construct all datalog programs in W by enumerating: $\{x{=}1, y{=}1\}$, $\{x{=}1, y{=}2\}$, $\{x{=}2, y{=}1\}$, and $\{x{=}2, y{=}2\}$.

A JudgeD program consists of a set of clauses. In JudgeD every clause c_i has an attached propositional sentence φ_i called a descriptive sentence. We use the shorthand $\langle c_i, \varphi_i \rangle$ to denote that sentence φ_i is attached to clause c_i. The descriptive sentence uses partitioning labels, of the form $\omega{=}v$, as atoms to A describe the set of traditional datalog programs for which the datalog clause holds: the clause is part of every datalog program for which φ_i evaluates to true given that the labels attached to the datalog program are the only labels that are true. For example, the clause $\langle A, x{=}2 \rangle$ is fully defined as follows:

$$A^h \overset{x=2}{\leftarrow} A_1, A_2, \ldots, A_i$$

This clause has the normal semantics that A^h holds if A_1 through A_i hold, and only in those datalog programs for which the descriptive sentence $x{=}2$ holds. Using the previous example partitionings, this clause is part of two datalog programs specified with the following sets labels: $\{x{=}2, y{=}1\}$ and $\{x{=}2, y{=}2\}$.

Dependencies between clauses The dependencies between clauses can be expressed with descriptive sentences logically combining different labels. Mutual dependency can be expressed by using the same sentence for the clauses.

For example $\langle a, \varphi\rangle$ and $\langle b, \varphi\rangle$ describe the situation where the clauses a and b always hold in the same datalog programs. Implication can be expressed by containment. For example $\langle a, \varphi\rangle$ and $\langle b, \varphi \wedge \psi\rangle$ describes the situation that whenever a is in a datalog program, then b is too. Mutual exclusivity can be expressed through mutually exclusive sentences. For example, $\langle a, \varphi\rangle$ and $\langle b, \psi\rangle$ are mutually exclusive if $\varphi \wedge \psi \equiv \mathit{false}$.

Probability calculation One can attach a probability $P(\omega{=}v)$ to each partition v of a partitioning ω^n provided that $\sum_{v=1}^{n} P(\omega{=}v) = 1$. As is known from the U-relations model [Antova et al., 2008] and variations thereof such as [van Keulen, 2012], calculating probabilities of possible worlds or the existence of an assertion among the worlds, can make use of certain properties that also apply here. For example $P(\omega_1{=}v_1 \wedge \omega_2{=}v_2) = P(\omega_1{=}v_1) \times P(\omega_2{=}v_2)$ and $P(\omega_1{=}v_1 \vee \omega_2{=}v_2) = P(\omega_1{=}v_1) + P(\omega_2{=}v_2)$ iff $\omega_1 \neq \omega_2$.

Given a JudgeD program J and a query q, the naive approach to calculate the probability of the query answer is to enumerate all possible datalog programs $P \in W_J$, and sum the probabilities of each program P for which there is a proof for q. This quickly becomes infeasible for any non-trivial amount of uncertainty.

4 Probabilistic Datalog

The syntax of JudgeD program closely resembles traditional datalog, with the addition of the descriptive sentences. Additionally, the probabilities attached to the labels are included in the syntax. An example of a simple coin-flip would be:

```
heads(c1) [x=1].
tails(c1) [x=2].
@P(x=1) = 0.5. @P(x=2) = 0.5.
```

The first two lines establish simple facts and attach sentences to make them mutually exclusive. The third line contain annotations that attach probabilities to the labels to allow the calculation of answer probabilities. When presented with the query `heads(C)?` the answer `heads(c1)` has a probability of 0.5.

To show how dependencies can be expressed in practice recall the example of uncertain facts on page 2: the example describes two sightings of vessels called XANDER and ZANDER, with doubt about whether they are the same vessel. The harbourmaster has three options: report one vessel named XANDER (`n=1`), report one vessel named ZANDER (`n=2`), or report them both as separate vessels (`s=2`). In JudgeD this can be expressed as follows:

```
report(r1, "XANDER") [ s=1 and n=1 ].
report(r1, "ZANDER") [(s=1 and n=2) or s=2].
report(r2, "XANDER") [s=2].
```

By creating a partitioning s^2 we effectively describe a choice between datalog programs: one where the two reports refer to the same vessel, and another where the two reports refer to different vessel. The choice of selecting the name XANDER or ZANDER, represented by the partitioning n^2, is dependent upon `s=1`, as expressed by the conjunction. Complex dependencies can be expressed by combining the `and`, `or` and `not` operations.

5 Implementation

The proof-of-concept implementation of JudgeD is based on SLG resolution for negative Prolog as described in [Chen, Swift, and Warren, 1995]. The focus of the implementation is not on raw performance, but on ease of prototyping, as such the system is implemented in Python[1] to allow for quick prototyping of new approaches.

The implementation is structured such that a basic implementation of Datalog with negation was created first. The basic implementation also allows the introduction of native predicates, i.e., predicates that are implemented in Python. Native predicates can be used to pull data from external data sources, such as a relational database or a graph database, into the query answering process.

The basic implementation was then used as a basis for two methods of evaluation: a Monte Carlo approximation, and an exact solver.

Monte Carlo Approximation

Monte Carlo approximation for a query q boils down to repeated weighted sampling of a traditional datalog program P_i from all implicitly specified datalog programs W_J in the JudgeD program J, and evaluating q for each sampled P_i. Sample weights are calculated by simple multiplication of the probabilities attached to the labels associated with P.

Instead of determining the weights of each datalog program, a lazy-evaluation scheme is used to construct a set of sampled labels only from those partitionings that are encountered during the search for a proof. This scheme allows the evaluation of q over knowledge bases with enormous amounts of uncertainty, as long as that uncertainty is 'local'. That is, if the uncertainty is expressed as large numbers of partitionings, each with a moderate number of labels.

The implementation features a rudimentary stopping criterion by determining the root mean square error of the samples observed up till now, and if the error moves below a configurable threshold the approximation is terminated.

The Monte Carlo approximation allows the use of the full expressiveness of negative Datalog, with the lazy-evaluation scheme allowing the application to knowledge bases with large amounts of uncertainty. Furthermore, because of the non-intrusive nature of the scheme, it can easily be applied to other types of solvers. A disadvantage of the Monte Carlo solver is that it will not provide the logical sentence that describes for which datalog programs the proof holds. It will only provide the probability of the answer.

Exact Solver

In contrast with the Monte Carlo solver, the Exact solver determines the exact sentence φ_a describing in which datalog programs the proof for the answer a was found. This is done based on the knowledge that for any answer the resolution proof can be restricted to a linear sequence of clauses $c_1, c_2, \ldots, c_i$, with attached sentences $\varphi_1, \varphi_2, \ldots, \varphi_i$. The

[1] `https://www.python.org`

sentence for the answer follows from the needed clauses as:

$$\varphi_a = \bigwedge_{n=1}^{i} \varphi_n$$

If the sentence φ_a is consistent, then answer a can be proven in all datalog programs for which the sentence holds. An inconsistent sentence shows that there are no datalog programs contained within the JudgeD program for which there is a valid proof.

Efficient construction of this sentence is done by constructing partial sentences during SLG resolution, i.e., unification, of two clauses G and C. The partial sentence φ_A for the resolvent A is equal to the conjunction of the sentences associated with G and C: $\varphi_A = \varphi_G \wedge \varphi_C$. If this sentence is inconsistent this means that G and C are not unifiable because there is no datalog program for which this proof will hold.

If a new fact is discovered during the search it is only necessary to expand on it if it is not subsumed by an already discovered fact. While datalog has no functions, and thus no functional subsumption, the introduction of descriptive sentences creates a different kind of subsumption. A new fact $\langle f, \varphi \rangle$ is subsumed by an already known fact $\langle a, \psi \rangle$ if $\varphi \wedge \psi \equiv \psi$. If this is the case, the new fact does not add new knowledge to the already expanded knowledge base, because any proof that leads to the new fact comes from already explored datalog programs.

The efficient detection of sentence subsumption is done through the use of Binary Decision Diagrams [Bryant, 1992]. A binary decision diagram is a graphical representation of a boolean function over a number of variables. Given a complete ordering over the variables a Reduced Ordered Binary Decision Diagram, more commonly known simply as a BDD, provides a canonical representation of the boolean function. A BDD can be constructed by starting with a binary decision tree in which all non-leaf nodes represent variables and all leaf nodes represent either 1 or 0. Non-leaf nodes have a 'high' and a 'low' child. Each path from the root to a leaf represents a full assignment of truth values to each variable, with variables encountered in the order determine by the full ordering. A BDD can be constructed from this tree by merging isomorphic subgraphs and reducing redundant nodes until no further reduction is possible. An example of a BDD for the function $f = (a \wedge b) \vee c$ can be seen in figure 1.

The current exact implementation is restricted to positive datalog, and does not yet calculate probabilities. See section 7 for a discussion on extension to negative datalog, and on two promising directions for probability calculation.

6 Related Work

The semantics of JudgeD can be seen as an extension to the semantics of ProbLog. In ProbLog [De Raedt, Kimmig, and Toivonen, 2007] each clause has an attached probability that they are true. These probabilities are assumed to be independent. We extend this semantic by decoupling the probabilities from the clauses through the descriptive sentences,

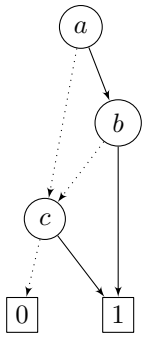

Figure 1: Example of a BDD for the function $f = (a \wedge b) \vee c$. Solid edges are high, dotted edges are low.

allowing the expression of complex dependencies. Furthermore, where the ability to assign probabilities to rules has to be exercised with caution in ProbLog — because, as De Raedt et al. state "the truth of two clauses, or non-ground facts, need not be independent, e.g. when one clause subsumes the other one" — this is not a concern in JudgeD where these clauses can be given multiple labels.

pD [Fuhr, 2000] also assumes independent probabilities, and allows the definition of sets of disjoint events. In this way it is possible to model arbitrary dependencies. This can be done by providing the disjoint probabilities for all possible combinations of dependent events. In practice, the required enumeration of all possible combinations makes this an infeasible solution.

The way labels are used in JudgeD is inspired in part by the random variable assignments of MayBMS [Koch, 2009]. MayBMS constraints the assignment labels to facts (by virtue of attaching them to tuples in a relational database) and requires that only conjunctive combinations of labels are used. MCDB [Jampani et al., 2008] uses a Monte Carlo approach to allow query answering over a sampled database, where they apply their concept of tuple-bundles to speed up the process. JudgeD uses a conceptually similar method through the lazy-evaluation scheme, which answers the query by monotonically constricting the answer to the set of datalog programs in which a proof can be found.

7 Future Work

Because JudgeD is a proof-of-concept implementation, there are several areas for improvement and investigation. Of specific interest is the computation and approximation of probabilities.

JudgeD is a probabilistic datalog derived from the framework described in [Wanders and van Keulen, 2015]. The decoupling between descriptive sentence and datalog clause closely adheres to this framework. However, with ProbLog's proven performance [Kimmig et al., 2008] on top of YAP-Prolog, reducing JudgeD programs to Problog programs is a promising and open topic of investigation.

Exact probability calculation for sentences in DNF is described by [Koch and Olteanu, 2008]. They propose an algorithm and heuristic to break down the DNF sentence into

independent subsentences, which allows computation of the exact probability. Another venue of investigation of probability calculation is ProbLog's approximation. [De Raedt, Kimmig, and Toivonen, 2007] describes both a BDD-based probability calculation for the exact probability and an approximation algorithm that can be applied during the computation of the SLD tree. Since the currently used solver is based on SLG [Chen, Swift, and Warren, 1995], which is a successor of SLDNF, this direction seems to be valuable.

The current implementation of the exact solver does not support negative datalog. Our work in the maritime evidence combination case has impressed the need for negation in real-life applications. Further investigation is needed to apply the formalism described in [Wanders and van Keulen, 2015] to negative datalog to allow a principled implementation of the exact solver for SLG resolution to support negation.

A different direction is the extension of JudgeD to allow for generalized probabilities. Currently, the modelling of two coin flips requires the explicit declaration of a second partitioning. For example, a single coin flip can be modelled by: $\{ \langle coin(c1), true\rangle \langle heads(C) \leftarrow coin(C), x{=}1\rangle, \langle tails(C) \leftarrow coin(C), x{=}2\rangle \}$ with a single partitioning x^2 describes how the coin $c1$ can go either heads or tails. Two coin flips must be made explicit with the addition of a new partitioning y^2. The simple addition of $coin(c2)$ to the previous scenario will result in x^2 representing a 'universal' coin flip: either all coins land on heads, or all coins land on tails. Extending the modelling of probabilities to allow the specification of implicit partitionings, i.e., the specification of "one partitioning per X for all answers of $coin(X)$", together with their probability mass functions may improve the way JudgeD can be applied to certain problems.

JudgeD has the option to use knowledge from external data sources, such as relational databases or graph databases, through native predicates. At the moment of writing, native predicates must be deterministic To leverage the full expressiveness of JudgeD, native predicates have to be extended to allow them to interface with probabilistic relational databases and other probabilistic data sources.

8 Conclusions

We present JudgeD, a proof-of-concept probabilistic datalog implementation. JudgeD can connect to external data sources through native predicates, and supports negative datalog based on SLG resolution. We have presented a Monte Carlo approximation for calculating answer probabilities, and presented an exact solver that works for positive datalog. The key contribution of JudgeD is the ability to express dependencies between arbitrary clauses, including both facts and rules in such dependencies.

There are several venues for future investigation, including improved probability calculation algorithms inspired by MayBMS and ProbLog, the use of external probabilistic data sources, and the addition of generalized probabilities to express independent probabilities associated with repeated or plural events.

JudgeD is released under the MIT license. It can be obtained from: `https://github.com/utdb/judged`

References

[Antova et al., 2008] Antova, L.; Jansen, T.; Koch, C.; and Olteanu, D. 2008. Fast and simple relational processing of uncertain data. In *Proc. of ICDE*, 983–992.

[Antova, Koch, and Olteanu, 2009] Antova, L.; Koch, C.; and Olteanu, D. 2009. $10^{(10^6)}$ worlds and beyond: Efficient representation and processing of incomplete information. *The VLDB Journal* 18(5):1021–1040.

[Bryant, 1992] Bryant, R. E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)* 24(3):293–318.

[Chen, Swift, and Warren, 1995] Chen, W.; Swift, T.; and Warren, D. S. 1995. Efficient top-down computation of queries under the well-founded semantics. *The Journal of logic programming* 24(3):161–199.

[Costa et al., 2002] Costa, V. S.; Page, D.; Qazi, M.; and Cussens, J. 2002. Clp (bn): Constraint logic programming for probabilistic knowledge. In *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*, 517–524. Morgan Kaufmann Publishers Inc.

[De Raedt, Kimmig, and Toivonen, 2007] De Raedt, L.; Kimmig, A.; and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, 2462–2467.

[Fuhr, 2000] Fuhr, N. 2000. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science* 51(2):95–110.

[Habib et al., 2015] Habib, M. B.; Wanders, B.; Flokstra, J.; and van Keulen, M. 2015. Data uncertainty handling using evidence combination: a case study on maritime data reasoning. In *Proceedings of the 5th DEXA Workshop on Information Systems for Situation Awareness and Situation Management (ISSASiM 2015), Valencia, Spain*. USA: IEEE Computer Society.

[Jampani et al., 2008] Jampani, R.; Xu, F.; Wu, M.; Perez, L. L.; Jermaine, C.; and Haas, P. J. 2008. MCDB: a monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 687–700. ACM.

[Kimmig et al., 2008] Kimmig, A.; Costa, V. S.; Rocha, R.; Demoen, B.; and De Raedt, L. 2008. On the efficient execution of problog programs. In *Logic Programming*. Springer. 175–189.

[Koch and Olteanu, 2008] Koch, C., and Olteanu, D. 2008. Conditioning probabilistic databases. *Proceedings of the VLDB Endowment* 1(1):313–325.

[Koch, 2009] Koch, C. 2009. Maybms: A system for managing large uncertain and probabilistic databases. *Managing and Mining Uncertain Data* 149.

[Muggleton, 1996] Muggleton, S. 1996. Stochastic logic programs. *Advances in inductive logic programming* 32:254–264.

[Poole, 1993] Poole, D. 1993. Probabilistic horn abduction and bayesian networks. *Artificial Intelligence* 64(1):81 – 129.

[Richardson and Domingos, 2006] Richardson, M., and Domingos, P. 2006. Markov logic networks. *Machine learning* 62(1-2):107–136.

[Sato and Kameya, 2001] Sato, T., and Kameya, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research* 391–454.

[van Keulen, 2012] van Keulen, M. 2012. Managing uncertainty: The road towards better data interoperability. *J. IT - Information Technology* 54(3):138–146.

[Wanders and van Keulen, 2015] Wanders, B., and van Keulen, M. 2015. Revisiting the formal foundation of probabilistic databases. In *Proceedings of the 2015 Conference of the International Fuzzy Systems Association and the European Society for Fuzzy Logic and Technology*. Atlantis Press.

[Widom, 2004] Widom, J. 2004. Trio: A system for integrated management of data, accuracy, and lineage. Technical Report 2004-40, Stanford InfoLab.