# Technical Report CS-01-23
Computing Provenance Using the Negated Chase

Andreas Görres, Andreas Heuer

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik
Lehrstuhl für Datenbank- und Informationssysteme

Long version of a 4-page conference paper
for VLDB PhD Workshop 2023,

erhältlich unter:
www.ls-dbis.de/digbib/dbis-tr-cs-01-23.pdf

# Computing Provenance Using the Negated Chase

Andreas Görres, Andreas Heuer
Database Research Group
University of Rostock
18051 Rostock
(andreas.goerres|andreas.heuer)(at)uni-rostock.de

**Abstract:** Since different challenges of data processing are interconnected, we describe them in a unified manner using a classic algorithm of database theory: the Chase. Explaining the origin of query results is one of the challenges considered in this research project. Previously, the Chase has been used to calculate why-provenance of simple conjunctive queries. However, applying the Chase to more realistic scenarios requires an extension of the algorithm, for example with negation. This work reveals opportunities for the extended Chase by calculating both why- and why-not provenance of conjunctive queries with negation.

## 1 Introduction

When processing large amounts of data in a systematic fashion, we usually solve the arising issues with algorithms tailored towards the specific challenge. Even though this strategy leads to increased efficiency on a local level, we miss connections between the existing problems. For instance, privacy and provenance are contradictory requirements usually solved in isolation from each other.

If we describe data processing with the concept of the data science pipeline[1], privacy and provenance can be mapped to individual steps of the pipeline, but at the same time, they are requirements for the other steps. Therefore, we need to consider them while designing the database schema, schema evolution, cleaning the data, transforming the queries and while processing the results of data analysis. Formulating separate issues with a single consistent language makes this possible. In our research, we chose the language of the Chase algorithm.

The Chase$_P(O)$ integrates a parameter $P$ into an object $O$, in the classical applications using integrity constraints as $P$ and database schemas or queries as $O$. The Chase is a classic algorithm of database theory. It was introduced more than four decades ago, and right from the start, the Chase processed two seemingly unrelated use cases – query optimization and schema construction – in a unified way [ASU79, MMS79]. In the following years, the Chase was slightly extended and the number of its application areas increased even further. The concept of "universal solution" connects the Chase's many application cases [DNR08]. Since then, intense research lead to a deeper understanding of the algo-

[1]The data science pipeline is also called data engineering/analytics pipeline, as in [KS22]

rithm's properties, for example its termination behavior. Despite its success in the field of theory, software tools making use of the algorithm are – for the most part – restricted to scientific prototypes. Ultimately, we intend to use the Chase to solve practice-related use cases, in particular, issues of privacy and provenance.

In this work, we focus on our results concerning the why- and why-not provenance. Previously, the Chase was already used to calculate the why-provenance of simple queries and scheme transformations. However, while we interpret the Chase as a universal algorithm, the algorithm needs to be extended for more realistic – and more complex – scenarios. Unfortunately, those extensions endanger confluence, termination and efficiency of the Chase. While our general research goal targets the universal Chase applicable to a broad variety of objects – like queries, database schemes, database instances or deductive databases – the semantics of its extensions depends on the Chase object and therefore the use case. While privacy issues can be solved with the Chase on queries, identifying relevant data in a database requires the Chase on database instances, which is therefore the focus of this work.

## 1.1 Data Science Pipeline

Data processing can be divided into a sequence of steps we call the data science pipeline. The initial step – data preparation – comprises schema design, data cleaning and data integration. Subsequently, data analysis takes place, which we abstract as a sequence of database queries. In a final post-processing phase, results are interpreted and issues of privacy and provenance are tackled. In the following, we will take a closer look at the individual steps highlight the contributions of the Chase algorithm.

During **schema design**, Chase parameter $P$ comprises data dependencies and Chase object $O$ is the database schema, possibly encoded as a tableau.

For **data cleaning**, $P$ are integrity constraints (e.g. key constraints) and $O$ is the database instance. The Chase might substitute null values with constants by making use of functional dependencies or insert missing tuples, e.g. to satisfy inclusion dependencies. While this approach fails if there are conflicting constants present in the database, there are Chase based approaches dealing with this issue as well [GMPS20].

**Database integration** (including **data exchange** [FKMP05]) takes data from one or several source instances – under the respective schemas – to generate a target instance. Again, $O$ is the set of database instances, whereas $P$ are rules describing a mapping from source to target schemas.

For **data analysis**, $O$ is again the database instance, while $P$ includes the database queries. Queries can be seen as data exchange rules from the research data to a virtual result instance. However, our interest in complex data analysis comprising e.g. statistical analysis, warrants an extension of the current Chase formalism, for example with mathematical operators or negation.

When databases and database schemas evolve over time, we need **schema evolution** techniques to describe the evolution steps and to perform schema and corresponding database changes. In terms of Chase, $P$ is a set of schema mappings and $O$ is the database schema and the database, the evolution process is applied to. Fagin [FKMP05] describes schema evolution with schema mappings in this way and combines it with other aspects of the data science pipeline, such as schema integration or data exchange. This composition of different steps of the database science pipeline is also our idea here, we extend it to other steps of the pipeline as well, such as provenance and privacy.

Before publication of analysis results, the responsible raw data needs to be identified using **provenance techniques**. This way, reproducibility of the results is guaranteed. At the same time, **privacy guidelines** might prohibit direct access to the raw data, for example when personal information is concerned. To calculate provenance, we invert the Chase rules used in the previous data analysis step. Here, $O$ refers to the achieved analysis results. In contrast to this, we improve privacy by applying Chase rules (e.g. view definitions) onto the query as the Chase object. By combining different Chase applications, we contribute to every step of the data science pipeline.

### 1.2 Research Data Management

One of the real world application areas motivating our research is research data management. Many of our use cases originate from our long term co-operation with the Institute for Baltic Sea Research (IOW). Here, reproducibility of published research results is a requirement for good scientific conduct. However, the schema of recorded data changes over time. Thus, while tracing back results to the responsible data, we need to account for schema evolution. After identifying involved tuples, scientists provide them – and not the entire data set – to a reviewer. If personal data is involved, data access might require additional restrictions.

### 1.3 Negation in Data Analysis

As we mentioned before, we interpret data analysis as a series of database queries, which in turn are expressed as Chase rules. In particular, we are interested in statistical analysis of scientific data. While negation is a central element in those analysis algorithms, general negation is not part of the standard Chase. First, negation might be used to avoid redundancy and thereby increase efficiency. Implicitly, this variant of negation is already part of the Chase algorithm: A rule is only not applied if its trigger is inactive, that is, if the rule is already satisfied by the Chase object. Second, some kind of negation might be explicit part of the analysis algorithm, for example in the form of set difference. Third, negation can be implicit part of basic aggregate function. The maximum of an attribute can, for example, be defined as the value for which no greater value exists in said attribute. Fourth, negation can be used to simulate an imperative algorithm using the declarative language of Chase rules. To iterate over all tuples of a database relation, identifiers of tuples already considered are inserted into a certain relation. By negating identifiers in this relation, we guarantee that each tuple is considered only once. This way, we can describe more complex aggregate functions. However, the fourth use case of negation is not relevant for provenance calculation.

### 1.4 Contribution

Using the Chase algorithm, we solve interconnected challenges of the data science pipeline in a coordinated manner. In this work, we study how previous results concerning provenance calculation are affected if we extend the Chase with negation. While the Conditional Chase we describe here has been studied in previous theoretical works [GO11], connecting it to Chase negation is quite unusual. Furthermore, this extension allows the calculation of instance based why-not provenance. While the calculation of why-not provenance itself is already possible with specialized algorithms, our Chase based solution is directly integrated into a framework solving a multitude of other data science challenges, for example schema evolution and query transformation.

## 2 State of the Art

The Chase is a fixpoint algorithm incorporating a set of rules, the Chase parameter, into a Chase object. In this work, the Chase parameter is a query or transformation rule, while the Chase object is a database instance. Chase rules considered here are logical implications called tuple generating dependencies (tgds), which have the general form $\phi(\vec{x}, \vec{y}) \rightarrow \exists \vec{z} : \psi(\vec{x}, \vec{z})$. Here, $\phi$ and $\psi$ are sets of relational atoms over the depicted variables. If the tgd encodes a query, we often existentially quantify $\vec{y}$, while at the same time prohibit existentially quantified variables $\vec{z}$ in the rule head (its right hand side). The head of a query is usually a single, unnamed atom $(\vec{x})$. The inversion of a tgd is the logical implication $\psi(\vec{x}, \vec{z}) \rightarrow \exists \vec{y} : \psi(\vec{x}, \vec{y})$.

If there is a homomorphic mapping of a Chase rule's body (the left hand side) and the rule is not satisfied yet (the trigger is active), the rule head is materialized. Consequently, the rule head's image under the homomorphism is materialized and a set of new tuples is generated. For each existentially quantified variable in the rule head, a fresh marked null value $\eta_i (i \in \mathbb{N})$ is generated. The Chase continues until a fix point is reached, however, termination is not guaranteed if existentially quantified variables in the rule head are allowed and the rules are cyclic. Still, there are several termination tests guaranteeing Chase termination even on cyclic rule sets. The Chase variant explained later in this work, the Conditional Chase, is for example guaranteed to terminate on richly acyclic rule sets in polynomial time [GO11].

In general, provenance can be seen as meta-data describing a production process [HDB17]. In the context of our generalizing research approach, why- and why-not data provenance are of particular interest. In contrast, a specialized prototypes computing more detailed variants of provenance can be found in [SJMR18].

Why-provenance provides the tuples involved in the generation of a certain result. Those tuples may be presented in the form of a witness basis. Each witness is a tuple of the original database justifying the given result. Quite often, for example if tuples become indistinguishable after projection, there are alternative sets of tuples witnessing the same result. Thus, a witness basis may consist of several sets of tuples. Alternatively, using the Chase on the inverted query leads us to a generic representative for a set of alternative witnesses (compare the relaxed Chase-inverse in [FKPT11]).

Why-not provenance provides an explanation why some expected tuples were not part of the result. This explanation can take three different forms: Instance based explanations, query based explanations and refinement based explanations [HDB17]. Since the Chase algorithm does not allow the deletion of tuples or update of constants, we restrict ourselves to explanations based on the insertion of tuples into the database. The algorithm calculating why-not provenance in [HH10] relies heavily on the concept of conditions and c-tables. By making use of the Conditional Chase, we combine this concept with the Chase-based solution calculating why-provenance.

Notice, however, that our concept of condition refers to comparisons with undetermined values of the Chase object (that is, null values), while in [HH10], conditions comprise comparisons in the query (in our concept: the Chase parameter) and constants in the expected result.

While [AHH22] formalizes both evolution rules and conjunctive queries using the language of the Chase, only a subset of practice relevant schema transformations is discussed. Furthermore, only non-recursive queries are actually inverted using the Chase (even though provenance of more complex expressions, e.g. recursive Chase programs encoding aggregate functions, is examined using other provenance techniques). In contrast to this, our starting point is the (extended) Chase algorithm itself, so we initially consider queries and schema transformations described by general tgds (with negation).

## 3 Implementation

With the Chase implementation ChaTEAU, different applications of the Chase are combined in a in a single toolkit [AHH22]. Currently, extensions like generalized negation on database instances and queries under the conditional semantics discussed in this work are integrated into the software.

## 4 Negation on Instances

While we regard the Chase as a universal algorithm handling different kinds of parameters and objects in a unified manner, semantics of negation still depends on the Chase object.

For the Chase on instances, negation in a Chase rule requires the absence of a certain set of tuples. In contrast, negation in Chase rules applied to queries requires the presence of an explicitly negated set of atoms in the query. As a consequence, we differentiate between negation as a negated boolean subquery (from integrity constraint to Chase object database instance), and negation as a boolean subquery with inverted direction (from Chase object query to integrity constraint). In this work, we will focus on the first case, since it is more relevant the use case data provenance. After finding a term mapping $h$ for all variables of the positive body of an integrity constraint, we select the atoms $\phi(\vec{x}, \vec{y})$ of a negative body $\neg\exists\vec{y} : \phi(\vec{x}, \vec{y})$ (that is, a negated conjunction of relational atoms). If the result of the boolean query $\phi(\vec{x}, \vec{y}) \rightarrow ()$ is $()$, we reject $h$, otherwise, we continue with the next negative body or complete the Chase step (e.g. generate some tuples).

In this work, we will focus on semi-positive negation, that is, negation of base relations. Otherwise, we can no longer guarantee a single generic witness base and calculations become rather complicated.

## 5 Conditions and Certain Answers Semantics

The standard Chase algorithm (without negation) operates under certain answers semantics. For this, we interpret (marked) null values as unknown, but existing values. We consider only results justified under any valuation of null values with concrete constants. For positive rules, it is sufficient to interpret null values as constants unequal to any constant in the database instance. However, this naive interpretation is not sufficient for negation under certain answers semantics. Consider a rule that is only triggered if a specific null value $\eta_1$ equals a certain constant $c$. Under naive interpretation (and under trinary semantics of SQL), $\eta_1$ is not equal to $c$ and no result tuple is generated. However, another rule with negation would be blocked by this result tuple. Since we did not generate the blocking tuple, the result tuple of the second rule is produced. Clearly, this does not follow certain answers semantics, since there is a valuation of null values (the valuation of $\eta_1$ with $c$) not justifying the generation of this tuple. To solve this issue, we not only consider certain tuples (true under any valuation of null values), but also tuples existing under certain conditions. For this paper, we define conditions as conjunctions of the logical comparisons $x\theta y$ ($\theta \in \{=, \neq\}, x \in \text{CONST}, y \in \text{CONST} \cup \text{NULL}$), with $CONST$ being the set of all constants of the domain and $\text{NULL}$ being the set of all marked null values. In the previous example, the blocking tuple would be generated under the condition $\eta_1 = c$, while the second result tuple would be generated under the inverted condition $\eta_1 \neq c$. The conditions of several equivalent tuples might complete each other to form a tautology. For example, the blocking tuple and the second result tuple of the previous example might have been identical. In this case, conditions can be omitted and the condition free tuples exist under certain answers semantics.

## 6 Provenance and the Minimal Witness Basis

### 6.1 Why Provenance

As mentioned before, we can use the standard Chase algorithm (without negation) to calculate the why-provenance of query results. The notion of why-provenance used here is based on the relaxed Chase-inverse found in [FKPT11].

For this, we invert the Chase rules that created the result by switching the rule's head and body. Attribute values not passed to the result (non frontier variables of the s-t tgd) become existentially quantified variables of the inverted Chase rule.

Applying the inverted query to the result using the so-called Backchase generates the minimal [2] witness basis. This witness basis is sufficient to create the same query result as the original instance. However, it is usually smaller than the latter and contains marked null values in places irrelevant for the query.

For the most part, we can ignore semi-positive negation when calculating why-provenance. However, the same witness might play different roles during query execution. Consider query $q$ on instance $I$:

$$\begin{array}{rl} I = & \{R(1), R(2), S(2)\} \\ q: & R(x) \wedge R(y) \wedge \neg S(y) \rightarrow (x, y). \end{array}$$

If we ignore the semi-positive negation of $S$, the witness basis is $\{R(1), R(2)\}$. Indeed, even if we materialize the image of $q$'s negative body, we only learn that $S(1)$ cannot exist, but we learn nothing about $S(2)$. Consequently, we can justify the existence of all previously published results $\{(1,1),(2,1)\}$, but we could additionally justify the result $(2,2)$. The absence of this expected result from the published result could be explained using why-not provenance. However, the instance based why-not provenance presented in this work is restricted to insertions – clearly, there is no way to generate $(2,2)$ by inserting additional tuples into the database.

While the method described above can be used for recursive queries, it should be noted that an inverted Chase program might not terminate even though termination of the original program is guaranteed.

This is, for example, the case with the following Chase program comprising the single tgds $r_1$ and its inversion $r_1^{-1}$:

$$\begin{array}{rl} r_1: & R(x,y) \wedge S(y) \rightarrow \exists z : R(y,z) \\ r_1^{-1}: & R(y,z) \rightarrow \exists x : R(x,y) \wedge S(y). \end{array}$$

Apart from this problem of Chase termination, different levels of selectivity in tgds triggering each other might prohibit inversion. If, for example, the first rule generates a fixed

[2]Strictly speaking, the witness basis is only guaranteed to be minimal if we use the Core Chase, a variant of the standard Chase.

constant in an attribute, while this attribute is ignored by the second rule, that is, there is neither a selection concerning this attribute, nor is the attribute value passed to the generated tuple. Even though the first rule triggers the second rule, the inversion of the second rule does not trigger the first one, since the generated null value [3] is not equal to the constant the inverted first rule selects for. While the reduction of selectivity causes problems for inversion in general, an increase in selectivity (constants in the body of the second rule that do not appear in the head of the first one) is without consequences for both why- and why-not provenance if we do not allow existentially quantified variables in any rule head.

The problems described until now are not specific for negation. If we extend the Chase-based calculation of why-provenance to stratifiable rule sets (with negation), nested negation might lead to alternative positive witnesses. Consider the following example of the transformation rule $r_{evol}$, describing a schema evolution, and query $q$, which addresses the target of the previous evolution:

$$\begin{array}{ll} r_{evol}\text{:} & R(1) \wedge \neg R(2) \rightarrow S(4) \\ q: & T(x) \wedge \neg S(4) \rightarrow Q(x). \end{array}$$

The witness base for the result $Q(1)$ is $\{T(1) \wedge \neg(R(1) \wedge \neg R(2))\}$, which can be simplified as the disjunction of witnesses $\{T(1) \wedge \neg R(1)\} \vee \{T(1) \wedge R(2)\}$. Even though we might eventually drop the negated part of the witness base, at first, we need it to to identify contradictions (e.g. bases containing both the positive and the negated variant of the same witness).

## 6.2 Why-not Provenance

Similarly to the why-provenance, the Chase algorithm can be used to calculate instance based why-not explanations. As a starting point, we apply the inverted query to the expected result [4]. The Chase result, if it exists, comprises the generic witness basis. However, the existence of an instance based explanation is not guaranteed. If the select part of our SQL query contains a specific constant, no changes in the database could generate a different constant in this position and we would be unable generate a generic witness basis. Deviating from the previous explanation of why-provenance, we map generic witnesses to concrete database tuples – those tuples are not part of the why-not explanation.

To calculate a mapping of a subset of witnesses (with null values) to the original database, we use the Conditional Chase. Initially, we tag every tuple of the original database with the same identifier, and each tuple of the generic witness basis (the explanation tuples) with an individual identifier. We interpret another copy of the witness basis as a query returning identifiers of all involved tuples (the witness query). Null values of the witness basis are substituted by fresh existentially quantified variables, and the identifiers by universally quantified variables (also appearing in the query head).

[3] Since the attribute value is not passed to the created tuple, the variable is a "non-frontier variable" and existentially quantified in the inverted second rule – therefore, it generates a null value.

[4] The expected result refers here to the union of missing tuples and the originally achieved result.

After applying this query to the database instance that is enriched with identifiers, we return the result with the smallest number of different witness ids.

If one of the results contains solely the default identifier of the existing database, there is no instance based (and insertion based) why-not explanation since all generic witnesses can be mapped to already existing tuples. Otherwise, identifiers of the result refer to the tuples in the generic witness basis that are part of the why-not explanation.

If a witness mapped to a tuple from the why-not explanation and a witness mapped to a tuple from the original database share existentially quantified variables, this variable is mapped to a marked null value (from the explanation tuple) and a constant (from the existing tuple). The Conditional Chase allows this mapping under the condition that null value and constant are equal. If those equality conditions are consistent, we interpret them as term mappings and substitute null values of explanation tuples with constants of the original database instance.

Finally, we incorporate negation into this method. Inverting a query with negative body will lead to a tgd with negation in its head. The image of this negated atom set will be used when constructing the witness query, but not the explanation tuples. When considering negation, a specific mapping of the generic witness basis might become impossible. Other mappings might only be possible under certain (inequality) conditions. Those conditions being incompatible with the established (equality) conditions invalidates additional mapping.

Consider the following simple example. Query $q_1$ was applied to the original database instance $I$, but did not generate the expected result $(1)$. Using the Backchase with inverted $q_1$ on $(1)$ generates a generic witness basis. $I_{wit}$ is the union of $I$ and explanation tuples from this witness basis. We mark the tuples in $I_{wit}$ with an identifier (0 for tuples from $I$ and integers greater than 0 for explanation tuples):

$$\begin{array}{ll} I = & \{R(1,2), S(2)\} \\ q_1: & R(x,y) \wedge \neg T(y) \rightarrow (x) \\ I_{wit} = & \{R(1,2,0), S(2,0), R(1,\eta_1,1)\} \\ q_{wit}: & R(1,e_1,id_1) \wedge \neg S(e_1,id_2) \rightarrow (id_1). \end{array}$$

There exist two mappings from $q_{wit}$'s positive body to $I_{wit}$. However, mapping $R(1,e_1,id_1)$ to $R(1,2,0)$ allows the mapping of $S(e_1,id_2)$ to $S(2,0)$. In contrast, mapping the positive body to $R(1,\eta_1,1)$ allows no consistent valuation of $S(e_1,id_2)$ under condition $\eta_1 \neq 2$. In conclusion, the minimal why-not explanation consists of $R(1,\eta_1,1)$ under condition $\eta_1 \neq 2$. Thus, inserting $R(1,\eta_1)$ into the original database leads, under naive interpretation of null values, to the expected result $(1)$.

While general Chase rules comprise tuple generating dependencies with existentially quantified head variables (generating fresh marked null values), queries (and schema transformation rules) considered here are restricted to universally quantified variables and constants in their conclusion. Otherwise, a tgd might generate a fresh null value tested for equality with an other value in the subsequent tgd. Since the null value is freshly generated, this test always fails and the negation should have no consequences for the why-not explanation. Examine the following query $q$ which negates the result of the schema evolution encoded by $r_{evol}$.

$$\begin{array}{ll} r_{evol}: & R(1) \wedge \neg R(2) \rightarrow \exists e : S(4, e) \\ q: & T(x, y) \wedge \neg S(4, y) \rightarrow Q(x). \end{array}$$

Since the null value generated by $q_{sub}$ in the second attribute of $S$ is not transferred to $T$, we cannot map both occurrences of $y$ in $q_{main}$ with this null value. The inverted transformation rule $r_{evol}$, however, generates the same disjunction of witnesses as the previous example ( $\{\neg R(1)\} \vee \{R(2)\}$). Since those witnesses do not contain the null value created by the inverted main query, mapping the witness base to the database instance will not generate conditions for this value. Therefore, the tuple $R(2)$ is an unneeded part of the explanation and the why not explanation is not minimal.

## 7 Future Work

In this work, we described for two types of provenance – why-provenance and instance based why-not provenance – Chase based solutions. In our next publications, we will include other types of why-not provenance – e.g. deletion based explanations – into our framework. Unfortunately, the Chase fails directly when detecting a violated primary key dependency involving unequal constants. Deletion based why-not explanations, however, need to identify the responsible tuple instead of failing (compare the second example in [Her13]). This warrants further Chase extensions. If we incorporate arithmetic comparisons – specifically, inequality – into the algorithm, we can detect (violated) primary keys using tgds. This way, we solve the primary key violation by deleting one of the responsible tuples, an entirely different strategy than the one used by classic standard Chase.

The explanation given in this work only considers semi-positive negation, even though the query itself might be recursive. If we negate the result of a subquery expressed as an individual tgd, we require a certain order of rule application. Even though, restricting the recursive query to a stratifiable rule set ensures a unique result. However, if negation is present in both query and subquery, why-provenance might be explained by alternative sets of generic witnesses.

The methods described in this work make use of a database instance as the Chase object. However, other steps of the data science pipeline require a query to be the Chase object. While Chase semantics are very similar for both types of objects, semantics of negation is entirely different. Negation on instances refers to missing tuples, but to explicitly negated atom sets in queries.

## 8 Conclusion

With the Chase algorithm, a large variety of data science challenges can be handled in unified manner. Provenance, for instance, can be calculated while keeping track of schema evolution. Real world applications, however, require extensions of the algorithm which affect its confluence and termination behavior. In this work, we demonstrate how why-provenance and why-not provenance of conjunctive queries with semi-positive negation can be explained using the extended Chase. While we chose the Conditional Chase to realize certain answers semantics with negations, this decision was advantageous for the explanation of why-not provenance even in the absence of negation.

**Acknowledgments**

This work was supported by a scholarship of the Landesgraduiertenförderung Mecklenburg-Vorpommern.

## Literatur

[AHH22] Tanja Auge, Moritz Hanzig und Andreas Heuer. ProSA Pipeline: Provenance Conquers the Chase. In *ADBIS (Short Papers)*, Jgg. 1652 of *Communications in Computer and Information Science*, Seiten 89–98. Springer, 2022.

[ASU79] Alfred V. Aho, Yehoshua Sagiv und Jeffrey D. Ullman. Efficient Optimization of a Class of Relational Expressions. *ACM Trans. Database Syst.*, 4(4):435–454, 1979.

[DNR08] Alin Deutsch, Alan Nash und Jeffrey B. Remmel. The chase revisited. In *PODS*, Seiten 149–158. ACM, 2008.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller und Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[FKPT11] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa und Wang Chiew Tan. Schema Mapping Evolution Through Composition and Inversion. In *Schema Matching and Mapping*, Data-Centric Systems and Applications, Seiten 191–222. Springer, 2011.

[GMPS20] Floris Geerts, Giansalvatore Mecca, Paolo Papotti und Donatello Santoro. Cleaning data with Llunatic. *VLDB J.*, 29(4):867–892, 2020.

[GO11] Gösta Grahne und Adrian Onet. On Conditional Chase Termination. *AMW*, 11:46, 2011.

[HDB17] Melanie Herschel, Ralf Diestelkämper und Houssem Ben Lahmar. A survey on provenance: What for? What form? What from? *VLDB J.*, 26(6):881–906, 2017.

[Her13] Melanie Herschel. Wondering why data are missing from query results?: ask conseil why-not. In *CIKM*, Seiten 2213–2218. ACM, 2013.

[HH10] Melanie Herschel und Mauricio A. Hernández. Explaining Missing Answers to SPJUA Queries. *Proc. VLDB Endow.*, 3(1):185–196, 2010.

[KS22] Meike Klettke und Uta Störl. Four Generations in Data Engineering for Data Science. *Datenbank-Spektrum*, 22(1):59–66, 2022.

[MMS79] David Maier, Alberto O. Mendelzon und Yehoshua Sagiv. Testing Implications of Data Dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, 1979.

[SJMR18] Pierre Senellart, Louis Jachiet, Silviu Maniu und Yann Ramusat. ProvSQL: Provenance and Probability Management in PostgreSQL. *Proc. VLDB Endow.*, 11(12):2034–2037, 2018.