Intelligent Query Answering with Contextual Knowledge for Relational Databases

Dietmar Seipel ✉
Department of Computer Science, Universität Würzburg, Germany

Daniel Weidner ✉
Department of Computer Science, Universität Würzburg, Germany

Salvador Abreu ✉
Nova–Lincs, University of Évora, Portugal

Abstract

We are proposing a *keyword–based query interface* for knowledge bases – including relational or deductive databases – based on contextual background knowledge such as suitable *join conditions* or synonyms. Join conditions could be extracted from existing referential integrity (foreign key) constaints of the database schema. They could also be learned from other, previous database queries, if the database schema does not contain foreign key constraints.

Given a textual representation – a word list – of a query to a relational database, one may parse the list into a structured term. The intelligent and cooperative part of our approach is to *hypothesize* the semantics of the word list and to find suitable *links* between the concepts mentioned in the query using contextual knowledge, more precisely *join conditions* between the database tables.

We use a knowledge–based parser based on an extension of Definite Clause Grammars (Dcg) that are interweaved with calls to the database schema to suitably *annotate* the tokens as table names, table attributes, attribute values or relationships linking tables. Our tool DdQl yields the possible queries in a special domain specific rule language that extends Datalog, from which the user can choose one.

2012 ACM Subject Classification Information systems → Data management systems

Keywords and phrases Knowledge Bases, Natural Language Interface, Logic Programming, Definite Clause Grammars, Referential Integrity Constraints

Digital Object Identifier 10.4230/OASIcs.SLATE.2021.16

Supplementary Material *Software (Source Code)*: `http://www1.pub.informatik.uni-wuerzburg.de/databases/research.html` [19]

1 Introduction

The growing wave of *digitization*, which the smart world of the future is facing, could be met by concepts from *artificial intelligence (AI)*. The field of AI can be divided into symbolic and subsymbolic approaches, e. g., [11,15]. *Symbolic or knowledge–based* approaches model central cognitive abilities of humans like *logic*, deduction and planning in computers – mathematically exact operations can be defined. *Subsymbolic or statistical* approaches try to learn a model of a process (e. g., an optimal action of a robot or the classification of sensor data) from the data.

Current knowledge–based information systems are increasingly becoming hybrid, including different formalisms for knowledge representation. In this paper, we use concepts from AI and logic programming for answering non–expert queries to hybrid knowledge bases. Still, the most frequent fromalism is relational databases, but it would be very interesting to include rule–bases, ontologies and Xml databases as well.

It is becoming popular to consider natural language queries [1]. In a simple form, this concept is well–known from *keyword–based queries* in search engines like Google. It can be very helpful for users who are not so familiar with the database schema, and for users on mobile devices, where it is difficult to enter complex–structured queries. For a preceeding speech–to–text transformation, currently subsymbolic approches, e.g. voice/speech assitants such as the commercial systems Siri, Alexa, or Dragon NaturallySpeaking or the publicly available tools Mozilla Common Voice/Deep Spech [12] are popular. In this paper, the complicated step of assigning a suitable semantics – i.e. of compling textual keyword–based queries to correct complex–structured knowledge base queries, e.g. in SQL or Datalog– is done using a symbolic, declarative knowledge–based approach with techniques from logic programming and deductive databases [3, 6, 7].

The rest of this paper is structured as follows: Section 2 gives an overview on database query languages and intelligent query answering. Section 3 presents our running example of a database schema, a database instance (tables) and a set of relational database queries; we sketch some possible ways of deriving suitable join conditions. Section 4 describes our new system and langauge DDQL for answering keyword–based queries using technology from logic programming and deductive databases; we use our running example database. Finally, Section 5 concludes with a summary.

2 Database Query Languages and Intelligent Query Answering

Natural language interfaces (NLI) are considered a useful end–user facing query language for knowledge bases, see Affolter et al. [1] and Damljanovic et al. [8]. This holds especially true for complex databases and knowledge bases, where the intricacies of both the information schema and the technicalities of the query language – SQL most of the time – put the task of issuing useful queries well beyond the skill of most prospective, non–technical users. NLIs can usually be catgorized into keyword–, pattern–, parsing–, and grammar–based systems. Recent case studies are also reported by Stockinger [23] who argues that the trend of building NLI databases is stimulated by the recent success stories of artificial intelligence and in particular deep learning. An important keyword–based system is SODA [2]. Li and Jagadish [14] hold that NLIs are superior to other approaches to ease database querying, such as keyword search or visual query–building. They present the parsing–based systems NaLIR and NaLIX.

The main gripe with a natural language interface is that it's inherently difficult to verify reliably: an ambiguous sentence might be incorrectly parsed and its meaning evaluated, without the end user ever becoming aware of the situation. As a consequence, much effort has been placed into devising user–friendly ways of removing the ambiguity and translating the query to a semantically equivalent one in the native database query language. In practice, this entails presenting alternatives to the user and asking him to decide; the process may be iterated.

Doing so with SQL as the target seems a natural choice, but hits many difficulties arising from the language's many quirks. This situation is exacerbated when one must present the query interpretation back to the user. Relying on a more abstract query language, such as a first order predicate logic–based one, turns out to be both easier and more effective, especially as the reflection of the user's utterance interpretation will be presented in a form which is closer to its presumed grammatical structure and therefore easier to recognize and understand.

Besides convenience in presentation, relying on a logic representation for the queries and schema has several enabling benefits: a major one is that it provides a unifying framework for heterogeneous sources of information, such as SQL databases but also deductive databases, XML databases, ontologies queried in SPARQL or RDF datasets [22]. This topic has been amply covered in the literature, see for example [16] for an overview on logic in deductive databases.

Interpreting a natural language sentence as a database query entails attempting to do several queries, ranging over the schema but also the data and even the query history. Contextual speech recognition is a very hard problem, which can be eased if one manages to make use of background knowledge. The inherent ambiguity in the task of parsing and tagging a sentence in natural language can be mitigated and complemented with concurrent knowledge base queries: domain knowledge may be used to constrain the admissible interpretations as well as to provide useful annotations. Having a logic–based framework also makes it easy to provide views, which may be further used in interpreting natural language queries. The logic dialect needs not be full first–order logic, as Datalog is sufficient to express queries originally formulated in simple natural language.

3 Relational Database Queries

It is difficult for database users to have to remember the strucuture of the database (the database schema) and the correct writing of the terms (table names and attributes) and the values in the tables. Nevertheless, they have a good notion of the queries that they would like to ask. One could, e.g., imagine the following database queries:

$\mathcal{Q}_1$ *Give me the salary of Borg.*
$\mathcal{Q}_2$ *Who is the father of Alice ?*
$\mathcal{Q}_3$ *What is the salary of Research ?*
$\mathcal{Q}_4$ *Give me the sum of the salaries of the departments by name.*
$\mathcal{Q}_5$ *Give me the supervisor name of an employee by name.*

The database user does not say that *salary* is an attribute of a database table or that *Borg* is a value of another attribute. Moreover, there could be slight spellings mistakes.

We are proposing an intelligent expert tool for query answering based on the deductive database system DDbase [21] of the declarative programming toolkit Declare [19]. We have developed a module DdQl of DDbase, that can first parse the textual representation of the query using Declare's extended definite clause grammars (Edcg) [18] in Prolog based on the background knowledge of the database schema and the database, then hypothesize the intended semantics of the query using expert knowledge, and finally present possible queries and answers, so that the user can select one. In future extensions, it might be possible to define Datalog–like rules in natural language.

3.1 The Database Schema

We use the relational database Company from [9] in an extensive case study for exemplifying our approach. The database schema in Figure 1 contains 6 entity/relationship types and 8 referential integrity constraints between them (links given by arrows). Some corresponding database tables will be given in the following Section 3.2. The query compilation in Section 4.2 will extract undirected connected subgraphs from the corresponding link graph in Figure 2.

The relationship types from the corresponding ER diagram of [9] are represented in the database schema as follows:

(a) in the table Employee, the 1:n relationship types Works_For and Supervision from the ER diagram are integrated as foreign keys Dno and SuperSsn (the Ssn of the supersisor), respectively;
(b) the manager of a department is given by the attribute MgrSsn in Department;
(c) the table Works_On gives the employees working on a project, and the attribute Dnum in Project gives the responsible department of a project.

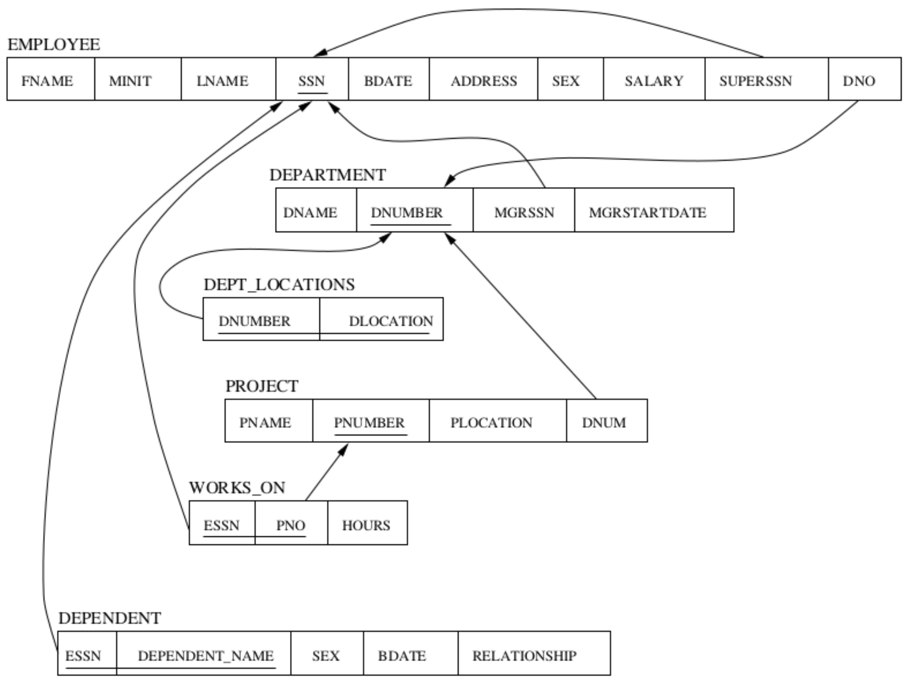


Figure 1 Referential Integrity Constraints for the Relational Database COMPANY.

Functionalities and existency constraints require: every employee works for exactly one department; every department must have exactly one manager; an employee can manage at most one department; every employee must work for at least one project; and every project must have exactly one responsible department. All constraints of the database schema can be used for optimizing queries.

3.2 Database Tables

In the following, we will use a slightly restricted version of the database, where some entity types and attributes are not present or renamed.

EMPLOYEE								
FNAME	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	Smith	4444	1955-01-09	731 Fondren, Houston	M	30000	2222	5
Franklin	Wong	2222	1945-12-08	638 Voss, Houston	M	40000	1111	5
Alicia	Zelaya	7777	1958-07-19	3321 Castle, Spring	F	25000	3333	4
Jennifer	Wallace	3333	1931-06-20	291 Berry, Bellaire	F	43000	1111	4
...	...	...	...	...	...	...	...	...
James	Borg	1111	1927-11-10	450 Stone, Houston	M	55000	NULL	1

The departments and their managers are given by the table DEPARTMENT with the primary key DNO (DNUMBER in Figure 1). The 1:1 relationship type MANAGES is integrated as a foreign key MGRSSN together with the describing attribute MGRSTARTDATE, the start date of its manager. The multi–valued attribute Locations of the entity type DEPARTMENT yields a separate table DEPT_LOCATIONS, which we do not consider here. The table WORKS_ON shows the HOURS that the employees work on the projects.

DEPARTMENT			
DNAME	DNO	MGRSSN	MGRSTARTDATE
Headquarters	1	1111	1971-06-19
Administration	4	3333	1985-01-01
Research	5	2222	1978-05-22

WORKS_ON		
ESSN	PNO	HOURS
1111	20	NULL
2222	2	10.0
2222	3	10.0
3333	30	20.0
...	...	...

The 1:n relationship type CONTROLS between DEPARTMENT and PROJECT is integrated as the foreign key DNUM in PROJECT. Every project is located at one of the locations of its controlling department.

PROJECT			
PNAME	PNUMBER	PLOCATION	DNUM
ProductX	1	Bellaire	5
Reorganization	20	Houston	1
...	...	...	...
Newbenefits	30	Stafford	4

In a deductive database variant, the rows of the relational tables would be represented by Datalog facts.

3.3 Datalog–Like Rules

In our system DDQL, the relational or deductive database can be enriched with further Datalog–like rules in field notation, or knowledge from ontologies in RDF, OWL, or SWRL. Likewise, some background knowledge from the database schema can be represented in a Datalog–like manner. For the foreign–key constraint from EMPLOYEE to DEPARTMENT, a Datalog–like rule with field notation can be generated in DDbase which links the social security number of an employee with the number of his or her department:

```
works_for(Employee, Department) :-
   employee:['SSN':Employee, 'DNO':Department].
```

From an ontology, it could be known that every employee is human. This would be expressed by the following Datalog–like rule with field notation, since SSN is the primary key of EMPLOYEE:

```
human(Employee) :-
   employee:['SSN':Employee].
```

The user of the database would like to ask queries in *Google style*, i.e. without precisely remembering the database schema and the links between the tables by referential integrity constraints. Here, the background knowledge given by the Datalog–like rules can be used.

3.4 Inference of Join Conditions

Suitable join conditions can be inferred from the foreign key constraints given in the database schema. The schema of the database `company` contained many foreign key constraints. For many other databases, no foreign key constraints are given. But join conditions can be inferred from previous SQL queries in the log file: e.g., a join condition can be assumed, if the primary key of a table (all attributes of the primary key) is joined with some attributes of another table.

In Declare, the schema of a table can be extracted automatically from a running relational MySQL database system and presented in XML to the user and analysed with Prolog:

```
<table name="employee">
   <attribute name="SSN" type="varchar(9)" is_nullable="NO"/>
   <attribute name="SALARY" ...>
   <attribute name="DNO" ...>
   ...
   <primary_key> <attribute name="SSN"/> </primary_key>
   <foreign_key>
      <attribute name="SUPERSSN"/>
      <references table="employee">
         <attribute name="SSN"/> </references> </foreign_key>
   <foreign_key>
      <attribute name="DNO"/>
      <references table="department">
         <attribute name="DNUMBER"/> </references> </foreign_key>
</table>
```

Currently, this XML representation is derived using ODBC in Declare, and join conditions are extracted in DDQL. If the second foreign key constraint was not given in the schema, we may still infer a corresponding join condition from the following SQL statement occuring in a query log file:

```
use company;
select employee.DNO, employee.SSN, employee.SALARY
from employee, department
where department.DNAME = 'Research'
and employee.DNO = department.DNUMBER;
```

Declare provides a tool named SQUASH [5] to parse SQL statements to Prolog terms, which may be mapped to XML. SQUASH proposes a domain specific language SQUASHML for SQL statements; this can be further processed in Declare to infer the join conditions. In a simplified version, the SQL statement above looks as follows:

```
<select>
   <select_list>
      <object table="employee" column="DNO"/>
      <object table="employee" column="SSN"/>
      <object table="employee" column="SALARY"/>
   </select_list_element>
   <from_list>
      <object table="employee"/>
      <object table="department"/>
   </from_list>
   <where_list>
      <condition junctor="and">
         <comparison operator="=">
            <object table="department" column="DNAME"/>
            <object value="Research"/>
         </comparison>
      </condition>
   </where_list>
</select>
```

4 The Declarative Database Query System and Language DDQL

In this section, we will present the new declarative database query system and language DdQl, which is based on concepts from logic programming. The knowledge-based compilation of keyword–queries to Datalog is done in three steps. In experiments with the `company` database, useful queries were generated; if a database does not contain referential integrity constraints, then we will need query logs for deriving suitable join conditions.

4.1 Syntax and Evaluation Using Logic Programming Concepts

The declarative programming toolkit Declare [19] and its deductive database system DDbase [21] already have functionality for evaluating database queries formulated using extensions of Datalog. Even hybrid queries including different knowledge representation formalisms are possible in DDbase. E.g., relational databases can be accessed using Sql queries and Odbc; for Xml processing, a query, transformation and update language FNQuery is given in [20].

We assume that the reader has some basic knowledge about logic programming [3, 7] as well as relational [9] and deductive databases [16]. DDbase allows for rules of the form A :- $L_1, \ldots, L_m$, where the head A is a logical atom and the body is a conjunction (the comma "," denoting the conjunction "$\wedge$") of literals L_i, $1 \leq i \leq m$. The literals can be logical atoms $L_i = B_i$, default negated literals $L_i = \mathit{not}\,(A_i)$, or aggregation literals $L_i =$ `ddbase_aggregate`$(X, (A_1, \ldots, A_n), Xs)$ over logical atoms A_i. In the domain specific language of DDbase, rules can have ordinary logical atoms – in Prolog notation – or *field notation* atoms $p : [a_1 : t_1, \ldots, a_k : t_k]$, where p is a predicate symbol, $a_1, \ldots, a_k$ are field names, and $t_1, \ldots, t_k$ are corresponding terms, which amounts to non–positional arguments in Prolog terms. The rules must fullfil the *safety condition* that variables in atoms A_i within default negated literals must be bound by preceeding ordinary atoms or aggregation literals in the same rule body. It is not the intent of this paper to formally define the semantics of DDbase. The next subsections will focus on the knowledge–based compilation of queries and give some intuitive examples without default negation. Only one ($\mathcal{Q}_4$) contains an aggregation literal

```
ddbase_aggregate( [C, sum(D)],
   ( employee(_, _, _, _, _, _, _, D, _, E),
     department(C, E, _, _) ), Xs ),
member([A, B], Xs).
```

In analogy to Sql and extending Prolog's predicate `findall/3`, this groups instantiations of the variable `C` with the sum (which is an aggregation function) of the corresponding instantiations of the variable `D` and returns pairs. Here, the template is `X = [C, sum(D)]` and pairs `[A, B]` are selected from the result `Xs`.

DDbase programs are *evaluated bottom–up* with *stratified fixpoint* computation like Datalog and they can – possibly – be compiled to Sql queries; often DDbase programs can also be evaluated top–down like in Prolog. For the evaluation in logic programming, the field notation atoms are compiled to ordinary, logical Datalog atoms based on background knowledge about the database schema. The ordinary Datalog rules can be compiled to Sql with DDbase, if there is no default negation – and for stratified default negation or aggregation. A stratified evaluation requires that none of the embedded atoms A_i is mutualle recursive with the head atom A. Then, the program is split into strata, such that default negated literals or aggregation literals refer to lower strata; the strata are evaluated successively, and the output of a lower stratum is fed into the strata above. For non–stratified default negation, DDbase could use *answer set* solvers, cf. [4, 13], if there are no aggregation literals.

4.2 Knowledge–Based Compilation of Queries

DdQl compiles a NL query $\mathcal{Q}_N$ to a Datalog query $\mathcal{Q}_D$ in three steps:

$\mathcal{Q}_N \rightarrow \mathcal{Q}_A \rightarrow \mathcal{Q}_F \rightarrow \mathcal{Q}_D$.

It shows them in a command line interface; the result tables are shown in a graphical interface.

Annotation of Key Words ($\mathcal{Q}_A$)

First, using Definite Clause Grammars (Dcgs, see, e.g., [3,7]), an annotated query $\mathcal{Q}_A$ is generated. Using, e.g., the following grammar rule in the extended Dcg formalism introduced in [18], also the resulting parse tree can be computed:

```
query ==> aggregation, of, attribute, of, table.
```

The Dcg rules are fully interleaved with database access operations of DDbase using Odbc. E.g., the derivations of `attribute` and `table` can result in Odbc calls, or – for potential speed–up – calls to a cached collection of facts previously extracted from the database. Some words of the query – such as "`of`" and "`the`" – are ignored.

DdQl generates the annotations one after the other on *backtracking*, starting with the most likely annotations. E.g., the query $\mathcal{Q}_1$ with the key words "`salary, of, Borg`" is first annotated to the following query $\mathcal{Q}_A$:

```
salary=company/employee/attribute
'Borg'=company/employee/row(@LNAME)
```

The keyword "`of`" is ignored. In DdQl, it can be detected easily from the database schema that `salary` is an attribute of the relation `employee`. The location of `'Borg'` has to be done based on the contents of the database, which is more expensive. But this can be done depending on the context of the table `employee`; it turns out that it is the value of the attribute `LNAME`. After the first annotation has been done and the first solution to the query has been produced, DdQl uses backtracking to generate further annotations and solutions. Of course, then DdQl will also search for `'Borg'` in other tables of the database.

Generation of Field Notation ($\mathcal{Q}_F$) and Datalog ($\mathcal{Q}_D$)

Then, the compilation of the annotated queries $\mathcal{Q}_A$ to Sql or Datalog is done using technology from DDbase based on the database schema. As an intermediate representation, conjunctive queries $\mathcal{Q}_F$ in Datalog are generated with atoms in field notation. The conjunctive queries are then refined and optimized to ordinary Datalog queries $\mathcal{Q}_D$ using background knowledge from the database schema or Datalog–likes rules. $\mathcal{Q}_D$ could be evaluated on a deductive variant of the relational database or a deductive database; an Sql variant of $\mathcal{Q}_D$ can be evaluated on the relational database.

In the following, we will show and explain the intermediate queries $\mathcal{Q}_F$ and $\mathcal{Q}_D$ for a few example queries to the database `company` – and we skip the annotated queries $\mathcal{Q}_A$.

The relevant links between the concepts mentioned in a user query might be an undirected tree, or even a cyclic graph. E.g., if the user asks for the salary of all employees working in a department that controlls the project `'ProductX'` and is located in `'Houston'`, then the red link tree in Figure 2 (arrows starting with a bullet) – which is an abstraction of Figure 1 – might be used. The direction of the labelled referential integrity constraints Manages and Controls is due to the fact that their corresponding relationships are integrated as the attributes MgrSsn and Dnum into Department and Project, respectively.

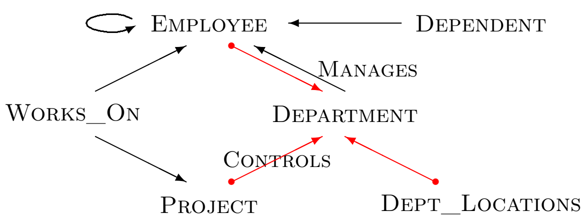


Figure 2 Link Graph for the Referential Integrity Constraints.

4.3 Example Queries

In the following, we will explain a spectrum of example queries to show different features of DDQL.

Selection Queries $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$

The two selection queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are relatively simple to solve, since they rely on only one table, namely EMPLOYEE and DEPENDENT, respectively (the latter table is not detailed in this paper, but the schema in Figure 1 contains it). The selection query $\mathcal{Q}_3$ results in the following NL query $\mathcal{Q}_N$:

```
salary, of, 'Research'
```

The database user might not know that SALARY is an attribute of the table EMPLOYEE, and that `Research` is the value of the attribute DNAME of the table DEPARTMENT. The query is annotated, and a list $\mathcal{Q}_F$ of query atoms in field notation is generated:

```
employee:['SSN':B, 'SALARY':C],
employee:['SSN':B, 'DNO':A],
department:['DNUMBER':A, 'DNAME':'Research'].
```

The variables `A`, `B`, and `C` are automatically generated by the system. The second atom links the first and the third; here, the WORKS_FOR relationship is integrated in EMPLOYEE using its last argument DNO. It would also be conceivable to link EMPLOYEE and DEPARTMENT by an atom associated with the MANAGES relationship to return the salary of the manager of the research department, or even by a longer path using the relationships WORKS_ON and CONTROLS to return the salary of all employees working for a project that is controlled by the research department. The former, alternative path through MANAGES might be offered to the user, but the latter, the longer path through WORKS_ON and CONTROLS seems unlikely.

The list $\mathcal{Q}_F$ could be optimized using the database schema; the two `employee` atoms could be combined into a single one, since they share the variable `B` for the key SSN: the result is the atom

```
employee:['SSN':B, 'SALARY':C, 'DNO':A].
```

Thus, from $\mathcal{Q}_F$, a Datalog query $\mathcal{Q}_D$ is generated:

```
select(A, B, C) :-
   employee(_, _, _, B, _, _, _, C, _, A),
   department('Research', A, _, _).
```

From $\mathcal{Q}_D$, an SQL query is compiled, which could be presented to the user together with its resulting table:

```
use company;
select employee.DNO, employee.SSN, employee.SALARY
from employee, department
where department.DNAME = 'Research'
and employee.DNO = department.DNUMBER;
```

The – intermediate – Datalog query $\mathcal{Q}_D$ can be used for evaluation or clarification. If the rule system of the user is complicated or refers to further Datalog rules, ontologies or XML tables, then $\mathcal{Q}_D$ cannot be translated to SQL and has to be evaluated in DDbase directly.

In general, the situation can be more complicated, as we can have multiple occurrences of the same table and we need aliases, e.g. for $\mathcal{Q}_5$. Moreover, the linking atoms can be ambiguous and we may need aggregations.

Aggregation Query $\mathcal{Q}_4$

Consider the following NL query $\mathcal{Q}_N$:

```
sum, of, salary, of, department, name
```

The database user might want to know the sum of the salaries of the employees grouped by the names of their departments. The query is annotated, and a list $\mathcal{Q}_F$ of query atoms in field notation is generated:

```
aggregation:[C, sum(D)],
employee:['SSN':F, 'SALARY':D],
employee:['SSN':F, 'DNO':E],
department:['DNUMBER':E, 'DNAME':C].
```

The system found out that the word `name` in the query refers to the attribute DNAME of DEPARTMENT. The third atom links the second and the fourth; here, the WORKS_FOR relationship is integrated in EMPLOYEE using its last argument DNO. The aggregation is encoded as a special atom. This list could be optimized using the database schema; the two `employee` atoms could be combined into a single one, since they share the value `F` for the key SSN, namely to the atom `employee:['SSN':F ,'SALARY':D ,'DNO':E]`. Thus, from $\mathcal{Q}_F$, a Datalog query $\mathcal{Q}_D$ is generated:

```
select(A, B) :-
   ddbase_aggregate( [C, sum(D)],
     ( employee(_, _, _, F, _, _, _, D, _, E),
       department(C, E, _, _) ),
     Xs ),
   member([A, B], Xs).
```

`ddbase_aggregate/2` is a meta–predicate provided with DDbase. From its result `Xs`, pairs `[A, B]` are selected using `member/2`. Here, this gets compiled to an aggregation with a `Group By` clause in SQL. From $\mathcal{Q}_D$, an SQL query is compiled, which can be presented to the user together with its resulting table:

```
use company;
select department.DNAME, sum(employee.SALARY)
from employee, department
where employee.DNO = department.DNUMBER
group by department.DNAME;
```

Moreover, it is possible in DDbase to use more general, user–defined aggregation functions. E.g., for `list(D)` (instead of `sum(D)`), $\mathcal{Q}_D$ would be directly evaluated in DDbase to produce a *non–first normal form (NfNf, NF^2)* relation showing a list of the salaries grouped by the departments `C`, which is not possible in SQL:

NF^2 Relation	
Dno	Salaries
1	'NULL'
4	25000, 43000, 25000
5	30000, 40000, 38000, 25000

Query $\mathcal{Q}_5$ with Aliases

The following NL query $\mathcal{Q}_N$ needs multiple occurrences of the same table and aliases:

```
name, of, supervisor, of, exmployee, name
```

The database user might want to know the names of the supervisors of the employees. The query is annotated, and a list $\mathcal{Q}_F$ of query atoms in field notation is generated:

```
employee:['SSN':D, 'LNAME':B],
employee:['SSN':D, 'SUPERSSN':C],
employee:['SSN':C, 'LNAME':A].
```

The second atom links the first and the third. The optimizer could combine the first two atoms, since they agree on the key SSN, to the atom

```
employee:['SSN':D, 'LNAME':B, 'SUPERSSN':C].
```

Notice, that the third atom cannot be merged since it differs on SSN. Thus, from $\mathcal{Q}_F$, a Datalog query $\mathcal{Q}_D$ is generated:

```
select(A, B) :-
   employee(_, _, B, D, _, _, _, _, C, _),
   employee(_, _, A, C, _, _, _, _, _, _).
```

Note that the SSNs of the supervisor (`C`) and the employee (`D`) are not part of the result. From $\mathcal{Q}_D$, an SQL query is compiled, which could be presented to the user together with its resulting table:

```
use company;
select employee__1.LNAME, employee__2.LNAME
from employee employee__1, employee employee__2
where employee__1.SUPERSSN = employee__2.SSN;
```

4.4 General Aspects of DDQL

The DdQl approach – exemplified in the case study – can be applied to knowledge databases – e.g. relational or deductive databases. The following aspects have to be taken into account.

Links Based on Query Order

For successor atoms of the annotated query $\mathcal{Q}_F$, links have to be found, which might not be unique or obvious. In fact, also for the queries $\mathcal{Q}_3$ and $\mathcal{Q}_4$ the links based on Works_For were not unique, but they were taken to follow the direction from Employee to Department in Figure 1. If Department were to be mentioned before Employee in the query, then it would be more likely that the user–intended semantics would be based on Manages.

Multiple Links

The atoms in $\mathcal{Q}_F$ might not appear consecutively in the graph of Figure 1. There might be several linking trees, or the atoms could be on a cycle. DdQl is collecting *heuristics* for finding suitable links. For $\mathcal{Q}_3$ and $\mathcal{Q}_4$ these links were affected by the order of the words in the query.

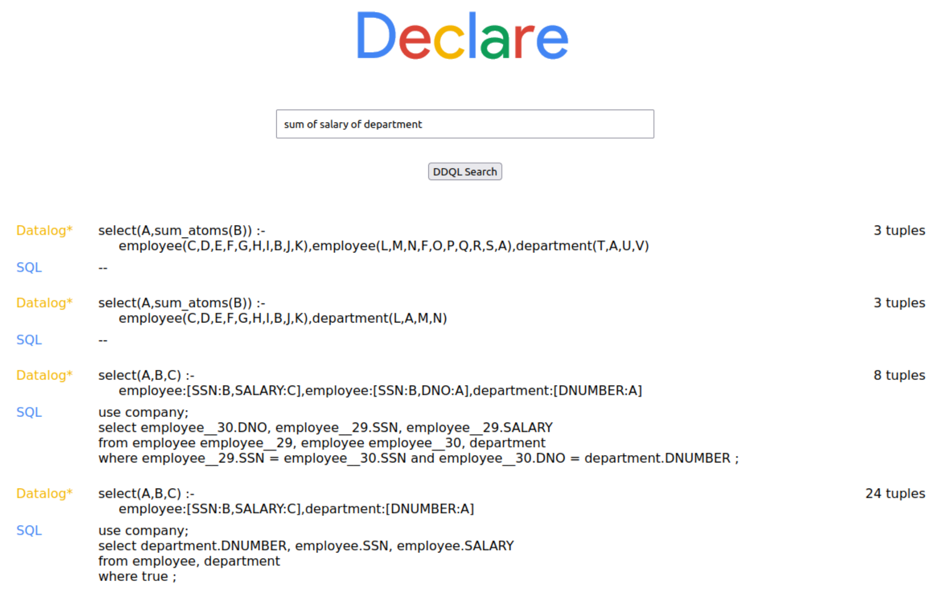
However, if we were to ask about Employee and Pname, then there would be two equally reasonable links, namely through Works_On and through Department (which could itself be supported by two different foreign–key constraints). These two cases can be expressed by the following Prolog clauses:

```
select(Lname, Pname) :-
   employee(_, _, Lname, Ssn, _, _, _, _, _, _),
   works_on(Ssn, Pno, _),
   project(Pname, Pno, _, _).
select(Lname, Pname) :-
   employee(_, _, Lname, _, _, _, _, _, _, Dnum),
   department(_, Dnum, _, _),
   project(Pname, _, _, Dnum).
```

In the second query, the linking atom `department(_, Dnum, _, _)`, which is produced because of the foreign key constraints between Employee and Department and between Department and Project, could be redundant. It makes a difference when there is no department with the number `Dnum` referenced by Employee and Project. In that case, the user has to decide about whether he wants to include this atom or not. If the query were to contain a keyword that is similar to a link, then the link would be preferred. In general, all corresponding queries can be presented to the user, for a choice to be made.

Similarities and Subsumptions

So far, we have not yet discussed similarities or subsumptions between words from the query and the terms used in the database. Here, ontologies or concepts from linguistics can be applied. A simple case would be the *Levenshtein Distance* (Edit Distance) between words or conversions between singular and plural. More complicated cases could be handled by background knowledge such as translations between languages.



```
Datalog*  select(A,sum_atoms(B)) :-                                                                  3 tuples
              employee(C,D,E,F,G,H,I,B,J,K),employee(L,M,N,F,O,P,Q,R,S,A),department(T,A,U,V)
SQL       --

Datalog*  select(A,sum_atoms(B)) :-                                                                  3 tuples
              employee(C,D,E,F,G,H,I,B,J,K),department(L,A,M,N)
SQL       --

Datalog*  select(A,B,C) :-                                                                           8 tuples
              employee:[SSN:B,SALARY:C],employee:[SSN:B,DNO:A],department:[DNUMBER:A]
SQL       use company;
          select employee__30.DNO, employee__29.SSN, employee__29.SALARY
          from employee employee__29, employee employee__30, department
          where employee__29.SSN = employee__30.SSN and employee__30.DNO = department.DNUMBER ;

Datalog*  select(A,B,C) :-                                                                          24 tuples
              employee:[SSN:B,SALARY:C],department:[DNUMBER:A]
SQL       use company;
          select department.DNUMBER, employee.SSN, employee.SALARY
          from employee, department
          where true ;
```

Figure 3 Graphical User Interface of DdQl: Variant of Query $\mathcal{Q}_4$.

Parsing Sentences and Rules

With a powerful speech–to–text component, we could aim to *parse* more complex structures for sentences. Then we could enable the user to define subqueries producing views. With DdQl, it is already possible to define additional Datalog predicates with rules to be used – like Sql views – in further queries. Also, sometimes referential integrity constraints can be inferred for these new predicates.

4.5 The Graphical User Interface

A prototype of the graphical user interface (GUI) of DdQl is shown in Figure 3. After entering the keywords separated by blanks, a list of possible search queries in Datalog* is generated. The generated queries might be further optimized. Currently, a corresponding Sql query is shown, if there are no aggregation functions. Obviously, for user–defined aggregation functions, only the Datalog* variant is possible. Figure 3 also shows the number of result tuples, such that the user can choose a possible alternative answer. A ranking of the results would be possible based on these numbers, the contextual knowledge, and the query history – but so far we have not finalized that.

4.6 Benchmark – Database Schemas and Queries

We have investigated the database schemas of a collection of relational databases. The tuples `[DB, T, A, F]` listed below show the following:

(a) `DB` is the name of the database,
(b) `T` is the number of tables in the database,
(c) `A` is the average number of attributes per table in the database,
(d) `F` is the average number of foreign keys per table in the database.

Most databases did not provide foreign key constraints. Only the database `company` provided foreign key constraints, namely 1.33 on average per table.

```
Tuples = [
   [alignment, 5, 3.4, 0], [company, 6, 4.67, 1.33],
   [evu, 20, 5.7, 0], [morphemes, 1, 7, 0], [selli, 1, 7, 0],
   [stock, 6, 3.33, 0], [wm_2002, 9, 7.33, 0] ]
```

I.e., for most databases DdQl has to rely on the query log files with previous queries, which we can parse to Xml using our tool Squash, to learn about possible join conditions.

In a benchmark with many queries to the relational database `company` – which has 1.33 foreign key constraints per table on average – about 4 to 5 different alternative suggestions were generated on average per query. The answer intended by the user was always one of them. For the single query

```
salary, of, exmployee, '1111'
```

many (i.e. 20) different alternative suggestions were generated. We are currently trying to reduce this number for similar cases. Without this query, only 3.5 alternative suggestions were generated on average.

5 Conclusions

In this paper, we have shown how queries to relational databases can be answered in keyword–based natural language interfaces using intelligent, cooperative techniques based on logic programming and deductive databases.

The concepts mentioned in the query are linked based on contextual background knowledge, mainly from the database schema. Also Datalog–like rules can be added as background knowledge, e.g. for deductive databases – without a database schema. In the deductive database DDbase, the knowledge could also be hybrid – including ontological, linked data (Rdf, Owl), Swrl knowledge and semi–structured Xml documents – and hybrid queries could become possible in DdQl.

Various styles or patterns of the database schema design can significantly influence the level of application of our approach. E.g., under the *universal relation scheme assumption* (URSA) [10] we could simply use natural join queries.

Currently, we are containerizing Declare – inluding DdQl – in a docker image. In the future, we are planning to add a voice recognition part, especially for using DdQl on mobile devices. Also the knowledge acquisition could be done with a voice assistant based on a domain–specific language, see [17].

We are constantly incorporating further aspects of AI into DdQl. In the future, also concepts from subsymbolic AI will be investigated and included where useful. For instance, by looking at the user behaviour from previous queries, we may derive heuristics for finding intended queries (e.g. linking atoms) using some form of *machine learning*.

References

1 Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. A Comparative Survey of Recent Nlis for Databases. *VLDB J.*, 28(5):793–819, 2019. doi:10.1007/s00778-019-00567-8.

2 Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. Soda: Generating Sql for Business Users. *Proc. VLDB Endowment*, 5(10):932–943, 2012. doi:10.14778/2336664.2336667.

3 Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison–Wesley Longman, 4th edition, 2011.

4 Gerhard Brewka, Thomas Eiter, and Mirek Truszczynski. Answer Set Programming at a Glance. *Communications of the ACM*, 54(12):92–103, 2011.

5 Andreas Böhm, Dietmar Seipel, Albert Sickmann, and Matthias Wetzka. Squash: A Tool for Designing, Analyzing and Refactoring Relational Database Applications. In *Proc. International Conference on Applications of Declarative Programming and Knowledge Management (INAP)*, pages 82–98, 2007.

6 Ceri, Stefano and Gottlob, Georg and Tanca, Laetitia. *Logic Programming and Databases*. Springer, 1990.

7 William Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer Science & Business Media, 2003.

8 Danica Damljanovic, Milan Agatonovic, and Hamish Cunningham. Nlis to Ontologies: Combining Syntactic Analysis and Ontology–based Lookup through the User Interaction. In *Extended Semantic Web Conf.*, pages 106–120. Springer, 2010.

9 Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 3rd Edition*. Addison–Wesley Longman, 2000.

10 Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom. *Database Systems: The Complete Book*, volume 638. Pearson Prentice Hall, 2009.

11 Ben Goertzel. Perception Processing for General Intelligence: Bridging the Symbolic/Subsymbolic Gap. In *International Conference on Artificial General Intelligence*, pages 79–88. Springer, 2012.

12 Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep Speech: Scaling up End–to–End Speech Recognition. *arXiv preprint*, 2014. arXiv:1412.5567.

13 Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

14 Fei Li and H. V. Jagadish. Understanding Natural Language Queries over Relational Databases. *SIGMOD Rec.*, 45(1):6–13, 2016. doi:10.1145/2949741.2949744.

15 Clayton McMillan, Michael C Mozer, and Paul Smolensky. Rule Induction through Integrated Symbolic and Subsymbolic Processing. In *Advances in Neural Information Processing Systems*, volume 4, pages 969–976, 1992.

16 Jack Minker, Dietmar Seipel, and Carlo Zaniolo. Logic and Databases: A History of Deductive Databases. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 571–627. Elsevier, 2014. doi:10.1016/B978-0-444-51624-4.50013-7.

17 Falco Nogatz, Julia Kübert, Dietmar Seipel, and Salvador Abreu. Alexa, How Can I Reason with Prolog? (Short Paper). In *Proc. 8th Symposium on Languages, Applications and Technologies (SLATE 2019)*, 2019.

18 Christian Schneiker, Dietmar Seipel, Werner Wegstein, and Klaus Prätor. Declarative Parsing and Annotation of Electronic Dictionaries. In *Proc. 6th International Workshop on Natural Language Processing and Cognitive Science (NLPCS 2009)*, 2009.

19 Dietmar Seipel. Declare – A Declarative Toolkit for Knowledge–Based Systems and Logic Programming. URL: http://www1.pub.informatik.uni-wuerzburg.de/databases/research.html.

20 Dietmar Seipel. Processing Xml–Documents in Prolog. In *Workshop on Logic Programming (WLP 2002)*, 2002.

21 Dietmar Seipel, Rüdiger von der Weth, Salvador Abreu, Falco Nogatz, and Alexander Werner. Declarative Rules for Annotated Expert Knowledge in Change Management. In *Proc. 5th Symposium on Languages, Applications and Technologies (SLATE 2016)*, 2016.

22 Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*, International Handbooks on Information Systems. Springer, 2009. doi:10.1007/978-3-540-92673-3.

23 Kurt Stockinger. The Rise of Natural Language Interfaces to Databases. ACM SIGMOD Blog, URL: https://blog.zhaw.ch/datascience/the-rise-of-natural-language-interfaces-to-databases/, 2019.