Which Patient to Treat Next? Probabilistic Stream-based Reasoning for Decision Support and Monitoring

Marcel Gehrke, Simon Schiff, Tanya Braun, and Ralf Möller
Institute of Information Systems
University of Lübeck
Lübeck, Germany
{gehrke, schiff, braun, moeller}@ifis.uni-luebeck.de

***Abstract*—Providing decision support for questions such as "Which Patient to Treat Next?" requires a combination of stream-based reasoning and probabilistic reasoning. The former arises due to a multitude of sensors constantly collecting data (data streams). The latter stems from the underlying decision making problem based on a probabilistic model of the scenario at hand. The STARQL engine handles temporal data streams efficiently and the lifted dynamic junction tree algorithm handles temporal probabilistic relational data efficiently. In this paper, we leverage the two approaches and propose probabilistic stream-based reasoning. Additionally, we demonstrate that our proposed solution runs in linear time w.r.t. the maximum number of time steps to allow for real-time decision support and monitoring.**

I. Introduction

Which patient to treat next is not an easy question, not only on a personal level but also on a computer science level. Answering the question (again and again) falls into the area of online decision making on probabilistic temporal models where actions influence obtainable utilities and one is interested in those actions that maximise overall utilities (maximum expected utility problem, MEU for short). While there already exist algorithms to solve a decision making problem, two kinds of obstacles arise. *One*, the real world presents challenges when trying to implement existing algorithms. Whether at home or in a ward, several sensors gather data continuously for each patient, yielding multiple streams of data that need to be fed into an underlying model as evidence. Further, different sensors can have different sampling rates and store information in different ways, e.g., under varying attribute names in different tables. *Two*, such algorithms are not designed to allow for monitoring of query results and issuing alerts when certain conditions are met. Therefore, in this paper, we study the problem of stream-based and probabilistic reasoning on temporal uncertain data streams.

To the best of our knowledge, so far there only exists theoretical work on answering various types of queries on temporal uncertain data streams [1]. An existing algorithm for exact reasoning in probabilistic temporal models is lifted dynamic junction tree algorithm (LDJT) [2], [3] for answering a set of queries for probability distributions of random variables (randvars). LDJT builds a helper structure to efficiently handle temporal aspects as well as multiple queries. A generalisation of LDJT to also answer MEU queries is meuLDJT [4], [5]. LDJT and meuLDJT also incorporate the lifting idea, in which an algorithm treats a pool of individuals as indistinguishable during calculations as long as no contrary evidence arises. But, they cannot handle streams of data and the problems the streams incur. Additionally, monitoring of streams of data or query results over time is also not part of the algorithm, which falls under stream-based reasoning. The streaming and temporal ontology access with a reasoning-based query language (STARQL) [6] is currently a state of the art for reasoning with temporal data streams from multiple sources, meaning, it can handle streams of data with varying sampling rate or different storing format. Varying attribute names are handled using the ontology-based data access (OBDA) paradigm [7], which maps the names of the different sources to a unified vocabulary. Additionally, STARQL allows for monitoring streams: If given a window size and an update frequency, STARQL is able to calculate queries on sliding windows, e.g., if values of a stream are rising or if a value exceeds a given threshold. However, STARQL currently does not support probabilistic reasoning including decision making.

The contribution of this paper is probabilistic stream-based reasoning (PSR), combining meuLDJT and STARQL. PSR is able to run in linear time w.r.t. the maximum number of time steps to allow for real-time decision support and monitoring. PSR is able to (i) calculate an MEU using current data from the incoming data streams, providing the key functionality of decision making, (ii) calculate additional queries for marginal distributions or probabilities, yielding query streams (a stream of query results for each query), (iii) calculate additional queries on data streams and query streams, enabling a monitoring of data and query results. The first two items require meuLDJT functionality, the third item requires STARQL functionality. The OBDA approach of STARQL automatically enables PSR to map the vocabulary of the data streams to the vocabulary of the underlying model. Additionally, meuLDJT requires a discretisation of continuous

This research originated from the Big Data project as part of Joint Lab 1, funded by Cisco Germany, at the centre COPICOH, University of Lübeck.

values in streams as well as an alignment of sampling rates, which STARQL also provides.

As a combination of meuLDJT and STARQL, PSR is able to provide decision support regarding the query "Which patient to treat next?". An empirical evaluation of PSR highlights how changes in parameters that influence the runtime of either meuLDJT or STARQL affect PSR as a whole. Of course, answering the question above requires reasoning over the "utility", i.e., health, of patients while only a limited amount of actions, i.e., the time and resources of medical personell, is available. Thus, setting up an appropriate model is not an easy task as utilities have to reflect that it is not acceptable to let one patient die in exchange for others to live.

In the following, we first look at related work. Afterwards, we recapitulate parameterised probabilistic dynamic models (PDMs), which is the underlying representation of meuLDJT, the algorithm itself, and STARQL. Then, we present PSR as a combination of meuLDJT and STARQL. Lastly, we evaluate the performance of PSR w.r.t. runtimes and conclude.

II. Related Work

First, we take a look at temporal data stream reasoning, with a focus on previous STARQL versions. Then, we consider inference in temporal probabilistic relational models. Last, we look into reasoning with probabilistic streams.

The STARQL engine complements the collection of state-of-the-art RDF stream processing engines, among them the engines for the languages C-SPARQL [8], CQELS [9], SPARQLStream [10], EP-SPARQL [11], TEF-SPARQL [12], and StreamQR [13]. An overview of all features supported by STARQL in comparison to other RDF stream engines can be found in [14]. STARQL is the only engine that does not require an RDF stream as input and allows for reasoning, though. STARQL uses OBDA [7], which has become of interest also for the industry [14], mainly due to recent research efforts of extending OBDA for handling temporal data [15], [16] and stream data [6], [9], [10], [14], [17] as well as efforts of addressing the needs for enabling statistical analytics [18].

Most approaches to inference in temporal relational models are approximate. Additionally, these approaches involve unnecessary groundings or are not designed to handle multiple queries efficiently. Ahmadi et al. [19] propose lifted belief propagation for dynamic Markov logic networks (DMLNs). Thon et al. [20] introduce CPT-L, a probabilistic model for sequences of relational state descriptions with a partially lifted inference algorithm. Geier and Biundo [21] present an online interface algorithm for DMLNs, similar to the work of Papai et al. [22]. Both approaches slice DMLNs to run well-studied MLN inference algorithms [23] on each slice. Two ways of performing online inference using particle filtering are described in [24], [25]. Vlasselaer et al. [26], [27] introduce an exact approach for temporal relational models, but perform inference on a ground knowledge base.

P. Koopmann [1] and S. Borgwardt, I. I. Ceylan, and T. Lukasiewicz [28] present first theoretical results for reasoning with probabilistic streams. However, they do not present an implementation of their work. Thus, stream-based and probabilistic reasoning on temporal uncertain data streams still requires work that accounts for practical considerations.

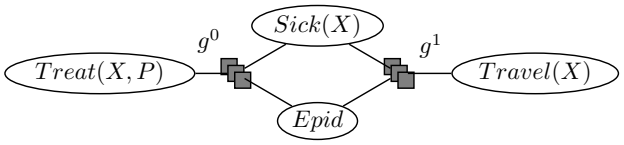

Fig. 1: Parfactor graph of G^{ex}

III. Parameterised Probabilistic Models

We shortly present parameterised probabilistic models (PMs) for relational static models, based on [29], and extend PMs to the temporal case, resulting in PDMs, based on [2].

A. Parameterised Probabilistic Models

PMs combine first-order logic with probabilistic models, representing first-order constructs using logical variables (logvars) as parameters. For illustrative purposes, we use an example of an epidemic. In the example, we model epidemic as a randvar. Further, we model persons being sick as a parameterised randvar (PRV) of a randvar for sick combined with a logvar for persons. In case, we are interested in whether there currently is an epidemic, it only matters whether a person is sick or not. Thus, all persons behave the same in our example and are, without additional evidence, indistinguishable.

Definition 1. Let $\mathbf{R}$ be a set of randvar names, $\mathbf{L}$ a set of logvar names, Φ a set of factor names, and $\mathbf{D}$ a set of constants. All sets are finite. Each logvar L has a domain $\mathcal{D}(L) \subseteq \mathbf{D}$. A *constraint* is a tuple $(\mathcal{X}, C_{\mathbf{X}})$ of a sequence of logvars $\mathcal{X} = (X^1, \ldots, X^n)$ and a set $C_{\mathcal{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X^i)$. The symbol $\top$ for C marks that no restrictions apply, i.e., $C_{\mathcal{X}} = \times_{i=1}^{n} \mathcal{D}(X^i)$. A *PRV* $R(L^1, \ldots, L^n), n \geq 0$ is a construct of a randvar $R \in \mathbf{R}$ possibly combined with logvars $L^1, \ldots, L^n \in \mathbf{L}$. If $n = 0$, the PRV is parameterless and forms a propositional randvar. The term $\mathcal{R}(A)$ denotes the possible values (range) of a PRV A. An *event* $A = a$ denotes the occurrence of PRV A with range value $a \in \mathcal{R}(A)$. We denote a *parametric factor (parfactor)* g by $\phi(\mathcal{A})_{|C}$ with $\mathcal{A} = (A^1, \ldots, A^n)$ a sequence of PRVs, $\phi : \times_{i=1}^{n} \mathcal{R}(A^i) \mapsto \mathbb{R}^+$ a function with name $\phi \in \Phi$, and C a constraint on $lv(\mathcal{A})$. A PRV A or logvar L under constraint C is given by $A_{|C}$ or $L_{|C}$, respectively. We may omit $|\top$ in $A_{|\top}$, $L_{|\top}$, or $\phi(\mathcal{A})_{|\top}$. A *PM* G is a set of parfactors $\{g^i\}_{i=1}^{n}$.

The term $lv(P)$ refers to the logvars in P, which may be a PRV, a constraint, a parfactor, or a model. The term $gr(P)$ denotes the set of all instances of P w.r.t. given constraints. An instance is an instantiation (grounding) of P, substituting the logvars in P with a set of constants from given constraints. If P is a constraint, $gr(P)$ refers to the second component $C_{\mathbf{X}}$. Given a parfactor $\phi(\mathcal{A})_{|C}$, ϕ is identical for $gr(\mathcal{A}_{|C})$.

Given $\mathbf{R} = \{Sick, Epid, Travel, Treat\}$ and $\mathbf{L} = \{X, P\}$, $\mathcal{D}(X) = \{x_1, x_2, x_3\}$ and $\mathcal{D}(P) = \{p_1, p_2\}$, we can build

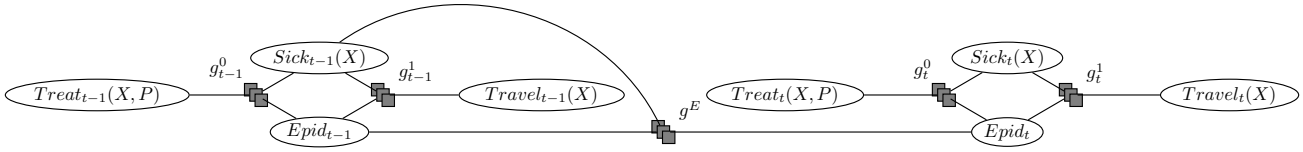

Fig. 2: Parfactor graph of the PDM based on G^{ex}

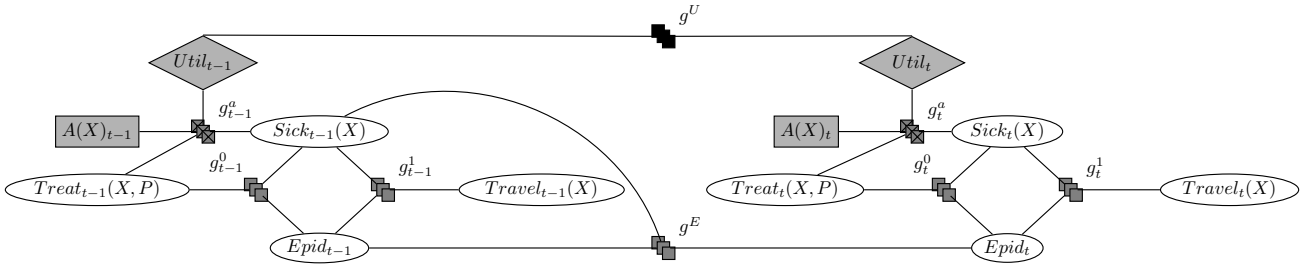

Fig. 3: Parfactor graph of the PDDecM based on G^{ex}

a boolean PRV $Sick(X)$. With $C = ((X), \{(x_1), (x_2)\})$, $gr(Sick(X)_{|C}) = \{Sick(x_1), Sick(x_2)\}$. The set of $gr(Sick(X)_{|\top})$ also contains $Sick(x_3)$. Adding boolean PRVs $Epid$, $Travel(X)$, and $Treat(X, P)$, we build a PM G_{ex}=$\{g^i\}_{i=0}^1$, with g^0=$\phi^0(Epid, Sick(X), Treat(X, P))_{|\top}$ and g^1=$\phi^1(Epid, Sick(X), Travel(X))_{|\top}$. Parfactors g^0 and g^1 have eight input-output pairs (omitted). Constraints are $\top$. Figure 1 depicts G^{ex} as a parfactor graph.

The semantics of a model is given by grounding and building a full joint distribution. In general, a query asks for a probability distribution of a randvar using a model's full joint distribution and given fixed events as evidence.

Definition 2. With Z as normalising constant, a model G represents the full joint distribution $P_G = \frac{1}{Z}\prod_{f \in gr(G)} f$. The term $P(Q|\mathbf{E})$ denotes a *query* w.r.t. G with Q a grounded PRV and $\mathbf{E}$ a set of events. Answering $P(Q|\mathbf{E})$ requires eliminating all randvars in G not occurring in $P(Q|\mathbf{E})$.

B. Parameterised Probabilistic Dynamic Models

We define PDMs based on the first-order Markov assumption. Further, the underlying process is stationary.

Definition 3. A PDM is a pair of PMs $(G_0, G_\rightarrow)$ where G_0 is a PM representing the first time step and $G_\rightarrow$ is a two-slice temporal parameterised model representing $\mathbf{A}_{t-1}$ and $\mathbf{A}_t$ where $\mathbf{A}_\pi$ is a set of PRVs from time slice π.

Figure 2 shows $G^{ex}_\rightarrow$ consisting of G^{ex} for time step $t-1$ and t with *inter*-slice parfactors for the behaviour over time. In this example, the parfactors g^A and g^U are the *inter*-slice parfactors, modelling the temporal behaviour.

Definition 4. Given a PDM G, a query term Q, and events $\mathbf{E}_{0:t} = \{E^i_t = e^i_t\}_{i,t}$, $P(Q_t|\mathbf{E}_{0:t})$ denotes a *query* w.r.t. G.

The problem of answering a marginal distribution query $P(A^i_\pi|\mathbf{E}_{0:t})$ w.r.t. the model is called *prediction* for $\pi > t$, *filtering* for $\pi = t$, and *smoothing* for $\pi < t$.

C. Parameterised Probabilistic Dynamic Decision Models

Let us extend PDMs with action and utility nodes, which we represent using PRVs, resulting in PDDecMs.

Definition 5. Let Φ^u be a set of utility factor names. The range of action PRVs is disjoint actions and the range of utility PRVs is $\mathbb{R}$. A parfactor with a utility PRV U is a *utility parfactor* u. We denote u with $\mu(\mathcal{A})_{|C}$, where $U \in \mathcal{A}$ and C a constraint on $lv(u)$. Function $\mu : \times_{A^i \in (\mathcal{A} \setminus \{U\})} \mathcal{R}(A^i) \mapsto \mathbb{R}$, $\mu \in \Phi^u$, is identical for $gr(\mathcal{A}_{|C})$. Its output is the additive change of U's value. The default initial utility value is 0. A *parameterised probabilistic decision model (PDecM)* G is a PM that also contains utility parfactors. Let G^u refer to the utility parfactors in G and $rv(G^u)$ refer to all probability PRVs in G^u. G^u represents the combination of all utilities $U_G = \sum_{f \in gr(G^u)} f$.

The μ functions output a utility, i.e., a scalar, which makes comparing utility values easy. After the evaluation of a μ function, the initial value $U = i$ is changed by the output value, j, resulting in the new value $U = i + j$. Hence, we can easily test how discriminable actions are.

Definition 6. A utility transfer function λ has utility PRVs $\mathbf{U}$ as input and one utility PRV U_o as output. Additionally, λ can have non utility PRVs as input. λ specifies how the value of U_o is additively changed when transferring from time step t to $t+1$. A *PDDecM* is a pair of PDecMs $(G_0, G_\rightarrow)$ with $G^u_\rightarrow$ also possibly containing utility transfer functions. Given a PDDecM G, a *temporal MEU query* asks for the action sequence (range values for each action PRV in G over time) that maximises the overall expected utility in G.

Figure 3 shows our example with one action (square) and one utility (diamond) PRV in grey, utility parfactors (crosses), and a utility transfer parfactor g^U in black. Further, assume that actions are: A^1 is *treat patient* and A^2 is *do nothing*.

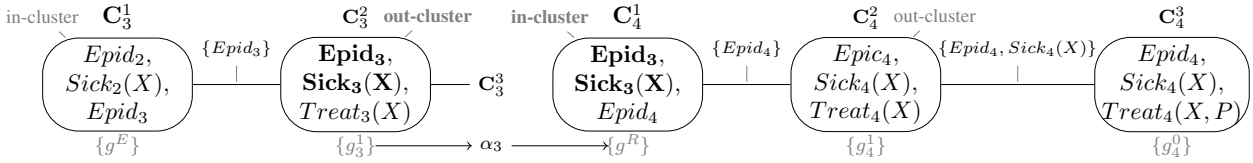

Fig. 4: FO jtree J_3 without $\mathbf{C}_3^3$ and FO jtree J_4 connected with α_3

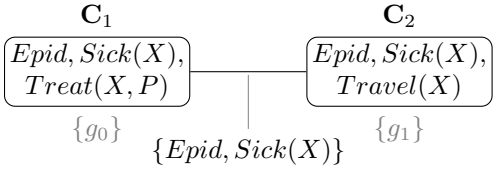

Fig. 5: An FO jtree for G_{ex}

IV. Lifted Inference Algorithm

We recapitulate the lifted junction tree algorithm (LJT) [29] to answer queries for PMs, LDJT [2], [3] to answer *hindsight*, *filtering*, and *prediction* queries for PDMs, and meuLDJT to answer temporal MEU queries [5]. LJT works as a subroutine of LDJT and meuLDJT is an extension of LDJT that is able to handle utilities and actions.

A. Lifted Junction Tree Algorithm

LJT provides efficient means to answer queries $P(Q^i|\mathbf{E})$, with $Q^i \in \mathbf{Q}$ a set of query terms, a PM G, and evidence $\mathbf{E}$ as input. LJT builds a helper structure called an FO jtree (cf. [30] for details). An FO jtree is a directed, acyclic graph whose nodes consist of PRVs, called parclusters. Its most important feature is that if a PRV appears into two nodes, it appears in every node on the path between the two nodes. The set of shared PRVs between two neighbouring nodes is called separator. Each node has parfactors associated whose arguments appear in the node (local model). Figure 5 shows an FO jtree of G^{ex} with the local models of the parclusters and the separators as labels of edges.

LJT performs the following steps: (i) Construct an FO jtree J for G. (ii) Enter $\mathbf{E}$ in J. (iii) Pass messages. (iv) Compute answer for each query $Q^i \in \mathbf{Q}$. Entering evidence adds the events in $\mathbf{E}$ to those nodes that contain the event PRV. LJT sends messages in two passes, from the periphery to the center and back (inbound/outbound). To compute a message, LJT eliminates all non-separator PRVs from the parcluster's local model and received messages. After message passing, the nodes are independent of each other and LJT answers queries based on the nodes. For each query, LJT finds a parcluster containing the query term and sums out all non-query terms in its local model and received messages.

In Fig. 5, LJT sends a message from $\mathbf{C}^1$ to $\mathbf{C}^2$ and back. If we want to know whether $Epid$ holds, we query for $P(Epid)$ for which LJT can use parcluster $\mathbf{C}^1$. LJT sums out $Treat(X,P)$ and $Sick(X)$ from $\mathbf{C}^1$'s local model G^1, $\{g^0\}$, combined with the received messages, one message from $\mathbf{C}^2$.

B. Lifted Dynamic Junction Tree Algorithm

LDJT efficiently answers queries $P(Q_\pi^i|\mathbf{E}_{0:t})$, with $Q_\pi^i \in \mathbf{Q_t}$ and $\mathbf{Q_t} \in \{\mathbf{Q}_t\}_{t=0}^T$, given a PDM G and evidence $\{\mathbf{E}_t\}_{t=0}^T$. LDJT uses the fact that only some PRV influence the next time step (interface PRVs). In our example PDM, $Epid_t$ and $Sick_t(X)$ influence the next time step. Using the interface PRVs, LDJT constructs FO jtree structures that are time independent by having the interface PRVs appear in one parcluster for $t-1$ and one for t (*in-* and *out-clusters*). Thus, LDJT can always only reason over one time step at once. Further, LDJT only needs to store the state of the interface PRVs while proceeding in time.

LDJT performs the following steps: (i) Construct two FO jtrees J_0 and J_t with *in-* and *out-clusters* from G. (ii) For $t=0$, use J_0, enter $\mathbf{E}_0$, pass messages, answer each query term $Q_\pi^i \in \mathbf{Q}_0$, and preserve the state in message α_0. (iii) For $t>0$, instantiate J_t for the current time step t, add α_{t-1} to the incluster, enter $\mathbf{E}_t$ in J_t, pass messages, answer each query term $Q_\pi^i \in \mathbf{Q}_t$, and preserve the state in message α_t.

Figure 4 depicts the passing on of the current state from time step three to four. As mentioned, only the PRVs $Epid_3$ and $Sick_t(X)$ are interface PRVs. To capture the state at $t=3$, LDJT sums out the non-interface PRV $Treat_3(Y)$ from $\mathbf{C}_3^2$'s local model and the received messages and saves the result in message α_3. After increasing t by one, LDJT adds α_3 to $\mathbf{C}_4^2$. α_3 is then distributed by message passing and accounted for during calculating α_4. Thus, LDJT can efficiently handle the temporal aspects while having the benefit of LJT including a rather small FO jtree to perform inference on.

C. meuLDJT for Decision Support

LDJT efficiently answers multiple marginal queries for temporal probabilistic relational models. However, solely with marginal queries, LDJT cannot support decision making. Here, a temporal MEU query comes into play.

To answer a temporal MEU query, Gehrke, Braun, and Möller introduce meuLDJT [31]. The basic idea of meuLDJT is to calculate a belief state and combine the belief state with corresponding utilities. Thus, meuLDJT calculates an action sequence for groups behaving the same that maximises the expected utility for a finite horizon. Finite horizon in this case means that we only plan a given number of time steps into the future. Further, in addition to solving the temporal MEU problem, meuLDJT still can efficiently answer multiple queries efficiently by using the FO jtree structures. To obtain an exact solution for a temporal MEU query, the computation is exponential in the number of time steps we plan ahead.

For the scenario in Fig. 3, assume that we are interested in the best action for the current time step. Then, meuLDJT has to check each possible action for each indistinguishable group of the logvar X. In case X is not split, meuLDJT can set $A(X)$ to A^1 and compute an expected utility and afterwards, set $A(X)$ to A^2 and again compute an expected utility to determine which action maximises the utility. In case we are also interested in the next time step, then meuLDJT needs to check 4 actions as for both actions in the current time step there again are 2 possible actions in the next time step. With evidence that splits X into more groups, meuLDJT also needs to check each possible combination of action assignments for the different groups, making the MEU problem a hard problem to solve. Nonetheless, with the lifting idea the problem already becomes manageable.

Let us now have a look at STARQL, which efficiently deals with temporal data streams.

V. STARQL

Based on [32], we recapitulate STARQL. STARQL is a stream-temporal query framework that was implemented as a submodule of the OPTIQUE software platform [14], [18], [33] and in stand-alone prototypes described in [34], [35]. It extends OBDA [7] to temporal and streaming data.

The main idea of OBDA query answering is to represent the knowledge of the domain of interest in a declarative knowledge, aka ontology, and access the data via a high-level query that refers to the ontology's vocabulary, aka signature. The non-terminological part of the ontology, called the abox, is a virtual view of the data produced by mapping rules. Formulated in a description logic, the abox can have many different first-order logic models that represent the possible worlds for the domain of interest. These can be constrained to intended ones by the so-called tbox, which contains the terminological part of the ontology. In an OBDA system, different components have to be set up, fined-tuned, and co-ordinated in order to enable robust and scalable query answering: (i) a query-engine which allows formulating ontology-level queries; (ii) a reformulation engine, which rewrites ontology-level queries into queries covering the entailments of the tbox; (iii) an unfolding mechanism that unfolds the queries into queries for the backend data sources, and, (iv) the backend sources which contain the data.

As one cannot reason over all data points in a stream, which is normally assumed to be infinite, STARQL provides a window operator with a range parameter $range$ (width of window) and a slide parameter $slide$ (update frequency). The range value defines a time interval STARQL reasons over at once and the slide value defines how fast the window proceeds over the data points. Together, the range and slide parameter allow for constructing a sliding window over a sequence of temporal data points. In case $range \geq slide$, the windows are overlapping. Otherwise, the windows do not overlap. For example, with $range = 5s$ and $slide = 1s$, STARQL starts with a window from $[-5s, 0s]$, then slides the window to $[-4s, 1s]$, and continues in this fashion.

VI. Probabilistic Stream-based Reasoning

STARQL can efficiently reason over temporal streams from multiple sources. However, STARQL cannot handle uncertainties and perform probabilistic inference. meuLDJT can efficiently answer multiple queries for temporal probabilistic relational models and support decision making. However, meuLDJT currently cannot handle data streams with different sampling rates, not unified data streams, or data streams that are not discretised. Therefore, we combine meuLDJT and STARQL, presenting in PSR. PSR leverages the benefits of both STARQL and meuLDJT.

PSR consists of a STARQL and a meuLDJT component. Figure 6 gives an overview about the data flow and tasks within PSR. The figure shows the data sources on the left, the STARQL component in the middle, and the meuLDJT component on the right. The main steps that PSR performs for each time step t are (i) input processing, (ii) input preparation, (iii) query answering, and (iv) monitoring.

PSR requires additional inputs next to data from sources, which we present first before diving into the steps themselves. The following inputs need to be defined:

- a PDDecM G as specified in Definition 6,
- queries $Q_\pi^1, \ldots, Q_\pi^m$ for G to evaluate at each time step,
- an ontology that unifies the vocabulary of the different sources of data streams into the vocabulary of G,
- thresholds for discretisation of continuous ranges,
- functions for combining multiple data points into one,
- window parameters $range$ and $slide$,
- monitoring queries for streams, and
- alert queries for streams.

The first input is passed to meuLDJT to build FO jtrees. The second input, queries for G, goes to STARQL such that STARQL can assemble evidence and queries as inputs for meuLDJT. The queries $Q_\pi^1, \ldots, Q_\pi^m$ are evaluated each time step, yielding streams of query results over time. With the ontology, STARQL is able to present a unified view of the data and meuLDJT is able to process the incoming data as evidence. The next two inputs provide STARQL with information about how to prepare incoming data from the streams for meuLDJT to be able to process the data as evidence. The last two inputs are queries that STARQL evaluates on the data streams as well as query streams given $range$ and $slide$. The results of monitoring queries are simply displayed. Alert queries lead STARQL to issue an alert to the user, which means the user can issue some more queries online. Next, we go through the steps and consider the specific tasks to perform.

During *input processing*, STARQL performs the following tasks for each t:

(i) Gather data v_i from different sources/devices dev_i.
(ii) Access v_i using OBDA.

Data v_i can be a single data point or a collection of points for t, depending on the sampling rate of dev_i. With OBDA, STARQL can use a terminology defined in an ontology independent of the actual terminology of the devices. To be more precise, the terminology in an ontology needs to map to the

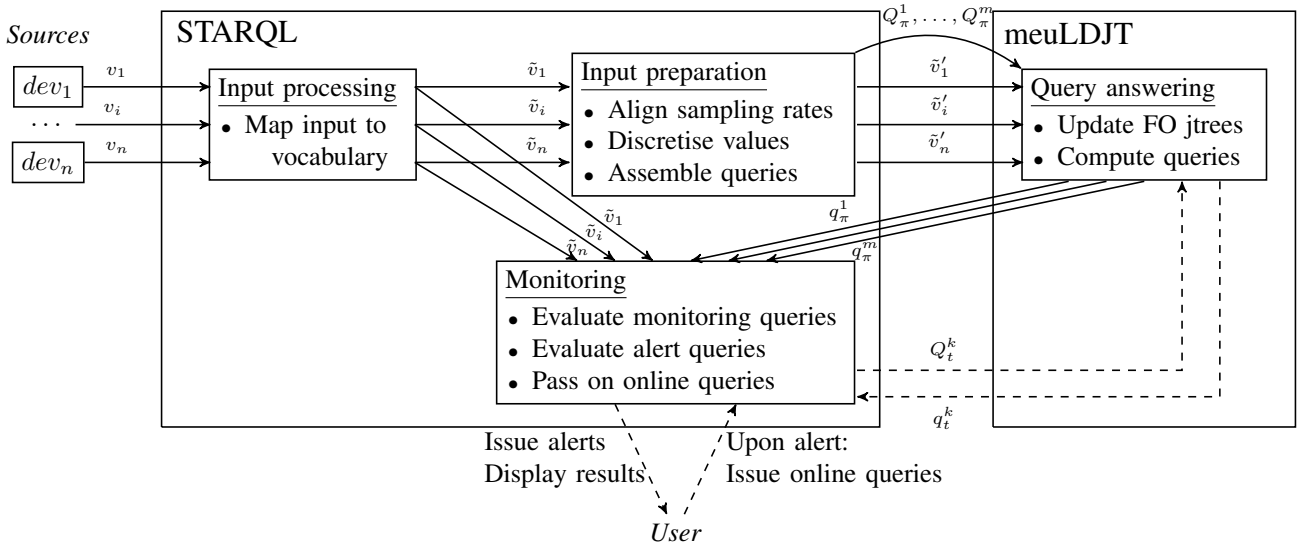

Fig. 6: PSR overview

randvar names $\mathbf{R}$ from G. By mapping the input sources to $\mathbf{R}$, STARQL can directly pass the data onwards to meuLDJT as evidence. To be able to pass on the data, STARQL also needs to map the individuals or input streams to instances in G, i.e., STARQL needs a mapping to the corresponding instance from $\mathbf{D}$. At the end of input processing, the different v_i are mapped to a unified vocabulary, leading to $\tilde{v}_i$.

The next step is *input preparation*, which works on the $\tilde{v}_i$ data. STARQL performs the following tasks for each t.

(i) Discretise those $\tilde{v}_i$ that have continuous ranges given predefined thresholds.
(ii) Align those $\tilde{v}_i$ with a different sampling rate given predefined functions.
(iii) Assemble corresponding queries $Q^1, \ldots, Q^m$ and evidence to pass on to meuLDJT.

The first task is due to meuLDJT only handling discrete ranges. The second task arises as sampling rates of sensors might differ and each randvar in G can only be assigned at most one value for each time step. In case some sensors or devices have a higher sampling rate than others, leading to more than one measurement in a time period, STARQL can, for example, average the information to obtain time discretised information. That is to say, STARQL produces an assignment of at most one event for each randvar for each time step, yielding data points $\tilde{v}'_i$ of the form $\mathbf{E} = \{E^i = e^i\}_i$ that meuLDJT can process as evidence.

The next step is *query answering*, which meuLDJT performs. The step includes the tasks

(i) Update the internal state of the FO jtrees.
(ii) Answer given queries $Q^1_\pi, \ldots, Q^m_\pi$.
(iii) Send query results $q^1_\pi, \ldots, q^m_\pi$ back to STARQL.

The first task means that meuLDJT stores the current model state in α_t, increases its internal time from t to $t \leftarrow t+1$, recovers the previous model state, enters the new evidence, and passes messages. Then, meuLDJT calculates a best action sequence for a finite horizon for an MEU query. For that best action sequence, meuLDJT then answers additional marginal queries. The results of the different queries are sent back to STARQL for monitoring and display.

The last step is *monitoring* by STARQL. The tasks are:

(i) Based on $range$ and $slide$, evaluate monitoring and alert queries on data and query streams.
(ii) Send online queries on to meuLDJT to evaluate.

STARQL evaluates monitoring and alert queries on the streams in a sliding window. STARQL displays the action sequence from meuLDJT as well as the results of the monitoring queries. If all alert queries evaluate to false, PSR continues with new data points arriving from the sources. If an alert query evaluates to true, PSR basically freezes the current state to allow for further query answering. If an alert query evaluates to true, STARQL issues an alert to the user. Upon an alert, a user may issue further queries Q^k_t for the current model state (online queries), which STARQL passes on to meuLDJT to evaluate. After answering the new queries, meuLDJT sends the query results q^k_t back to STARQL for display.

Monitoring can be used for different aspects. For example, we can use it for pattern recognition. Another idea would be to see if the probabilities for a certain event are increasing within a window. Further, we could also be interested in if the probabilities are over a certain threshold for a given number of time step in a window and request STARQL to issue an alert if true. Thus, with STARQL, we can reason over the inference results of meuLDJT to see how the probabilities change over time, providing another indication which action should be performed next.

VII. Evaluation

In this empirical evaluation, we analyse the efficiency and scalability of PSR. The window parameters, which depend on user requirements, influence the efficiency of STARQL

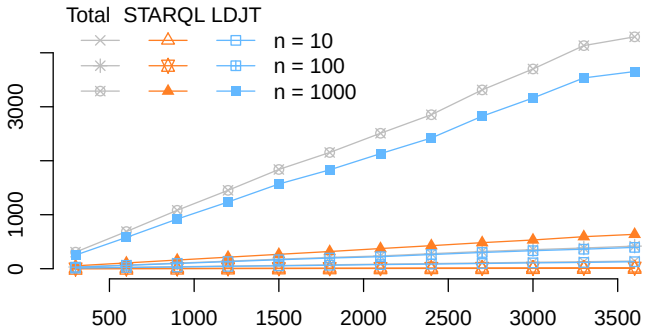

Fig. 7: Runtimes for $range = 5$ and $slide = 1$ with changing domain sizes, y-axis: [seconds], x-axis: maximum time steps

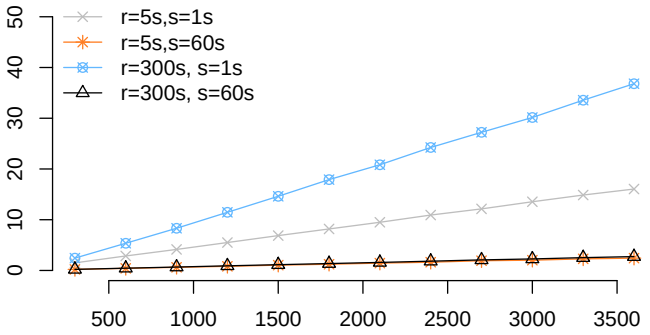

Fig. 8: Runtimes for $n = 10$ with changing range and sliding values, y-axis: [seconds], x-axis: maximum time steps

as small windows with high frequency mean more frequent calculations. For meuLDJT, the domain size and the maximum number of time steps are the largest contributing factors, the former depending on the problem at hand, the latter on user requirements. Therefore, we look at these parameters. We expect meuLDJT to take up a large part of the runtime as reasoning takes time. Nonetheless, we expect PSR runtimes to depend only linearly on the maximum number of time steps and only polynomially on domain sizes (*not* exponentially).

For the evaluation, we use G^{ex} with domain sizes $|\mathcal{D}(X)|$ set to 10, 100, and 1000. We assume that we obtain stream data from one source with the terminology of G^{ex}. Further, we split the evidence into two uniform groups, i.e., after evidence entering, there are two groups of persons in G^{ex} that has indistinguishable members. Queries for meuLDJT are an MEU query for a current time step as well as filtering queries for $Epid$ and $Sick(X)$ for each group of persons. Based on the query results, STARQL checks whether the probability of an epidemic is above 80%. We test the parameter $range$ with 5 and 300 and the parameter $slide$ with 1 and 60.

Figure 7 shows the runtimes of PSR with $range = 5$ and $slide = 1$ and $|\mathcal{D}(X)|$ to 10, 100, and 1000. We can see multiple aspects in the figure. The first aspect is that PSR runs in linear time w.r.t. the maximum number of time steps. The next aspect is that most time is spent inside of meuLDJT. meuLDJT has to solve an MEU query for the current time step and answer 3 *filtering* queries. Thus, meuLDJT has to perform $4 - 5$ message passes for each time step and answer overall 11 queries. With the evidence, we split the logvar X into two groups. Thus, we need to iterate over 4 actions for each time steps, as for both groups we can perform both actions. Hence, we have at least 4 message pass and might need 5 in case the last action assignment does not lead to the maximum utility. The 11 queries are a combination of 3 filtering and 8 utility queries for each time step. STARQL only needs to reason over the query results of meuLDJT. Further, as meuLDJT is a lifted algorithm, it suffices to reason over representatives. Thus, STARQL also only needs to reason over representatives instead of each individual. Lastly, we can see that PSR is highly scalable due to meuLDJT. When increasing $|\mathcal{D}(X)|$ from 10 to 100, runtimes only need about 3 times longer. Therefore, PSR is polynomial w.r.t. the domain sizes instead of exponential, which applies for a ground inference algorithm.

How changing $range$ and $slide$ influences the runtime of STARQL is depicted in Fig. 8 for a domain size of 10. We only show the runtimes of STARQL since the runtimes of meuLDJT remain as shown in Fig. 7. In Fig. 8, we see that the $slide$ value has the biggest impact on the reasoning times of STARQL. The $slide$ value basically determines over how frequent STARQL has to reason in a window. With $slide = 1s$ and $range = 5s$, STARQL frequently moves along its sliding window. By setting the range value to a high value, in this case, $300s$, STARQL has huge windows to reason over each second, leading to the highest runtime. If STARQL only reasons over a few small windows, e.g., $slide = 60s$ and $range = 5s$, the runtimes are dominated by providing meuLDJT with the input and the reasoning aspect hardly plays any role.

As mentioned, due to meuLDJT, STARQL also only needs to reason over representatives. Hence, increasing the domain sizes of the logvar X does not increase the reasoning time of STARQL and the STARQL reasoning times remain the same for $n = 100$ and $n = 1000$. Nonetheless, the time STARQL requires to provide meuLDJT with inputs increases as more evidence needs to be passed along.

Overall, PSR is efficient and scalable given our setup as it runs in linear time w.r.t. the maximum number of time steps and in polynomial time w.r.t. domain sizes. Real-time processing with one second corresponding to one time step is possible depending on domain sizes. E.g., meuLDJT only takes around $0.12s$ to evaluate all queries for one time step with a domain size of 100. With PSR, STARQL and meuLDJT benefit from the strengths of the other approach, allowing for performing inference and supporting decision making on temporal uncertain data streams.

VIII. Conclusion

We show how one combines a stream-based reasoning approach, in this case STARQL, and an approach to perform inference on temporal probabilistic relational models, in this case meuLDJT, to perform reasoning on temporal uncertain data streams, resulting in PSR. PSR also reasons over a sliding window, allowing for monitoring. Thus, with PSR, we can

easily identify the next best action and also perform a long term monitoring, which could be used to identify the next patient to treat. First results demonstrate that PSR runs in linear time w.r.t. the maximum number of time steps.

Future work includes to extend STARQL with an incremental reasoning over a sliding window. That is to say, STARQL only adjusts for the data that is new in a window and the data that is removed from the window.

REFERENCES

[1] P. Koopmann, "Ontology-Based Query Answering for Probabilistic Temporal Data," in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19)*, P. V. Hentenryck and Z.-H. Zhou, Eds. Honolulu, USA: AAAI Press, 2019.

[2] M. Gehrke, T. Braun, and R. Möller, "Lifted Dynamic Junction Tree Algorithm," in *Proceedings of the 23rd International Conference on Conceptual Structures*. Springer, 2018, pp. 55–69.

[3] M. Gehrke, T. Braun, and R. Möller, "Relational Forward Backward Algorithm for Multiple Queries," in *Proceedings of the 32nd International Florida Artificial Intelligence Research Society Conference (FLAIRS-19)*. AAAI Press, 2019.

[4] M. Gehrke, T. Braun, R. Möller, A. Waschkau, C. Strumann, and J. Steinhäuser, "Lifted Maximum Expected Utility," in *Artificial Intelligence in Health*. Springer International Publishing, 2019, pp. 131–141.

[5] M. Gehrke, T. Braun, and R. Möller, "Lifted Temporal Most Probable Explanation," in *Proceedings of the International Conference on Conceptual Structures 2019*. Springer, 2019.

[6] Özgür. L. Özçep, R. Möller, and C. Neuenstadt, "A stream-temporal query language for ontology based data access," in *KI 2014*, ser. LNCS, vol. 8736. Springer International Publishing Switzerland, 2014, pp. 183–194.

[7] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodríguez-Muro, and R. Rosati, "Ontologies and databases: The DL-Lite approach," in *5th Int. Reasoning Web Summer School (RW 2009)*, ser. LNCS. Springer, 2009, vol. 5689, pp. 255–356.

[8] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "C-sparql: a continuous query language for rdf data streams," *Int. J. Semantic Computing*, vol. 4, no. 1, pp. 3–25, 2010.

[9] D. L. Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth, "A native and adaptive approach for unified processing of linked streams and linked data," in *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, ser. Lecture Notes in Computer Science, L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, Eds., vol. 7031. Springer, 2011, pp. 370–388.

[10] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray, "Enabling ontology-based access to streaming data sources," in *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ser. ISWC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 96–111. [Online]. Available: http://dl.acm.org/citation.cfm?id=1940281.1940289

[11] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "Ep-sparql: a unified language for event processing and stream reasoning," in *WWW*, 2011, pp. 635–644.

[12] J.-U. Kietz, T. Scharrenbach, L. Fischer, M. K. Nguyen, and A. Bernstein, "Tef-sparql: The ddis query-language for time annotated event and fact triple-streams," University of Zurich, Department of Informatics (IFI), Tech. Rep. IFI-2013.07, 2013.

[13] J.-P. Calbimonte, J. Mora, and O. Corcho, "Query rewriting in rdf stream processing," in *Proceedings of the 13th International Conference on The Semantic Web. Latest Advances and New Domains - Volume 9678*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 486–502.

[14] E. Kharlamov, T. Mailis, G. Mehdi, C. Neuenstadt, O. L. Özçep, M. Roshchin, N. Solomakhina, A. Soylu, C. Svingos, S. Brandt, M. Giese, Y. Ioannidis, S. Lamparter, R. Möller, Y. Kotidis, and A. Waaler, "Semantic access to streaming and static data at Siemens," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 44, pp. 54–74, 2017.

[15] S. Borgwardt, M. Lippmann, and V. Thost, "Temporal query answering in the description logic DL-Lite," in *FroCos 2013*, ser. LNCS, vol. 8152, 2013, pp. 165–180.

[16] A. Artale, R. Kontchakov, F. Wolter, and M. Zakharyaschev, "Temporal description logic for ontology-based data access," in *IJCAI 2013*, 2013, pp. 711–717.

[17] E. Della Valle, S. Ceri, D. Barbieri, D. Braga, and A. Campi, "A first step towards stream reasoning," in *Future Internet – FIS 2008*, ser. LNCS. Springer, 2009, vol. 5468, pp. 72–81.

[18] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. L. Özçep, C. Svingos, D. Zheleznyakov, S. Brandt, I. Horrocks, Y. E. Ioannidis, S. Lamparter, and R. Möller, "Towards analytics aware ontology based access to static and streaming data," in *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, P. T. Groth, E. Simperl, A. J. G. Gray, M. Sabou, M. Krötzsch, F. Lécué, F. Flöck, and Y. Gil, Eds., vol. 9982, 2016, pp. 344–362.

[19] B. Ahmadi, K. Kersting, M. Mladenov, and S. Natarajan, "Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training," *Machine learning*, vol. 92, no. 1, pp. 91–132, 2013.

[20] I. Thon, N. Landwehr, and L. De Raedt, "Stochastic relational processes: Efficient inference and applications," *Machine Learning*, vol. 82, no. 2, pp. 239–272, 2011.

[21] T. Geier and S. Biundo, "Approximate Online Inference for Dynamic Markov Logic Networks," in *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2011, pp. 764–768.

[22] T. Papai, H. Kautz, and D. Stefankovic, "Slice Normalized Dynamic Markov Logic Networks," in *Proceedings of the Advances in Neural Information Processing Systems*, 2012, pp. 1907–1915.

[23] M. Richardson and P. Domingos, "Markov Logic Networks," *Machine learning*, vol. 62, no. 1, pp. 107–136, 2006.

[24] C. E. Manfredotti, "Modeling and Inference with Relational Dynamic Bayesian Networks," Ph.D. dissertation, Ph. D. Dissertation, University of Milano-Bicocca, 2009.

[25] D. Nitti, T. De Laet, and L. De Raedt, "A particle Filter for Hybrid Relational Domains," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013, pp. 2764–2771.

[26] J. Vlasselaer, W. Meert, G. Van den Broeck, and L. De Raedt, "Efficient Probabilistic Inference for Dynamic Relational Models," in *Proceedings of the 13th AAAI Conference on Statistical Relational AI*, ser. AAAIWS'14-13. AAAI Press, 2014, pp. 131–132.

[27] J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, and L. De Raedt, "TP-Compilation for Inference in Probabilistic Logic Programs," *International Journal of Approximate Reasoning*, vol. 78, pp. 15–32, 2016.

[28] S. Borgwardt, I. I. Ceylan, and T. Lukasiewicz, "Ontology-Mediated Query Answering over Log-linear Probabilistic Data," in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19)*, P. V. Hentenryck and Z.-H. Zhou, Eds. Honolulu, USA: AAAI Press, 2019.

[29] T. Braun and R. Möller, "Parameterised Queries and Lifted Query Answering," in *Proceedings of IJCAI 2018*, 2018, pp. 4980–4986.

[30] ——, "Lifted Junction Tree Algorithm," in *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer, 2016, pp. 30–42.

[31] M. Gehrke, T. Braun, and R. Möller, "Lifted Temporal Maximum Expected Utility," in *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*. Springer, 2019.

[32] S. Schiff, Ö. L. Özçep, and R. Möller, "Ontology-based Data Access to Big Data," *Open Journal of Databases (OJDB)*, vol. 6, pp. 21–32, 2018, postproceeding of Hidest'18.

[33] M. Giese, A. Soylu, G. Vega-Gorgojo, A. Waaler, P. Haase, E. Jiménez-Ruiz, D. Lanti, M. Rezk, G. Xiao, Ö. L. Özçep, and R. Rosati, "Optique: Zooming in on big data," *IEEE Computer*, vol. 48, no. 3, pp. 60–67, 2015. [Online]. Available: http://dx.doi.org/10.1109/MC.2015.82

[34] R. Möller, C. Neuenstadt, and Özgür. L. Özçep, "Deliverable D5.2 – OBDA with temporal and stream-oriented queries: Optimization techniques," EU, Deliverable FP7-318338, October 2014.

[35] C. Neuenstadt, R. Möller, and Özgür. L. Özçep, "OBDA for temporal querying and streams with STARQL," in *HiDeSt '15—Proceedings of the First Workshop on High-Level Declarative Stream Processing (co-located with KI 2015)*, ser. CEUR Workshop Proceedings, D. Nicklas and Özgür. L. Özçep, Eds., vol. 1447. CEUR-WS.org, 2015, pp. 70–75.