PDQ 2.0: Flexible Infrastructure for Integrating Reasoning and Query Planning

Michael Benedikt and Fergus Cooper and Stefano Germano
University of Oxford, UK

Gabor Gyorkei and Efthymia Tsamoura
Samsung AI, UK

Brandon Moore
1X2 Network

Camilo Ortiz
Google

ABSTRACT

Reasoning-based query planning has been explored in many contexts, including relational data integration, the Semantic Web, and query reformulation. But infrastructure to build reasoning-based optimization in the relational context has been slow to develop. We overview PDQ 2.0, a platform supporting a number of reasoning-enhanced querying tasks. We focus on a major goal of PDQ 2.0: obtaining a more modular and flexible architecture for reasoning-based query optimization.

1. INTRODUCTION

For many decades database researchers have explored the interaction of reasoning in query planning. Particularly relevant to our work are: 1. *Optimizing queries with respect to constraints*. We are given integrity constraints Σ, which we know that our input data satisfies. Given a query Q, our goal is to rewrite Q to a lower cost query, or directly to a physical plan, taking advantage of the constraints in Σ. 2. *Querying over limited interfaces with constraints*: We have constraints as above, but access to our dataset restricted. A model for access restrictions is to associate each relation R with a set of function calls or *access methods*, each requiring a subset of attributes of R, and performing a lookup on R. Access restrictions clearly complicate the query planning problem, even in the absence of constraints. The presence of access restrictions can make a naïve implementation of Q impractical or even infeasible even when Q is in principle answerable with the available data. The goal is to convert Q to a physical plan that makes use of the given access methods, again with the equivalence being with respect to constraints Σ.

These topics have received extensive attention in the database literature. For query optimization with respect to constraints, there is a well-studied approach, the "chase and backchase" (C&B) [9], which can be seen as a special case of interpolation [13]. For querying with respect to access restrictions, there is considerable theory [8, 7], with the most common approaches for dependencies being closely-related to the C&B. While the theory of these topics has continued to evolve, software support for the three topics above has not matured the way one would hope. The authors are aware of no open source system for querying with respect to constraints, access methods, or (as a standalone tool) views, even though algorithms have been available since the 1990s [2]. The lack of mature systems relates to an even broader issue. Although individual problems have been studied in the literature, a vision of how components for the problem might interrelate is lacking.

In this work, we try to make a small step in the direction of software maturity. We introduce an open source evolution of the PDQ system, originally targeted towards the second task above [5, 6]. Although the evolved 2.0 system focuses on the two tasks above, it can also be used as a standalone reasoning tool supporting "open world query answering" or chasing [3]. The **goals** of the software are to: 1. support the tasks above with performance comparable to PDQ 1.0, which is to our knowledge, the state of the art, 2. provide modularity, allowing a developer to mix and match different components of a solution, including components related to how reasoning is performed, how source data is stored and accessed, and how reasoning within query optimization is combined with non-logic factors like cost, 3. provide a base of well-documented code [1] to build on.

EXAMPLE 1. *Consider a schema listing relations* Actor, Movie, *and* $\mathsf{MovieActor}$. *The intended semantics of the tables would be captured with integrity constraints, including constraints capturing the semantics of* $\mathsf{MovieActor}$*:* $\mathsf{Movie}(\mathsf{mid},\mathsf{mname},\mathsf{aid}) \wedge \mathsf{Actor}(\mathsf{aid},\mathsf{aname}) \rightarrow \mathsf{MovieActor}(\mathsf{mname},\mathsf{aname})$, $\mathsf{MovieActor}(\mathsf{mname},\mathsf{aname}) \rightarrow \exists \mathsf{aid}\ \mathsf{aname}\ \mathsf{Movie}(\mathsf{mid},\mathsf{mname},\mathsf{aid}) \wedge \mathsf{Actor}(\mathsf{aid},\mathsf{aname})$, *and* $\mathsf{Movie}(\mathsf{mid},\mathsf{mname},\mathsf{aid}) \wedge \mathsf{Actor}(\mathsf{aid},\mathsf{aname}) \rightarrow \mathsf{MovieActor}(\mathsf{mid},\mathsf{aid},\mathsf{aname})$.

Consider the query asking for the names of all actors in the movie "Inside Man". Depending on which tables the user is familiar with, this could be expressed *in different ways, for example SELECT* aname *FROM* MovieActor *WHERE* mname=*'Inside Man'. However, there are a number of different ways to answer the query, in addition to the naïve join plan: for example, it is possible to first access* MovieActor *and then filter.* PDQ *will explore all equivalent plans, looking for the one with the lowest cost.*

Finally, assume these tables are available only as web services that require particular inputs, and the MovieActor *table requires an* aid *as input. Then some of the plans will no longer be valid, and* PDQ *will restrict the search accordingly.* PDQ *can also evaluate the plans produced by the search.*

We focus on the architecture of PDQ 2.0: how it would be used to solve a task, what flexibility it supports, and what issues arise. Algorithmic and implementation issues are discussed elsewhere in the context of PDQ 1.0 [5, 6], and for space reasons we defer a discussion of the 1.0 to 2.0 delta to online material [1].

2. PDQ CORE FUNCTIONALITY

The motivating application for PDQ is enhancing query planning via reasoning. Given query Q and Σ, we seek a query plan P equivalent to P for all instances satisfying Σ, where P has lower cost according to some cost function. A variation of the problem allows access restrictions in the schema, where the restrictions are specified by each relation being associated with a collection of access methods, with each access method implementing a lookup with some require arguments: the *inputs* to the method. The goal is now to get a plan where access to the data is only allowed through the given access methods. The abstraction of access methods serves to model a number of application scenarios [2].

The query planning problem can, in principle, be reduced to the well-known problem of computing *certain answers to query Q on database D with respect to constraints* Σ. These are the tuples $\vec{t}$ such that in every database D' that contains D and satisfies Σ $\vec{t}$ is an answer to Q in D' in the usual sense. The decidability and complexity of this task depends on the properties of Σ. The version of PDQ described here deals only with one of the most well-studied cases, where Σ consists of tuple-generating dependencies (TGDs). These are logical sentences of the form: $\forall \vec{x}\, \lambda(\vec{x}) \rightarrow \exists \vec{y}\, \rho(\vec{x},\vec{y})$ where λ and ρ are conjunctions of atoms, or *equality generating dependencies* (EGDs), sentences of the form $\forall \vec{x}\, \lambda(\vec{x}) \rightarrow x_i = x_j$ where λ is as above. See Example 1 for examples of TGDs: in the examples, the outer universal quantifiers have been omitted. For such sentences the certain query computation problem can be tackled via the well-known *chase algorithm* [10], in which $D = D_0$ is iteratively enhanced to database $D_1 \dots D_n$ via applying chase steps. A chase step for a TGD τ extends D_i by adding witnesses for the conclusion ρ of τ, while for an EGD it identifies two elements of D_i to make the conclusion hold.

We state the reduction only in the case where Q is a Boolean Conjunctive Query (CQ), but the reduction generalizes to non-Boolean CQs. Given Q, Σ, and the access methods, one can derive a set of constraints Σ^* and another query Q' such that there is a plan for Q using the access patterns modulo Σ exactly when Q' is certain for the database $\mathsf{CanonDB}(Q)$ with respect to Σ^*. Here $\mathsf{CanonDB}(Q)$ is the *canonical database of* Q, whose elements are the variables and constants mentioned in Q, containing facts for each atom of Q. Furthermore, one can find plans for Q by exploring chase proofs proving Q' from Σ^* and $\mathsf{CanonDB}(Q)$. This process was developed in the context of views and integrity constraints by Popa, Tannen, and Deutsch [9]. The variant for access patterns was developed in [8, 5, 6]. The idea is that one first chases $\mathsf{CanonDB}(Q)$ with the constraints in Σ, producing a set of facts I_0. One can then begin to apply certain axioms capturing the interface restrictions. In [7] these latter axioms are called *accessibility axioms*. Firing accessibility axioms at the chase level corresponds to exploring a plan with certain accesses. While exploring these plans, one also checks whether they are equivalent to the original query Q, by chasing again. Following [9], this process of iteratively exploring plans is called *back-chasing*.

Most of the details of this reduction of planning to certain answer computation will not concern us, but there are several aspects that we want to highlight: 1. The certain answer process that is generated for planning is initiated with the incomplete dataset $\mathsf{CanonDB}(Q)$: that is, a dataset only as large as the user's original query. While the chase process will generate new facts, generally the amount of data involved is very small, in contrast to standalone certain answer computation, typically directed towards large-scale data. 2. In order to find one plan, we need simply to find one chase proof. But to find good plans we will need to explore multiple chase proofs. Thus, we need to follow prior work on constraint-based query-planning [12] in finding efficient ways to explore multiple proofs. 3. Determining which plans are low cost requires costing plans. When the process is run on top of a DBMS, this may be done via a function call to an `ANALYZE` method. In a data integration setting the costing will have to be done in middleware, which will usually involve metadata, such as histograms or catalogs.

3. SOFTWARE ARCHITECTURE

The PDQ 2.0 architecture, released recently on GitHub [1], is built around 6 major packages: **Com-**

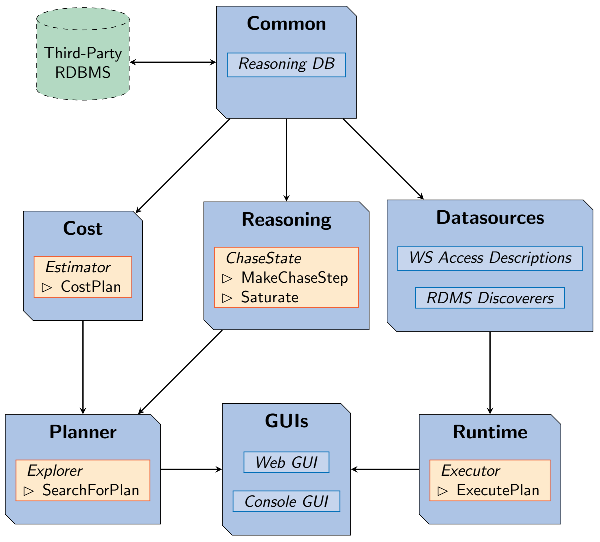


Figure 1: PDQ 2.0 software architecture: blue boxes are sub-packages, other inner boxes are classes.

mon, **Datasources**, **Reasoning**, **Planner**, **Cost**, and **Runtime**. The dependencies are shown in Figure 1.

The **Common** package has base classes for database schemas, database queries, database query plans, and first-order logic. This includes PDQ's internal data management infrastructure: the management classes which store the intermediate results of reasoning. PDQ allows internal data management to be deferred to an external DBMS, and some of these database classes simply translate query evaluation for logical formulas into SQL evaluation, passing them off to an external engine. But PDQ also has a bare-bones main-memory DBMS that can evaluate basic SQL SELECT queries.

The **Reasoning** package supports reasoning with common classes of database constraints. It exposes an abstraction called a chase state, which makes use of the database infrastructure in the **Common** package. A reasoner object can transform a chase state by either making a single chase step or chasing until termination (or timeout). The package provides an entry point allowing PDQ to be used for standalone reasoning tasks: computing the chase of an instance, determining the certain answers of a query under constraints, or the related task of determining whether one conjunctive query follows from another in the presence of constraints (*query containment with constraints*).

The **Datasources** package represents datasource descriptions and their translations to PDQ's internal representation. It includes code for extracting metadata from other formats – e.g., from the catalog of a DBMS. It also includes PDQ's framework for wrapping web services as access patterns.

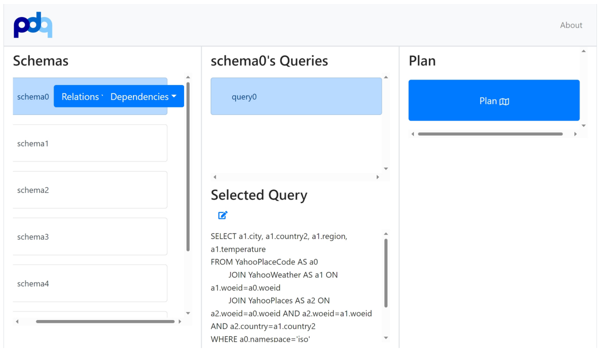
The **Cost** package represents cost estimators for physical plans. PDQ can defer the estimation to a DBMS; simply translating the plan to SQL and sending it to the `ANALYZE` method of a DBMS. This technique is appropriate if using PDQ to add constraint-based optimization on top of an existing DBMS. PDQ also includes hand-rolled cost estimation techniques; for example, using catalog estimates and heuristics.

The **Planner** package is the most complex. The main abstraction within it is an *explorer*, which is given an input query Q as well as a schema S describing a collection of abstract access methods and constraints Σ, and searches for the lowest cost physical plan making use of the accesses that is equivalent to Q modulo constraints Σ. The planner makes use of the certain answer computation provided by the **Reasoning** package and the cost calculations provided by the **Cost** package. An explorer can restrict the search to the traditional left-deep plans that are often the focus in traditional query planners; it can also search for bushy plans, imposing additional restrictions to make the search more manageable.

Finally, the **Runtime** package is middleware for evaluating physical plans. It provides *executors* that represent implementation strategies for plans. At the leaf level of a plan tree, each executor makes use of the access methods specified by the **Datasources** package. The strategy for evaluating other plan operators is encapsulated in the executor, which implements traditional and dependent joins. Note that the **Runtime** package has no dependence on the **Planner** package: it can be used to evaluate any plan, even one constructed operator-by-operator via the plan constructors.

In addition to these core packages, PDQ 2.0 has packages for front-ends. A web-based GUI serves as a demonstration of the planning functionality available for a fixed schema. A console-based GUI allows for both planning and some administration of the schema. It also serves as purely Java-based infrastructure for building front ends to the system. There are also packages (omitted from Figure 1) for demos, regression testing, and web services wrapping system functionality.

We show how some of the scenarios in Example 1 would play out using the PDQ 2.0 architecture. In setting up an application, a DBA would create metadata that describes the web services: for example they would describe a REST web service that takes as input an aid and returns other MovieActor info. PDQ offers a simple specification language that allows DBAs to describe how inputs are inserted into a service call and how the outputs are extracted, using XPath with placeholders. This specification bridges between the format of the concrete web services and the more abstract view of the services as taking in a parameter and returning a table. The parsing of these services as well as the marshalling and unmarshalling code generated by them is done via



```
SELECT a1.city, a1.country2, a1.region,
a1.temperature
FROM YahooPlaceCode AS a0
      JOIN YahooWeather AS a1 ON
a1.woeid=a0.woeid
      JOIN YahooPlaces AS a2 ON
a2.woeid=a0.woeid AND a2.woeid=a1.woeid
AND a2.country=a1.country2
WHERE a0.namespace='iso'
```

Figure 2: Web GUI

the **Datasources** package. The DBA must also supply a schema that includes integrity constraints, those listed in Example 1. PDQ provides a number of syntaxes for these, a native XML format as well as the "ASCII" format supported by the ChaseBench project [3]. The processing of these inputs is done within the IO subpackage of the **Common** package. The DBA may also make available catalog information about the underlying web services: e.g., how selective an access on MovieActor with a given aid is. This can be described using a *catalog file* processed by the **Cost** package.

We describe how a querier would interact with the system. One option is to utilize the command line interface, where various parameters (described below) will need to be passed. Alternatively one of PDQ's GUIs can be utilized. Both GUIs allow a user to select a schema, bring up an existing query or write a query from scratch, validate the query against the schema, and then perform planning. The Web GUI in Figure 2 shows a query written in SQL referencing several data sources for geographic information, available via Yahoo web services. The query can be answered via data from other services (e.g., Google): PDQ will detect multiple plans and select the one with the lowest cost. For the Web GUI, a fixed schema, planning method and cost function is available. In the console GUI the user can configure internal parameters as in the command line; it can be run on a set of metadata files available on a local machine. In both GUIs, a user can see the best plan, while within the console GUI they can see the entire search space of plans. The user can choose any available plan and run it, using the **Runtime** package. A small number of output tuples are displayed in the GUI, while the entire output can be downloaded as a CSV file.

We now turn to the parameters and their relation to the architecture. One set of parameters controls the type of reasoning to use: basically, how to perform multiple chases, with each chase corresponding to a potential plan. One parameter controls whether this is stored in an external database or PDQs in-memory DBMS. Note that *PDQ's internal database is distinct from the datasources with which PDQ interfaces*. A second reasoning parameter specifies what "saturating with consequences" means: it can mean that reasoning is done until complete, or using a fixed number of steps. A *reasoner* object within the **Reasoning** package encapsulates these decisions. A second set of parameters concerns planning search heuristics. They are encapsulated in the *explorer* classes that the **Planner** package provides. For example, the *linear planner* searches only for tree-shaped plans, while the *dag planner* allows bushy plans to be considered. A third set of parameters determine which cost function provided by the **Cost** package is used. Some cost functions that make use only of the syntax of the plans (e.g., based on size of the plan or number of joins), while others make use of catalog information, like the selectivity of the MovieActor access method mentioned above. PDQ provides a default *executor* from the **Runtime** package for running the plans, but a plan can also be saved for later running by other means (e.g., translation to SQL and execution via a DBMS).

4. ARCHITECTURAL ISSUES

In the process of constructing PDQ, we encountered a number of design decisions that are worth highlighting. They give a hint at the complexity of query reformulation/query planning software, which we believe has been one of the major obstacles in creating a codebase for this problem.

Interfacing reasoning and data management. The most basic issue we needed to deal with was how the reasoner would use a DBMS. We considered this problem in the case where the constraints are dependencies where the chase algorithm terminates. In implementing the chase there are a number of design decisions, including whether one interfaces to a data management system with SQL, whether one works natively in main memory or on top of an external DBMS, and whether one implements *nulls* via strings or via integers. Many of the trade-offs are discussed in [3]. For the purposes of PDQ, the question was not so much which design is better, but to what extent we can insulate the reasoner from these decisions.

In our original implementation, we attempted to abstract away completely from database management issues, creating a homomorphism detection class that could be implemented either in bulk within SQL – to find all triggers at once and then use SQL update commands to do a full round of chasing, or tuple-at-a-time, allowing one to perform the restricted chase. The resulting architecture was complex, and since our abstractions were not standard, it was difficult for a developer to maintain the structure. We thus simplified the architecture to assume an SQL interface to a DBMS. The reasoning project of PDQ provides an implementation of the terminating chase on top of an SQL abstraction. We developed two implementations of the abstraction, one using the open source DBMS PostgreSQL, and another using our own main-memory implementation, supporting only conjunctive queries with inequalities.

Insulating the reasoner from the encoding of nulls in the DBMS is challenging. In performing the chase it is necessary to generate fresh values for each unbounded datatype. One could use different value-generation strategies for each datatype, but a common solution is to make use of strings, where creating a fresh value and encoding the parameters it depends on (e.g., the dependency and the homomorphism) can be done using string operations. PDQ takes a variant of this approach: we have a special class of values corresponding to null, and they are associated with string. In our own main memory implementation, tables are not strongly-typed, so in executing a chase step we are free to add tuples containing nulls in positions where the schema specifies a datatype other than string. When we use an external DBMS, we simply pre-process to encode the entire database as strings and perform the entire reasoning process with the encoded database. The encoding/decoding steps are given in a single place within the code, and thus the heart of the chase implementation code need not be aware of whether the encoding has occurred.

Interfacing planning and reasoning. The reduction of planning to reasoning mentioned in Section 2 is simple, but a few caveats must be noted. For finding *some* plan, it gives a very general reduction for an arbitrary first-order theory. But for exploring low-cost plans it works only for restricted cases, like dependencies and disjunctive dependencies, and there are modifications to the planning algorithms needed for each case. Even when restricting to dependencies with terminating chase, there are some design decisions relating to the fact that we need to explore many proofs. We need to maintain multiple chase states of the exploration, all of which are subsets of the facts in $\mathsf{Chase}_\Sigma(\mathsf{CanonDB}(Q))$. This required us to broaden the DBMS abstraction used for query answering, introducing a *multi-database* (omitted from Figure 1) consisting of multiple virtual instances of the same schema. In our main-memory DBMS this is implemented by just creating multiple Java object instances. In our external DBMS implementation, a multi-database is realized by adding an additional id field, and there is then some additional translation of queries and updates.

Another issue involves the extent of parallelization. While many aspects of chasing are inherently sequential, much of the exploration of plans making use of the chase – the "back-chasing" – can be done in parallel. PDQ supports this, but the cost is difficulty in system testing, since with parallelization enabled, different runs will often produce different plans.

The interface to a reasoner was a key design decision. We oriented PDQ towards chase-based reasoners that work "in place" and expose the evolving chase state – that is, supporting the *MakeChaseStep* and *Saturate* methods. This simplified the design, but did restrict reformulation options. In the course of the PDQ project we have experimented with a number of more advanced reasoning methods: 1. The "infinite chase" or "blocking chase", applicable to tuple-generating dependencies that satisfy conditions such as guardedness; 2. The disjunctive chase, which generalizes the classical chase to rules with disjunction in the head; 3. Resolution-based theorem provers [4], applicable to arbitrary first-order integrity constraints. All of these are available in the PDQ 1.0 codebase. While the first could be re-implemented in PDQ 2.0 (by enhancing the ChaseState), the second and third require changes to the interface.

Interfacing planning, cost, and runtime. Query reformulation systems based on the C&B were geared towards "high-level planning" – logical reformulation with constraints. The notion of cost supported was minimal size of the reformulation [12]. In PDQ a key goal was to allow more general notions of cost, including cost functions that can make use of database statistics and the ordering of operations. On the other hand, it is desirable for planning to be decoupled from runtime processing. One should be able to take a plan and utilize it on any runtime that implemented that accesses in the plan.

Our solution is to support high-level planning, but with some abstraction for the notion of "better plan". The PDQ planner assumes access to a cost function. We provide a number of cost functions, some that just use count or weighted sum of certain operators. For example, one cost function takes a weighted sum of the subterms corresponding to data access. The weight of an access method defaults to 1, but it can be assigned a numerical weight in a catalog file which is read in at planning time. More sophisticated cost functions work recursively on an access plan, making use of catalog information in a standard way. The cost functions work on "logical plans", expressions in a variant of relational algebra. The runtime package converts these expressions to physical plans – for instance, by choosing a join implementation. Cost information is not taken into account in the "low-level plan creation" given by the runtime process. PDQ also allows costing to be deferred to a DBMS, using an SQL ANALYZE command. This is appropriate if PDQ is used only for planning.

Interfacing planning and data APIs. A unique feature of PDQ is that it performs query reformulation over access methods. An access method is abstractly represented by an identifier, a relation, and a subset of attributes of the relation (the *input attributes* for the relation). Plans based on the access methods are built up from access primitives via the usual relational algebra commands along with *dependent joins* [11], in which the ordering of the joined terms is fixed. The high-level description of access methods, along with the integrity constraints, are sufficient for the purposes of determining what the valid plans are for a given query. Assuming a cost function on such plans, we can begin to search for low-cost plans implementing the query.

But to run the plan we need an implementation of each access method on relation R with input attributes $a_1 \dots a_k$; this is a function call that takes values for these attributes and returns a set of tuples matching R. PDQ provides a Java interface for defining implementations of access methods, so in principle a developer can implement an access method using an arbitrary Java function. A thornier issue, and a longstanding one in the database community, is how to ease the burden on DBAs in wrapping datasources for PDQ. Automated support is provided for a few kinds of access methods. Access to data in main memory is supported in a straightforward way. For database lookups, the developer just needs to declare that the method is implemented via an SQL DBMS, and associate it with JDBC-style connection information. A more involved type of access method implementation is via web services. Even sticking to REST-based services, we unsurprisingly found a great diversity in the encoding schemes used to wrap real-world services. After experimenting with a number of more ambitious specification formalisms, PDQ currently allows only a simple kind of wrapper, where the developer needs to specify regular expressions that determine how values are encoded in a URL. Returning to Example 1, the wrapper implementing a lookup of Movie on aid might include the "template" expression `http://www.imdb/actor={1}`. When a service call is made with a particular actorid, that value will be inserted in place of 1 in the template.

5. DISCUSSION

We presented the open source version of PDQ, focusing on introducing the functionality offered and presenting some issues — many of them admittedly still open — in creating a simple but flexible architecture for reasoning-based optimization.

We consider the system in light of the goals outlined in Section 1. Despite the challenges noted in Section 4, PDQ 2.0 offers much in the way of re-use. Developers can re-use our reasoner and runtime independent of planning. They can use our cost estimators and datasource infrastructure independent of both reasoning or planning. It is also possible to swap in another chase engine in place of our reasoner, if the engine provides the *ChaseState* interface from Figure 1. The codebase is well-documented, and we are using it as a foundation for our ongoing projects in integrating querying and reasoning. We hope others can build on it as well.

Turning to the final goal, for space reasons we could not discuss the algorithms, implementation, and performance of PDQ 2.0 in depth. The search algorithms are those from PDQ 1.0: see [6]. The performance of PDQ 1.0 for standalone reasoning is evaluated in [3]. We do note that PDQ 2.0 configured with the internal in-memory database improves substantially on the reasoning performance of 1.0; this translates into better performance on the ChaseBench benchmark and the ability to perform complex planning tasks in seconds. We refer the reader to the Wiki in [1] for details of the performance on both standalone reasoning and planning.

6. REFERENCES

[1] PDQ on GitHub, 2021. `https://github.com/ProofDrivenQuerying/pdq`.

[2] F. N. Afrati and R. Chirkova. *Answering Queries Using Views*. Morgan Claypool, 2019.

[3] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, 2017.

[4] M. Benedikt, E. V. Kostylev, F. Mogavero, and E. Tsamoura. Reformulating queries: Theory and practice. In *IJCAI*, 2017.

[5] M. Benedikt, J. Leblay, and E. Tsamoura. PDQ: Proof-driven query answering over web-based data. In *VLDB*, 2014.

[6] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. In *VLDB*, 2015.

[7] M. Benedikt, B. ten Cate, J. Leblay, and E. Tsamoura. *Generating Plans from Proofs*. Morgan Claypool, 2016.

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

[9] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[10] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

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

[12] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for min. query reform. under constraints. In *SIGMOD*, 2014.

[13] D. Toman and G. Weddell. *Fund. of Phys. Design and Query Compilation*. Morgan Claypool, 2011.