Interactive Inference of Join Queries

Angela Bonifati, Radu Ciucanu, Slawomir Staworko

Interactive Inference of Join Queries

Angela Bonifati Radu Ciucanu Sławek Staworko
University of Lille & INRIA, France
{angela.bonifati, radu.ciucanu, slawomir.staworko}@inria.fr

ABSTRACT

We investigate the problem of inferring join queries from user interactions. The user is presented with a set of candidate tuples and is asked to label them as *positive* or *negative* depending on whether or not she would like the tuples as part of the join result. The goal is to quickly infer an arbitrary n-ary join predicate across two relations by keeping the number of user interactions as minimal as possible. We assume no prior knowledge of the integrity constraints between the involved relations. This kind of scenario occurs in several application settings, such as data integration, reverse engineering of database queries, and constraint inference. In such scenarios, the database instances may be too big to be skimmed. We explore the search space by using a set of strategies that let us prune what we call "uninformative" tuples, and directly present to the user the *informative* ones i.e., those that allow to quickly find the goal query that the user has in mind. In this paper, we focus on the inference of joins with equality predicates and we show that for such joins deciding whether a tuple is uninformative can be done in polynomial time. Next, we propose several strategies for presenting tuples to the user in a given order that lets minimize the number of interactions. We show the efficiency and scalability of our approach through an experimental study on both benchmark and synthetic datasets. Finally, we prove that adding projection to our queries makes the problem intractable.

1. INTRODUCTION

The amount of data and the number of available data sources continue to grow at an ever astounding rate allowing the users to satisfy more and more complex information needs. However, expressing complex information needs requires the use of formalisms for querying and integrating data sources that are typically mastered by only a small group of adept users. In real life, casual users often have to combine raw data coming from disparate data sources, with little or no knowledge of metadata and/or querying formalisms. Such unqualified users need to resort to brute force solutions of manipulating the data by hand. While there may exist providers of integrated data, the users may be unsatisfied with the quality of their results.

In this paper, we consider very simple user input via Boolean membership queries ("Yes/No") to assist unfamiliar users to write their queries. In particular, we focus on two fundamental operators of any data integration or querying tool: *equijoins* – combining data from two sources, and *semijoins* – filtering data from one source based on the data from another source. Such operators are crucial in several application settings, such as data integration, reverse engineering of database queries, and constraint inference, whenever the user has little or no knowledge of the database schemas.

Inference algorithms for expert users have been recently studied to design mappings [3, 4] via data examples. Such examples are expected to be built by the mapping designer, who is also responsible of selecting the mappings that best fit them. Query learning for relational queries with quantifiers has recently been addressed in [1, 2]. There, the system starts from an initial formulation of the query and refines it based on primary-foreign key relationships and the input from the user. We discuss in detail the differences with our work at the end of this section. To the best of our knowledge, ours is the first work that considers inference of joins via simple tuple labeling and with no knowledge of integrity constraints.

Consider a scenario where a user working for a travel agency wants to build a list of flight&hotel packages. The user is not acquainted with querying languages and can access the information on flights and hotels in two tables (Figure 1).

From	*To*	*Airline*
Paris	Lille	AF
Lille	NYC	AA
NYC	Paris	AA
Paris	NYC	AF

City	*Discount*
NYC	AA
Paris	None
Lille	AF

Figure 1: Instances of *Flight* and *Hotel*, respectively.

The airline operating every flight is known and some hotels offer a discount when paired with a flight of a selected airline. Two queries can be envisioned: one that selects packages consisting of a flight and a stay in a hotel and another one that additionally ensures that the package is combined in a way allowing a discount. These two queries correspond to the following equijoin predicates:

$$Flight.To = Hotel.City, \quad (Q_1)$$
$$Flight.To = Hotel.City \wedge Flight.Airline = Hotel.Discount. \quad (Q_2)$$

Note that since we assume no knowledge of the schema and of the integrity constraints, a number of other queries can possibly be formulated but we remove them from consideration for the sake of simplicity and clarity of the example.

While the user may be unable to formulate her query, it is reasonable to assume that she can indicate whether or not a given pair of flight and hotel is of interest to her, or she can even pair a flight and hotel herself. We view this as labeling with + and − the tuples of the Cartesian product of the two tables (Figure 2). For instance, suppose the user pairs the flight from Paris to Lille operated by Air France (AF) and the hotel in Lille. This corresponds to labeling by + the tuple (3) in the Cartesian product below.

	From	*To*	*Airline*	*City*	*Discount*	
	Paris	Lille	AF	NYC	AA	(1)
	Paris	Lille	AF	Paris	None	(2)
+	Paris	Lille	AF	Lille	AF	(3)
+	Lille	NYC	AA	NYC	AA	(4)
	Lille	NYC	AA	Paris	None	(5)
	Lille	NYC	AA	Lille	AF	(6)
	NYC	Paris	AA	NYC	AA	(7)
−	NYC	Paris	AA	Paris	None	(8)
	NYC	Paris	AA	Lille	AF	(9)
	Paris	NYC	AF	NYC	AA	(10)
	Paris	NYC	AF	Paris	None	(11)
	Paris	NYC	AF	Lille	AF	(12)

Figure 2: Cartesian product of tables from Figure 1.

Observe that both queries Q_1 and Q_2 are consistent with this labeling i.e., both queries select the tuple (3). Naturally, the objective is to use the labeling of further tuples to identify the goal query i.e., the query that the user has in mind. Not every tuple can however serve this purpose. For instance, if the user labels next the tuple (4) with +, both queries remain consistent. Intuitively, the labeling of the tuple (4) does not contribute any new information about the goal query and is therefore *uninformative*, an important concept that we formalize in this paper. Since the input tables may be big, it may be unfeasible for the user to label every tuple in the Cartesian product.

In this paper, we aim at limiting the number of tuples that the user needs to label in order to infer the goal query. We propose solutions that analyze and measure the potential information about the goal query that labeling a tuple can contribute and present to the user tuples that maximize this measure. In particular, since uninformative tuples do not contribute any new information, they are not presented to the user. In the example of the flight&hotel packages, a tuple whose labeling can distinguish between Q_1 and Q_2 is, for instance, the tuple (8) because Q_1 selects it and Q_2 does not. If the user labels the tuple (8) with −, then the query Q_2 is returned; otherwise Q_1 is returned. We also point out that the use of only *positive* examples, tuples labeled with +, is not sufficient to identify all possible queries. As an example, query Q_2 is contained in Q_1, and therefore, satisfies all positive examples that Q_1 does. Consequently, the use of *negative* examples, tuples with label −, is necessary to distinguish between these two.

We make the following *main contributions*:

- We focus on *equijoins* and we characterize the potential information that labeling a given tuple may contribute in the join inference process and identify uninformative tuples. We provide an alternative characterization of uninformative tuples that does not assume the knowledge of the goal query and can be efficiently tested.
- We propose a set of strategies for interactively inferring a goal join predicate and we show their efficiency and scalability within an experimental study on both TPC-H and synthetic data.
- With the goal of extending our queries with the projection operator, we then study *semijoins*. We investigate the consistency problem, a fundamental problem underlying query inference, which is to decide whether there exists a query consistent with a given set of examples. While for equijoins this problem is in PTIME, it becomes NP-complete for semijoins, which precludes the possibility of an efficient inference of semijoin queries from examples.

Since our goal is to minimize the number of interactions with the user, our research is of interest for novel database applications e.g., joining datasets using crowdsourcing [11], where minimizing the number of interactions entails lower financial costs. Moreover, our research also applies to schema mapping inference, assuming a less expert user than in [3, 4]. Indeed, in our case the annotations correspond to simple membership queries [5] to be answered even by a user who is not familiar with schema mapping. While the restriction to joins of two relations only may seem very limiting, such queries can be of practical use in the context of denormalized databases having a small number of relations with large numbers of attributes.

The paper is organized as follows. In Section 2, we introduce some preliminary notions. In Section 3, we state our problem setting and characterize our problems of interest. In Section 4, we propose practical strategies of presenting tuples to the user, while in Section 5, we experimentally evaluate their performance. In Section 6, we show that consistency checking becomes intractable when we add the projection. Finally, we summarize the conclusions and outline directions of future work in Section 7.

Related work

Our work follows a very recent line of research on the inference of relational queries [19, 17, 7]. Zhang et al. [19] have focused on computing a join query starting from a database instance, its complete schema, and an output table. Clearly, their assumptions are different from ours. In particular, we do not assume any knowledge of the integrity constraints or the query result. In our approach, the latter has to be incrementally constructed via multiple interactions with the user, along with the join predicate itself. Zhang et al. [19] consider more expressive join queries than we do, but when the integrity constraints are unknown, one can leverage our algorithms to yield those and apply their approach thereafter. Moreover, Tran et al. [17] have investigated the query by output problem: given a database instance, a query statement and its output, construct an instance-equivalent query to the initial statement. Das Sarma et al. [7] have studied

the view definition problem i.e., given a database instance and a corresponding view instance, find the most succinct and accurate view definition. Both [17] and [7] essentially use decision trees to classify tuples as selected or not selected in the query output or in the view output, respectively. We differ from their work in two ways: we do not know a priori the query output, and we need to discover it from user interactions; we have no initial query statement to start with.

Fan et al. [8] have worked on discovering conditional functional dependencies using data mining techniques. We focus on simpler join constraints, and exploit an interactive scenario to discover them by interacting with the users.

Since our goal is to find the most informative tuples and ask the user to label them, our research is also related to the work of Yan et al. [18]. However, we do not use active learning and we do not consider keyword-based queries. Another work strongly related to ours has been done by Abouzied et al. [1, 2], who have formalized a query learning model using membership questions [5]. They focus on learning quantified Boolean queries for the nested relational model and their main results are optimal algorithms for learning some subclasses of such queries [1] and a system that helps users specify quantifiers [2]. Primary-foreign key relationships between attributes are used to place quantified constraints and help the user tune her query, whereas we do not assume such knowledge. The goal of their system is somewhat different, in that their goal is to disambiguate a natural language specification of the query, whereas we focus on raw data to guess the "unknown" query that the user has in mind. The theoretical foundations of learning with membership queries have been studied in the context of schema mappings by ten Cate et al. [15]. Moreover, Alexe et al. [3, 4] have proposed a system which allows a user to interactively design and refine schema mappings via data examples. The problem of discovering schema mappings from data instances have been also studied in [9] and [12]. Our queries can be eventually seen as simple GAV mappings, even though our problem goes beyond data integration. Moreover, our focus is on proposing tuples to the user, while Alexe et al. [3, 4] assume that an expert user chooses the data examples. Additionally, our notions of certain and uninformative tuples have connections with the approach of Cohen and Weiss [6] for XPath queries, even though joins are not considered there. Furthermore, our notion of entropy of a tuple is related to the work of Sellam and Kersten [14] on exploratory querying big data collections.

2. PRELIMINARIES

We assume the setting of two *relations* R and P with disjoint sets of attributes $attrs(R) = \{A_1, \ldots, A_n\}$ and $attrs(P) = \{B_1, \ldots, B_m\}$. We assume no other knowledge of the database schema, in particular no knowledge of the integrity constraints between the two relations. By Ω we denote the set $attrs(R) \times attrs(P)$. Given a subset $\theta \subseteq \Omega$, by $R \bowtie_\theta P$ we denote the following relational algebra expression:

$$R \bowtie_\theta P = R \bowtie_{\bigwedge_{(A_i,B_j)\in\theta} R[A_i]=P[B_j]} P$$

and we refer to θ as the *equijoin predicate*. Similarly, by $R \ltimes_\theta P$ we denote the following relational algebra expression:

$$R \ltimes_\theta P = \Pi_{attrs(R)}(R \bowtie_\theta P)$$

and we refer to θ as the *semijoin predicate*.

A *database instance* is a pair of sets of tuples $I = (R^I, P^I)$. The semantics of relational algebra expressions is standard and in particular:

$$(R \bowtie_\theta P)^I = \{(t, t') \in R^I \times P^I \mid \forall (A_i, B_j) \in \theta.\ t[A_i] = t'[B_j]\},$$
$$(R \ltimes_\theta P)^I = \{t \in R^I \mid \exists t' \in P^I.\ \forall (A_i, B_j) \in \theta.\ t[A_i] = t'[B_j]\}.$$

We also use the Cartesian product of the two relations and denote it by $D^I = R^I \times P^I$. In the sequel, when the instance is known from the context, we omit the superscript ^{I} when it does not lead to confusion, and in particular, we write simple $R \bowtie_\theta P$ to represent the set $(R \bowtie_\theta P)^I$.

For two join predicates θ_1 and θ_2 such that $\theta_1 \subseteq \theta_2$, we say that θ_1 is more *general* than θ_2 and θ_2 is more *specific* than θ_1. The most general join predicate is $\varnothing$ and the most specific join predicate is Ω. We point out the *anti-monotonicity* of the join operators w.r.t. the join predicate i.e., if $\theta_1 \subseteq \theta_2$, then $R \bowtie_{\theta_2} P \subseteq R \bowtie_{\theta_1} P$ and $R \ltimes_{\theta_2} P \subseteq R \ltimes_{\theta_1} P$.

Example 2.1 Consider R_0 and P_0 with $attrs(R_0) = \{A_1, A_2\}$ and $attrs(P_0) = \{B_1, B_2, B_3\}$ and the following instance:

$R_0 =$

	A_1	A_2
t_1	0	1
t_2	0	2
t_3	2	2
t_4	1	0

and $P_0 =$

	B_1	B_2	B_3
t'_1	1	1	0
t'_2	0	1	2
t'_3	2	0	0

Take $\theta_1 = \{(A_1, B_1), (A_2, B_3)\}$, $\theta_2 = \{(A_2, B_2)\}$, and $\theta_3 = \{(A_2, B_1), (A_2, B_2), (A_2, B_3)\}$. We obtain:

$R_0 \bowtie_{\theta_1} P_0 = \{(t_2, t'_2), (t_4, t'_1)\}$, $R_0 \ltimes_{\theta_1} P_0 = \{t_2, t_4\}$,
$R_0 \bowtie_{\theta_2} P_0 = \{(t_1, t'_1), (t_1, t'_2), (t_4, t'_3)\}$, $R_0 \ltimes_{\theta_2} P_0 = \{t_1, t_4\}$,
$R_0 \bowtie_{\theta_3} P_0 = \varnothing$, $R_0 \ltimes_{\theta_3} P_0 = \varnothing$. □

Next, we present the interactive inference of equijoins, the main class of queries that we deal with in this paper. The only exception is Section 6, where we consider semijoins.

3. INFERENCE OF EQUIJOINS

Take an instance I of two relations R and P and let $D = R \times P$. An *example* is a pair (t, α), where $t \in D$ and $\alpha \in \{+, -\}$. We say that an example of the form $(t, +)$ is a *positive example* while an example of the form $(t, -)$ is a *negative example*. A *sample* S is a set of examples i.e., a subset of $D \times \{+, -\}$. Given a sample S, we denote the set of positive examples $\{t \in D \mid (t, +) \in S\}$ by S_+ and the set of negative examples $\{t \in D \mid (t, -) \in S\}$ by S_-. An equijoin predicate θ is *consistent* with S iff θ selects all positive examples and none of the negative ones i.e., $S_+ \subseteq R \bowtie_\theta P$ and $S_- \cap (R \bowtie_\theta P) = \varnothing$. Naturally, the goal of the inference should be the construction of a consistent equijoin predicate.

We introduce an elementary tool that we employ for inference of equijoins. We assume that the schema and the instance are known from the context. Let $t = (t_R, t_P) \in D$ be a tuple with $t_R \in R$ and $t_P \in P$. We define the *most specific equijoin predicate selecting* t as follows:

$$T(t) = \{(A_i, B_j) \mid t_R[A_i] = t_P[B_j]\}.$$

Additionally, we extend T to sets of tuples $T(U) = \bigcap_{t\in U} T(t)$. Our interest in T follows from the observation that for a given set of tuples U, if θ is a equijoin selecting U, then $\theta \subseteq T(U)$.

3.1 Consistency checking

Given a sample, one would like to decide whether there exists a join predicate that selects all positive examples and none of the negative ones. This permits for instance to check whether the user who has provided the examples is honest, has not made any error, and therefore, has labeled the tuples consistently with some goal join predicate that she has in mind. More formally, the *consistency checking* is the following decision problem: given a database instance I and a sample S, decide whether there exists a consistent join predicate i.e., a equijoin predicate θ such that $S_+ \subseteq R \bowtie_\theta P$ and $S_- \cap (R \bowtie_\theta P) = \varnothing$.

For equijoins this problem has a simple solution that employs the construction of the most specific equijoin predicate: it suffices to check that $R \bowtie_{T(S_+)} P$ selects no negative example. The soundness of this procedure follows from the fact that the most specific join predicate $T(S_+)$ selects all positive examples. To show its completeness, assume there exists a predicate θ selecting all positive examples and none of the negative ones. Since $T(S_+)$ is the most specific equijoin predicate selecting all positive examples, $\theta \subseteq T(S_+)$, and since θ selects no negative example, neither does $T(S_+)$. Hence, $T(S_+)$ is also an equijoin predicate consistent with the set of examples.

Example 3.1 Take the relations R_0 and P_0 and their instances from Example 2.1. In Figure 3 we present the Cartesian product $D_0 = R_0 \times P_0$, the value of T for each tuple from D_0, and the sample S_0 s.t. $S_{0,+} = \{(t_2, t'_2), (t_4, t'_1)\}$ and $S_{0,-} = \{(t_3, t'_2)\}$. The sample is consistent and the most specific consistent join predicate is $\theta_0 = \{(A_1, B_1), (A_2, B_3)\}$. Another consistent join predicate (but not minimal) is $\theta'_0 = \{(A_1, B_1)\}$. On the other hand, the sample S'_0 s.t. $S'_{0,+} = \{(t_1, t'_2), (t_1, t'_3)\}$ and $S'_{0,-} = \{(t_3, t'_1)\}$ is not consistent. From now on, we only consider consistent samples. ▫

	A_1	A_2	B_1	B_2	B_3	T
(t_1, t'_1)	0	1	1	1	0	$\{(A_1, B_3), (A_2, B_1), (A_2, B_2)\}$
(t_1, t'_2)	0	1	0	1	2	$\{(A_1, B_1), (A_2, B_2)\}$
(t_1, t'_3)	0	1	2	0	0	$\{(A_1, B_2), (A_1, B_3)\}$
(t_2, t'_1)	0	2	1	1	0	$\{(A_1, B_3)\}$
$+$ (t_2, t'_2)	0	2	0	1	2	$\{(A_1, B_1), (A_2, B_3)\}$
(t_2, t'_3)	0	2	2	0	0	$\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$
(t_3, t'_1)	2	2	1	1	0	$\varnothing$
$-$ (t_3, t'_2)	2	2	0	1	2	$\{(A_1, B_3), (A_2, B_3)\}$
(t_3, t'_3)	2	2	2	0	0	$\{(A_1, B_1), (A_2, B_1)\}$
$+$ (t_4, t'_1)	1	0	1	1	0	$\{(A_1, B_1), (A_1, B_2), (A_2, B_3)\}$
(t_4, t'_2)	1	0	0	1	2	$\{(A_1, B_2), (A_2, B_1)\}$
(t_4, t'_3)	1	0	2	0	0	$\{(A_2, B_2), (A_2, B_3)\}$

Figure 3: The Cartesian product $D_0 = R_0 \times P_0$, the value of T for each tuple from D_0, and sample S_0.

3.2 Interactive scenario

Let us now consider the following *interactive scenario* of join query inference. The user is presented with a tuple from the Cartesian product and indicates whether the tuple is selected or not by the join predicate that she has in mind by labeling the tuple as a positive or negative example. This process is repeated until a sufficient knowledge of the goal join predicate has been accumulated (i.e., there exists at most one join predicate consistent with the user's labels). This scenario is inspired by the well-known framework of *learning with membership queries* proposed by Angluin [5]. Since the instance may be of big size, we do not want to ask the user to label all tuples from the Cartesian product, but only a part of them. Our goal is to minimize the number of interactions with the user while being computationally efficient. In this context, an interesting question is what strategy of presenting tuples to the user we should adopt. To answer this question, our approach leads through the analysis of the potential information that labeling a given tuple may contribute from the point of view of the inference process.

We first need to introduce some auxiliary notions. We assume the existence of some goal θ^G and that the user labels the tuples in a manner consistent with θ^G. Furthermore, we identify the fully labeled database S^G s.t.

$$S^G_+ = R \bowtie_{\theta^G} P \quad \text{and} \quad S^G_- = (R \times P) \setminus (R \bowtie_{\theta^G} P).$$

Given a sample S we also identify all join predicates consistent with the sample

$$\mathcal{C}(S) = \{\theta \subseteq \Omega \mid S_+ \subseteq R \bowtie_\theta P \text{ and } S_- \cap R \bowtie_\theta P = \varnothing\}.$$

Initially, $S = \varnothing$, and hence, $\mathcal{C}(S) = \mathcal{P}(\Omega)$. Because S is consistent with θ^G, $\mathcal{C}(S)$ always contains θ^G. Ideally, we would like to devise a strategy of presenting elements of D to the user to get us "quickly" from $\varnothing$ to some S s.t. $\mathcal{C}(S) = \{\theta^G\}$.

3.3 Instance-equivalent join predicates

It is important to note that the content of the instance I may not be rich enough to allow the exact identification of the goal join predicate θ^G i.e., when $\mathcal{C}(S^G)$ contains elements other than θ^G. In such a case, we want to return to the user a join predicate which is *equivalent to θ^G w.r.t. the instance I*, and hence, indistinguishable by the user. We shall return $T(S_+)$ which is equivalent to θ^G over the instance I i.e., $(R \bowtie_{\theta^G} P)^I = (R \bowtie_{T(S_+)} P)^I$, and hence indistinguishable by the user.

For example, take the relations R_1 and P_1 below with their corresponding instances:

$$R_1 = \begin{array}{c|cc} & A_1 & A_2 \\ \hline t_1 & 1 & 1 \end{array} \qquad P_1 = \begin{array}{c|c} & B_1 \\ \hline t'_1 & 1 \end{array}$$

and $\theta^G_1 = \{(A_1, B_1)\}$. If we present the only tuple of the Cartesian product to the user, she labels it as a positive example, which yields the sample $S_1 = \{((t_1, t'_1), +)\}$. Then, $\mathcal{C}(S_1) = \mathcal{P}(attrs(R_1) \times attrs(P_1))$ and all its elements are equivalent to θ^G_1 w.r.t. this instance. In particular, in this case we return to the user the join predicate $T(S_{1,+}) = \{(A_1, B_1), (A_2, B_1)\}$, where $\theta^G_1 \subsetneq T(S_{1,+})$.

Another example when we return an instance-equivalent join predicate is when $R \bowtie_{\theta^G} P$ is empty, and therefore, the user labels all given tuples as negative examples. We also return $T(S_+)$, which in this case equals $\Omega = attrs(R) \times attrs(P)$, which again is equivalent to θ^G over I.

3.4 Uninformative tuples

In this section, we identify the tuples that do not yield new information when presented to the user. For this purpose, let us assume for a moment that the goal θ^G is known. We say that an *example* (t, α) from S^G is *uninformative* w.r.t. a sample S if $\mathcal{C}(S) = \mathcal{C}(S \cup \{(t, \alpha)\})$. In this case, we say that t is an *uninformative tuple* w.r.t. S. We denote by $Uninf(S)$

the set of all uninformative examples w.r.t. S:

$$Uninf(S) = \{(t,\alpha) \in S^G \mid \mathcal{C}(S) = \mathcal{C}(S \cup \{(t,\alpha)\})\}.$$

For instance, if we take instance of the relations R_0 and P_0 from Example 2.1, the goal join predicate $\theta_0^G = \{(A_2, B_3)\}$, and S_0 s.t. $S_{0,+} = \{(t_2, t_2')\}$ and $S_{0,-} = \{(t_1, t_3')\}$, then the examples $((t_4, t_1'), +)$ and $((t_2, t_1'), -)$ are uninformative.

Ideally, a smart inference algorithm should avoid presenting uninformative tuples to the user, but it is impossible to identify those tuples using the definition above without the knowledge of θ^G. This motivates us to introduce the notion of *certain tuples* w.r.t. a sample S, which is independent of the goal join predicate θ^G. Then we prove that the notions of uninformative and certain tuples are equivalent and we show that testing membership is in PTIME. We also mention that the notion of certain tuples is inspired by possible world semantics and certain answers [10] and already employed for XML querying for non-expert users by Cohen and Weiss [6]. Formally, given a sample S, we define the set $Cert(S)$ as follows:

$$Cert_+(S) = \{t \in D \mid \forall \theta \in \mathcal{C}(S).\ t \in R \bowtie_\theta P, \},$$
$$Cert_-(S) = \{t \in D \mid \forall \theta \in \mathcal{C}(S).\ t \notin R \bowtie_\theta P, \},$$
$$Cert(S) = Cert_+(S) \times \{+\} \cup Cert_-(S) \times \{-\}.$$

As already mentioned, we assume w.l.o.g. that all samples that we manipulate are consistent. In case of an inconsistent sample S, we have $\mathcal{C}(S) = \varnothing$, in which case the notion of certain tuples is of no interest. Next, we show that the notions of uninformative and certain tuples are equivalent.

Lemma 3.2 *Given a sample S, $Uninf(S) = Cert(S)$.*

Proof. First, we show the inclusion $Uninf(S) \subseteq Cert(S)$. *Case 1.* Take a tuple t s.t. $(t,+) \in Uninf(S)$. From the definition of $\mathcal{C}$ we know that for any join predicate θ from $\mathcal{C}(S \cup \{(t,+)\})$ it holds that $t \in R \bowtie_\theta P$. Because $\mathcal{C}(S) = \mathcal{C}(S \cup \{(t,+)\})$, we infer that for any join predicate θ from $\mathcal{C}(S)$ it holds that $t \in R \bowtie_\theta P$, and therefore, $t \in Cert_+(S)$. *Case 2.* Take a tuple t s.t. $(t,-) \in Uninf(S)$. From the definition of $\mathcal{C}$ we know that for any join predicate θ from $\mathcal{C}(S \cup \{(t,-)\})$ it holds that $t \notin R \bowtie_\theta P$. Because $\mathcal{C}(S) = \mathcal{C}(S \cup \{(t,-)\})$, we infer that for any join predicate θ from $\mathcal{C}(S)$ it holds that $t \notin R \bowtie_\theta P$, and therefore, $t \in Cert_-(S)$.

Next, we prove the inclusion $Cert(S) \subseteq Uninf(S)$. *Case 1.* Take a tuple t in $Cert_+(S)$, which means that for any join predicate θ in $\mathcal{C}(S)$ it holds that $t \in R \bowtie_\theta P$, which implies $\mathcal{C}(S) = \mathcal{C}(S \cup \{(t,+)\})$, hence $(t,+) \in Uninf(S)$. *Case 2.* Take a tuple t in $Cert_-(S)$, which means that for any join predicate θ in $\mathcal{C}(S)$ it holds that $t \notin R \bowtie_\theta P$, which implies $\mathcal{C}(S) = \mathcal{C}(S \cup \{(t,-)\})$, in other words $(t,-) \in Uninf(S)$. ▫

Next, we characterize the tuples from $Cert_+$.

Lemma 3.3 *Given a sample S and a tuple t from D, t belongs to $Cert_+(S)$ iff $T(S_+) \subseteq T(t)$.*

Proof. For the *if* part, assume $T(S_+) \subseteq T(t)$. From the definitions of $\mathcal{C}$ and T, we infer that for any θ in $\mathcal{C}(S)$, it holds that $\theta \subseteq T(S_+)$. This implies that for any θ in $\mathcal{C}(S)$, $\theta \subseteq T(t)$, hence $t \in R \bowtie_\theta P$, in other words $t \in Cert_+(S)$.

For the *only if* part, assume $t \in Cert_+(S)$, which means that for any $\theta \in \mathcal{C}(S)$ it holds that $t \in R \bowtie_\theta P$. From the definitions of $\mathcal{C}$ and T, we infer that $T(S_+) \in \mathcal{C}(S)$, and therefore, $t \in R \bowtie_{T(S_+)} P$, which yields $T(S_+) \subseteq T(t)$. ▫

We also characterize the tuples from $Cert_-$.

Lemma 3.4 *Given a sample S and a tuple t from D, t belongs to $Cert_-(S)$ iff there exists a tuple t' in S_- s.t. $T(S_+) \cap T(t) \subseteq T(t')$.*

Proof. For the *if* part, take a tuple t' in S_- s.t. $T(S_+) \cap T(t) \subseteq T(t')$. This implies that for any θ in $\mathcal{C}(S \cup \{(t,+)\})$ it holds that $t' \in R \bowtie_\theta P$, hence $\mathcal{C}(S \cup \{(t,+)\}) = \varnothing$. Because $\mathcal{C}(S \cup \{(t,+)\}) \cup \mathcal{C}(S \cup \{(t,-)\}) = \mathcal{C}(S)$, we obtain $\mathcal{C}(S \cup \{(t,-)\}) = \mathcal{C}(S)$, which means that $(t,-) \in Uninf(S)$, and therefore, $t \in Cert_-(S)$.

For the *only if* part, assume $t \in Cert_-(S)$, which means that for any θ in $\mathcal{C}(S)$ it holds that $t \notin R \bowtie_\theta P$. This implies that $\mathcal{C}(S) \subseteq \{\theta \subseteq \Omega \mid t \notin R \bowtie_\theta P\}$. Moreover, given a sample S, note that the set $\mathcal{C}(S)$ can be expressed equivalently as $\mathcal{P}(T(S_+)) \setminus (\bigcup_{t' \in S_-} \mathcal{P}(T(t')))$, which yields

$$\mathcal{P}(T(S_+)) \setminus (\bigcup_{t' \in S_-} \mathcal{P}(T(t'))) \subseteq \{\theta \subseteq \Omega \mid t \notin R \bowtie_\theta P\} \quad (*).$$

From the definition of $\mathcal{C}$ we infer that for any $\theta \in \mathcal{P}(\Omega) \setminus \mathcal{C}(S)$ none of the join predicates $\theta' \subseteq \theta$ belongs to $\mathcal{C}(S)$. From this remark and (*) we conclude that there exists a tuple t' in S_- s.t. $T(S_+) \cap T(t) \subseteq T(t')$. ▫

Recall that we have defined $Uninf(S)$ w.r.t. the goal join predicate, then we have shown in Lemma 3.2 that $Uninf(S) = Cert(S)$, which means that we are able to characterize the uninformative tuples by using only the given sample, without having the knowledge of the goal join predicate. Furthermore, given a sample S and a tuple t from D, deciding whether $(t,+)$ belongs to $Uninf(S)$ can be done in polynomial time using the characterization from Lemma 3.3. Similarly, deciding whether $(t,-)$ belongs to $Uninf(S)$ can be done in polynomial time due to Lemma 3.4. We say that a tuple t from D is *informative* w.r.t. a sample S if there does not exist a label $\alpha \in \{+,-\}$ s.t. $(t,\alpha) \in S$ or $(t,\alpha) \in Uninf(S)$. When the sample S is clear from the context, we may write simply that a tuple is informative instead of informative w.r.t. S. Using all remarks above, we can state the main result of this section.

Theorem 3.5 *Testing if a tuple is informative is in PTIME.*

4. STRATEGIES

In this section, we use the developments from the previous section to propose efficient strategies for interactively presenting tuples to the user. We introduce first the *general inference algorithm*, then we claim that there exists an *optimal strategy* that is however exponential. Consequently, we propose several *efficient strategies* that we essentially classify in two categories: *local* and *lookahead*.

4.1 General inference algorithm

A *strategy* Υ is a function which takes as input a Cartesian product D and a sample S, and returns a tuple t in D. The *general inference algorithm* (Algorithm 1) consists of selecting a tuple w.r.t. a *strategy* Υ and asking the user to label it as a positive or negative example; this process continues until the *halt condition* Γ is satisfied. The algorithm continuously verifies the consistency of the sample, if at any moment the user labels a tuple s.t. the sample becomes inconsistent, the algorithm raises an exception.

We have chosen to investigate strategies that ask the user to label informative tuples only because we aim to minimize the number of interactions. Therefore, the sample that we incrementally construct is always consistent and our approach does not yield any error in lines 6-7. In our approach, we choose the strongest halt condition i.e., to stop the interactions when there is no informative tuple left:

$$\Gamma := \forall t \in D.\ \exists \alpha \in \{+,-\}.\ (t,\alpha) \in S \cup \mathit{Uninf}(S).$$

At the end of the interactive process, we return the inferred join predicate $\theta = T(S_+)$ i.e., the most specific join predicate consistent with the examples provided by the user. However, the *halt condition* Γ may be weaker in practice, as the user might decide to stop the interactive process at an earlier time if, for instance, she finds the most specific consistent query $T(S_+)$ to be satisfactory.

Algorithm 1 General inference algorithm.

Input: the Cartesian product D
Output: a join predicate consistent with the user's labels
Parameters: strategy Υ, halt condition Γ

1: **let** $S = \varnothing$
2: **while** $\neg\Gamma$ **do**
3: **let** $t = \Upsilon(D, S)$
4: **query** the user about the label α for t
5: $S := S \cup \{(t, \alpha)\}$
6: **if** S is not consistent (with $T(S_+)$) **then**
7: **error**
8: **return** $T(S_+)$

An optimal strategy exists and can be built by employing the standard construction of a minimax tree [13]. While the exact complexity of the optimal strategy remains an open question, a straightforward implementation of minimax requires exponential time (and is in PSPACE), which unfortunately renders it unusable in practice. As a consequence, we propose a number of time-efficient strategies that attempt to minimize the number of interactions with the user. All of the proposed strategies are based on the notion of lattice of join predicates, which we present next. For comparison we also introduce the *random strategy* (RND) which chooses randomly an informative tuple.

4.2 Lattice of join predicates

The *lattice of the join predicates* is $(\mathcal{P}(\Omega), \subseteq)$ with $\varnothing$ as its bottom-most node and Ω as its top-most node. We focus on *non-nullable* join predicates i.e., join predicates that select at least one tuple, because we expect the user to label at least one positive example during the interactive process. We also consider Ω in case the user decides to label all tuples as negative. Naturally, the number of non-nullable join predicates may still be exponential since all join predicates are non-nullable iff there exist two tuples $t \in R$ and $t' \in P$ s.t. $t[A_1] = \ldots = t[A_n] = t'[B_1] = \ldots = t'[B_m]$.

Figure 4 presents the non-nullable nodes (and Ω) of the lattice corresponding to the instance from Example 2.1. We point out a correspondence between the nodes and the tuples in the Cartesian product D: a tuple $t \in D$ corresponds to a node of the lattice θ if $T(t) = \theta$. Not every node of the lattice has corresponding tuples and in Figure 4 only nodes in boxes have corresponding tuples (cf. Figure 3).

In the remainder of this section, we present the strategies of presenting tuples to the user. The main insight is that we propagate labels in the lattice using Lemma 3.3 (for positive examples) and Lemma 3.4 (for negative examples), which allow us to *prune* parts of the lattice corresponding to the tuples that become uninformative. Basically, labeling a tuple t corresponding to a node θ as positive renders tuples corresponding to all nodes above θ uninformative and possibly some other nodes depending on tuples labeled previously. Conversely, labeling t as negative prunes at least the part of the lattice below θ. For instance, take the lattice from Figure 4, assume an empty sample, and take the join predicate $\{(A_1, B_2), (A_1, B_3)\}$ and the corresponding tuple $t^\circ = (t_1, t'_3)$. If the user labels t° as a positive example, then the tuple (t_2, t'_3) corresponding to $\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$ becomes uninformative (cf. Lemma 3.3). On the other hand, if the user labels the tuple t° as a negative example, then the tuples (t_2, t'_1) and (t_3, t'_1) corresponding to $\{(A_1, B_3)\}$ and $\varnothing$ respectively, become uninformative (cf. Lemma 3.4). If we reason at the lattice level, the question "Which is the next tuple to present to the user?" intuitively becomes "Labeling which tuple allows us to prune as much of the lattice as possible?"

4.3 Local strategies

The principle behind the *local strategies* is that they propose tuples to the user following a simple order on the lattice. We call these strategies local because they do not take into account the quantity of information that labeling an informative tuple could bring to the inference process. As such, they differ from the lookahead strategies that we present in the next section. In this section we propose two local strategies, which essentially correspond to two basic variants of *navigating* in the lattice: the *bottom-up strategy* and the *top-down strategy*.

The *bottom-up strategy* (BU) (Algorithm 2) intuitively navigates the lattice of join predicates from the most general join predicate ($\varnothing$) towards the most specific one (Ω). It visits a minimal node of the lattice that has a corresponding informative tuple and asks the user to label it. If the label is positive, (at least) the part of the lattice above the node is pruned. If the label is negative, the current node is pruned (since the nodes below are not informative, they must have been pruned before). Recall the instance from Example 2.1 and its corresponding lattice in Figure 4. The BU strategy asks the user to label the tuple $t_0 = (t_3, t'_1)$ corresponding to $\varnothing$. If the label is positive, all nodes of the lattice are pruned and the empty join predicate returned. If the label is negative, the strategy selects the tuple (t_2, t'_1) corresponding to the node $\theta_1 = \{(A_1, B_3)\}$ for labeling, etc. The BU strategy discovers quickly the goal join predicate $\varnothing$, but is inadequate to discover join predicates of bigger size. In the worst case, when the user provides only negative examples, the BU strategy might ask the user to label every tuple from the Cartesian product.

Algorithm 2 Bottom-up strategy $\mathrm{BU}(D, S)$

1: **let** $m = \min(\{|T(t)| \mid t \in D \text{ s.t. } t \text{ is informative}\})$
2: **return** informative t s.t. $|T(t)| = m$

The *top-down strategy* (TD) (Algorithm 3) intuitively starts to navigate in the lattice of join predicates from the most specific join predicate (Ω) to the most general one ($\varnothing$). It has basically two behaviors depending on the contents of the current sample. First, when there is no positive example yet

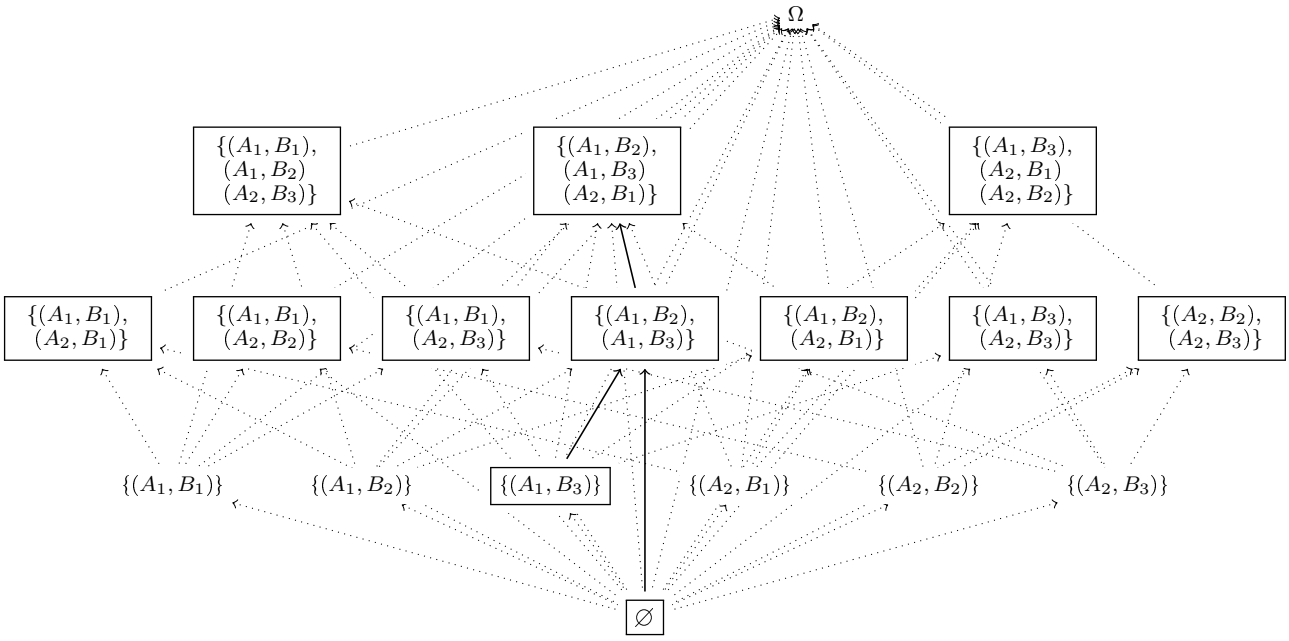


Figure 4: Lattice of join predicates for the instance from Example 2.1

(lines 1-2), this strategy chooses a tuple t corresponding to a $\subseteq$-maximal join predicate i.e., whose $T(t)$ has no other non-nullable join predicate above it in the lattice (line 2). For example, for the instance corresponding to the lattice from Figure 4, we first ask the user to label the tuple corresponding to $\{(A_1, B_1), (A_1, B_2), (A_2, B_3)\}$, then the tuple corresponding to $\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$, etc. Note that the relative order among these tuples corresponding to $\subseteq$-maximal join predicates is arbitrary. If the user labels all $\subseteq$-maximal join predicates as negative examples, we are able to infer the goal Ω without asking her to label all the Cartesian product (using Lemma 3.4). Thus, the TD strategy overcomes the mentioned drawback of the BU. On the other hand, if there is at least one positive example, then the goal join predicate is a non-nullable one, and the TD strategy turns into BU (lines 3-5). As we later show in Section 5, the TD strategy seems a good practical compromise.

Algorithm 3 Top-down strategy TD(D, S)

```
1: if S+ = ∅ then
2:     return informative t s.t. ∄t' ∈ D. T(t) ⊊ T(t')
3: else
4:     let m = min({|T(t)| | t ∈ D s.t. t is informative})
5:     return informative t s.t. |T(t)| = m
```

4.4 Lookahead strategies

In this section, we present the *lookahead strategies*. The key difference between them and the local strategies is that the lookahead strategies take into account the quantity of information that labeling an informative tuple could bring to the process of inference. We need to introduce first some auxiliary notions. Given an informative tuple t from D and a sample S, let $u^{\alpha}_{t,S}$ be the number of tuples which become uninformative if the tuple t is labeled with α:

$$u^{\alpha}_{t,S} = |Uninf(S \cup \{(t, \alpha)\}) \setminus Uninf(S)|.$$

Now, the entropy of an informative tuple t w.r.t. a sample S, denoted $entropy_S(t)$, is the pair $(\min(u^+_{t,S}, u^-_{t,S}), \max(u^+_{t,S}, u^-_{t,S}))$, which captures the quantity of information that labeling the tuple t can provide. The entropy of uninformative tuples is undefined, however we never make use of it. In Figure 5 we present the entropy for each tuple from the Cartesian product of the instance from Example 2.1, for an empty sample.

	T	$u^+_{t,S}$	$u^-_{t,S}$	$entropy_S$
(t_1, t'_1)	$\{(A_1, B_3), (A_2, B_1), (A_2, B_2)\}$	0	2	(0,2)
(t_1, t'_2)	$\{(A_1, B_1), (A_2, B_2)\}$	0	1	(0,1)
(t_1, t'_3)	$\{(A_1, B_2), (A_1, B_3)\}$	1	2	**(1,2)**
(t_2, t'_1)	$\{(A_1, B_3)\}$	2	1	**(1,2)**
(t_2, t'_2)	$\{(A_1, B_1), (A_2, B_3)\}$	1	1	(1,1)
(t_2, t'_3)	$\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$	0	4	(0,4)
(t_3, t'_1)	$\varnothing$	11	0	(0,11)
(t_3, t'_2)	$\{(A_1, B_3), (A_2, B_3)\}$	0	2	(0,2)
(t_3, t'_3)	$\{(A_1, B_1), (A_2, B_1)\}$	0	1	(0,1)
(t_4, t'_1)	$\{(A_1, B_1), (A_1, B_2), (A_2, B_3)\}$	0	2	(0,2)
(t_4, t'_2)	$\{(A_1, B_2), (A_2, B_1)\}$	1	1	(1,1)
(t_4, t'_3)	$\{(A_2, B_2), (A_2, B_3)\}$	0	1	(0,1)

Figure 5: The Cartesian product corresponding to the instance from Example 2.1 and the entropy for each tuple, for an initial empty sample.

Given two entropies $e = (a, b)$ and $e' = (a', b')$, we say that e *dominates* e' if $a \geqslant a'$ and $b \geqslant b'$. For example, $(1, 2)$ dominates $(1, 1)$ and $(0, 2)$, but it does not dominate $(2, 2)$ nor $(0, 3)$. Next, given a set of entropies E, we define the *skyline* of E, denoted $skyline(E)$, as the set of entropies e that are not dominated by any other entropy of E. For example, for the set of entropies of the tuples from Figure 5, the skyline is $\{(1, 2), (0, 11)\}$.

Next, we present the *one-step lookahead skyline strategy* (L^1S) (Algorithm 4). We illustrate this strategy for the instance from Example 2.1, for an initial empty sample. First (line 1), we compute the entropy for each informative tuple from the Cartesian product. This corresponds to computing the last column from Figure 5. Then (line 2), we calculate the maximal value among all minimal values of the entropies computed at the previous step. For our example, this value is 1. Finally (lines 3-4), we return an informative tuple whose entropy is in the skyline of all entropies, and moreover, has as minimal value the number computed at the previous step. For our example, the skyline is $\{(1,2),(0,11)\}$, thus the entropy corresponding to the value computed previously (i.e., 1) is $(1,2)$. Consequently, we return one of the tuples having the entropy $(1,2)$, more precisely either (t_1,t'_3) or (t_2,t'_1). Intuitively, according to L^1S strategy, we choose to ask the user to label a tuple which permits to eliminate at least one and at most two additional tuples. Note that by min (resp. max) we denote the minimal (resp. maximal) value from either a given set or a given pair of numbers, depending on the context.

Algorithm 4 One-step lookahead skyline $L^1S(D,S)$

1: **let** $E = \{entropy_S(t) \mid t \in D$ s.t. t is informative$\}$
2: **let** $m = \max(\{\min(e) \mid e \in E\})$
3: **let** e the entropy in $skyline(E)$ s.t. $\min(e) = m$
4: **return** informative t s.t. $entropy_S(t) = e$

The L^1S strategy naturally extends to *k-steps lookahead skyline strategy* (L^kS). The difference is that instead of taking into account the quantity of information that labeling *one* tuple could bring to the inference process, we take into account the quantity of information for labeling k tuples. Note that if k is greater than the total number of informative tuples in the Cartesian product, then the strategy becomes optimal and thus inefficient. For such a reason, in the experiments we focus on a lookahead of two steps, which is a good trade-off between keeping a relatively low computation time and minimizing the number of interactions. Therefore, we present such strategy in the remainder.

Algorithm 5 $entropy^2_S(t)$

1: **for** $\alpha \in \{+,-\}$ **do**
2: **let** $S' = S \cup \{(t,\alpha)\}$
3: **if** $\nexists t' \in D$ s.t. t' is informative w.r.t. S' **then**
4: **let** $e_\alpha = (\infty,\infty)$
5: **continue**
6: **let** $E = \varnothing$
7: **for** $t' \in D$ s.t. i is informative w.r.t. S' **do**
8: **let** $u^+ = |Uninf(S \cup \{(t,\alpha),(t',+)\}) \setminus Uninf(S)|$
9: **let** $u^- = |Uninf(S \cup \{(t,\alpha),(t',-)\}) \setminus Uninf(S)|$
10: $E := E \cup \{(\min(u^+,u^-),\max(u^+,u^-))\}$
11: **let** $m = \max(\{\min(e) \mid e \in E\})$
12: **let** e_α the entropy in $skyline(E)$ s.t. $\min(e_\alpha) = m$
13: **let** $m = \min(\{\min(e_+),\min(e_-)\})$
14: **return** e_α s.t. $\min(e_\alpha) = m$

We need to extend first the notion of entropy of a tuple to the notion of $entropy^2$ of a tuple. Given an informative tuple t and a sample S, the $entropy^2$ of t w.r.t. S, denoted $entropy^2_S(t)$, intuitively captures the minimal quantity of information that labeling t and another tuple can bring to the inference process. The construction of the $entropy^2$ is quite technical (Algorithm 5) and we present an example below. Take the sample $S = \{((t_1,t'_3),+),((t_3,t'_1),-)\}$. Note that $Uninf(S) = \{((t_2,t'_3),+),((t_1,t'_2),-),((t_2,t'_2),-),((t_3,t'_3),-),((t_4,t'_3),-)\}$. There are five informative tuples left: (t_1,t'_1), (t_2,t'_1), (t_3,t'_2), (t_4,t'_1), and (t_4,t'_2). Let compute now the $entropy^2$ of (t_2,t'_1) w.r.t. S using Algorithm 5. First, take $\alpha = +$ (line 1), then $S' = S \cup \{((t_2,t'_1),+)\}$ (line 2), note that there is no other informative tuple left (line 3), and therefore, $e_+ = (\infty,\infty)$ (lines 4-5). This intuitively means that given the sample S, if the user labels the tuple (t_2,t'_1) as positive example, then there is no informative tuple left and we can stop the interactions. Next, take $\alpha = -$ (line 1), then $S' = S \cup \{((t_2,t'_1),-)\}$ (line 2), and the only tuples informative w.r.t. S' are (t_4,t'_1) and (t_4,t'_2), we obtain $E = \{(3,3)\}$ (lines 6-10), and $e_- = (3,3)$ (lines 11-12). Finally, $entropy^2_S((t_2,t'_1)) = (3,3)$ (lines 13-14), which means that if we ask the user to label the tuple (t_2,t'_1) and any arbitrary tuple afterwards, then there are at least three other tuples that become uninformative. The computation of the entropies of the other informative tuples w.r.t. S is done in a similar manner. The *2-steps lookahead skyline strategy* (L^2S) (Algorithm 6) returns a tuple corresponding to the "best" $entropy^2$ in a similar manner to L^1S. In fact, Algorithm 6 has been obtained from Algorithm 4 by simply replacing *entropy* by $entropy^2$. As we have already mentioned, the approach can be easily generalized to $entropy^k$ and L^kS, respectively.

Algorithm 6 2-steps lookahead skyline $L^2S(D,S)$

1: **let** $E = \{entropy^2_S(t) \mid t \in D$ s.t. t is informative$\}$
2: **let** $m = \max(\{\min(e) \mid e \in E\})$
3: **let** e the entropy in $skyline(E)$ s.t. $\min(e) = m$
4: **return** informative t s.t. $entropy^2_S(t) = e$

5. EXPERIMENTS

In this section, we present an experimental study devoted to proving the efficiency and effectiveness of our join inference strategies. Precisely, we compare the three classes of strategies presented above: the random strategy (RND), the local strategies (BU and TD), and the lookahead strategies (L^1S and L^2S). For each database instance I and for each goal join predicate θ, we have considered two measures: the *number of user interactions* (i.e., the number of tuples that need to be presented to the user in order to infer the join predicate), and the total *time* needed to infer the goal join predicate, using each of the above strategies as strategy Υ (cf. Section 4.1), and reiterating the user interactions until no informative tuple is left (halt condition Γ).

In the experiments, we have used two datasets: the TPC-H benchmark datasets (Section 5.1) and randomly generated synthetic datasets that we have built (Section 5.2). For the synthetic datasets we have used all non-nullable join predicates (cf. Section 4.2) as goal predicates, while for TPC-H we could settle some specific goals, as suggested by the key-foreign key relationships. We show in Table 1 the description of all datasets along with the summary of results.

Our algorithms have been implemented in Python. All our experiments were run on an Intel Core i7 with 4×2.9 GHz CPU and 8 GB RAM.

5.1 Setup of experiments on TPC-H

We have considered the following goal join predicates over the TPC-H benchmark [16]:

Join 1: *Part[Partkey] = Partsupp[Partkey]*,

Join 2: *Supplier[Suppkey] = Partsupp[Suppkey]*,

Join 3: *Customer[Custkey] = Orders[Custkey]*,

Join 4: *Orders[Orderkey] = Lineitem[Orderkey]*,

Join 5: *Partsupp[Partkey] = Lineitem[Partkey]* $\wedge$
$\wedge$ *Partsupp[Suppkey] = Lineitem[Suppkey]*.

They indeed correspond to key-foreign key relationships between different combinations of relations from TPC-H. Note that the strategies are not aware of these constraints and select tuples to present to the user only by reasoning on the user annotations. The goal of such experiments on TPC-H is to evict the goal join predicates that rely on integrity constraints. It may easily happen in the benchmark to have other attributes that match with the keys and foreign keys as they exhibit compatible types. For instance, a value "15" of an attribute of a tuple may as well represent a key, a size, a price, or a quantity, etc. We have repeated these experiments on all scaling factors (SF) for TPC-H i.e., in the interval between 1 and 100000. For conciseness, we only present the results on the minimum and maximum scaling factors in Figure 6 and a summary of these TPC-H experiments in the top half of the table from Table 1. We discuss these experiments along with the synthetic ones in Section 5.3.

5.2 Setup of experiments on synthetic data

Since the size of TPC-H join predicates is only 1 or 2, we have implemented a synthetic datasets generator. The goal of these experiments was to tweak our strategies on various sizes of the join predicate. A *configuration* of our generator is a quadruple $(|attrs(R)|, |attrs(P)|, l, v)$, where $|attrs(R)|$ (resp. $|attrs(P)|$) is the number of attributes in the relation R (resp. P), l is the number of tuples in the instance of each relation, and v is the number of possible values of the attributes of the relations. For example, the configuration $(3, 3, 50, 100)$ generates instances of two relations s.t. each of them has 3 attributes and 50 lines, and the values in the instance can be only numbers in the set $\{0, 1, \ldots, 99\}$. The results presented for our synthetic datasets are obtained after averaging over 100 runs. We include in Figure 7 and the bottom half of Table 1 the results for the following six configurations: $(3, 3, 100, 100)$, $(3, 3, 50, 100)$, $(3, 4, 50, 100)$, $(2, 5, 50, 100)$, $(2, 4, 50, 50)$, and $(2, 4, 50, 100)$. The first two of them are particularly interesting in practice as they could represent triples of RDF stores.

5.3 Discussion

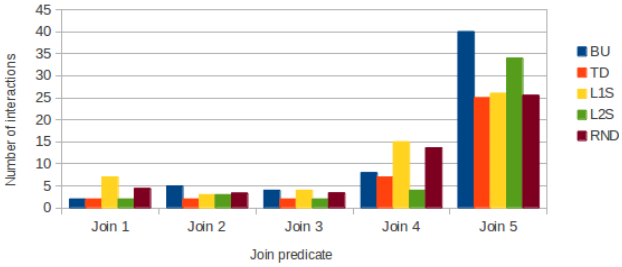
In this section, we discuss the experimental results for the two settings presented above. We first introduce an additional notion. Given an instance I, the *join ratio* of I is the average of the sizes of the unique join predicates θ for which there exists a tuple t in the Cartesian product s.t. $T(t) = \theta$. Formally, let $N = \{\theta \in \mathcal{P}(\Omega) \mid \exists t \in D.\ T(t) = \theta\}$. Then, the join ratio of I is $(\sum_{\theta \in N} |\theta|)/(|N|)$. In the above definition, unique join predicates denotes predicates selecting tuples and considered only once. Indeed, if two tuples are selected by the same most specific join predicate, then they are basically equivalent w.r.t. the inference process. In such a case, we will consider such a predicate only once. For example, the instance from Example 2.1 has a total of 12 tuples in the Cartesian product, each of them yielding a unique most specific join predicate: 1 of size 0, 1 of size 1, 7 of size 2, and 3 of size 3. Thus, the join ratio of this instance is $(0 + 1 + 7 \times 2 + 3 \times 3)/12 = 2$. The join ratio intuitively captures the complexity of an instance i.e., the bigger the join ratio of an instance is, the more non-nullable join predicates are in the lattice (cf. Section 4.2), and therefore, more interactions are needed to infer a join predicate on that instance.

We show in Table 1 the summary of the experimental results. We can observe that in the majority of cases TD and L^2S are better than the other strategies w.r.t. minimizing the number of interactions. However, none of them seems to win over the other. In fact, their performance essentially depends on both the size of the goal join predicate and the join ratio of the instance. Concerning the size of the goal join predicate, note that a goal join predicate of smaller size can be generally inferred with less interactions. For example, in Figure 6 for the TPC-H benchmark, four joins have size 1 while the fifth one has size 2 (cf. Section 5.1). For both presented scaling factors, join predicates of size 1 are inferred with less interactions than the join predicate of size 2. A similar behavior can be observed on the synthetic datasets in Figure 7, where joins of size 0 and 1 are inferred with less interactions than joins of size greater or equal to 2. Moreover, if we look more in detail at the joins of size greater or equal to 2, we observe that in fact those of size 2 need slightly more interactions than those of size 3 and 4. Intuitively, this happens because in the lattice corresponding to the synthetic instances, non-nullable join predicates have sizes between 0 and 4. Thus, the joins of size 2 are somewhere in the intermediate part of the lattice, and therefore much more difficult to infer.

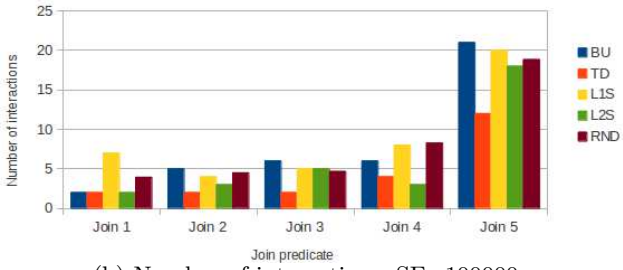
Next, let us discuss how the number of interactions needed for each strategy depends on the size of the goal join predicate and the impact of the join ratio. Trivially, the goal join predicate of size 0 (i.e., $\varnothing$), can be inferred using only one interaction, thus making the BU the best strategy for it as expected (cf. Section 4.3). By opposite, for goal join predicates of size greater or equal to 1, TD and L^2S give the smallest number of interactions. In particular, for the goal join predicates of size 1, L^2S is better than TD unless the join ratio is very small (i.e., around 1). A small join ratio means that there are very few non-nullable join predicates in the lattice hence the lookahead might not be necessarily useful. Of course, L^2S exhibits a better performance than TD in all those cases when the join ratio is more significant. A bigger join ratio entails more non-nullable join predicates in the lattice, and therefore, inferring a join predicate requires more interactions. More precisely, L^2S is the best strategy for joins of size 1 and joins of size 3 and 4. This trend is confirmed by experiments on both kinds of datasets[1]. By opposite, for joins of size 2 TD is the best strategy as it requires the smallest number of interactions for both synthetic data and TPC-H data.

Finally, the time to execute the strategies stays reasonable for all the strategies, within the order of seconds. Of course, the L^2S strategy is the most expensive compared to the other strategies, but its worst run is not more than 73.57 seconds.

[1]Notice that joins of size 3 and 4 occur only within the synthetic data, since in TPC-H such joins do not make sense (cf. Section 5.1).



(a) Number of interactions, SF=1.



(b) Number of interactions, SF=100000.

	BU	TD	L^1S	L^2S	RND
Join 1	0.001	0.001	0.015	0.072	0.001
Join 2	0.001	0.001	0.008	0.046	0.001
Join 3	0.001	0.001	0.01	0.042	0.001
Join 4	0.012	0.01	3.452	56.167	0.013
Join 5	0.019	0.014	2.53	73.57	0.013

(c) Inference time (seconds), SF=1.

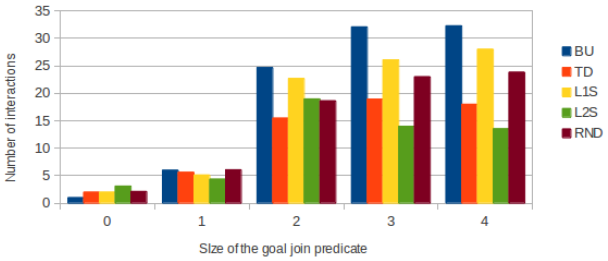
	BU	TD	L^1S	L^2S	RND
Join 1	0.001	0.001	0.017	0.072	0.001
Join 2	0.001	0.001	0.013	0.074	0.001
Join 3	0.001	0.001	0.006	0.033	0.001
Join 4	0.007	0.004	0.627	9.694	0.006
Join 5	0.004	0.003	0.312	4.423	0.004

(d) Inference time (seconds), SF=100000.

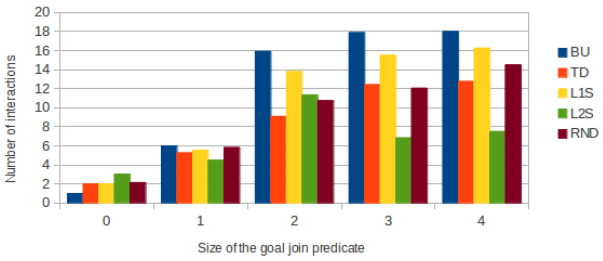
Figure 6: Number of interactions and inference time for TPC-H experiments for two SF.

			Size of the Cartesian product (number of tuples)	*Join ratio*	*Best strategy (w.r.t. number of interactions)*	*Time of best strategy (in seconds)*
TPC-H experiments	SF = 1	Join 1 (size 1)	2.5×10^5	1	BU/TD/L^2S (2 int.)	0.001/0.001/0.072
		Join 2 (size 1)	2.5×10^5	1	TD (2 int.)	0.001
		Join 3 (size 1)	2.5×10^6	1.142	TD/L^2S (2 int.)	0.001/0.042
		Join 4 (size 1)	9.1×10^7	2.109	L^2S (4 int.)	56.167
		Join 5 (size 2)	9.1×10^6	1.681	TD (25 int.)	0.014
	SF = 100000	Join 1 (size 1)	2.5×10^5	1	BU/TD/L^2S (2 int.)	0.001/0.001/0.072
		Join 2 (size 1)	2.5×10^5	1	TD (2 int.)	0.001
		Join 3 (size 1)	1.5×10^7	1.166	TD (2 int.)	0.001
		Join 4 (size 1)	9.6×10^8	2.03	L^2S (3 int.)	9.694
		Join 5 (size 2)	1.5×10^7	1.523	TD (12 int.)	0.003
Synthetic dataset experiments	(3, 3, 100, 100)	Joins of size 0	10^4	1.647	BU (1 int.)	0.002
		Joins of size 1	10^4	1.647	L^2S (4 int.)	8.95
		Joins of size 2	10^4	1.647	TD (15 int.)	0.006
		Joins of size 3	10^4	1.647	L^2S (14 int.)	10.241
		Joins of size 4	10^4	1.647	L^2S (13 int.)	9.924
	(3, 3, 50, 100)	Joins of size 0	2.5×10^3	1.341	BU (1 int.)	0.001
		Joins of size 1	2.5×10^3	1.341	L^2S (4 int.)	1.373
		Joins of size 2	2.5×10^3	1.341	TD (9 int.)	0.002
		Joins of size 3	2.5×10^3	1.341	L^2S (7 int.)	1.28
		Joins of size 4	2.5×10^3	1.341	L^2S (8 int.)	1.332
	(3, 4, 50, 100)	Joins of size 0	2.5×10^3	1.458	BU (1 int.)	0.001
		Joins of size 1	2.5×10^3	1.458	L^2S (5 int.)	6.698
		Joins of size 2	2.5×10^3	1.458	TD (13 int.)	0.004
		Joins of size 3	2.5×10^3	1.458	L^2S (10 int.)	7.1
		Joins of size 4	2.5×10^3	1.458	L^2S (9 int.)	7.344
	(2, 5, 50, 100)	Joins of size 0	2.5×10^3	1.377	BU (1 int.)	0.001
		Joins of size 1	2.5×10^3	1.377	L^2S (5 int.)	2.502
		Joins of size 2	2.5×10^3	1.377	TD (10 int.)	0.003
		Joins of size 3	2.5×10^3	1.377	L^2S (9 int.)	2.859
		Joins of size 4	2.5×10^3	1.377	L^2S (10 int.)	3.719
	(2, 4, 50, 50)	Joins of size 0	2.5×10^3	1.596	BU (1 int.)	0.004
		Joins of size 1	2.5×10^3	1.596	L^2S (4 int.)	10.71
		Joins of size 2	2.5×10^3	1.596	TD (13 int.)	0.011
		Joins of size 3	2.5×10^3	1.596	L^2S (13 int.)	14.058
		Joins of size 4	2.5×10^3	1.596	L^2S (13 int.)	14.177
	(2, 4, 50, 100)	Joins of size 0	2.5×10^3	1.633	BU (1 int.)	0.001
		Joins of size 1	2.5×10^3	1.633	L^2S (4 int.)	0.666
		Joins of size 2	2.5×10^3	1.633	TD (8 int.)	0.001
		Joins of size 3	2.5×10^3	1.633	L^2S (7 int.)	0.954
		Joins of size 4	2.5×10^3	1.633	L^2S (9 int.)	1.072

Table 1: Description and summary of all experiments.



(a) Number of interactions, $(3, 3, 100, 100)$.



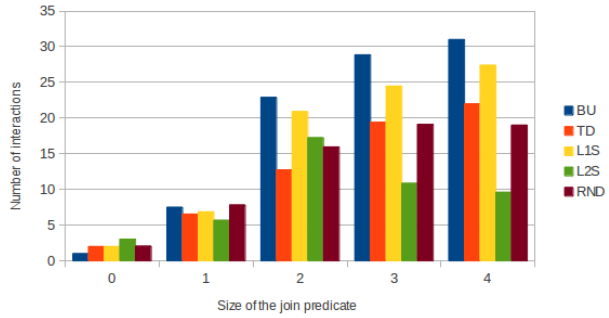
(b) Number of interactions, $(3, 3, 50, 100)$.

$\lvert\theta^G\rvert$	BU	TD	L^{1}S	L^{2}S	RND
0	0.002	0.002	0.127	6.147	0.002
1	0.004	0.004	0.335	8.95	0.004
2	0.008	0.006	0.916	17.648	0.006
3	0.01	0.008	1.085	10.241	0.008
4	0.01	0.008	1.132	9.924	0.008

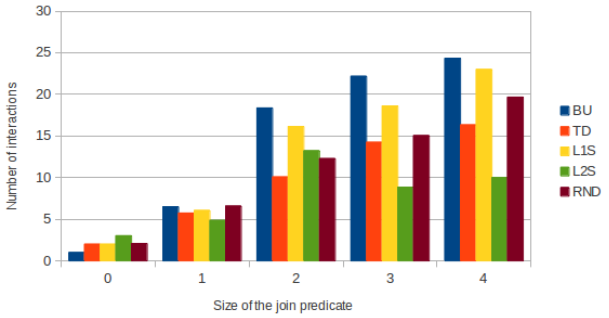
(c) Inference time (seconds), $(3, 3, 100, 100)$.

$\lvert\theta^G\rvert$	BU	TD	L^{1}S	L^{2}S	RND
0	0.001	0.001	0.04	0.999	0.001
1	0.002	0.002	0.097	1.373	0.002
2	0.003	0.002	0.189	2.19	0.002
3	0.003	0.002	0.185	1.28	0.002
4	0.003	0.002	0.185	1.332	0.003

(d) Inference time (seconds), $(3, 3, 50, 100)$.



(e) Number of interactions, $(3, 4, 50, 100)$.



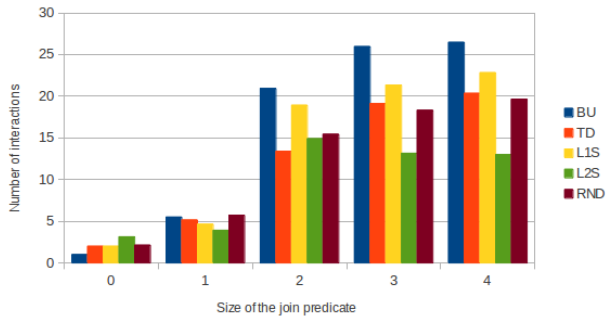
(f) Number of interactions, $(2, 5, 50, 100)$.

$\lvert\theta^G\rvert$	BU	TD	L^{1}S	L^{2}S	RND
0	0.001	0.001	0.1	3.949	0.001
1	0.004	0.003	0.32	6.698	0.003
2	0.007	0.004	0.693	11.26	0.005
3	0.008	0.006	0.856	7.1	0.006
4	0.01	0.007	1.049	7.344	0.006

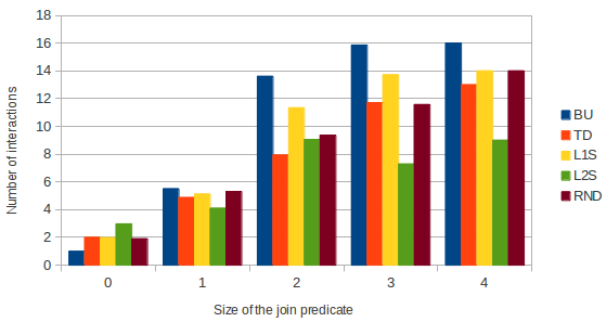
(g) Inference time (seconds), $(3, 4, 50, 100)$.

$\lvert\theta^G\rvert$	BU	TD	L^{1}S	L^{2}S	RND
0	0.001	0.001	0.057	1.718	0.001
1	0.002	0.002	0.155	2.502	0.002
2	0.004	0.003	0.316	4.074	0.003
3	0.005	0.004	0.385	2.859	0.004
4	0.006	0.004	0.516	3.719	0.005

(h) Inference time (seconds), $(2, 5, 50, 100)$.



(i) Number of interactions, $(2, 4, 50, 50)$.



(j) Number of interactions, $(2, 4, 50, 100)$.

$\lvert\theta^G\rvert$	BU	TD	L^{1}S	L^{2}S	RND
0	0.004	0.005	0.216	8.739	0.005
1	0.008	0.008	0.505	10.71	0.008
2	0.016	0.011	1.306	18.713	0.012
3	0.019	0.015	1.492	14.058	0.014
4	0.019	0.015	1.576	14.177	0.014

(k) Inference time (seconds), $(2, 4, 50, 50)$.

$\lvert\theta^G\rvert$	BU	TD	L^{1}S	L^{2}S	RND
0	0.001	0.001	0.027	0.544	0.001
1	0.001	0.001	0.059	0.666	0.001
2	0.002	0.001	0.112	1.046	0.002
3	0.003	0.002	0.138	0.954	0.002
4	0.003	0.002	0.141	1.072	0.002

(l) Inference time (seconds), $(2, 4, 50, 100)$.

Figure 7: Number of interactions and inference time for six synthetic datasets. A configuration is: ($|attrs(R)|$, $|attrs(P)|$, number of tuples in the instance of each table, number of possible values).

6. INTRACTABILITY OF SEMIJOINS

In this section, we approach the problem of inferring *semijoin predicates* from examples given by the user. First, we have to adapt some notions that we have previously introduced in Section 3 to take into account the *projection* on $attrs(R)$. More precisely, an *example* is now a pair (t, α), where $t \in R$ and $\alpha \in \{+, -\}$. Similarly to Section 3, we say that an example of the form $(t, +)$ is a *positive example* while an example of the form $(t, -)$ is a *negative example*. A *sample* is a set of examples i.e., a set $S \subseteq R \times \{+, -\}$. Given a sample S, we denote the set of positive examples $\{t \in R \mid (t, +) \in S\}$ by S_+ and the set of negative examples $\{t \in R \mid (t, -) \in S\}$ by S_-. For example, take the relations and their instances from Example 2.1. Consider the sample S' s.t. $S'_+ = \{t_1, t_2\}$ and $S'_- = \{t_3\}$. The semijoin predicate $\theta' = \{(A_1, B_2)\}$ is consistent with the sample S' i.e., $S'_+ \subseteq R \ltimes_{\theta'} P$ and $S'_- \cap R \ltimes_{\theta'} P = \varnothing$.

The basic problem of interest is the *consistency checking* i.e., given a database instance and a sample, decide whether there exists a semijoin predicate that selects all positive examples and none of the negative ones. Formally, we have the following decision problem:

$$\text{CONS}_\ltimes = \{(R, P, S) \mid \exists \theta.\, S_+ \subseteq R \ltimes_\theta P \wedge S_- \cap (R \ltimes_\theta P) = \varnothing\}.$$

Unfortunately, this fundamental decision problem is intractable, as we state below.

Theorem 6.1 $\text{CONS}_\ltimes$ *is NP-complete.*

Theorem 6.1 shows that consistency checking is intractable for semijoins, which implies that deciding whether a tuple is uninformative is also intractable. This precludes a tractable adaptation for semijoins of the interactive scenario proposed in Section 3.

7. CONCLUSIONS AND FUTURE WORK

We have focused on interactive inference of join queries without assuming referential integrity constraints. We have precisely characterized join predicates with equality and defined the potential information that labeling a tuple may contribute to the inference process. Whether a tuple is informative or not can be indeed decided in polynomial time. Then, we have proposed several efficient strategies of presenting tuples to the user and we have discussed their performance on TPC-H benchmark and synthetic datasets. Finally, towards the goal of extending our queries to include projection, we have shown the intractability of consistency checking.

As future work, we would like to investigate lookahead strategies using probabilistic graphical models, which will in principle improve the current strategies. We also would like to design heuristics for the interactive inference of semijoins, and we would like to check whether the presence of only positive examples makes the problem more tractable. Our early attempt indicates that deciding the minimality of a semijoin predicate in the presence of only positive examples is coNP-complete, but it is not yet clear whether such a minimal semijoin predicate is unique or not. Another interesting direction for future work is to extend our approach to other algebraic operators and to join paths. Finally, our study makes sense in realistic crowdsourcing scenarios, thus we could think of crowdsourcing users to provide positive and negative examples for join inference.

8. REFERENCES

[1] A. Abouzied, D. Angluin, C. H. Papadimitriou, J. M. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In *PODS*, pages 49–60, 2013.

[2] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful query specification with DataPlay. *PVLDB*, 5(12):1938–1941, 2012.

[3] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. Designing and refining schema mappings via data examples. In *SIGMOD Conference*, pages 133–144, 2011.

[4] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. EIRENE: Interactive design and refinement of schema mappings via data examples. *PVLDB*, 4(12):1414–1417, 2011.

[5] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[6] S. Cohen and Y. Weiss. Certain and possible XPath answers. In *ICDT*, pages 237–248, 2013.

[7] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010.

[8] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011.

[9] G. Gottlob and P. Senellart. Schema mapping discovery from data instances. *J. ACM*, 57(2), 2010.

[10] T. Imielinski and W. Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[11] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

[12] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD Conference*, pages 73–84, 2012.

[13] S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[14] T. Sellam and M. L. Kersten. Meet Charles, big data query advisor. In *CIDR*, 2013.

[15] B. ten Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. *ACM Trans. Database Syst.*, 38(4):28, 2013.

[16] TPC. TPC benchmarks, `http://www.tpc.org/`.

[17] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD Conference*, pages 535–548, 2009.

[18] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *PVLDB*, 6(3):205–216, 2013.

[19] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD Conference*, pages 809–820, 2013.