Declarative Analysis of Noisy Information Networks

Walaa Eldin Moustafa [#1], Galileo Namata [#2], Amol Deshpande [#3], Lise Getoor [#4]

[#]*Department of Computer Science, University of Maryland*
College Park, MD, USA

[1]walaa@cs.umd.edu
[2]namatag@cs.umd.edu
[3]amol@cs.umd.edu
[4]getoor@cs.umd.edu

***Abstract*—There is a growing interest in methods for analyzing data describing networks of all types, including information, biological, physical, and social networks. Typically the data describing these networks is observational, and thus noisy and incomplete; it is often at the wrong level of fidelity and abstraction for meaningful data analysis. This has resulted in a growing body of work on extracting, cleaning, and annotating network data. Unfortunately, much of this work is ad hoc and domain-specific. In this paper, we present the architecture of a data management system that enables efficient, declarative analysis of large-scale information networks. We identify a set of primitives to support the extraction and inference of a network from observational data, and describe a framework that enables a network analyst to easily implement and combine new extraction and analysis techniques, and efficiently apply them to large observation networks. The key insight behind our approach is to *decouple*, to the extent possible, (a) the operations that require traversing the graph structure (typically the computationally expensive step), from (b) the operations that do the modification and update of the extracted network. We present an analysis language based on *Datalog*, and show how to use it to cleanly achieve such decoupling. We briefly describe our prototype system that supports these abstractions. We include a preliminary performance evaluation of the system and show that our approach scales well and can efficiently handle a wide spectrum of data cleaning operations on network data.**

I. Introduction

In today's world, networks abound. Examples include social networks, communication networks, financial transaction networks, gene regulatory networks, disease transmission networks, ecological food networks, sensor networks, and more. There is a growing interest in methods for analyzing such network data for scientific discovery, anomaly detection, vulnerability prediction, and assessing the potential impact of interventions. Although observational data describing these networks can often times be obtained, an inherent problem with much of this data is that it is noisy and incomplete, and at the wrong level of fidelity and abstraction for meaningful data analysis. Thus there is a need for methods which extract and infer "clean" annotated networks from noisy observational network data. This involves inferring missing attribute values (attribute prediction), adding missing links and removing spurious links between the nodes (link prediction), and eliminating duplicate nodes (entity resolution).

While methods have been proposed for doing each of these extractions/inferences in isolation, there has been little work on fully integrated approaches. The little work that has been done has been ad hoc, domain specific, and typically performed outside a declarative data management framework. This makes it cumbersome to store and compare results of different approaches, or to handle dynamic updates to the underlying observation network. Furthermore, the sizes of real-world networks are growing at a rapid pace, with networks with millions of nodes and edges becoming ubiquitous. To support analysis and cleaning of these networks, a framework is needed for efficiently storing, managing, and analyzing such large, dynamic network data.

In this paper, we present the design and architecture of a data management system that enables efficient, declarative analysis of large-scale information networks. Our goal is to provide a declarative framework for common operations required in cleaning and extracting networks, a mechanism for combining them in various ways, and an implementation for efficiently applying them to large observation networks. The three main challenges in building such a framework are: (a) network analysis is heavily dependent on the actual graph structure and typically requires traversal of the node neighborhoods and computation of structural features; (b) most network analysis techniques are inherently *iterative*, and require repeated passes over the graph; and (c) network cleaning and analysis often needs to be "collective" (where a decision in one part of the network affects the information flow in other parts of the network).

The key to our approach is to *decouple* the graph traversal operations from the modification operations; the traversal operations are typically computationally expensive, especially for large disk-resident graphs. We present a declarative analysis language based on *Datalog*, and show how it can be used to cleanly achieve such decoupling. This decoupling enables us to develop a framework for declarative analysis over large networks, and facilitates efficient execution, by allowing us to push much of the computation inside a database system. Further, the declarative framework allows us to efficiently incorporate, and propagate through the analysis task, dynamic updates to the network data. We have implemented a prototype system called GRDB (Graph Database) that supports our declarative framework. Our preliminary results illustrate the computational and usability advantages of our system.

Our main contributions can be summarized as follows:

- We identify the commonalities between different graph extraction and cleaning tasks and derive a decoupling that enables efficient integration of these tasks.

- We propose an interface for specifying network analysis tasks that makes it easy for the users (network analysts) to experiment with different methods and combinations of features to decide how to best analyze and clean a network. Our framework supports defining prediction domains, features, and functions, which allows a declarative interface for the coupled inferences required for network cleaning.
- We present several extensions to Datalog giving it operational semantics rather than fixed-points semantics, where necessary, to make it more suitable for network analysis.
- We develop algorithms for efficiently computing the features, and for incrementally maintaining them in the presence of updates introduced by the predictions (a requirement given the iterative nature of network analysis). Due to space limitations, we discuss these techniques only briefly.
- We present the results of a preliminary experimental study over a real dataset.

II. Network Inference Operations

The operations that are commonly required in cleaning network data include filling in missing information, correcting inaccurate information, and consolidating and reconciling redundant information. In this work, we frame these operations as *prediction* problems, and use machine learning classification algorithms to perform them. We make a distinction between *local* classification algorithms, which make predictions based on known attributes of the prediction element and *collective* classification algorithms whose predictions can depend on the output of other classifiers. The prediction problems supported in our system can be broadly classified into three categories:

Attribute prediction: Predicting the value for an attribute of a node. Predictions can be made based on the values of other attributes of the node (local classification) or based on the predicted neighbors' attribute values (collective classification). The underlying assumption in attribute prediction is that the links between nodes carry important information for inferring the attribute values.

Link prediction: Predicting the edges in the network [10], [12]. The link prediction problem can be formulated as a classification problem where we associate a binary variable for each pair of nodes which is true if a link exists between the two nodes and false otherwise. The prediction can depend on structural features computed based on the network (e.g. number of common neighbors) and attribute values of nodes.

Entity resolution: Identifying when two nodes in the graph are referring to the same real-world entity. In this case, the nodes should be merged, and their attributes and links should be updated accordingly. Common approaches to entity resolution use a variety of similarity measures, often based on approximate string matching criteria [8], [4], [6]. These work well for correcting typographical errors and other types of noisy reference attributes. More sophisticated approaches make use of domain-specific attribute similarity measures and often learn such mapping functions from resolved data. Other approaches take graph structure and similarity into account [2], [9] and allow dependencies among the resolutions, e.g., *collective* entity resolution [5].

III. Specification Language and Data Model

Our specification language for defining inference tasks builds upon Datalog. A Datalog program consists of a set of rules and a set of facts. Facts represent statements that are true, whereas rules allow us to deduce new facts from other true facts that are already known (or deduced), and exist in the knowledge base. A Datalog rule has the following syntax:

$$\mathtt{L}_0 \text{ :- } \mathtt{L}_1, \ \ldots, \ \mathtt{L}_n$$

where each of $\mathtt{L}_i$ is a *literal* of the form $P_i(X_1, \ldots, X_n)$, or $\sim P_i(X_1, \ldots, X_n)$, where P_i is a predicate symbol, and $X_1, \ldots, X_n$ are terms. For the purposes of our GRDB specification, we consider only definite clauses, in which there are no negations. Also, in some places, we use the shorthand $P(\overline{X})$ where $\overline{X}$ stands for $X_1, \ldots, X_n$. Terms can be variable terms or constant terms. Informally, rules are read as 'if $\mathtt{L}_1, \ \ldots, \ \mathtt{L}_n$ are true, then $\mathtt{L}_0$ is true.' $\mathtt{L}_0$ is called the rule's LHS or *head*, and $\mathtt{L}_1, \ \ldots, \ \mathtt{L}_n$ are called the rule's RHS, or *body*. Each $\mathtt{L}_i$ on the rule's RHS is called a subgoal. A fact is a rule with an empty body and is always true. A fact that has all its terms constant is called a ground fact. In database terminology, each predicate symbol corresponds to a relation name. An *extensional database (EDB)* is the set of relation names corresponding to ground facts. An *intensional database (IDB)* is the set of relation names corresponding to inferred facts. Our graph is stored in an EDB, while rules defining various inference tasks are expressed as IDBs.

We use Datalog as the base language for our graph analysis framework for several reasons.

- Datalog can naturally capture both graph structure and properties of graph elements (i.e., nodes and edges). For example, one may query two hop neighbors from node `X` using the rule `TwoHops(X,Z):-Edge(X,Y),Edge(Y,Z).`
- Compared to SQL, Datalog is a natural language to answer path-based graph queries because it is a recursive language.
- On the other hand, compared to XPath, Datalog deals with graph edges as first-class citizens, where they can have identifiers and attributes that can be queried. In XPath, an edge is just expressed by the "/" (slash) operator.
- Compared to RDF query languages like SPARQL, Datalog can be naturally extended to handle concepts like feature domains (Section IV) and updates.
- Finally, compared to imperative languages, a declarative language like Datalog relieves the user from the burden of specifying how to evaluate the query by pushing this work to the evaluation engine. Furthermore, its algebraic properties allow the system to incrementally compute the changes in query results when the base graph changes, while in imperative languages, it is not as clear how to track dependencies and perform incremental maintenance.

Our data model supports multiple node and edge types where each type has its own set of attributes. Although our approach and framework can be applied to any EDB schema representing a graph structure, for brevity, we will assume just two EDBs, $Node(X, \overline{A})$ and $Edge(X, Y, \overline{B})$, where the *Node* relation contains a key, X, along with a set of attributes $\overline{A}$ and the *Edge* relation contains a key (X, Y) along with edge

attributes $\overline{B}$. We have the following shorthand: `Node(X)` stands for `Node(X,_,...)` and means that node `X` exists in the EDB. Similarly, `Edge(X, Y)` means that node `X` points to the node `Y`. `Node(X,Att=V)` stands for `Node(X,_,..,V,..)` and means that node `X` has the value `V` for attribute `Att`, and similarly, `Edge(X, Y, Att=V)` means that edge `(X, Y)` has the value `V` for attribute `Att`.

We extend Datalog with several constructs to enable our analysis framework. Some bear close similarity to existing Datalog extensions (e.g. aggregation), whereas others are new.

- **Aggregates:** An aggregate is a term of the form $Agg\langle\overline{Y}\rangle$ where Agg is an aggregation function, and $\overline{Y}$ are the aggregate function arguments. For a rule: $P(\overline{X}, Agg\langle\overline{Y}\rangle) :- P_1(\overline{X_1}), \ldots, P_n(\overline{X_n})$, where $\overline{X}, \overline{Y} \subseteq \bigcup_i \overline{X_i}$, a set is created for each value of $\overline{X}$, the aggregate operation Agg is applied on each set, and a corresponding fact is added.
- **Update Rules:** We use update rules to express graph updates that result from inference operations. Since updates have side effects, the order in which these side effects should take place must be specified explicitly in the program. Hence, our programs are divided into two parts: (1) the non-update rules (i.e. query rules) where evaluation order does not matter, and (2) the update rules where it matters. We use following syntax to express updates:
  `[INSERT | DELETE | UPDATE]` $P(\overline{X}) :- P_1(\overline{X_1}), \ldots, P_n(\overline{X_n})$
  where the predicate name P corresponds to an EDB, where the changes will take place. The semantics are that the rule is evaluated and the results are then added or deleted from P's EDB for `INSERT` or `DELETE`, and updated (based on their keys) for `UPDATE`.
- **`ITERATE` Construct:** We introduce the `ITERATE` construct as a looping construct to allow updates to be performed iteratively:
  `ITERATE(N) { Block of Update Rules }`
  where N is either the number of iterations or $*$. The semantics of the `ITERATE` construct are that it applies the update rules in its body in the specified ordering iteratively, and recomputes (or maintains) the results of any "query" rules, until no change takes place or for at most N iterations, whichever happens first. If $*$ is specified, then the evaluation proceeds indefinitely until no changes take place.
- **Other extensions:** There are other Datalog extensions specific to our framework, like **`DOMAIN` constructs**, and **top K ranking**. We discuss these extensions as we encounter them.

IV. Declarative Analysis Framework

While the three network cleaning operations described in Section II result in different updates to the network, and can be combined in complex ways, network analysis processes can be seen, at a high level, as interleaved application of three basic modules as shown in Figure 1(i). In this section, we describe these components, and then present our proposed declarative language constructs for specifying these components.

A. Defining Prediction Domains and Features

For graph inference tasks, we typically need to compute the values of various "features" for a set of relevant objects, called *prediction elements*. Depending on the nature of features, this step is typically the most computationally expensive step in the overall process. The prediction elements are either nodes (for attribute prediction) or pairs of nodes (for link prediction or entity resolution). For scalability, we must somehow constrain the set of prediction elements, especially in the latter case. We call the set of prediction elements considered during inference the *prediction domain* of the task.

Features: We can divide the features broadly into three categories based on their complexity:

Local: Attributes of the prediction elements themselves can be used as features for input to a prediction function. For nodes, these are node attributes; for pairs of nodes, these can be binary features which describe whether the attribute values of the nodes match, or real-valued features which measure the distance between the nodes' attribute values. The key distinguishing characteristic of these features is that they require only local information about the attribute values of the nodes, or pairs of nodes.

Local Structural: These are features that require exploration of a small fixed neighborhood around the prediction element. For node predictions, commonly used features include the **degree** of the node, the count of the number of neighbors within c hops with a specific property (where c is a constant), etc. Another commonly used feature is the **clustering coefficient**. The clustering coefficient $C(x)$ of a node x in a graph is a measure of how close the node and its neighbors are to forming a clique; more precisely, it is the ratio of edges observed over the number of possible edges. In addition, for pairs of nodes, we often measure some sort of neighborhood similarity. Common neighborhood similarity measures include: **No. of common neighbors**; **Jaccard coefficient**, which is the number of common neighbors normalized by the size of the union of the neighbors of the nodes; and, a related measure, introduced by Adamic and Adar [1], which gives more weight to rare features (those that are not shared by many other entities).

Global Structural: Examples of features that depend on the global structure of the graph include the **Katz score**, **betweenness centrality**, and **PageRank**. The Katz score for a pair of nodes is computed based on the number of different paths between the two nodes, with the shorter paths given higher weight than the longer paths. See Table I for the formal definition. The betweenness centrality of a node is determined by the number of shortest paths that contain that node; it is the frequency with which a node appears along the shortest path between other pairs. PageRank of a node captures the probability that a random walk will end up at the node.

The features can be specified using Datalog in a straightforward manner (see Table I).

Domains: While features may be defined for all prediction elements, often we want to restrict our attention to only a subset of the elements to make analysis tractable. We refer to

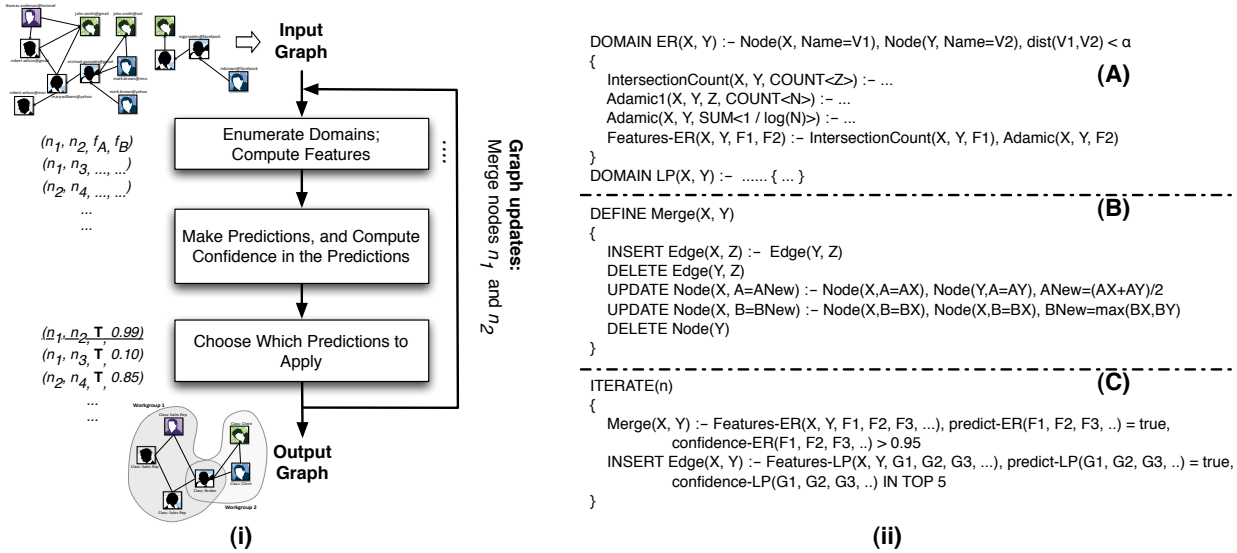

Fig. 1. (i) Illustrative workflow depicting the main steps in an iterative statistical inference task (using the example of *entity resolution*); (ii) An example Datalog program that specifies an interleaved execution of an ER task and an LP task in a decoupled fashion by separating the feature computation operations ((**A**) from the graph modification operations ((**B,C**)).

Degree: $Degree(x) = \|\Gamma(x)\|$	`Degree(X, COUNT<Y>) :- Edge(X, Y)`
No. of neighbors w/ Att = 'A'	`NumNeighbors(X, COUNT<Y>) :- Edge(X, Y), Node(Y, Att='A')`

Clustering Coefficient	`NeighborCluster(X, COUNT<Y,Z>) :- Edge(X,Y), Edge(X,Z), Edge(Y,Z)`
	`Degree(X, COUNT<Y>) :- Edge(X, Y)`
	`ClusteringCoeff(X, C) :- NeighborCluster(X,N), Degree(X,D), C=2*N/D*(D-1)`

Number of common neighbors	`IntersectionCount(X, Y, COUNT<Z>) :- Edge(X, Z), Edge(Y, Z)`

Jaccard's coefficient $Jaccard(x,y) = \frac{\|\Gamma(x)\cap\Gamma(y)\|}{\|\Gamma(x)\cup\Gamma(y)\|}$	`Degree(X, COUNT<Z>) :- Edge(X, Z)`
	`IntersectionCount(X, Y, COUNT<Z>) :- Edge(X, Z), Edge(Y, Z)`
	`UnionCount(X, Y, D) :- Degree(X,D1), Degree(Y,D2), D=D1+D2-D3`
	`IntersectionCount(X, Y, D3)`
	`Jaccard(X, Y, J) :- IntersectionCount(X, Y, N), UnionCount(X, Y, D), J=N/D`

Adamic measure $Adamic(x,y) = \sum_{z\in\Gamma(x)\cap\Gamma(y)} \frac{1}{\log\Gamma(z)}$	`Degree(X, COUNT<Z>) :- Edge(X, Z)`
	`Adamic1(X, Y, Z, N) :- Edge(X, Z), Edge(Y, Z), Degree(Z,N)`
	`Adamic(X, Y, SUM<1/log(N)>) :- Edge(X, Z), Edge(Y, Z), Adamic1(X, Y, Z, N)`

Similarity based on a func. $f(v1, v2)$	`Similarity(X, Y, S) :- Node(X, Att=V1), Node(Y, Att=V1), S=f(V1, V2)`

Katz measure $Katz(x,y) = \sum_{l=1}^{\infty} \beta^{-l}.\|paths(x,y)^{<l>}\|$	`Path(X, Y, 1) :- Edge(X, Y)`
	`Path(X, Y, L1) :- Edge(X, Z), Path(Z, Y, L), L1=L+1`
	`Path_count(X, Y, L, COUNT<1>) :- Path(X, Y, L)`
	`Katz1(X, Y, L, K) :- Path_count(X, Y, L, N), K=N * power(β, -L)`
	`Katz(X, Y, SUM<K>) :- Katz1(X, Y, L, K)`

TABLE I
COMMON RELATIONAL FEATURES AND THEIR DATALOG REPRESENTATION. WE USE $\Gamma(x)$ TO INDICATE THE SET OF NEIGHBORS OF NODE x.

such a subset of elements as the *prediction domain*.

For attribute prediction, the prediction is over attribute values of the nodes and we can use the domain construct to restrict our attention to some subset of the nodes. This allows us, for example, to predict attribute values only for nodes with missing attribute values, or to predict attribute values only for nodes which have some percentage of neighboring values observed (not missing). Judicious use of prediction domains is especially important for tasks such as link prediction and entity resolution, where the prediction takes place over pairs of nodes. For a reasonably-sized network, it is infeasible to check every possible prediction element, and we must be able to limit the possible node pairs that are considered.

We use the keyword `DOMAIN` for defining a domain for features. The general syntax for specifying a domain is:

```
DOMAIN D(X1, X2, ...) :-   ...
    {    ⟨ List of features to be computed ⟩    }
```

For example, during entity resolution, we may want to restrict ourselves to pairs of nodes that are sufficiently close to each other based on the *string similarity distance* between their names. This can be specified as shown in Figure 1(ii)(A). Although it may seem that this domain requires listing all the pairs of nodes and filtering them, our framework supports efficient methods for avoiding that. The last rule (with head `Features-ER`) combines all the features into a single predicate using which we can do inference. Note that although we have focused on unary or binary prediction domains thus far, our

framework allows for using n-ary domains; this may be needed for situations where we want to make predictions for groups of three or more entities.

B. Iterative Inference and Updating

The next step in the analysis process is to perform the required inferences and updates. For each prediction element, the prediction is made by applying a *user supplied function* over the features computed in the previous step and returning a *prediction* and a *confidence* (or *score*) value. This function can either be a user defined function or a function that is the output of some machine learning system; in the context of GRDB, we treat it as a black box. A key observation we make here is that, at this point, the *prediction can be done independently for each domain element in parallel*.

For attribute prediction, commonly used prediction functions include classifiers like naïve Bayes, logistic regression, and decision trees. Similarly, for link prediction, the problem of deciding whether to add an edge between a pair of nodes is often treated as a *binary* classification problem, and the functions listed above can be used as well. In some cases, especially for entity resolution, a similarity function might be used instead to compute a similarity score for a pair of nodes, and then a thresholding mechanism may be used to decide which nodes to merge or which edges to add.

The next step depends on the nature of the inference task. In some cases, we may just make one pass and commit all of the predictions made. In other cases, we may only choose to commit a subset of the predictions, and may want to iteratively recompute the features and perform inference on the updated graph. The updates include attribute value changes (for attribute prediction), edge insertions/deletions (for link prediction), and node merges (for entity resolution), and we must recompute the values of the features in response to these updates. Such iterative application often results in more accurate predictions and robust behavior. The most common approach to choosing which predictions to commit is to choose either the top k of the predictions (by score) or all predictions with confidence above a given threshold.

In general, for each individual inference task, the user must specify:

- *Prediction* function to be used and the predicate containing the features. The prediction function is written as a user-defined function (UDF) **Predict**: $\overline{FT} \rightarrow P$, where $\overline{FT}$ is the feature vector and P is the set of possible predictions.
- *Confidence* or *score* function to be used to choose a subset of the predictions to commit. This is also typically written as a UDF **Confidence**: $\overline{FT} \rightarrow [0,1]$ (or more generally, **Score**: $\overline{FT} \rightarrow \Re$).
- *Prediction Confidence Cut-off:* In addition, the user must specify how to choose the subset of the predictions to be committed. A cut-off value for the confidence is provided by defining a predicate over the confidence function. A predicate can take the form of a minimum given threshold (e.g. `confidence(`$\overline{FT}$`) > C`), or can be expressed by picking the top K predictions. We define a Datalog extension for this purpose (`confidence(`$\overline{FT}$`) IN TOP K`).
- *Graph Update Operations* to be performed as a result of the inference. These are expressed as Datalog update rules.
- *Number of Iterations* used when updates are executed iteratively so that only high confidence predictions are applied in each iteration. As we described earlier, update rules are enclosed in an `ITERATE` block to achieve this control.

As an example, an entity resolution task where we only commit high-confidence predictions can be specified as (Figure 1(ii)(C)):

```
Merge(X, Y) :- Features-ER(X, Y, F1, F2, F3, ...),
               predict-ER(F1, F2, F3, ..) = true,
               confidence-ER(F1, F2, F3, ..) > 0.95
```

Here `Features-ER` contains all the features that are needed for inference. `predict-ER` and `confidence-ER` are the prediction and confidence functions respectively. To differentiate between functions and predicates in our Datalog programs, we use upper case initials for predicates and lower case initials for functions. `Merge(X, Y)` indicates that the graph update operation to be performed is a merge (corresponding to entity resolution or duplicate elimination). Other examples include `INSERT Edge(X, Y)`, indicating edge addition between nodes `X` and `Y` (for link prediction, see Figure 1(ii)(C)), and `UPDATE Node(X,Att=V)`, indicating that the attribute value of `Att` should be changed to `V` for node `X`, (for classification or attribute prediction).

Note that update operations corresponding to link prediction and attribute prediction are simple (i.e. a single rule). However, the `Merge` operation can be composite, i.e., defined in terms of other operations. An example of `Merge` definition is shown in Figure 1(ii)(B). This allows the user specify exactly how to update the attribute values for the new node that is created. The semantics of composite updates is that the update rules inside them are executed in order; however, there is no need to recompute the features after each single update rule. Features are recomputed only after the entire composite block is executed.

Finally, the user may specify an interleaving of two or more different inference tasks. For example, the syntax for specifying an interleaving of entity resolution and link prediction is as shown in Figure 1(ii)(C). Here for the second inference task, we specify that only the `TOP 5` of the predictions (based on the `confidence-LP` function results) be committed at end of each iteration.

C. Implementation

To implement our framework, we built a deductive database system on top of the Java Edition of the Berkeley DB key/value store. We implemented a full fledged non-transactional relational database system on top of Berkeley DB that has *a query parser*, *a rule-based query optimizer*, *a relational expression converter* for converting Datalog rules to canonical relational expressions, and *a plan executor*. We omit further details of this component due to space constraints. Second, we implemented the necessary special logic to enable our framework, such as the `DOMAIN`, and the `ITERATE` constructs.

An important optimization is incremental maintenance. We materialize the result of every Datalog rule in the system, and we treat these results as materialized views over the base relations. As the base relations change in response to the predictions made during analysis, we need to maintain these views. In our prototype, we devise different methods to handle feature views, DOMAIN views, and cascaded views (where the output of one rule is propagated to another rule).

V. Experimental Evaluation

We crawled a portion of the PubMed online dataset – a citation network in the medical domain. The size of the network is 50,634 papers and it has 115,323 citation edges. The dataset has four categories describing the topic of the paper (Cognition, Learning, Perception, and Thinking). We used GRDB to infer the category (class label) of each paper. We used 2-fold cross validation, where we trained a classifier using about half of the network, and tested it on the other half. We used logistic regression for the prediction function, which we supply with the presence of words of the paper abstract (we use a bag of 1678 words) and the count of citations of each category. When committing the top 10% of the predictions and iterating for 10 iterations, the incremental maintenance method takes 28 minutes (average over the two folds) to finish, while recomputation from scratch takes 42 minutes. Note that there is some inherent overhead due to the large number of local features (i.e., words of the abstract) which logistic regression has to reason about for each node in each iteration. Using this approach leads to 84% accuracy in predicting the paper categories. [1]

VI. Related Work

In recent studies, Datalog has been the centerpiece in enabling declarative specification in various domains, like network protocol specification [11], sensor networking [7], recommendation in social networks [13], and data cleaning [3]. Arasu et al. [3] employ Datalog to solve the problem of collective entity resolution using domain-specific constraints. The constraints are in the form of user-defined soft and hard rules. The system performs the deduplication by satisfying all the hard rules and minimizing the number of violations to the soft rules. Our approach also supports a declarative approach toward collective entity resolution, however our approach is capable of performing a more general set of network inferences. In [4], the authors consider the problem of generic entity resolution, where they define entity resolution in terms of two functions, *match* that matches two records and *merge* that merges two records if they match. These two functions are treated as black-boxes, and the authors define classes for the their properties, studying efficient algorithms for different classes. Our work is in a similar spirit, in attempting to define black boxes for the prediction problems, however we focus on a declarative specification for the interactions among the predictions, and efficient incremental maintenance.

[1] We also performed an extensive performance evaluation using synthetic data. Details can be found in the extended version of the paper at: http://www.cs.umd.edu/~walaa/grdb.pdf.

San Martin and Gutierrez [14] present a social network data model for representing, querying and transforming social networks. The proposed data model is based on RDF and the proposed query and transformation language is based on a composition of SPARQL and SQL. The authors gather the requirements of the proposed language from operations common in social network software tools and published social network research, such as network projection, subnetwork extraction and ranking. However, our focus here is on supporting the full range of graph extraction and inference problems including attribute prediction, entity resolution and link prediction, which prior work does not support.

VII. Conclusions

We described the design of a data management system, called GRDB, for supporting declarative graph analysis over noisy information networks. Our system supports new constructs for defining graph-based inference operations, and heavily exploits the common properties shared by these operators to enable efficient storage and execution. We chose to base our language on Datalog because of its expressive power in representing computation over graphs in an intuitive and easy-to-understand manner. The key insight behind our approach is to decouple the operations that require traversing the graph structure (typically the computationally expensive step) from the operations that do the modification and update of the extracted network. We built a prototype system that implements this functionality and briefly discussed its implementation. We showed a preliminary performance evaluation study on a real-world data network.

Acknowledgements: This work was supported in part by NSF under Grants IIS-0546136 and IIS-0916736.

References

[1] L. Adamic and E. Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2003.
[2] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, 2002.
[3] A. Arasu, C. Re, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, 2009.
[4] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *The VLDB Journal*, 18:255–276, 2008.
[5] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM TKDD*, 1:1–36, 2007.
[6] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.
[7] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, 2007.
[8] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proc. of IJCAI Workshop on Information Integration*, August 2003.
[9] D. Kalashnikov, S. Mehrotra, and Z. Chen. Exploiting relationships for domain-independent data cleaning. In *SIAM SDM*, 2005.
[10] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *CIKM*, 2003.
[11] B. Loo, T. Condie, M. Garofalakis, D. Gay, J. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.
[12] M. J. Rattigan and D. Jensen. The case for anomalous link discovery. *SIGKDD Explorations Newsletter*, 7:41–47, 2005.
[13] R. Ronen and O. Shmueli. Evaluating very large datalog queries on social networks. In *EDBT*, 2009.
[14] M. San Martín and C. Gutierrez. Representing, querying and transforming social networks with rdf/sparql. In *ESWC*, 2009.