PDQ: Proof-driven Query Answering over Web-based Data*

Michael Benedikt, Julien Leblay, and Efthymia Tsamoura
Oxford University, UK
name.surname@cs.ox.ac.uk

ABSTRACT

The data needed to answer queries is often available through Web-based APIs. Indeed, for a given query there may be many Web-based sources which can be used to answer it, with the sources overlapping in their vocabularies, and differing in their access restrictions (required arguments) and cost. We introduce PDQ (Proof-Driven Query Answering), a system for determining a query plan in the presence of web-based sources. It is: (i) *constraint-aware* – exploiting relationships between sources to rewrite an expensive query into a cheaper one, (ii) *access-aware* – abiding by any access restrictions known in the sources, and (iii) *cost-aware* – making use of any cost information that is available about services. PDQ takes the novel approach of *generating query plans from proofs that a query is answerable*. We demonstrate the use of PDQ and its effectiveness in generating low-cost plans.

1. INTRODUCTION

This work concerns answering queries on top of remote data-sources, such as web forms and web services, that have restricted access, i.e., one must provide some input to retrieve some data. Our system starts with a query and a schema, consisting of integrity constraints and access restrictions and (a) determines whether the information in the sources is sufficient to completely answer the query and (b) assuming the answer to (a) is yes, finds the plan which *minimizes the cost*, with respect to some cost function. Our queries need not be written explicitly in terms of the sources, but can be written in terms of other relations related to the accessible sources via mappings or integrity constraints. The system automatically considers pulling in data from related sources in searching for a plan. Indeed, the use of integrity constraints may be critical for finding any plan.

EXAMPLE 1. *Consider a scenario where queries are being written directly using the Yahoo! GeoPlanet API. The* YPlaces *service is at the center of the API, featuring basic information about places of various granularities. This includes a unique place ID (*yId*), as well as other attributes* name, type, latitude, longitude, *etc.* YPlaces *requires a* yId *as input, or a place* name *with an optional* type. *A plethora of other services also exists providing, for example, the relationships among places (*YBelongsTo, YChildren, *etc.). The following query asks for all countries belonging to "Asia":*

```
SELECT p1.name FROM YBelongsTo AS belong
       JOIN YPlaces AS p1 ON p1.id=belong.source,
       JOIN YPlaces AS p2 ON p2.id=belong.target
WHERE  p1.type = 'Country' AND p2.name = 'Asia'
```

where source *is a required input of* YBelongsTo. *At first sight, the query is not answerable since not all required inputs to* YPlaces *and* YBelongsTo *are provided. However, there exists a relation* YCountries *with input-free access listing all places of type "Country". Therefore, it is possible to answer the query by (i) retrieving all countries from* YCountries *and projecting on their* yId *and* name*, (ii) retrieve all places named "Asia" from* YPlaces *(iii) using each returned* yId *from (i) as input to* YBelongsTo*, returning* name*s whose associated* yId*s join with the result of (ii).*

This is a simple example, but it shows that a complete query plan may or may not be obtainable, and that information on the semantics of the data plays a key role in determining a plan. Above there was one reasonable plan, but often there can be many, with different costs.

EXAMPLE 2. *Consider a data integration scenario, where the user writes queries on a global schema, without any knowledge of the underlying web-accessible relations. In this example,* Continent, Country *and* CapitalCity *are (virtual) global relations, thus we model them as* inaccessible.
The following query asks for capital cities in Asia:

```
SELECT countryName, capital FROM Continent AS c1
       JOIN Country AS c2 ON c1.name=c2.continentName
       JOIN CapitalCity AS c3 ON countryCode
WHERE  c1.name = 'Asia'
```

We express relationships between virtual relations and online services as integrity constraints. There exists a Yahoo service named YContinents *with an input-free access returning all places of type "Continent". One of our constraints states that the natural join* Continent $\bowtie$ Country *is equivalent to the join* YCountries $\bowtie$ YBelongsTo $\bowtie$ YContinents, *after projecting on* id*s and* name*s. Similar constraints can be defined between our virtual relations and the Geonames and WorldBank APIs. For instance,* Country $\bowtie$ CapitalCity *is equivalent to* WBCountries *after projecting on* name*s and* countryCode. *Yahoo does not hold any information about capital cities. However, mappings from country codes to capital cities can be found in two other services, namely both Geoname's* GNCountries *and* WBCountries*, provided by the World-Bank API.*

*Supported by EPSRC grant EP/H017690/1, Query-driven Data Acquisition from Web-based Data Sources

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 13
Copyright 2014 VLDB Endowment 2150-8097/14/08.

There are multiple plans to evaluate this query. First, one can leverage the first constraint described above to extract countries located in Asia from the Yahoo API, then join with either GNCountries *or* WBCountries *to obtain capital cities. Since these two plans are very similar, their difference in cost will be depend on the cost of accessing* GNCountries *vs.* WBCountries*. Alternatively, one could rely exclusively on relations from Geonames.*

We use a novel approach to finding plans for queries based on *searching for a proof that a query can be answered*. Starting with a query Q and a schema with access patterns and integrity constraints, we first generate a *proof goal*, where the goal informally states "assuming that Q holds of a tuple t, show that a user could learn that it holds". The goal will have the property that *every proof of the goal corresponds to a way of answering the query*. We have thus reduced the plan-generation problem to that of proof search, and our Proof-driven Query Answering system (PDQ) searches for a proof of the goal while concurrently measuring the cost of the corresponding plans. Two other features that discriminate PDQ over state-of-the-art approaches are its ability to find low-cost plans with respect to an arbitrary user-defined cost function, and it ability to deal with a broader class of constraints. We allow classes for which the chase terminates (e.g. stratified, weakly-acyclic) and also *guarded tuple-generating dependencies* (GTGDs), of the form $\forall \vec{x}\ G(\vec{x}) \wedge \bigwedge_{i<m} B_i(\vec{x}) \rightarrow \exists \vec{y} \bigwedge_{i<n} H_i(\vec{x},\vec{y})$, with G, B_i, H_i atoms and $G(\vec{x})$ containing every variable occurring in some B_i.

Related Work. Query answering in the presence of access patterns has been studied in the past with cost-agnostic [5] and cost-aware approaches [3]. The first work to look at access patterns and integrity constraints is that of Deutsch et al. [2], where the authors couple axioms that capture access restrictions with integrity constraints, and then apply a query reformulation technique to obtain a rewriting that accords with access restrictions. In [1], it is shown that this technique is a special case of a correspondence between proofs and plans that extends to arbitrary first-order constraints, and which can be applied effectively beyond the case of "terminating chase" (a restriction of [2]). [1] outlines an algorithm that searches for plans (rather than queries) applying a plan cost function as one searches. Like [2], it is a theoretical work that does not propose a system or explore any applications (as here). In contrast, Kambhampati et al. [4], Srivastava et al. [7] and Preda et al. [6] present systems to optimize queries over web services, minimizing some cost function, but without consideration for integrity constraints.

To the best of our knowledge, PDQ is the first system that achieves the combined goals of checking the anwerability of a query and producing an optimal plan, taking into account both access restrictions on data sources and integrity constraints.

2. PDQ

Our system takes as input a schema S that consists of relation descriptions (attributes and data types) and integrity constraints that are GTGDs or have terminating chase. Relations are endowed with *access methods*, each of which associates a relation with a set of required input attributes. Relations and access methods are assumed to carry cost information, such as selectivity and per-tuple access cost. The second input is a Select-Project-Join query Q (as in most prior work on view querying, we assume set semantics). Other inputs to PDQ include parameters to our search strategies and heuristics, but we omit them here for simplicity. Section 4 has more details about these parameters and how they can be used to tune the planning algorithm from the user interface.

The first step of the planning phase is to *augment* the input schema S with new relations and with rules that tell how these new relations are related to the original ones. For each relation R in the original schema S, we have a relation InferredAcc$R$ that informally denotes all tuples which can be inferred to be in R using access methods and constraints. We also have a relation accessible(x) which holds all values that can be retrieved via accesses. In addition to the new relations, our augmented schema will have new integrity constraints, which we refer to as "accessibility axioms", that capture the informal semantics of relations of the form InferredAcc$R$, and the relation accessible. For instance, the augmented schema for Example 1 will add an axiom:

$$\mathsf{accessible}(name) \wedge \mathsf{accessible}(type) \wedge YPlaces(\ldots,name,type,\ldots)$$
$$\rightarrow \mathsf{InferredAcc}YPlaces(\ldots,name,type,\ldots) \wedge \mathsf{accessible}(zipc) \wedge \ldots$$

Informally this asserts that if name and type values have become accessible, and we have a hidden YPlaces fact F involving those values, then a) F can be made known to the user and b) the other attribute values in the fact can become accessible. It thus encodes that there is an access method requiring name and type as inputs to YPlaces. We also have a copy of the original constraints on the inferred accessible copy of the relations. The new schema generated is referred to as the *accessible schema*, S_{acc}. In addition, we create a copy of Q, named $Q_{inferred}$, adding atoms accessible(x) for each free variable x and replacing each R with InferredAcc$R$, i.e., any atom $R(\vec{x})$ in Q corresponds to InferredAcc$R(\vec{x})$ in $Q_{inferred}$. The basic result (proven in [1]) is that plans for Q over the schema S correspond to proofs of $Q_{inferred}$ from Q using axioms of S_{acc}. When integrity constraints are given by TGDs, a proof can be taken to be a sequence of databases $D_1, D_2, \ldots$ that starts with the "canonical database of Q" – the database that has elements for each variable and constant in Q, and facts for each atom of Q – and adds on facts by applying rule-firings of S_{acc}. A proof is *successful* if it leads to a database where the goal query $Q_{inferred}$ holds.

A proof is a sequence of facts produced by firing either integrity constraint rules or accessibility axioms. The sequence of facts in the proof will be needed to determine whether a proof is successful and, if it is not yet successful, to determine what new rules could be applied to extend it. PDQ's *proof-to-plan* algorithm is in charge of producing a plan from a proof. This is achieved by looking at the sequence of accessibility axiom rule firings (and corresponding set of facts) that are contained in the proof. The full set of possible proofs forms a tree. *Proof search* proceeds by exploring this tree, expanding one branch at a time. As a branch is expanded, the proof is checked to see if it is successful, a plan is generated from the proof and the cost of the plan is computed. The exploration of a branch stops when a successful proof is found or when the cost of the associated plans is too high. For a tree node that corresponds to a not-yet-successful proof, the accessibility axioms that can still be fired represent *candidates* for extending the tree.

Algorithm 1 outlines these steps. Here we have omitted a number of details, such as what it means to obtain "all consequences" in various steps. Our system can handle cases where the number of such consequences is infinite, determining a cut-off point where further derivation is not useful. The detailed algorithm is available in [1]. Figure 1 shows a portion of the proof space for the query in Example 2. Nodes in Figure 1 represent partial proofs. For clarity, we only show a subset of the facts that each node contains, and abbreviate (e.g. accessible(x) as $Acc(x)$). Also note that each node includes the facts of its parent. The initial node S_0 at the top simply contains a *grounded* version of the query. From there, we exhaustively apply rules (up to a termination condition), other than accessibility axioms, to derive new facts. In S_1, we have already derived several facts that could trigger candidate accessibility axioms. In this case, the axiom corresponding to an input-free access to YCountries is picked. We represent the firing of this axiom by an edge leading to a new node S_2. Reasoning on this new set of

Algorithm 1: Planning algorithm

Input: query Q, schema S, cost threshold t
Output: plan BestPlan

1. S_{acc} := augmented schema formed from S
2. $Q_{inferred}$:= augmented query formed from Q
3. v_0 :=derive all consequences of Q w.r.t. constraints of S
4. ProofTree := $\{v_0\}$ // initial state
5. BestPlan := $\perp$ BestCost := ∞
6. **while** *there is a proof node in* ProofTree *that still has candidate rules to explore* **do**
7.   Choose such a node v.
8.   Choose a candidate rule firing c in v. // The choice is guided by search strategies
9.   Fire the rule to create a new node v_c, adding it to ProofTree as a child of v.
10.   Close v_c under firing inferred accessible integrity constraint rules.
11.   Generate $\mathsf{Plan}(v_c)$ and compute $\mathsf{Cost}(\mathsf{Plan}(v_c))$.
12.   Mark v_c as aborted if the cost is higher than BestCost, or above t.
13.   Mark v as fully explored if it has no more candidates.
14.   Check if $Q_{inferred}$ holds in v_c, i.e., there is a complete proof.
15.   **if** v_c *is successful and* $\mathsf{Cost}(\mathsf{Plan}(v_c)) < \mathsf{BestCost}$ **then**
16.   BestPlan := $\mathsf{Plan}(v_c)$
17.   BestCost := $\mathsf{Cost}(\mathsf{Plan}(v_c))$
18. return BestPlan

facts renders Country's ids accessible. Therefore, the accessibility axiom that corresponds to an access to YBelongsTo, with its first attribute as input, can now be fired, leading to S_3. This can be directly followed by a rule-firing of an axiom corresponding to an input-free access to YContinents. In S_4, we have already inferred that the Country and Continent facts from Q are accessible. Finally, the two pink nodes S_5 and S_6 at the bottom of the figure show alternative proofs reached through accessibility axioms firings corresponding to distinct accesses: one on GNCountries and one on WBCountries, both with input-free accesses. In either case, we can immediately infer that the CapitalCity relation is accessible, which also means there is a match with $Q_{inferred}$.

3. ARCHITECTURE AND TEST DATA

The architecture of our application is shown in Figure 2. It comprises two main components: a planner and a runtime.

Plan exploration. The planner is the central part of the software, with a reasoning module at its core. Incoming queries are provided to the reasoner, which starts searching for proofs as described in Section 2. Proofs, successful or incomplete, are sent to a proof-to-plan module in charge of producing sub-plans. The planner enumerates complete plans, each corresponding to a complete proof, and returns an optimal one w.r.t. to some cost functions. PDQ supports a variety of cost functions whose applicability depends on the nature of the underlying resources. The cost estimation module is responsible for determining which function can be used in a given context, and delegating the cost computation to the appropriate implementation during the planning phase. For instance, if all relations in the considered schema reside in a common relational database and this database provides a cost estimation interface, this interface can be used directly during the planning phase. The cost estimation module also provides its own set of generic cost metrics. When these are used, the computation is performed within the module itself.

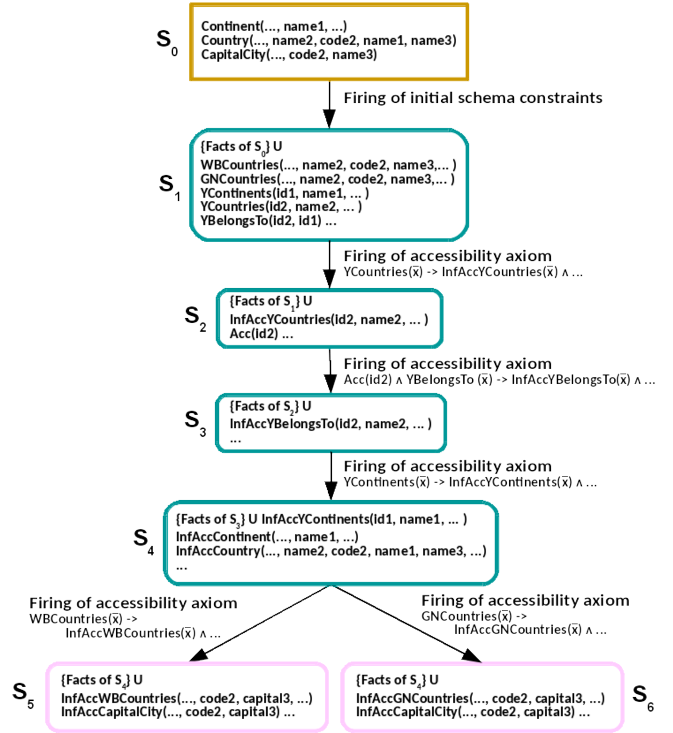

Figure 1: A portion of the plan search space for Example 2.

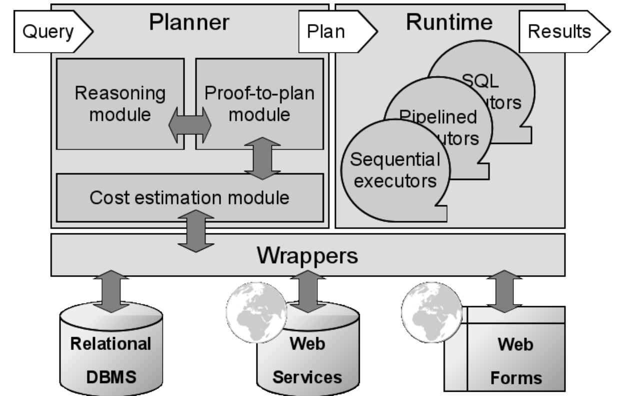

Figure 2: System's architecture.

Plan evaluation. The runtime component is a set of executors, which take plans as input and output collections of tuples as results. Available executors include: (i) an *in-memory linear plan executor*, which evaluates linear plans naïvely, producing a temporary table in-memory at each step of the execution, (ii) a *materializing linear plan executor*, which materializes every intermediary tables in a relational database and delegates other operations (select, project, join, etc.) to the RDBMS, (iii) a *pipelined executor*, which converts linear plans to conventional relational plans, making use of standard caching and pipelining techniques wherever applicable, (iv) two SQL translators, which convert plans to single SQL statements that are sent of the underlying RDBMS for evaluation. The first one uses standard SPJ queries, while the second relies on the non-standard "SELECT WITH" syntax. The latter enables defining intermediary tables explicitly and produces queries whose execution strictly follows that of the initial definition. In some cases, the choice of the executor is guided by the cost function that was used during the planning phase. For instance, if the cost estimation was

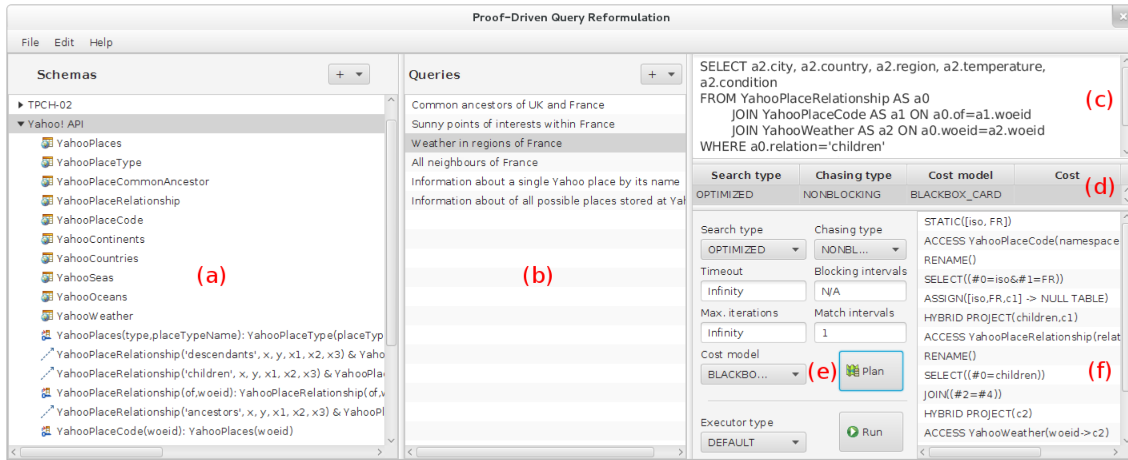

Figure 3: Screenshot of the user interface's main window.

achieved through a RDBMS interface, then the execution will take place using the appropriate SQL translators.

Interfaces with data sources. The final component, depicted at the bottom of Figure 2, provides wrapping functionality for various types of datasources. We currently support relational databases and RESTful services. As data coming from web services and online forms are generally semi-structured, this module is in charge of providing a "relational view" on such sources, converting data back and forth from trees to sets of tuples. It also encapsulates the concrete access mechanism for a service, forming the appropriate requests for any given relation and set of input tuples.

Datasets. Our demo comes with testsets that show the performance of PDQ on synthetic and real world examples. For the former, we use the TPCH benchmark stored in a PostgreSQL database, assigning randomly picked access methods. For queries, we use purely conjunctive variants of those provided with the benchmark. We had direct access to PostgreSQL's own cost estimators, and dependencies were generated from TPCH foreign key definitions. In addition, we defined views over the base relations, which generate two-way constraints between base relations and views. Our real world datasets include REST APIs of Google, Yahoo, Geonames and the World Bank. In this case, access methods were given by the APIs themselves. We manually defined dependencies between those services.

4. DEMONSTRATION WALK-THROUGH

The demonstration showcases PDQ's ability to explore different plans. PDQ provides a graphical interface (Figure 3) enabling users to load schemata. The left-hand panel (a) of the main window lists all available schemata. Unfolding a schema reveals the relations and dependencies in it. Users can use the menu at the top right-hand corner of the panel or context menus to inspect them or delete them. Upon selecting a schema, a second panel, (b) in the center of the window, displays a list of available queries over it. Similarly, queries can be imported or deleted using the top right-hand corner menu. Currently, PDQ requires both the schemata and the queries to be written in XML format. We plan to extend PDQ with a user-interface that will enable users to graphically edit queries and schemata. Selecting a query refreshes the right-hand side of the window. At the top (c), an SQL representation of the query appears, while the table below (d) lists out plans that have been generated so far for this query, along with the planner configuration used. When a configuration is selected in the table, the bottom right-hand area of the window is updated; planner configuration details are shown in (e) and the best plan found is displayed in (f). Two buttons appear at the bottom of panel (e). Pressing the button labelled "Plan" opens a new window, where one can run a new planning session for this configuration and visualize the planner's search space in real-time. The search is represented as a graph in which each node corresponds to a partial proof and its associated plan. Visual cues highlight the choices made by the planner, indicating, e.g. nodes that have been pruned or the next candidates for the exploration. Nodes can be selected to obtain detailed information about them and show the facts that have been inferred so far during the chase procedure.

Conference attendees will be invited to create their own schemas and queries from a set of predefined data sources, then run the planner with various search settings. Once an optimal plan has been found, it is stored in table (d) for later use. The second button in panel (e), labelled "Run", opens another window from where the user can run the plan using one of the available executors. Attendees will have the opportunity to compare evaluation times for plans obtained with various search settings, as well as comparing optimal plans to alternative plans. Since we know of no other available systems with comparable functionality, the demo will focus on feasibility of the approach rather than comparative evaluation.

5. REFERENCES

[1] M. Benedikt, B. T. Cate, and E. Tsamoura. Generating low-cost plans from proofs. In *PODS*, 2014. Available at `www.cs.ox.ac.uk/people/michael.benedikt/pdq/access.pdf`.

[2] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. *TCS*, 371(3):200–226, 2007.

[3] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, 1999.

[4] S. Kambhampati, E. Lambrecht, U. Nambiar, Z. Nie, and G. Senthil. Optimizing recursive information gathering plans in EMERAC. *JIIS*, 22(2):119–153, 2004.

[5] C. Li and E. Chang. Answering queries with useful bindings. *TODS*, 26(3):313–343, 2001.

[6] N. Preda, G. Kasneci, F. Suchanek, T. Neumann, W. Yuan, and G. Weikum. Active knowledge: dynamically enriching RDF knowledge bases by web services. In *SIGMOD*, 2010.

[7] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB*, 2006.