How Can Reasoners Simplify Database Querying (And Why Haven't They Done It Yet)?

Michael Benedikt
University of Oxford

ABSTRACT

The last few decades have seen vast progress in computational reasoning. This has included significant developments in theory, increasing maturity of tools both in performance and usability, and the evolution of standards and benchmarks. The purpose of this article is to reflect on the use of reasoning for rewriting and simplifying relational database queries. We undertake a review of some of the results and reasoning algorithms that have been developed with a motivation from query evaluation, and add to this a look at open problems in the area as well as a critique of prior work from the point of view of practice.

ACM Reference Format:
Michael Benedikt. 2018. How Can Reasoners Simplify Database Querying (And Why Haven't They Done It Yet)?. In *PODS'18: 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3196959.3196989

1 INTRODUCTION

Codd's work almost half a century ago transformed data management in many ways. The most significant outcome was the development of relational database systems, which came to dominate the market for data management software. But another consequence was an emphasis on the role of declarative methods in databases, and particularly the role of logic, in both textbook treatments of databases and within database research. After Codd's seminal work [65], data management became a leading application area of logic within computer science. The subsequent decades have seen numerous proposals for applications of logic to databases, but one recurring theme has been that logical tools can be used to simplify query processing. Indeed, the ability to apply logic-based simplifications to queries is often mentioned as a key advantage of grounding database languages in logic. The simplification and rewriting tasks considered include:

- **Minimizing queries:** given an SQL query, determine whether it can be simplified by dropping some subquery. This kind of redundancy is unlikely for a query composed manually. But it can arise in queries generated by transformations implementing a query written over an object model on top of a relational database, or shredding a higher-arity relational query into a query over a graph store. Queries requiring extensive simplification can also arise from processing queries with respect to ontologies, an area where there has been increasing activity both in theory [56] and practice [55]. In this setting we are interested in getting not only the traditional answers to a query over a dataset but also the answers that can be inferred from the data using ontological constraints. A standard approach to this problem is to "expand" the query to incorporate the impact of the constraints, but this expansion process can generate large queries that are challenging to evaluate in a DBMS [50, 51].
- **Eliminating operators:** Given a query using some expensive operation, rewrite it to eliminate that operation. For example, if the query uses recursion we can try to rewrite without recursion. As with eliminating redundant subqueries, queries that can benefit from eliminating recursion arise readily from ontological querying, as well as from querying over restricted interfaces.
- **Restricting tables:** Given an SQL query and a collection of tables $\mathcal{T}$, determine whether the query can be rewritten so that it uses only the tables in $\mathcal{T}$. Typically, this rewriting should be equivalent to the original query only for inputs that satisfy certain integrity constraints. For example, if the tables are materialized views, the rewriting need only be equivalent for inputs in which the views are correctly materialized. A variant of this problem looks at a physical plan or web query plan rather than an SQL query: given a query and a set of interfaces, create an equivalent query plan making calls to these interfaces. The problem is well motivated by querying with respect to restricted interfaces, such as web services.

The decades since Codd's work have also seen an explosion of activity in computational logic, with much of it focusing on *reasoning systems*. The rapid maturation of these systems has been dubbed a "logic revolution" [136]. A well-known instance of this progress has been the development of highly-tuned solvers for propositional logic — SAT solvers. But the key algorithmic ideas underlying SAT solving have been extended to provide decision procedures for logics using well-behaved "built-in functions" (e.g. integer addition) within first-order logic [74, 110]. There are also decision procedures acting on formulas over un-interpreted relational vocabularies, particularly for logics working on labelled graphs, such as description logics [17]. Even for logics for which reasoning problems are known to be undecidable, there has been progress on incomplete or non-terminating reasoning procedures. Examples include automated theorem provers for first-order logic [7, 108] and interactive theorem provers for both first-order and higher-order logics [2, 43, 75]. In parallel with the growth of tools and standards, the applications of reasoning tools within computing have multiplied in surprising ways [96].

PODS'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4706-8/18/06…$15.00
https://doi.org/10.1145/3196959.3196989

This seems like a good time to take a step back and consider the question: how does data management, and more specifically the prospect of applying logical reasoning to query evaluation, differ after the logic revolution from its state prior to the revolution? The topic of reasoning with queries is not only still active in foundations of data management, but many strands of current research propose to expand its scope even further. For example, the analysis of queries generated from rewriting with respect to ontologies now spans the data management, the semantic web, and the description logic communities. Research on uncertain data includes a number of attempts to generalize relational reasoning to a probabilistic context (e.g. [78]). It has been proposed that "knowledge-enriched data management" and extending classical reasoning techniques to queries over uncertain data will be major thrusts of research in foundations of data management in the next decade [9].

If there is a need to take on new and larger challenges concerning the incorporation of reasoning into database querying in the future, it is important to understand what has happened in the past, and what is the "state of play" of research in this area today. In this paper, I make an attempt to start reflecting on the state of this research. It will not be a full survey – the topic is just too big. And it will not be a summary of my own work in this area, though of course it is heavily biased towards the topics I know best. The aim is only to:

- discuss some significant results and accomplishments of the research community in the area of applying reasoning to simplify database queries;
- mention some open theoretical problems in this space;
- critique the results with regard to their practical applicability to database systems; and
- spur discussion about how the community should proceed moving forward.

Organization and reader's guide. To make this a bit more manageable, I will focus only on the use of reasoning tools to simplify queries on arbitrary relational schemas, thus omitting enormous amounts of work focused exclusively on RDF data and SPARQL. The article will start with an overview of logical languages, database query and constraint languages, and the relationship between them (Section 2). After highlighting a few common techniques developed by the database community for reasoning and its application to rewriting queries (Section 3), I will present a quick tour of a few well-studied query simplification problems motivated by database issues (Section 4). For each problem I will start by explaining some techniques that have been developed, then give an idea of what has been done using them. Since many of the results will be of interest only to specialists, I have highlighted a few examples and also given a "bottom line" for each topic. The hope is that the take-aways and perhaps the examples would make sense to a wider audience. I also include a discussion of open theoretical issues, before returning to practice with an attempt at critiquing research on each topic, asking how well it has linked back to the applications that motivated it.

The paper is in the spirit of other recent reflection activities in database theory [8, 113]. The idea is to take a step back and look at the development of the field, as an initial step towards deciding what might be changed. It will differ from most prior keynote papers in being lighter on celebration of prior work and heavier on assessment of limitations. And it will differ from prior whitepapers, such as [9], in containing no proposal for a new research program. The article will include some non-technical suggestions about how to make more progress going forward in Section 5, before concluding in Section 6.

2 LOGICAL LANGUAGES AND DATABASE QUERY LANGUAGES

This paper is about the ability to use reasoners for manipulating and analyzing database queries. So I begin with a brief synopsis of logic and database query languages and some comments on how these relate to one another.

A *logic* includes a grammar defining the set of syntactic objects — formulas — and also a semantics saying when a formula holds in some semantic object. For the logics that concern us here, the semantic objects will consist of a *structure* — an interpretation for each constant and relation in the vocabulary of the formula, and also a domain, representing elements that quantifiers should range over — along with a binding for any free variables of the formula. The main computational problems that reasoning tools solve are *satisfiability and validity*. A formula $\phi(\vec{x})$ is *satisfiable* if there is a structure M and binding σ for the free variables $\vec{x}$ of ϕ such that ϕ holds in M when variables are mapped using σ. A formula is *valid* if for every structure M and binding σ for the variables $\vec{x}$, ϕ holds in M with σ. Two formulas $\phi_1(\vec{x})$ and $\phi_2(\vec{x})$ are *equivalent* if ϕ_1 holds for a structure M and binding σ exactly when ϕ_2 holds for M and σ. For logics that are closed under boolean operations, these problems are closely connected. Formulas ϕ_1 and ϕ_2 are equivalent exactly when the formula $\phi_1 \leftrightarrow \phi_2$ is valid, while formula ϕ is valid if and only if $\neg\phi$ is not satisfiable. We get variants of these problems by restricting the witness structures M to be finite: the *finite satisfiability problem*, *finite validity problem*, and *finite equivalence problem*.

The most widely studied "classical" logic is *first-order logic* (FO). In first-order logic quantifiers range only over elements in the structure, as opposed to more complex data structures such as sets or sequences of elements. A fundamental negative result dating to the work of Church, Gödel, and Turing in the 1930's is that reasoning problems are undecidable for first-order logic:

Theorem 2.1. *[64, 133, 134] The satisfiability, validity, and equivalence problems for first-order logic are undecidable. The same is true for the finite variants.*

That is, there can not be an algorithm run on first-order formulas that always terminates and tells whether the formula is satisfiable; and similarly there can be no such algorithm for the other problems. In the second half of the twentieth century, ways to circumvent these undecidability results have emerged. Much of the development is linked to two observations:

- The validity and equivaelnce problems for first-order logic are *semi-decidable*. That is, we can develop procedures that search for a proof witnessing validity, such that if the formula is valid a proof will be found eventually, although we will never be sure when to stop looking.
- The undecidability argument for validity and satisfiability requires complex first-order formulas. This leaves open the

possibility that these problems could be decidable for restricted fragments of first-order logic.

These observations have spurred the creation of tools for solving the satisfiability or validity problem for logics, and these tools are what I refer to in this paper as *reasoners*. The first observation leads to the development of *first-order theorem provers*. For example, several theorem provers are based on the proof system *resolution* [108, 124]; many others are based on *tableau* [4, 138]. The second observation leads to *decision procedures* for specialized logics [110].

Note that finite validity and finite equivalence are not semi-decidable. However finite satisfiability is semi-decidable — one can search for a finite model for a first-order sentence, although one would not know when to terminate the search. Tools and techniques for finite validity have been developed as well [102, 116]. But our focus here will be on query simplification scenarios that require solving equivalence and validity, and there the major tool activity considers validity and equivalence over all structures, not just finite ones. So *throughout the paper, I use the "for all structures" version of each problem as the default.* We will return to the question of finite vs arbitrary structures in Subsection 4.1.

Least fixpoint logic (LFP) extends first-order logic with second-order variables, which are bound by a *fixpoint operator*. Least fixpoint logic can express recursion. For example, it allows one to express that there a path from one element to another in a graph. Since LFP contains first-order logic, the validity problem for LFP is undecidable. Worse, it is not semi-decidable, so constructive proof systems have been developed only for very limited cases, primarily on arity-2 signatures, such as the μ-calculus [12, 109, 129, 130]. As far as I know, there has been little tool development supporting these proof systems.

Database query languages with – and against – logic. Logic has a close connection with database query languages. A way this is often phrased is that first-order logic is "essentially" relational calculus, and that first-order logic is the "relational core of SQL": it represents SQL, ignoring features that are non-relational. But the quotation marks here obscure a host of difficulties.

First, even if we stick to relational features of SQL, pretending for the moment that this means the relational calculus, there are differences with classical first-order logic. To name a few:

- **Domain of quantification.** First-order logic evaluates formulas over a structure, which consists of interpretations of relations along with a *domain*. Quantified variables in first-order logic range over the domain. The database variants, such as relational calculus, are evaluated using interpretations of relations without any domain, and quantifiers range only over constants mentioned in the query and data values stored in relations.
- **Unique name assumption.** First-order logic allows two distinct constants to refer to the same element, while in databases this is commonly forbidden.
- **Safety.** First-order logic allows formulas such as $x = x$ or $R(x) \vee S(y)$ that can be satisfied by infinitely many elements. Database languages only allow expressions that return finitely many answers.

If we look at the goal of applying reasoners to database queries, the differences above are notable but not fatal. They represent ways in which database languages restrict first-order logic. Thus these distinctions are not an obstacle to applying first-order logic reasoners to analyze queries. But in applying logic-based transformations to optimize queries these differences can be problematic: one has to be careful that the transformations preserve the restrictions that database languages impose.

A scarier thing is that there are many features of SQL that are far from first-order logic over relations. To take two of the most well-known:

- **Bag semantics versus set semantics.** SQL semantics assumes by default that query results are bags.
- **Aggregates.** The aggregation features of SQL, like SUM and COUNT, are critical to many applications. These cannot be expressed in first-order logic.

And this is only the tip of the iceberg: there are also NULL values [95, 113] which play a large role in SQL.

How bad are these latter distinctions? If your goal is to *model* SQL features, one can still be hopeful. For example, powerful logics with counting have been developed to model SQL aggregation [97], and they have given us some understanding of the expressive power of SQL. For applying reasoning tools to database languages, these features lead to major problems. The most interesting attempt I am aware of to apply modern reasoners to SQL in its entirety is a very recent one, associated with the tool Cosette [62, 63]. Cosette does not aim at automatically simplifying queries, but at interactively verifying transformations (e.g. the magic sets transformation used in Datalog), making use of a broad interactive theorem proving framework. Even this preliminary attempt at tackling SQL is an exception: the bulk of the work on reasoning with database queries has stuck with the relational core.

One can contrast the situation for SQL with the database language *Datalog*, which has been implemented in academia for decades, with several variants used in industry [3, 6]. Datalog has set semantics, and is a subset of the well-studied formalism LFP mentioned earlier. In fact, Datalog is the fragment of LFP where negation and universal quantification are absent. *Datalog with Stratified Negation* (Datalog$^\neg$) captures a larger subset of LFP. Unfortunately reasoning with variants of Datalog is much more difficult than reasoning with first-order logic. Datalog and Datalog$^\neg$ query evaluation, which is a kind of inference, is not only decidable but well-developed experimentally. But other reasoning problems, such as containment and equivalence, are undecidable even for Datalog.

A schematic picture of the relationship of database query languages to logic-based languages is shown in Figure 1. The solid lines represent containments at the level of expressiveness; the wavy lines represent a correspondence which is not quite an equivalence. Warning and apology: this diagram is super-simplistic! It is only intended to convey one basic message: the tools that computational logicians have built over the past decades apply to a restricted subset of queries considered in databases, and for those queries there are non-trivial discrepancies between the semantics assumed by the tools and the semantics implemented by database engines.

SQL also defines *integrity constraints*, which represent restrictions on input databases, such as keys and foreign keys. There is a corresponding set of logic-based constraint languages in the theory community that shadow these. The most commonly-studied

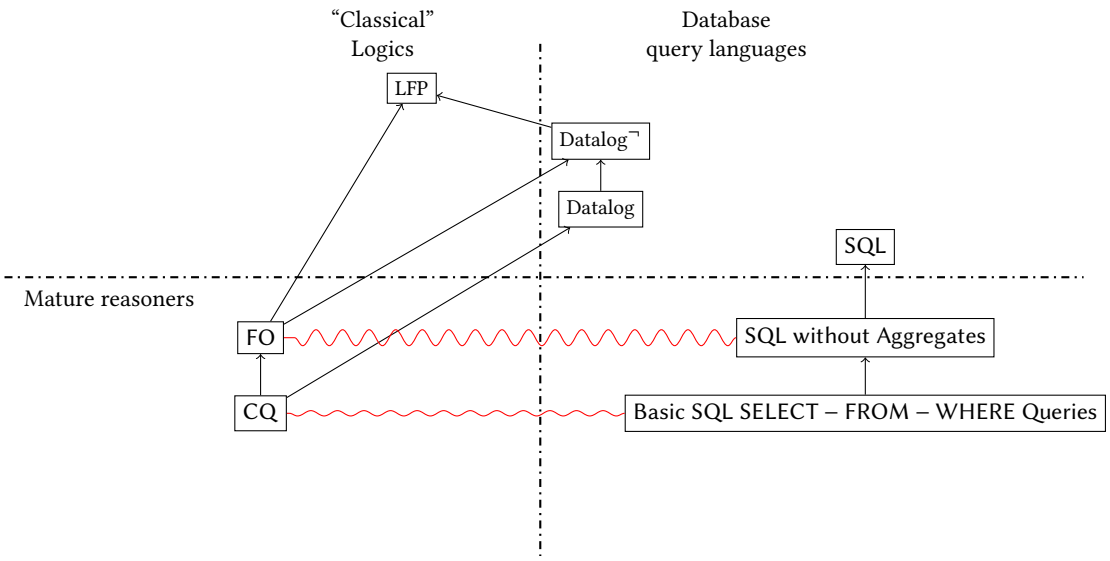


Figure 1: "Cartoon version" of Logic and database languages

classes of logic-based constraints are Tuple-generating Dependencies (TGDs) and Equality-generating Dependencies (EGDs). SQL keys can be seen as special cases of EGDs. SQL foreign keys can be seen as a special case of inclusion dependencies (IDs), which in turn are a special case of TGDs. Functional dependencies (FDs) are a restriction of EGDs and a generalization of keys; reasoning problems for FDs have been studied since the 1970's. As with queries, the equivalence between SQL-based constraint languages and the logic-based ones is only a rough one because of differences in the data model, since issues revolving around NULLs play a significant role in the semantics of SQL constraints.

3 PRELUDE TO THE TOUR: SOME BROAD TECHNIQUES

The jury may be out on how much computational logic has done for simplifying database queries. But in the other direction, it is easy to argue that 40^+ years of research motivated by simplifying queries has produced many ideas of interest to computational logicians. Before beginning a tour of the simplification problems mentioned in the introduction, I would like to highlight a few techniques related to reasoning that have been developed extensively within the database community. Each of them will arise frequently in the later discussion.

To apply reasoning to relational data management, one first needs algorithms for reasoning on arbitrary arity relational structures. In the introduction I mentioned rapid progress on several kinds of reasoning tools. There are SMT solvers, which provide full automation but with a focus on formulas in which every predicate has a fixed meaning – for example, integer addition or inequality. There are general-purpose theorem provers, which handled uninterpreted relations, but give up any guarantee of termination or completeness. And there were special-purposes reasoners focusing on uninterpreted relations, but only on arity-two schemas. None of the research above focuses on fully automated tools for logics having uninterpreted relations of arbitrary arity, and here the database community has contributed important techniques.

Set-at-a-time reasoning. One key contribution consists of techniques for reasoning with large sets of facts, exploiting the capabilities of data management systems. Simulating backward-chaining algorithms such as SLD resolution using forward chaining [28] is one of the most well-known examples, applied heavily in Datalog evaluation. The family of algorithms known as *the chase* [79, 115] allows the application of forward-chaining techniques beyond Datalog, allowing a DBMS to be used for reasoning with TGDs and EGDs over large collections of facts. One application of the chase is to determine whether a query result follows from constraints and incomplete data. The chase extends the data by iteratively adding facts to satisfy the constraints. When the constraints are dependencies with existential quantifiers in the head, the added facts will contain "dummy values" representing witnesses for the quantifiers. After the chase process is finished, one can determine the implied query results using standard query evaluation on the extended database. A second application is to containment of a conjunctive query Q_1 in another conjunctive query Q_2 under constraints Σ, a problem I will discuss in more detail in Subsection 4.1. In using the chase to solve query containment with constraints, instead of adding implied facts to an initial database, the chase can be seen as adding implied atoms to query Q_1. After the process is complete, resulting in a new query Q_1', the query containment problem reduces to evaluating Q_2 on a database formed from Q_1'.

Decidable fragments and automata translation. The chase gives a solution to several database-related reasoning problems, but for constraints that are "cyclic" it cannot be applied straightforwardly, since the process of iteratively adding new facts might not terminate. This has prompted the identification of syntactic conditions that assure termination, with the best known being Weakly Acyclic

(WkAcy) TGDs [79]. But what about classes where a forward-chaining approach like the chase does not terminate? There are now decision procedures for many expressive logics outside the scope of forward-chaining, and many of them make use of the technique of *satisfiability via automata translation*. The idea is that a formula ϕ can be translated to an automaton A_ϕ running over trees that code input structures for ϕ. A tree code is an abstraction of an input structure in which multiple elements are grouped into a single tree node. The tree node representing a group of elements E is given a label that reflects the local structure of E. Adjacent nodes in the tree may represent overlapping groups, and the overlap is also encoded in the node's label. The top of the left pane of Figure 2 exhibits the idea of a tree coding.

In tree codes we fix the number of elements that can be abstracted in a single node. For example, in the left of Figure 2 each node of the tree represents at most 3 elements of the structure. Because of this limit, not every structure can be coded as a tree. Thus the automaton A_ϕ that runs over tree codes is not equivalent to ϕ over all inputs. But it accepts some tree exactly when ϕ is satisfiable by a structure that can be encoded by a tree. This is sufficient for satisfiability testing for certain logics, those that have the "tree-like model property": whenever a formula is satisfied, it is satisfied by a model that has a tree code. The translation taking ϕ to A_ϕ is shown schematically in the bottom of the left pane within Figure 2.

The idea has many sources, including analysis of logic on graphs [69], and decidability results in modal logic. An early seminal work overviewing the use of automata and the tree-like model property for getting decidability results was Vardi's article [135]. The technique was later developed extensively by Grädel and Otto [88, 90, 91, 93]. But much of the inspiration and development of the method has emerged out of database problems. One milestone was the work of Johnson and Klug [104], who showed that the problem of query containment under referential constraints (inclusion dependencies) could be solved by looking at models that could be represented as trees. The formalization of this using automata was made more explicit in the work of Gottlob and his colleagues [114], who extended the containment analysis of Johnson and Klug to wider classes of dependencies. The database community also were early adopters in applying the technique of automata translation in reasoning about fragments of relational calculus [22]. I will give a number of examples of this in Section 4. The method has also been used to analyze Datalog, with the relevance of automata being implicit in Courcelle's [70] and then made explicit in a series of papers on Datalog containment by Chaudhuri and Vardi [60, 61].

Reducing query rewritability to validity. Even if one has devised some reasoning methods, there is the question of how to apply them to simplify query evaluation. There are some straightforward ways to simplify queries using a reasoner, by looking for subqueries that are redundant (see Subsection 4.2). But there is one not-so-obvious method that has been developed primarily in database theory: *reduction of simplification problems to validity checking*. We take the problem of rewriting a query ϕ into a certain restricted form, and we reduce it to a validity problem.

A rewriting problem is specified by a *target* language $\mathcal{T}$, describing some restriction on queries. Perhaps we want queries that only use certain tables, or ones that only use certain relational operators. The reduction technique takes as input a query Q and $\mathcal{T}$, and produces a first-order sentence $\phi^{\mathsf{ReduceVal}}_{Q,\mathcal{T}}$ such that:

There is a query Q' in $\mathcal{T}$ that is equivalent to Q if and only if $\phi^{\mathsf{ReduceVal}}_{Q,\mathcal{T}}$ is valid.

A variation of the problem assumes that we have some integrity constraints Σ restricting our inputs. Our rewriting goal is to obtain a Q' in the target language that is *equivalent to Q modulo* Σ: that is, Q' agrees with Q on all inputs satisfying Σ. A modification of the reduction technique takes as input query Q, target $\mathcal{T}$, and Σ, producing a first-order sentence $\phi^{\mathsf{ReduceVal}}_{Q,\mathcal{T},\Sigma}$ such that:

There is a query Q' in $\mathcal{T}$ that is equivalent to Q modulo Σ if and only if $\phi^{\mathsf{ReduceVal}}_{Q,\mathcal{T},\Sigma}$ is valid.

Further, the rewriting Q' can be effectively reconstructed from a certificate that $\phi^{\mathsf{ReduceVal}}_{Q,\mathcal{T},\Sigma}$ is valid, in the form of a proof of validity. The reduction is represented schematically at the top of Figure 3. The theorem prover is shaded just to emphasize that it is a generic reasoning component. We will not spell out how the construction of a rewriting from a proof works. It relies on *interpolation algorithms for first-order logic*; a detailed explanation can be found in [41, 131]. The interpolation-based reduction originates from the work of the logician Craig [71], but its significance in databases is due to Segoufin and Vianu, who along with Nash [118, 126] connected Craig's work with rewriting problems. Toman and Weddell [131] were the first to isolate the reduction explicitly in the context of reformulating queries over a physical database design, and the later broad formalization presented here is from [41, 42].

4 A TOUR OF PROBLEMS AND RESULTS

So let's begin the tour of problems related to simplifying queries via reasoning. In each stop there will be some comments about the techniques used, and a sample of some results obtained. Since the presentation of results may be cryptic for people outside the area, there is a brief "bottom-line summary" focusing on positive results. After this I will mention some open questions for theorists, closing with a critique of what has been accomplished from the point of view of practice.

4.1 Deciding equivalence and containment

The most obvious place where reasoning systems can play a role is in verifying equivalences. Equivalence testing is useful for verifying that transformations are correct, and equivalence and containment algorithms will be a component in the solution to several of the algorithmic problems described in the introduction. Two queries Q, Q' are said to be *set-equivalent* if for all inputs I, Q and Q' return the same set of tuples. If Q and Q' are formulas in a logic, then this corresponds to the standard notion of logical equivalence. Closely related to query equivalence is *query containment*: Q is *contained in* Q' if for all inputs I, the set of tuples returned by Q on I is a subset of the set of tuples return by Q' on I. Clearly Q and Q' are equivalent exactly when each is contained in the other. As mentioned in the introduction, for richer languages, such as relational algebra, equivalence is semi-decidable but not decidable. But a number of islands of decidabililty have been discovered.

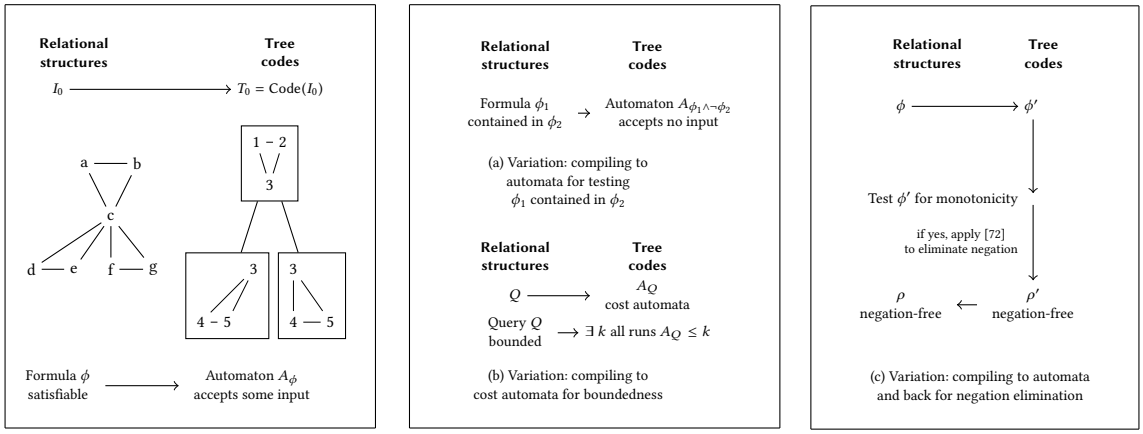
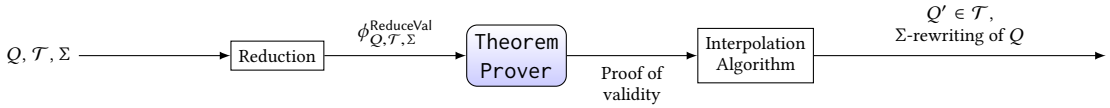
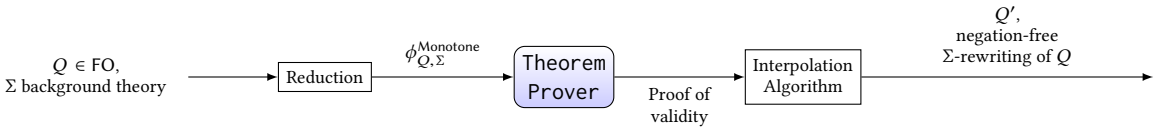


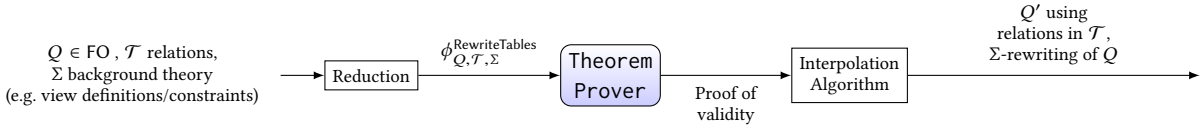
Figure 2: General template for translating logic to automata, and variations



(a) Template for reducing to validity checking



(b) Variation: reducing negation-elimination to validity testing



(c) Variation: reduction of rewriting over tables $\mathcal{T}$ to validity checking

Figure 3: Template for reducing rewriting to validity, and variations

Above I have talked about containment and equivalence of queries over arbitrary data inputs. A related topic is containment and equivalence when restricting to inputs satisfying a set of integrity constraints, which has been heavily studied over the last few decades. When equivalence of queries with respect to constraints has been studied, the queries considered are typically CQs while the constraint languages considered are usually classes of TGDs or EGDs. While the problem of CQ equivalence with respect to constraints is known to be undecidable when constraints are general TGDs [10] or even combinations of IDs and FDs, there are a number of classes of dependencies for which the problem is decidable.

Techniques developed. One basic observation, due to Chandra and Merlin [58] and extended by Sagiv and Yannakakis [122] is that for first-order queries without negation or universal quantification, equivalence is decidable. In fact, for conjunctive queries verifying equivalence can be performed using database techniques.

Later work on equivalence and containment relied on the automata translation technique mentioned in Section 3. For certain restricted languages, counterexamples to containment can always be taken to be "tree-like" and hence can be coded in trees. We can translate each query Q to an automaton A_Q that runs over trees, accepting exactly the codes of tree-like models of Q. By using operations on automata, we can arrive at a single automaton that accepts all tree codes for counterexamples to containment. Checking for containment then reduces to checking that the automaton accepts no trees, and this can be verified using standard automata algorithms. This variation of the automata-theoretic technique to show decidability of containment is shown in the top of the middle pane within Figure 2.

Some example results. *We give an example of an accomplishment in this space, due to Bárány, ten Cate, and Segoufin. The* guarded negation fragment *(*GNF*) is the subset of first-order logic built up from relation symbols via existential quantification (i.e. projection)*

and guarded negation: $R(\vec{x}, \vec{y}) \wedge \neg\phi(\vec{x})$. *In relational algebra terms,* GNF *says that you restrict relational algebra difference so that you can only subtract off tuples from an input table. If you restrict negation this way, containment and equivalence become decidable:*

THEOREM 4.1. *[21] Equivalence and containment of* GNF *sentences is decidable in doubly exponential time.*

We now turn to an example result concerning CQ equivalence with constraints. A frontier-guarded TGD *(*FGTGD*) is an integrity constraint of the form*

$$\forall\vec{x}\,[\phi(\vec{x}) \rightarrow \exists\vec{y}\,\rho]$$

where ϕ *and* ρ *are conjunctions of atoms, and where there is a single atom in* ϕ *containing the variables that occur on both sides of the implication. Roughly speaking, these constraints can only assert something about the set of values within a table, but allowing side-conditions that might span multiple tables. One can think of this as a generalization of standard referential constraints, which only allow you to constrain values in a single source table to appear in a target table. The constraint*

$$\mathsf{Contact}(\mathsf{custid}, \mathsf{address}, \ldots) \wedge \mathsf{Purchased}(\mathsf{custid}, \mathsf{itemid}, \ldots) \rightarrow \mathsf{Customer}(\mathsf{custid}, \mathsf{address}, \ldots)$$

is frontier-guarded, since only information in the Contact *table appears on both sides. The frontier-guarded restriction allows us to get decidable query equivalence:*

THEOREM 4.2. *[18] We can decide whether two CQs are equivalent with respect to a set of* FGTGD*s.*

Additional results. The automata-theoretic technique has also been applied to query equivalence for languages having a form of recursion. While Datalog equivalence is undecidable [127]. Monadic Datalog [60, 60] and Conjunctive 2-way Regular Path Queries (C2RPQs) [57] were two early examples of recursive languages with decidable equivalence. Later work has unified these examples [47] and also provided a generalization that subsumes the known classes of non-recursive queries with decidable containment, such as the guarded negation fragment [32]. A summary of some results on decidable equivalence, along with the corresponding complexity of equivalence, can be found in the left of Figure 4.

When we turn to CQ equivalence with constraints, we get a number of decidable classes defined by acyclicity conditions, such as weak acyclicity. Here decidability is proven using the chase technique mentioned earlier. For other classes, such as the Frontier-guarded TGDs mentioned in Theorem 4.2 above, equivalence with respect to the constraints can be reduced to satisfiability of logics shown on the left of Figure 4. A diagram showing the decidability/undecidability boundary for equivalence of CQs with respect to constraints is given in the right of Figure 4.

The results listed thus far deal only with equivalence under set semantics. For conjunctive queries, one can also look at equivalence under bag semantics. Bag equivalence is important for SQL, in that it gets closer to the ultimate goal of *contextual equivalence*: one query can be replaced by another within any SQL query. For CQs, bag equivalence is known to be decidable [59]. Decidability of containment for CQs under bag semantics is open, but it is known that slight enhancements to the query language, such as moving to UCQs [101] or adding inequalities [103], make the problem undecidable.

Another variation is to consider equivalence over finite query inputs rather than over all inputs. The finite/infinite distinction has been studied mainly under set semantics. For most of the well-known decidable fragments, the two semantics agree. For a few exceptional cases (e.g. the logic GNFP-UP from the left of Figure 4) decidability of equivalence over arbitrary inputs is known but decidability over finite inputs is open. An interesting case where the two semantics disagree, but both are known to be decidable, is the case of CQ equivalence under unary inclusion dependencies and functional dependencies [13, 68]. Here one can decide equivalence over finite inputs via a polynomial reduction to equivalence over unrestricted inputs.

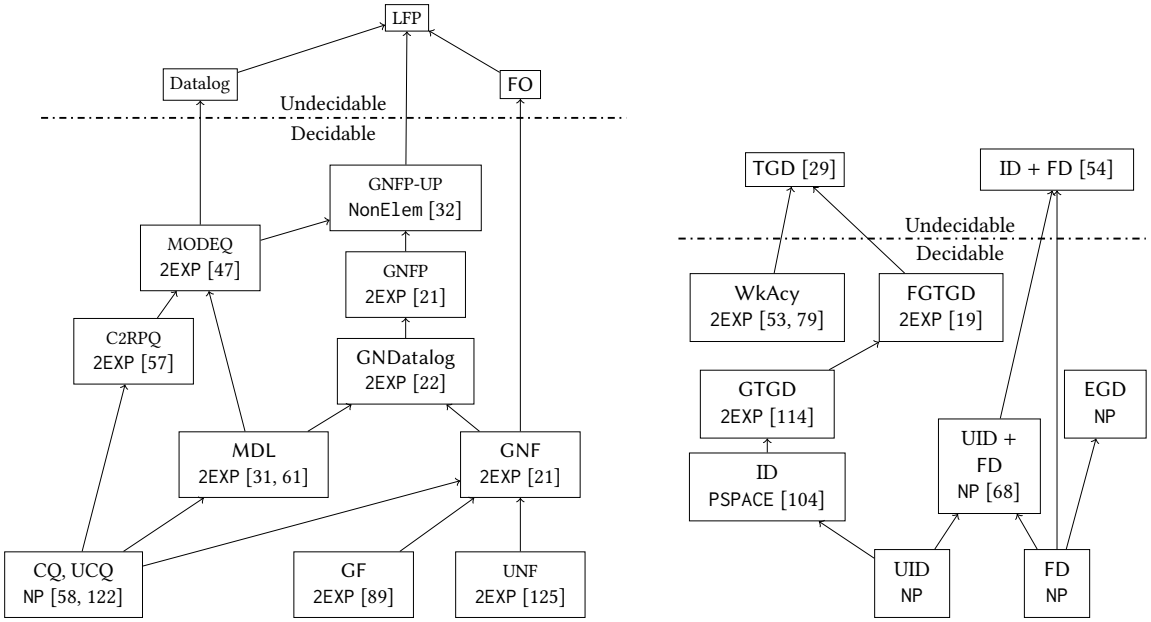
Some positive take-aways. The bottom line for a database person is that query equivalence is decidable for several expressive fragments of relational calculus, assuming set semantics. For example, the result about GNF states roughly that deciding equivalence is doable for queries that only filter out information from single relations, not from joins. For recursive queries, it suffices to be careful about negation and also avoid intensional predicates that maintain joins.

Some open problems. For the richest decidable fragments, such as the language GNFP-UP on the left of Figure 4, equivalence over finite inputs is open. Little is known about decidability of CQ equivalence with respect to constraints containing both EGDs and TGDs, outside of a few very special cases, such as UIDs and FDs.

A broad open question is how to get fragments that are relevant to databases, have good closure properties, and admit containment algorithms that go beyond the automata-theoretic paradigm — that is, fragments where one can not reduce to reasoning over trees. There are other paradigms for decidability of query containment and equivalence in the presence of database integrity languages, for example, based on acyclicity conditions [79], and query-rewriting [52, 56, 114, 119]. Recent work has looked for classes of constraints that can subsume several of these paradigms [14, 30]. However, none of these works has proposed a logic that subsumes all of the prior classes. One interesting starting point is the extension of decidability for Monadic Datalog to Monadic Disjunctive Datalog. This line originates in Feder and Vardi's seminal paper [80] with a streamlined argument and complexity analysis in [48]. The example is intriguing in that Monadic Datalog is canonical example of a language whose decidability can be tackled via the method of automata, while Feder and Vardi's decidability argument for the extension with disjunction uses fundamentally different ideas.

Critique. The isolation of a rich family of logics over arbitrary relational signatures with decidable equivalence is an important achievement of the theoretical computer science community in the last few decades. A significant component of that achievement — perhaps surprisingly — has been motivated by data management applications. But few of these algorithms have been implemented even stand-alone, much less integrated into a broader reasoning system, not to even mention being applied in data management.

In the case of the logics without recursion, the automata-theoretic decision procedures are closely connected to tableau decision procedures for description logics, which have been extensively implemented. For the readers who do not know description logics,



Abbreviations used in the logic diagram	
LFP=Least Fixpoint Logic	GNFP= Guarded Negation Fixpoint Logic
GNFP-UP=Guarded Negation Fixpoint Logic with unrestricted parameters	GNDatalog=Guarded Negation Datalog
MODEQ=Monadically Defined Queries	C2RPQ=Conjunctive Two-way Regular Path Queries
MDL=Monadic Datalog	GNF=Guarded Negation Fragment
GF=Guarded Fragment	UNF=Unary Negation Fragment
NonElem= Non-elementary complexity	2EXP= Double exponential time complexity
Abbreviations used in the constraint language diagram	
TGD=Tuple Generating Dependencies	EGD=Equality Generating Dependencies
FD=Functional Dependencies	WkAcy=Weakly Acyclic TGDs
FGTGD= Frontier-Guarded TGDs	GTGD=Guarded TGDs
ID= Inclusion Dependencies	UID= Unary Inclusion Dependencies

Figure 4: Logics classified by decidability of equivalence (Left) and Constraint languages classified by decidability of equivalence of CQs (Right)

it may be helpful to think of these as the analogous logics to the ones in Figure 4, but only for arity-two schemas. There was a brief interest in extending some of the approaches used for implementation of description logics to the guarded fragment [76, 84, 106]. But other than this, there has been little attention given to equivalence-testing procedures for the logics of Figure 4 from the point of view of implementation.

Let's also reflect a bit on what we have learned about the different variants of equivalence and containment. I listed several results on bag containment and containment over finite structures earlier in this subsection. An obvious question is: what does it all mean? Which variant of equivalence is the appropriate one to test in a given situation? Drawing conclusions about the appropriateness of bag equivalence versus set equivalence is difficult while the main problem in the area, bag containment for CQs, is open. But in addition to not knowing an answer to the question, we lack an understanding of what impact a positive or a negative answer might have on either optimization or verification problems related to databases. What could we do with bag containment that we can not do with bag equivalence or set containment? And if solving bag containment is not a significant next step in understanding logical simplification of richer queries, like SQL aggregates, then would constitute a plausible route to simplifying them?

The results accumulated so far do make a strong case for the superiority of equivalence over arbitrary structures to the variant that considers only finite structures. There are many languages where the unrestricted case is semi-decidable but the finite case is not, with first-order logic being the most notable. And the semantics that most theorem proving tools use is based on arbitrary structures. To the best of my knowledge, there are no languages proposed for use in databases where equivalence over finite inputs is known to be more feasible than equivalence over all inputs. But what about the

loss in precision by moving to unrestricted equivalence? Will we miss important equivalences by restricting to equivalences that hold over arbitrary inputs? As mentioned earlier, in the current cases where we know how to decide equivalence, the answer is no. More generally, to date nobody has come up with compelling examples of what we lose in practice by using unrestricted equivalence, even in the cases where we do not know how to decide either notion. But there has been no experimental investigation into the issue.

4.2 Query minimization

The most obvious query simplification that can be done using a reasoning technique is the removal of formula components that are redundant. A conjunct ϕ_2 in $\phi_1 \wedge \phi_2$ can be considered redundant if $\phi_1 \wedge \phi_2$ is equivalent to ϕ_1, and similarly for a disjunct. Both of these redundancy-elimination analyses reduce to a containment check. For Datalog, where disjunction is implicit in the syntax, redundancy of a disjunct corresponds to the ability to get rid of a rule.

Techniques developed. Minimization generally proceeds by greedily removing a rule or subformula and checking equivalence. Thus whenever we have decidability of containment, we can effectively execute a form of minimization.

How about for cases where containment is undecidable, such as general Datalog? One of the nicest pragmatic steps in this area was the development of *uniform equivalence* by Sagiv [121]. In the context of minimization, the use of uniform equivalence amounts to a simple sufficient condition for dropping a rule $\beta := \alpha$ from a Datalog program P. One creates a database D_α using the facts of α, where variables are changed to constants. If running the pruned program $P - \{\beta := \alpha\}$ on D_α produces a copy of the head β, then the rule can be safely eliminated from the program.

Some example results. *A nice example of the end-to-end use of minimization is in the context of "knowledge-embedded database queries". Consider a data integration setting, where a global schema represents information available to a user, while the local data is available via web services. There are integrity constraints that capture the relationship of local and global data. To answer a query over the global schema, the EMERAC system [105] first creates a Datalog program that represents the answers that can be obtained by accessing services and then reasoning using the constraints and mappings. The system optimizes this program by using minimization under uniform equivalence, as described just above. As one might expect when dealing with a Datalog program that is not hand-written, [105] demonstrates that minimization has a significant impact on pruning the set of rules.*

Some positive take-aways. For queries written in the languages underneath the decidability line within the left of Figure 4, we know how to eliminate redundant conjuncts or disjuncts. We can usually extend this to elimination of subqueries that are redundant when restricting to inputs satisfying integrity constraints, as long as the constraints are in languages below the decidability line on the right of Figure 4.

Some open problems. Minimization under uniform equivalence gives a non-trivial incomplete method for simplifying Datalog. But recursive queries emerging from data integration over web services and other functional interfaces [105] often have redundancy that can not be detected by minimization under uniform equivalence. The last decades have seen significant progress in proof theory for recursion, but focusing on logics over arity two signatures, such as the μ-calculus [12, 129, 130]. An interesting theoretical challenge is to lift this to arbitrary arity, obtaining an incomplete method for detecting redundancy for fragments of Datalog. This could be useful in the undecidable cases where the automata-theoretic method does not apply, and in expressive decidable fragments where the automata-theoretic decision procedures apply in principle but may prove impractical.

Critique. For propositional logic there is a well-developed theory of minimization dating back to the work of Quine and McCluskey in the 1950's [99]. Propositional minimization has rich connections to other areas of theory as well as numerous applications in practice. For propositional Horn formulas (implications of conjunctions – the propositional case of Datalog) the problem is NP-complete, and it is hard even to approximate it [44]. Minimization for more general propositional logic formulas was shown to be hard for the second level of the polynomial hierarchy a few years ago [49]. For predicate logic minimization and database query minimization there is no general theory. Instead formal problem statements and complexity analysis crop up in any number of places – e.g. Datalog minimization, minimal equivalent rewritings. And although a few papers on minimization of queries have connections to the propositional case [36, 123], the relationship of propositional minimization to first-order minimization still needs to be explored in depth.

4.3 Reducing the set of operators

For queries Q in a fixpoint logic, an obvious notion of a structural simplification is to convert to an equivalent Q' that does not use recursion. A fundamental result of Barwise and Moschovakis [26] implies that a Datalog query Q is equivalent to a relational calculus query exactly when it is *bounded*: Q is equivalent to some query Q' formed by unfolding Q a fixed number of times. Boundedness is undecidable for general Datalog queries [83]. But it has been shown decidable for many restricted Datalog fragments.

Instead of — or in addition to — eliminating recursion, one can simplify by eliminating the use of certain logical operators. Most of the results I know of concern eliminating negation, and thus I focus on this below.

Techniques developed. For boundedness, the automata-theoretic technique outlined in Section 3 can be applied, as was the case for equivalence. For the language Monadic Datalog, one can just convert the query into an automaton that accepts tree-like models (see Figure 2) and analyze it for cycles. In the case of the richer decidable fixpoint languages in Figure 4, instead of translating a fixpoint query Q to an automaton that computes a yes/no value from a tree, we can translate it to a *cost tree automaton* A_Q, an automaton that computes a value from a tree, by computing a number along each path and then minimizing across paths. The translation from Q to A_Q has the property that Q is bounded if and only if the range of numbers that can be output by A_Q is finite. The latter property (*limitedness* of a cost automaton) is decidable for the flavor of cost automata emerging from this translation [34]. See the middle of the bottom pane in Figure 2.

For eliminating negation, we can apply the final technique mentioned in Section 3, reduction of rewriting to validity testing. It turns out that a first-order query is equivalent over all inputs to a

formula without negation exactly when the formula is *monotone*: the set of satisfiers grows when the set of facts in the input increases. We can express monotonicity of a query Q by the sentence ϕ_Q^{Monotone} defined as $\forall \vec{x}\ [Q(\vec{x}) \wedge \bigwedge_R (\forall \vec{y}\ R(\vec{y}) \rightarrow R'(\vec{y})) \rightarrow Q'(\vec{x})]$ where Q' is formed by replacing any relations R in Q by R'. The primed and unprimed copies represent two inputs, and the implications $R(\vec{y}) \rightarrow R'(\vec{y})$ state that the data in one input extends the data in the other. Query Q is monotone if and only if the sentence ϕ_Q^{Monotone} is valid. The connection with rewritability is given by the following result (see, e.g., Section 2.3 of [41]):

Proposition 4.3. *For a first-order formula Q, the sentence ϕ_Q^{Monotone} is valid if and only if Q can be rewritten to a query without negation.*

The above result says nothing about how we can get the rewriting, much less about the complexity. In order to obtain the rewriting, one needs a "certificate" of validity, in the form of a proof.

Proposition 4.4. *There is a polynomial time algorithm that takes as input a resolution proof of ϕ_Q^{Monotone} and produces a rewriting of Q without negation. The same holds for a tableau proof.*

A similar transformation holds in the presence of background information (e.g. integrity constraints) Σ. This variation of the technique of reduction to validity is shown in Figure 3b.

Some example results. *The reduction to theorem proving in Proposition 4.4 provides a method to search for a rewriting without negation, one which is applicable to general first-order queries. But since first-order theorem-proving is undecidable, we will never be sure when to terminate the search. We can apply this technique to more specialized query languages where we have decidable validity problems, such as the subsets of first-order logic mentioned in Figure 4. Here is one such result, taken from [41]:*

Proposition 4.5. *If a query Q is in* GNF*, then ϕ_Q^{Monotone} is in* GNF*, and hence we can apply the decision procedure for* GNF *validity mentioned in Subsection 4.1 to determine whether negation can be removed from Q.*

Additional results. For Monadic Datalog, boundedness was shown decidable in [67]. Decidability was extended to Guarded Negation Fixpoint Logic in [22, 45], with the complexity likewise 2EXP [34]. A further extension was to the fixpoint logic GNFP-UP, which subsumes all the previous logics as well as regular path queries [32]. Thus for all the fixpoint logics in Figure 4 where equivalence is decidable, boundedness is also decidable.

Some positive take-aways. We can determine whether recursion can be replaced by iterative inlining of some unfoldings for Datalog queries where intentional predicates do not mix data from multiple tables.

For eliminating negation, a rough bottom line is that —- in theory — we can run a theorem prover to check whether a subquery that has a NOT EXISTS in it can be rewritten to exclude such subqueries. And for many common classes of queries we can know when to give up looking for such a rewriting.

Some open problems. For languages with complex nesting of fixpoints, such as Datalog with stratified negation, the ability to convert a recursive query to a non-recursive query does not agree with boundedness. Determining whether recursion can be eliminated in richer decidable fixpoint logics (e.g. stratified Monadic Datalog or the language GNFP of Figure 4) appears to be connected to a similar question over logics on trees: whether one can effectively determine whether a tree automaton defines a first-order language over trees. This question has been open for decades; see [40, 46] for some partial results.

Removing unnecessary negation is all well and good, but negation is not the only expensive operation. Eliminating disjunction may also be useful, but little is known about this problem. There are semantic characterizations of queries that can be rewritten without disjunction [107]. But I know of no results mapping the decidability line.

Little is known about the question of negation removal for fixpoint logics. Recall that the decidable logics in Figure 4 include several fixpoint logics. One can check whether a formula in these logics is monotone, but it is not known whether a monotone formula can be converted to a negation-free one in the same logic. Some partial results can be obtained by combining the automata-theoretic method, the method of reduction to validity, and prior results for logics over trees. Specifically, it is known that monotone formulas can be converted to negation-free formulas for the μ-calculus, [72]. This allows one to apply the automata-translation method of Section 3, with a new twist that combines it with the method of reduction to validity. The variation is shown in Figure 3b. First, we translate the formula to an automaton, and then from an automaton to a μ-calculus formula Q'. We can determine whether Q' is monotone by forming the sentence $\phi_{Q'}^{\text{Monotone}}$ mentioned earlier, and checking it for validity. The check is performed by converting $\phi_{Q'}^{\text{Monotone}}$ to an automaton and then applying standard automata-theoretic tools. If the formula is monotone, we can apply [72] to remove negation from the formula. We can then *translate back* to a relational fixpoint logic, using a translation technique originating with [92] and explored further in [33]. This will allow us to get a negation-free formula in LFP. What is not known is whether one can perform this process staying in the logic we started with – for example, if we start with Monadic Datalog with stratified negation, can we obtain a Monadic Datalog program?

Critique. For recursive query languages the main "operator elimination" problem studied has been boundedness, which had its heyday in the 90's. With advances in ontology-based rewriting, we have more examples of recursive queries that are in need of simplification. However, some of the more recent research suggests that even when we have the option of replacing a Datalog representation with its unfolding, it might be better not to do so (see e.g. [87]). Knowing a bit more about the kinds of recursive queries we might want to simplify, we should reconsider which problems really matter.

Rewriting to eliminate certain logical operators like negation and recursion is just one example of the kind of simplification analysis where the literature has produced difficult and intriguing results. In the same spirit there has been progress on a number of other problems concerning transforming a query to a more restricted form: for example, converting a CQ to an acyclic one in the presence of constraints [23, 24, 81]. All of these efforts suffer from the same weakness. The most compelling application of these algorithms is

in the presence of really rich constraints, which make it more likely that a query could undergo dramatic simplification. However, in the presence of such constraints the complexity of the rewriting algorithms is formidable. Further, in these cases it may be necessary to blow up the query significantly in order to get into the target language, thus negating any gain in efficiency.

4.4 Changing the set of tables

Another way of making a formula simpler is the elimination of "bad" relations in favor of "good" ones. Given a formula Q and a distinguished subset of the relations $\mathcal{T} = T_1 \ldots T_n$, we want to rewrite Q to an equivalent $Q_{\mathcal{T}}$ using only relations in $\mathcal{T}$.

Techniques developed. The technique of reduction to validity can be applied here, using the variation shown in Figure 3c. The ability to restrict to tables in $\mathcal{T}$ turns out to be equivalent to Q being *determined* by $\mathcal{T}$: its output depends only on the contents of the tables in $\mathcal{T}$. Just by formalizing determinacy in logical terms, we see that a query Q is determined by $\mathcal{T}$ exactly when the sentence $\phi^{\text{RestrictTables}}_{Q,\mathcal{T}}$ is valid, where $\phi^{\text{RestrictTables}}_{Q,\mathcal{T}}$ is:

$$\forall \vec{x}\, [Q(\vec{x}) \wedge \bigwedge_{T_i \in \mathcal{T}} (\forall \vec{y}\, T_i(\vec{y}) \leftrightarrow T_i'(\vec{y})) \rightarrow Q'(\vec{x})]$$

where Q' is formed from Q by replacing all relations R by a copy R'. The following result is implicit in Segoufin and Vianu's work [126]:

Proposition 4.6. *For a first-order logic formula Q, $\phi^{\text{RestrictTables}}_{Q,\mathcal{T}}$ is valid if and only if Q can be rewritten to a first-order query using only relations in $\mathcal{T}$.*

Again, the rewriting can be extracted from a certificate (see, for example [36]):

Proposition 4.7. *There is a polynomial time algorithm that given a tableau proof of $\phi^{\text{RestrictTables}}_{Q,\mathcal{T}}$, produces a rewriting of Q using relations in $\mathcal{T}$. The same holds for a resolution proof.*

This transformation can be modified for a set of integrity constraints Σ. We can produce sentence $\phi^{\text{RestrictTables}}_{Q,\mathcal{T},\Sigma}$ defined as

$$\forall \vec{x}\, [\Sigma \wedge \Sigma' \wedge Q(\vec{x}) \wedge \bigwedge_{T_i \in \mathcal{T}} (\forall \vec{y}\, T_i(\vec{y}) \leftrightarrow T_i'(\vec{y})) \rightarrow Q'(\vec{x})]$$

where Σ' is formed from Σ by changing each relation to its primed copy, and similarly for Q'.

Proposition 4.8. *For any first-order logic formula Q, the sentence $\phi^{\text{RestrictTables}}_{Q,\mathcal{T},\Sigma}$ is valid if and only if there is a query $Q_{\mathcal{T}}$ using only relations in $\mathcal{T}$ that is equivalent to Q on all inputs satisfying Σ.*

A case of particular interest is where the constraints Σ consist of *view definitions for each T_i*: sentences defining each $T_i \in \mathcal{T}$ as a query over relations not in $\mathcal{T}$. Then specializing the results above we get a reduction of view-rewriting to validity checking:

Proposition 4.9. *Given views $\mathcal{V}$ associated with view definitions in relational calculus and a first-order logic formula Q we can form a sentence $\phi^{\text{RestrictViews}}_{Q,\mathcal{V}}$ which is valid if and only if there is a query $Q_{\mathcal{V}}$ equivalent to Q using only the views in $\mathcal{V}$.*

Some example results. *We can make use of the reductions in Proposition 4.6, 4.7, and 4.8 for each of the main decidable languages in Figure 4. Using these we can determine whether a query can be rewritten to use a specific set of tables, where equivalence could be relative to a set of nicely-behaved integrity constraints.*

To apply this to views, we need a class of views for which the corresponding constraints expressing the view definitions are in a nice class. Here is one interesting case:

Theorem 4.10. *[20] If we have a set of CQ views where all free variables sit in a single query atom, then we can determine whether a conjunctive query can be rewritten with respect to the views.*

Additional results. Proposition 4.9 provides a reduction of view rewriting to theorem proving. Unfortunately, even in the case where the views $\mathcal{V}$ and the queries Q are given by CQs, the corresponding formula $\phi^{\text{RestrictViews}}_{Q,\mathcal{V}}$ is not in a decidable fragment of first-order logic. In fact, a recent fundamental result places a sharp limit on what one can do in terms of solving rewriting problems automatically. Gogacz and Marcinkowski [86] showed that we can not decide whether a CQ is first-order rewritable with respect to a set of CQ views. In contrast, if Q and $\mathcal{V}$ are CQs and we further require that the rewriting is monotone, then $\phi^{\text{RestrictViews}}_{Q,\mathcal{V}}$ lies in a decidable fragment. This corresponds to CQ-rewritability of Q with respect to the views, proven decidable by Levy, Mendelzon, Sagiv, and Srivastava [112].

Some open problems. As with negation removal, the question of deciding rewriting using only a given set of tables or views for queries in the main standard decidable fixpoint logics, such as Monadic Datalog, is open to the best of my knowledge. Some partial results for regular path queries can be found in [82].

Another issue for this transformation concerns query safety, one of the key distinctions between "database logic" and "classical logic" mentioned in Section 2. The transformation that takes a proof of validity for $\phi^{\text{RestrictTables}}_{Q,\mathcal{T},\Sigma}$ and produces a rewriting Q' works fine when Q is a boolean query, since no safety issues crop up. It also produces safe rewritings when Σ is a set of TGDs or TGDs with disjunction, and thus in particular when Σ defines each table in $\mathcal{T}$ using a CQ or a UCQ. For more general first-order logic sentences as Σ, applying this transformation can produce an unsafe query. What is needed are further conditions on the proof of validity that guarantee that the corresponding formula produced by interpolation is safe.

The algorithm described in Proposition 4.9 gives a semi-decision procedure for determining FO-rewritability of a query with respect to views. For general CQ views, one cannot expect more. But our understanding of the exact decidability line is in a very primitive state. The results of Gogacz and Marcinkowski mentioned above show that the problem is undecidable for general CQ views, and yet it is very difficult to identify the "hard cases": the family of examples in [86] takes pages to describe. Given this, it is surprising that so few decidable subcases have been identified to date.

Critique. An instance of rewriting to restrict tables is view-based rewriting, which is perhaps the only algorithmic problem discussed in this section where there are substantial implementations, including in commercial systems.

The more general approach outlined above, via calls to a reasoner, has an advantage in scope. It can be applied to arbitrary first-order queries and views, given a call to a general-purpose reasoner. And it can be applied with termination guarantees for restricted queries and views (those for which the validity problem for $\phi^{\text{RestrictTables}}_{Q,\mathcal{V}}$ is decidable). But the case for general first-order views is problematic in practice. To mention just one difficulty, commercial systems have little support for views beyond CQs.

Aggregate views are the focus of many data warehousing tools. The problem of rewriting aggregate queries with respect to views has received some attention from the research community, including in-depth complexity analyses as well as special-purpose algorithms [11, 66]. But it is not clear what role broader reasoning tools can play there. Going back to the discussion prior to Figure 1, recall that I had mentioned the lack of tools that support reasoning for aggregate logics. In fact, there are few logics with aggregation over arbitrary arity relational schemes where we have even obtained decidability of containment in theory.

4.5 Rewriting exploration

Query minimization, discussed in Subsection 4.2 was a "local" query simplification technique that could make use of a reasoner. There is a more global approach to query simplification that can make use of equivalence and containment checks, as well as the rewriting techniques of the previous two subsections. It consists of *exploring equivalent queries*: one enumerates queries Q^* equivalent to Q, perhaps focusing on queries with special properties, such as a distinguished set of operators or tables. The goal is to find a query that is better than Q in some respect. Of course, there is vast research on how to do this for join queries in the absent of constraints, and also for variants such as outer-joins and group-joins [117]. But reasoning-based techniques give an opportunity to do it for relational calculus queries, even in the presence of rich constraints. This enumeration could be done on top of another query optimizer, or could plug into a query optimization framework that can generate requests for such queries. Interfaces that include such requests are components of a number of optimizers [94, 128], although in traditional frameworks this exploration is done via rewrite rules. The advantage, in principle, for using a more general technique for exploring logically equivalent queries is that one need not restrict to applying algebraic rules that just transform a query locally (e.g. commuting conjunctions).

Techniques developed. One obvious method to explore the space of equivalent queries is "generate-and-check". We enumerate queries Q^* that are *consequences* of Q and then check each generated Q^* to see if it is in fact equivalent to Q. The most well-known instance of generate-and-check is the *chase and backchase* (C&B) developed by Tannen, Popa, and Deutsch [77, 120], which applies in the setting of specialized constraints: dependencies with terminating chase. The C&B generates consequences of Q and then enumerates conjunctions of consequences satisfying certain additional conditions: e.g. if our goal is to rewrite with respect to views, we will conjoin only consequences that use view predicates. The generation of consequences uses the chase algorithm mentioned in Section 3. Given a set of consequences A of the appropriate form, we check A for equivalence with Q, which again involves generating consequences of A using the chase algorithm. For dependencies where the chase process terminates, the consequence-generation process can be instrumented to allow the checking stage to be heavily optimized. For example, one can exploit the sharing between the computation done in checking different sets of consequences, and also the sharing between the computation done in the checking stage and that performed in generating consequences [100].

For enumerating equivalent rewritings for more general queries, or for enumerating rewritings CQ queries in the presence of constraints that are more general than TGDs, an alternative is to apply one of the reductions from rewriting to validity that were mentioned earlier in this section. For example, we can choose a set of target tables $\mathcal{T}$ and enumerate formulas $Q_{\mathcal{T}}$ over $\mathcal{T}$ that are equivalent to Q modulo constraints Σ, by enumerating proofs of the sentence $\phi^{\text{RestrictTables}}_{Q,\mathcal{T},\Sigma}$ discussed in Subsection 4.4. If we are interested in exploring negation-free queries $Q_{\mathcal{T}}$, we can get a corresponding sentence whose proofs generate negation-free rewritings.

Some example results. *The C&B is developed in a number of papers, although the closet thing I know to an available implementation of it is the PDQ system [5], which deals with cost-based exploration of rewritings over access patterns.*

The more general enumeration approach via reduction to validity is discussed in [41, 42, 131]. Implementations of the approach have been developed by Toman and Weddell [98, 132] as well as [36].

Some positive take-aways. Given a query we can enumerate rewritings that are equivalent in the presence of background knowledge, even when the queries are complex (e.g with negation) and even when the background knowledge is in a rich language. We can impose additional constraints on the kinds of rewritings we want to explore – the tables used or the logical operators allowed.

Some open problems. Traditional algebraic simplification focuses on transformations that are thought to be "clear wins" – for example, pushing selections or projections through joins. The main advantage of more general simplification techniques is that they can find equivalent queries which differ radically in structure from the input query. But this is also a major disadvantage, since one has no guarantee that the rewritten query is an improvement over the original. While any query optimization technique requires cost estimation as a component, these more "global" transformations put more of a burden on it. Surprisingly, the problem of cost and size estimation has not received much formal treatment from the database theory community. Consider one of the most basic problems: estimating the worst-case size of a conjunctive query, given upper bound information on the constituent relations. The complexity of this problem (e.g. for a fixed query) is, to the best of my knowledge, open, although Atserias, Grohe, and Marx [16] provides a lower bound on the combined complexity.

Critique. Exploration frameworks based on the reduction to validity checking can be considered as a *synthesis technology*, which starts with a query that has no obvious implementation at all, and finds a rewriting that is implementable. They can also be seen as an *optimization technology* that starts with a query that is implementable, and looks for an equivalent one that performs better. These two modes correspond roughly to the two variations of the results I have mentioned using the method of reduction-to-validity: for example, compare Proposition 4.3 and Proposition 4.4.

For an example of synthesis, the PDQ system of [37–39] takes as input a query written on top of an integrated schema, connected to web services via constraints. It will decide whether there is a plan that implements the query. A limitation of this technique in a data integration setting is that the notion of "exact reformulation" that it demands is extremely strong. Systems based on the more common certain answer semantics (e.g. as in LAV data integration [111]) arguably lead to more natural specification of mappings.

Let me turn to the use of exploration based on the reduction to validity as an optimization technology. The main limitation here is that it requires exploring many proofs of the formula produced by the reduction. That is, we must consider many proofs of $\phi^{\text{RestrictTables}}_{Q,\mathcal{T},\Sigma}$ or its analogs with access patterns and/or monotonicity requirements. Thus for optimization we do not have a reduction to a validity check; rather we need a *proof iterator API* that can enumerate proofs of validity of a sentence. In fact, it can be shown, assuming standard complexity hypotheses, that finding good rewritings is strictly harder than validity testing [36]. As a practical matter, theorem provers do not offer these APIs, and it is non-trivial to adapt implementations of standard tools to provide them. In the case where the constraints are TGDs with terminating chase, the space of possible proofs can be compactly represented and instrumented, which facilitates easier enumeration of possible rewritings; this is exploited in the C&B [77, 100, 120]. Beyond this setting, efficiently navigating the proof space is a daunting challenge.

5 SOME COMMON SIGHTINGS ON THE TOUR

The main message of the tour of prior work is: some hard problems have been solved, interesting and sophisticated techniques have been developed. But in terms of our initial question of what reasoners can do for database queries, the best answer would probably be: we don't know. I warned in the introduction that this paper would not end with a bold program. There will be no attempt to answer any of the questions I have raised in the critiques of earlier work given in the previous section. Instead I will wrap up by proposing some non-technical lessons that we can take away from the tour.

The starting point for this work was the "logic revolution" mentioned in the introduction. Recall that this referred to progress in computational logic on several fronts, each going far beyond foundational studies. Researchers have been building platforms that experimentalists can tinker with, developing standards (e.g. for syntax), and constructing a wide range of benchmarks and experimental methodologies. SMT solvers do it, [25, 73], interactive theorem provers do it [2, 75]; even modal logicians do it [1].

Reasoning for query evaluation is markedly different in all of these respects. Groups working on logic and data management have often built experimental prototypes, but these have not become platforms. Recently there has been the slightest sign of progress: tools for chasing [85], reformulation [5] and verifying equivalence [62] are currently available. What is lacking is a broad software platform for experimenting with logic and databases – not just checking for equivalence, but also trying out logical simplifications and seeing how they impact runtime in a DBMS.

For most of the constraint languages and all of the logic-based query languages mentioned earlier, there is no standard syntax. This is true even for the most basic and widely-studied ones, such as Datalog. There have been a few benchmarking activities around dependencies [15, 35], while [35] looked at some rudimentary methodological questions around benchmarking in this area, such as what exactly one would want to measure. But these efforts only deal with a few reasoning-related problems. Further they rely on synthetic data, synthetic integrity constraints, and synthetic queries. And unlike the synthetic generators used in database systems, the generation techniques have no basis in any intuition or experience concerning what we can expect in practice or what the current state of tools can handle — because currently we do not have any intuition and we do not have any experience. The situation in logic-based research for the semantic web should be both a sobering comparison and also an inspiration. The standardization of the ontology language OWL [27] had a dramatic impact on research in description logics. The existence of these standards spurred the creation of benchmarks that feature real constraints, realistic queries, and even real datasets [137]. An often-repeated refrain in many fields is "we need (more, better) standards/benchmarks". But in the case of reasoning with database queries, the lack of activity is at another level of urgency, and is perhaps symptomatic of the lack of consensus concerning the goals of the research program.

It is also worth noting the way other computational logic communities deal with the gap between the "pure" logic-based formalisms that exactly match the theory and the languages used in practice. The distinction between real database languages and the logics that we currently know how to analyze is really quite a chasm: the spacing in the crude diagram of Figure 1 is intended to highlight the distance between the two. And even restricting attention to the logic side of the diagram, our tour identified a number of disconnects between the solutions offered and the problems that motivated them. Gaps between academic work and reality always exist. But in other areas the understanding of these gaps, and the effort expended in bridging them, has increased proportionally with the growth of the underlying theory. Hopefully reasoning with query languages will move towards a better balance in this respect.

6 CONCLUSION

Reasoning in databases is in a curious state. We have a multitude of interesting results – the references of this paper contain only a small fraction! Some of them are undeservedly obscure, as the community has moved on to other topics and left them behind. Many of them are well-known, but few researchers could say what the consequences of them might be for data management — or for anything else.

The only conclusion of this paper is a modest one. The rapid progress of reasoning systems outside of data management is a great opportunity to revisit these topics and reflect: consolidate what we know, acknowledge what we don't know or haven't explored yet, look at what these other reasoning communities did right which we can emulate, and begin a community effort to address the gaps. We should also look at what these other communities could use from our trove of prior results and techniques.

Throughout the history of this topic, papers have deferred much of this effort — particularly on the practical side — "for future work". Hopefully the future will arrive shortly.

Acknowledgements. I want to express my deep gratitude to Marcelo Arenas, Michaël Cadilhac, Phokion Kolaitis, Dan Olteanu, and Victor Vianu for extensive feedback on this work. The research was supported by EPSRC grants EP/M005852/1, EP/N014359/1, and EP/L012138/1.

REFERENCES

[1] COOL- the Coalgebraic Ontology Logic Reasoner. cal8.cs.fau.de/redmine/projects/cool.
[2] Coq theorem prover. coq.inria.fr.
[3] DLVSYSTEM. www.dlvsystem.com.
[4] The leanCoP theorem prover. www.leancop.de.
[5] PDQ. Available at www.cs.ox.ac.uk/projects/pdq/home.html.
[6] Semmle. semmle.com.
[7] TPTP. www.cs.miami.edu/~tptp.
[8] PODS reflections workshop, 2013. www.sigmod.org/2013/pods_reflections.shtml.
[9] S. Abiteboul, M. Arenas, P. Barceló, M. Bienvenu, D. Calvanese, C. David, R. Hull, E. Hüllermeier, B. Kimelfeld, L. Libkin, W. Martens, T. Milo, F. Murlak, F. Neven, M. Ortiz, T. Schwentick, J. Stoyanovich, J. Su, D. Suciu, V. Vianu, and K. Yi. Research directions for principles of data management. *SIGMOD Record*, 45(4):5–17, 2016.
[10] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[11] F. N. Afrati and R. Chirkova. Selecting and using views to compute aggregate queries. *JCSS*, 77(6):1079–1107, 2011.
[12] B. Afshari and G. E. Leigh. Cut-free completeness for modal mu-calculus. In *LICS*, 2017.
[13] A. Amarilli and M. Benedikt. Finite open-world query answering with number restrictions. In *LICS*, 2015.
[14] M. Arenas, G. Gottlob, and A. Pieris. Expressive languages for querying the semantic web. In *PODS*, 2014.
[15] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The iBench integration metadata generator. In *VLDB*, 2015.
[16] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comp.*, 42(4), 2013.
[17] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
[18] J.-F. Baget, M.-L. Mugnier, S. Rudolph, and M. Thomazo. Walking the complexity lines for generalized guarded existential rules. In *IJCAI*, 2011.
[19] J.-F. Baget, M.-L. Mugnier, S. Rudolph, and M. Thomazo. Walking the complexity lines for generalized guarded existential rules. In *IJCAI*, 2011.
[20] V. Bárány, M. Benedikt, and B. ten Cate. Rewriting guarded negation queries. In *MFCS*, 2013.
[21] V. Bárány, B. t. Cate, and L. Segoufin. Guarded negation. *JACM*, 62(3), 2015.
[22] V. Bárány, B. ten Cate, and M. Otto. Queries with guarded negation. In *VLDB*, 2012.
[23] P. Barceló, G. Gottlob, and A. Pieris. Semantic acyclicity under constraints. In *PODS*, 2016.
[24] P. Barceló, A. Pieris, and M. Romero. Semantic optimization in tractable classes of conjunctive queries. *SIGMOD Record*, 46(2):5–17, Sept. 2017.
[25] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.
[26] J. Barwise and Y. Moschovakis. Global inductive definability. *Journal of Symbolic Logic*, 43(3):521–534, 1978.
[27] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language reference. W3C Recommendation, 2004. Available at http://www.w3.org/TR/owl-ref/.
[28] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3&4):255–299, 1991.
[29] C. Beeri and M. Y. Vardi. The implication problem for data dependencies. In *ICALP*, 1981.
[30] L. Bellomarini, G. Gottlob, A. Pieris, and E. Sallinger. Swift logic for big data and knowledge graphs. In *IJCAI*, 2017.
[31] M. Benedikt, P. Bourhis, and P. Senellart. Monadic Datalog containment. In *ICALP*, 2012.
[32] M. Benedikt, P. Bourhis, and M. Vanden Boom. A step up in expressiveness of decidable fixpoint logics. In *LICS*, 2016.
[33] M. Benedikt, P. Bourhis, and M. Vanden Boom. Characterizing definability in decidable fixpoint logics. In *ICALP*, 2017.
[34] M. Benedikt, T. Colcombet, B. ten Cate, and M. Vanden Boom. The complexity of boundedness for guarded logics. In *LICS*, 2015.
[35] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, 2017.
[36] M. Benedikt, E. V. Kostylev, F. Mogavero, and E. Tsamoura. Reformulating queries: Theory and practice. In *IJCAI*, 2017.
[37] M. Benedikt, J. Leblay, and E. Tsamoura. PDQ: Proof-driven query answering over web-based data. In *VLDB*, 2014.
[38] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. In *VLDB*, 2015.
[39] M. Benedikt, R. Lopez-Serrano, and E. Tsamoura. Biological web services: Integration, optimization, and reasoning. In *IJCAI-BAI*, 2016.
[40] M. Benedikt and L. Segoufin. Regular tree languages definable in FO and in FO_{mod}. *TOCL*, 11(1):4:1–4:32, 2009.
[41] M. Benedikt, B. ten Cate, J. Leblay, and E. Tsamoura. *Generating Plans from Proofs: the Interpolation-based Approach to Query Reformulation*. Morgan Claypool, 2016.
[42] M. Benedikt, B. ten Cate, and E. Tsamoura. Generating plans from proofs. In *TODS*, 2016.
[43] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer, 1st edition, 2010.
[44] A. Bhattacharya, B. DasGupta, D. Mubayi, and G. Turán. On approximate horn formula minimization. In *ICALP*, 2010.
[45] A. Blumensath, M. Otto, and M. Weyer. Decidability results for the boundedness problem. *Logical Methods in Computer Science*, 10(3), 2014.
[46] M. Bojańczyk and L. Segoufin. Tree languages defined in first-order logic with one quantifier alternation. *Logical Methods in Computer Science*, 6(4), 2010.
[47] P. Bourhis, M. Krötzsch, and S. Rudolph. Reasonable highly expressive query languages. In *IJCAI*, 2015.
[48] P. Bourhis and C. Lutz. Containment in Monadic Disjunctive Datalog, MMSNP, and expressive description logics. In *KR*, 2016.
[49] D. Buchfuhrer and C. Umans. The complexity of boolean formula minimization. *JCSS*, 77(1):142–153, 2011.
[50] D. Bursztyn, F. Goasdoué, and I. Manolescu. Reformulation-based query answering in RDF: alternatives and performance. In *VLDB*, 2015.
[51] D. Bursztyn, F. Goasdoué, and I. Manolescu. Teaching an RDBMS about ontological constraints. In *VLDB*, 2016.
[52] A. Calì, G. Gottlob, and A. Pieris. Advanced processing for ontological queries. In *VLDB*, 2010.
[53] A. Calì, G. Gottlob, and A. Pieris. Query answering under non-guarded rules in Datalog+/-. In *Web Reasoning and Rule Systems*, 2010.
[54] A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS*, 2003.
[55] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.
[56] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.
[57] D. Calvanese, G. De Giacomo, and M. Y. Vardi. Decidable containment of recursive queries. *Theoretical Computer Science*, 336(1):33–56, May 2005.
[58] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.
[59] S. Chaudhuri and M. Y. Vardi. Optimization of *Real* conjunctive queries. In *PODS*, 1993.
[60] S. Chaudhuri and M. Y. Vardi. On the complexity of equivalence between recursive and nonrecursive Datalog programs. In *PODS*, 1994.
[61] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive Datalog programs. *JCSS*, 54(1):61–78, 1997.
[62] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.
[63] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*, 2017.
[64] A. Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 03 1936.
[65] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.
[66] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS*, 1999.
[67] S. S. Cosmadakis, H. Gaifman, P. C. Kanellakis, and M. Y. Vardi. Decidable optimization problems for database logic programs. In *STOC*, 1988.
[68] S. S. Cosmadakis, P. C. Kanellakis, and M. Y. Vardi. Polynomial-time implication problems for unary inclusion dependencies. *JACM*, 37(1), 1990.
[69] B. Courcelle. Graph Rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, chapter 5, pages 193–242. 1990.
[70] B. Courcelle. Recursive queries and context-free graph grammars. *Theoretical Computer Science*, 78(1):217 – 244, 1991.
[71] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.
[72] G. D'Agostino and M. Hollenberg. Logical Questions Concerning The mu-Calculus: Interpolation, Lyndon and Los-Tarski. *Journal of Symbolic Logic*,

65(1):310–332, 2000.
[73] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
[74] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and applications. *CACM*, 54(9):69–77, Sept. 2011.
[75] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In *CADE*, 2015.
[76] H. de Nivelle. A resolution decision procedure for the guarded fragment. In *CADE*, 1998.
[77] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.
[78] P. M. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.
[79] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.
[80] T. Feder and M. Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through Datalog and group theory. *SIAM Journal of Computing*, 28(1):57–104, 1998.
[81] D. Figueira. Semantically acyclic conjunctive queries under functional dependencies. In *LICS*, 2016.
[82] N. Francis, L. Segoufin, and C. Sirangelo. Datalog rewritings of regular path queries using views. *Logical Methods in Computer Science*, 11(4), 2015.
[83] H. Gaifman, H. G. Mairson, Y. Sagiv, and M. Y. Vardi. Undecidable optimization problems for database logic programs. *J. ACM*, 40(3):683–713, 1993.
[84] H. Ganzinger and H. de Nivelle. A superposition decision procedure for the guarded fragment with equality. In *LICS*, 1999.
[85] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's All Folks! LLUNATIC Goes Open Source. In *VLDB*, 2014.
[86] T. Gogacz and J. Marcinkowski. The hunt for a red spider: Conjunctive query determinacy is undecidable. In *LICS*, 2015.
[87] G. Gottlob and T. Schwentick. Rewriting ontological queries into small nonrecursive datalog programs. In *KR*, 2012.
[88] E. Grädel. Decision procedures for guarded logics. In *CADE*, 1999.
[89] E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64(4):1719–1742, 1999.
[90] E. Grädel. Why is modal logic so robustly decidable?, 1999.
[91] E. Grädel. Guarded fixed point logics and the monadic theory of countable trees. *Theoretical Computer Science*, 288(1):129 – 152, 2002.
[92] E. Grädel, C. Hirsch, and M. Otto. Back and forth between guarded and modal logics. *TOCL*, 3(3):418–463, 2002.
[93] E. Grädel and M. Otto. The freedoms of (guarded) bisimulation. In *Johan van Benthem on Logic and Information Dynamics*, pages 3–31. 2014.
[94] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.
[95] P. Guagliardo and L. Libkin. Correctness of SQL queries on databases with nulls. *SIGMOD Record*, 46(3):5–16, 2017.
[96] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.
[97] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *JACM*, 48(4):880–907, 2001.
[98] A. K. Hudek, D. Toman, and G. E. Weddell. On enumerating query plans using analytic tableau. In *TABLEAUX*, 2015.
[99] A. Ignatiev, A. Previti, and J. Marques-Silva. SAT-based formula simplification. In *SAT*, 2015.
[100] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, 2014.
[101] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *TODS*, 20(3):288–324, 1995.
[102] D. Jackson. Automating first-order relational logic. In *SIGSOFT/FSE*, 2000.
[103] T. S. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for REAL conjunctive queries with inequalities. In *PODS*, 2006.
[104] D. S. Johnson and A. C. Klug. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *JCSS*, 28(1):167–189, 1984.
[105] S. Kambhampati, E. Lambrecht, U. Nambiar, Z. Nie, and G. Senthil. Optimizing recursive information gathering plans in EMERAC. *Journal of Intelligent Information Systems*, 22(2):119–153, 2004.
[106] Y. Kazakov and H. de Nivelle. A resolution decision procedure for the guarded fragment with transitive guards. In *IJCAR*, 2004.
[107] H. J. Keisler. Reduced products and Horn classes. *Transactions of the American Mathematical Society*, 117:307–328, 1965.
[108] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *CAV*, 2013.
[109] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
[110] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 1 edition, 2008.
[111] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.
[112] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.
[113] L. Libkin. Incomplete data: What went wrong, and how to fix it. In *PODS*, 2014.
[114] T. Lukasiewicz, A. Calì, and G. Gottlob. A general Datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics*, 14(0):57–83, 2012.
[115] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *TODS*, 4(4):455–469, 1979.
[116] W. McCune. A davis-putnam program and its application to finite-order model search: Quasigroup existence problems. Technical report, 9 1994.
[117] G. Moerkotte, P. Fender, and M. Eich. On the correct and complete enumeration of the core search space. In *SIGMOD*, 2013.
[118] A. Nash, L. Segoufin, and V. Vianu. Views and queries: Determinacy and rewriting. *TODS*, 35(3), 2010.
[119] A. Pieris. *Ontological query answering : new languages, algorithms and complexity*. PhD thesis, University of Oxford, 2011.
[120] L. Popa. *Object/Relational Query Optimization with Chase and Backchase*. PhD thesis, U. Penn., 2000.
[121] Y. Sagiv. Optimizing Datalog programs. In *PODS*, 1987.
[122] Y. Sagiv and M. Yannakakis. Equivalences among Relational Expressions with the Union and Difference operators. *JACM*, **27**(4):633–655, 1980.
[123] M. Schäfer and O. de Moor. Type inference for Datalog with complex type hierarchies. In *POPL*, 2010.
[124] S. Schulz. System Description: E 1.8. In *LPAR*, 2013.
[125] L. Segoufin and B. ten Cate. Unary negation. *Logical Methods in Computer Science*, 9(3), 2013.
[126] L. Segoufin and V. Vianu. Views and queries: determinacy and rewriting. In *PODS*, 2005.
[127] O. Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, 1993.
[128] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A modular query optimizer architecture for big data. In *SIGMOD*, 2014.
[129] C. Stirling. A tableau proof system with names for modal mu-calculus. In *HOWARD-60:*, 2014.
[130] T. Studer. On the proof theory of the modal mu-calculus. *Studia Logica*, 89(3):343–363, 2008.
[131] D. Toman and G. Weddell. *Fundamentals of Physical Design and Query Compilation*. Morgan Claypool, 2011.
[132] D. Toman and G. E. Weddell. An interpolation-based compiler and optimizer for relational queries (system design report). In *IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations*, 2017.
[133] B. Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. *Proceedings of the USSR Academy of Sciences*, (4):569âĂŞ572, 1950.
[134] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.
[135] M. Y. Vardi. Why is modal logic so robustly decidable? In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 31, pages 149–184. American Mathematical Society, 1997.
[136] M. Y. Vardi. A logical revolution (keynote). In *FSE*, 2013.
[137] D. Vrandečić. Wikidata: A new platform for collaborative data collection. In *WWW*, 2012.
[138] C. Wernhard. The PIE system for proving, interpolating and eliminating. In *PAAR*, 2016.