Querying Relational Databases with Speech-Recognition Driven by Contextual Knowledge

Dietmar Seipel
Department of Computer Science, Universität Würzburg, Germany

Benjamin Förster
Department of Computer Science, Universität Würzburg, Germany

Magnus Liebl
Department of Computer Science, Universität Würzburg, Germany

Marcel Waleska
Department of Computer Science, Universität Würzburg, Germany

Salvador Abreu
NOVA–LINCS, University of Évora, Portugal

Abstract

We are extending the *keyword–based query interface* DDQL for relational databases which is based on contextual background knowledge such as suitable *join conditions* and which was proposed in [10]. In the previous paper, join conditions were extracted from existing referential integrity (foreign key) constraints of the database schema, or they could be learned from other, previous database queries.

In this paper, we describe a *speech–to–text* component for entering the query keywords based on the system Whisper. Keywords, which have been recognized wrongly by Whisper can be corrected to similarly sounding words. Again, the context of the database schema can help here.

For users with a limited knowledge of the schema and the contents of the database, the approach of DDQL can help to provide useful suggestions for query implementations in SQL or Datalog, from which the user can choose one. Our tool DDQL can be run in a *docker image*; it yields the possible queries in SQL and a special domain specific rule language that extends Datalog. The Datalog variant allows for additional *user-defined aggregation functions* which are not possible in SQL.

2012 ACM Subject Classification Information systems → Relational database query languages; Information systems → Relational database model

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

Digital Object Identifier 10.4230/OASIcs.SLATE.2023.6

Supplementary Material *Software (Source Code)*: `https://gitlab2.informatik.uni-wuerzburg.de/Wissensbasierte-Systeme/declare/declare.git`
archived at `swh:1:dir:fc9f1d42f6306261bee5a7fae45776845d7b1025`

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., [12]. *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 formalism 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 assistants such as the commercial systems Siri, Alexa, or Dragon NaturallySpeaking or the publicly available tools Whisper and Mozilla Common Voice/Deep Spech [13] are popular. In this paper, the complicated step of assigning a suitable semantics – i.e. of compiling 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.

In a previous paper on DDQL at SLATE 2021 [10], we had presented the compilation of keywords to SQL or Datalog queries using concepts from logic programming [4, 6, 8]. In this paper, we will add a speech–to–text component that is based on the well–known tool Whisper for speech–to–text conversion. Subsequently, based on the context of the queried database, another text–to–text conversion has been developed to convert the keywords, which might have been understood a little bit wrongly by Whisper, to similar keywords that might have been intended by the user. This is done using concepts from language processing and the context of the database.

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 a running example of a database instance, i.e. the representation by tables, and a set of relational database queries. Section 4 deals with the speech–to–text conversion with Whisper and the following text–to–text corrections based on the context of the database. Section 5 summarizes our system DDQL for answering keyword–based queries using technology from logic programming and deductive databases. We show how DDQL and Whisper are integrated in a *docker–based* approach to optimize the usability, portability and availability of the tool. Finally, Section 6 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. [9]. This is 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 [22] 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 [3]. Li and Jagadish [15] 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 is 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 this 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.

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. We are proposing an intelligent expert tool for query answering based on the *deductive database system* DDbase of the declarative programming toolkit Declare [20]. We have developed a module DDQL, that can first parse the textual representation of the query using Declare's extended definite clause grammars (EDCG) [19] 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.

One could, e.g., imagine the following database queries to the well-known relational database COMPANY from [11] for exemplifying our approach.

$\mathcal{Q}_1$ *Give me the salary of Borg.*
$\mathcal{Q}_2$ *What is the salary of Research ?*
$\mathcal{Q}_3$ *Give me the sum of the salaries of the departments by name.*
$\mathcal{Q}_4$ *Give me the list 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 spelling mistakes.

The complete database schema will be given in Figure 2 in the appendix; the *ER diagram* contains 6 entity/relationship types and 8 referential integrity constraints between them (links given by arrows). Some corresponding database tables will be given in this Section.

In the background, the query compilation in Section 5.1 will extract undirected connected subgraphs from the ER diagram. The relationship types from the corresponding ER diagram of [11] 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; both tables are not shown here, but they can be seen in the ER diagram in the appendix.

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.

In the following, we show 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. 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.

Department			
Dname	Dno	MgrSsn	MgrStartDate
Headquarters	1	1111	1971-06-19
Administration	4	3333	1985-01-01
Research	5	2222	1978-05-22

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. 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.

In this paper, we will only go into details for the shown tables Employee and Department.

4 Speech–to–Text with Whisper and Word Corrections

In this section, the possibility of using DdQl in cooperation with a speech–to–text tool is examined in more detail. For this purpose, Whisper, which was released in September 2022 [18], is presented and examined. For DdQl, it is crucial whether especially the keywords are

correctly translated by a speech-to-text tool. Therefore DDQL can accordingly continue to work correctly with the text transcribed from the spoken sentence. Whisper is apparently better than the previous open-source speech-to-text tools, with a lower error rate regardless of context. This is also described in the paper on Whisper in addition to an explanation of the training model used to train Whisper. This training model is crucial for the high robustness of the resulting speech-to-text tool [2].

Apart from that, we have deliberately decided against using commercial speech-to-text variants. Experience has shown that voice assistants such as Amazon Alexa, Google Assistant or Apple's Siri work quite reliably, but they have the disadvantage that it is not possible to trace what happens to the data that is transmitted to the company's internal servers. Furthermore we do not want to use ChatGPT due to its fatal privacy issues. Apart from that, it would be possible to run a variant of ChatGPT locally. However, this would be accompanied by significantly increased hardware requirements.

4.1 Technology of Whisper

As already mentioned, Whisper is a software built by a training model for machine learning. This neural network is composed of a so called pipeline model [2]. This approach combined the parallel learning of the aspects *language identification*, *voice activity detection*, *transcription of the spoken to a written text* and *transcription from any available language into english text*. Therefore, the model uses several tokens during the training process to specify, which task especially is handled at the moment. All in all the model was trained with 680.000 hours of audio, of which 117.000 hours are 96 other languages than english. The resulting speech-to-text-tool Whisper is really robust and detached from any context of the spoken text, that would have been usually given by other training models [2].

4.2 Various Aspects of Speech–to–Text with Whisper

In the following, a case study will be presented to give an overview on various aspects of the extent to which Whisper would be suitable for use as a speech-to-text-tool for DDQL. The `employee` table (see Subsection 3) from the already mentioned `company` database gives the basis for this purpose.

The table name, the column names and then all attributes were read out loud three times and recorded in audio files. The values of the attributes `fname`, `minit` and `lname` were read out as triples. The audio files were then evaluated and transcribed individually by Whisper. The Whisper models `base`, `small` and `medium` were used one after the other. In summary, each entry from the table was read out loud three times, recorded and then transcribed by the three mentioned Whisper models. It would also have been possible to read out all the words in the `employee` table in one take to record and then have it transcribed by Whisper. According to the Whisper development team, however, this could have led to an unwanted side effect. Whisper could let already evaluated audio data influence later evaluations within an audio file [2]. Therefore, it was decided to record the different attributes and names individually in the respective audio files and then have Whisper transcribe them.

Since Whisper currently does not have the functionality to configure it easily so that it recognises certain words preferentially by the user, a representative case distinction is now presented below using the `employee` table as an example. This case distinction considers the possibilities that could occur, how Whisper mis-transcribes words and how they would then have to be further processed in order to ensure a meaningful further use by DdQl. Seen in this way, the mappings opposite to the direction of the arrow in the following represent the work of an intermediate tool between Whisper and DdQl in order to make the throughput of spoken queries to the desired results of DdQl as high as possible.

Since DdQl is case-insensitive, capitalisation is ignored in the following. Generally, Whisper transcriptions are complete sentences with punctuation. In the following, it is assumed that Whisper transcriptions are passed as word lists. Therefore punctuation on the edges of the whitespaces is also omitted, i.e. the characters '.', ',', ';', '!' and '?' inbetween words of the word lists are not shown.

The arrows $\mapsto$ of the mappings below correspond to the transcription by Whisper. Each left side of the arrows represent a table entry. The right side represents the possible Whisper transcriptions of each entry. The mentioned word lists are shown as words separated by whitespaces. Various transcriptions are separated by the character '|', while various mappings are separated by a semicolon. The mappings only show a selection of the wrong transcriptions of the `medium` model. This model, according to its own information [2], relatively often translated what was said into what was wanted.

In the following, we show for a selection of types of speech-to-text by Whisper and in the following Subsection 4.3 we present the subsequent text-to-text corrections by DdQl depending on the context of the queried database.

Case 1: Separation of Words by Whisper. In the most cases, Whisper basically transcribes the spoken words correctly. Apart from the correct order of letters, Whisper occasionally adds whitespaces or other characters inbetween letters of words, that originally are written as one word.

Examples:

`fname` (spoken f-name) $\mapsto$ `f name`; `minit` (spoken m-init) $\mapsto$ `m in it`;
`lname` (spoken l-name) $\mapsto$ `l name`; `dno` (spoken d-n-o) $\mapsto$ `d-n-o`;
`superssn` (spoken super-s-s-n) $\mapsto$ `super ssn`

Therefore an intermediate tool between Whisper and DdQl would have to ensure that the keywords are partially reassembled in order to then pass them to DdQl. Incidentally, this should also make it possible to spell out entries letter by letter.

Case 2: Tanscriptions of Similar or Phonetically Similar Words. Due to the fact that Whisper is derived from a speech-to-text driven learning environment, it sometimes transcribes words with smaller deviations. These discrepancies consist of either single or multiple letters and can also take place across several words, as in Case 1.

Examples:

`salary` $\mapsto$ `celery`; `bdate (spoken b-date)` $\mapsto$ `beat it` | `be date`;
`dno (spoken d-n-o)` $\mapsto$ `tno`; `franklin t wong` $\mapsto$ `franklin t wom`;
`alicia j salaya` $\mapsto$ `alicia j zelaya` | `alicia jade salaya`;
`joyce a english` $\mapsto$ `choice a english`;
`ahmad v jabbar` $\mapsto$ `amad vi jabar` | `amad vi javar` | `amad vi jabal`;
`james e borg` $\mapsto$ `james e bork`; `m` $\mapsto$ `um` | `mmm`; `null` $\mapsto$ `nah` | `no`

In this case, an intermediate tool between Whisper and DDQL would have to check for a written similarity or even phonetic similarity with the words from the exemplary `employee` table. If similarities are rated very highly according to a percentage value, these could then be passed on to DDQL as word lists accordingly.

Case 3: Speech-to-Text for Numbers. Here, it is shown how Whisper transcribes spoken numbers. Apart from characters like '<whitespace>', '.', ',' and '-' that could appear between the digits after the transcription, a closer look reveals that in the following examples with nine identical digits, an additional digit is added occasionally. Furthermore Whisper seems to translate single digits into written-out numbers or even words that are phonetically very close to these numbers.

Examples:

SSN (ID Number): `111111111` ↦ `1 1 1 1 1 1 1 1 1`;
`444444444` ↦ `4 4 4 4 4 4 4 4 4`;
`555555555` ↦ `5 5 5 5 5 5 5 5 5` | `5-5-5-5-5-5-5`;
`666666666` ↦ `6666666666`; `777777777` ↦ `7777 7777 7` | `7777 7777 7 7`;
Money: `30000` ↦ `3 0 0 0 0` | `30,000` | `3-0-0-0-0`;
`25000` ↦ `25,000` | `2500` | `2 5 0 0 0`; `43000` ↦ `43,000` | `4 3 0 0 0`;
Single Digits: `5` ↦ `five` | `fives`; `4` ↦ `four` | `for`; `1` ↦ `one`

An intermediate tool between Whisper and DDQL could eliminate the separation of the numbers by intermediate characters as described in Case 1. The occurrence of too many or too few digits could be fixed by checking for equality of the first digits compared to the entries of the exemplary `employee` table. This would also make it possible not to have to enter all the digits for a number. For the written-out representation of a number, the intermediate tool would have to pass the equivalent in digits on to DDQL. However, evaluating `for` as `4` and passing it on to DDQL as such would probably not make much sense, since the word `for` occurs frequently in English sentences and would therefor be evaluated as a number each time the speaker uses the word `for`.

Case 4: Transcription of Date Values. Whisper can transcribe dates read out loud number by number. The cases in which the months were pronounced or the spoken order of a date were differed is beyond consideration here.

Examples:

`1955-01-09` ↦ `1955.1.9`; `1945-12-08` ↦ `1945 12 8`;
`1958-07-19` ↦ `1958 7 19` | `1958 719`; `1931-06-20` ↦ `1931 620`;
`1952-09-15` ↦ `1952 9 15`; `1927-11-10` ↦ `1927 11 10`

An intermediate tool for transforming the recognized words to more suitable words would have to derive all possible dates that can be formed from the input and compare them with the values in the database.

Case 5: Transcription of Addresses. Here the transcription of addresses is looked on more closely. It can be seen that similar errors occur here, as they already emerged in the previous cases.

Examples:

`tx` (spoken texas) ↦ `texas`;
`3321 castle spring tx` ↦ `3,3,2,1 castle spring texas` |
`3 3 2 1 kassel spring texas` | `3321 carson spring texas`;
`291 berry bellaire tx` ↦ `291 berry bel air texas`

Additionally `tx` was always transcribed to `texas`, but this is because it was recorded that way. For an intermediate tool, several challenges would arise at the same time, as an address can consist of numbers, words and also abbreviations. Accordingly, an intermediate tool would have to combine several of the approaches from the previous cases. Apart from that, another table could be introduced that divides the address into its different components. This could be handled more easily by an intermediate tool, as described in Cases 1 to 4.

Case 6: Diverse. Proper Nouns like `ProductX`, abreviations like `SSN`, `MGRSSN`, and ambiguities like `SME` (subject meta expert, small medium enterprises) have to be handled. Moreover, `MGR` can abbreviate `Manager`. Nowaday, `Laser`, `X-Ray`, `LGBTQ`, `ChatGPT` and `Open AI` are commonly known names. Furthermore, as in the example above with `tx`, it should be ensured that common designations, such as `USA`, `EU` and `ASEAN` should be recognised and brought into a context processable by DDQL.

In conclusion, it can be said that an intermediate tool between Whisper and DDQL should be mostly dependent on data type in order to effectively enable corrections for DDQL. It is important that all data types that occur in the accessed database are considered. A person with expert knowledge about the database could then, for example in the case of this case study, set the intermediate tool accordingly so that addresses are recognised and handled like a compound data type.

The above mappings were composed mainly in relation to the transcription of the `medium` model of Whisper. In fact, the same categories of error types occurred in the `small` and `base` models, but with higher frequency and partially more bias. This is clearly due to the fact that `small` and `base` are smaller and less accurate speech-to-text models than `medium`, but thus are faster with respect to runtime [2]. When designing the intermediate tool between Whisper and DDQL, this trade-off of accuracy and runtime can be utilized and should be further investigated in the implementation.

4.3 Text–to–Text Corrections with Language Technology

As we have seen in the previous section, it is not only important to correctly interpret a user's request in natural language, but it is also of great importance to correct words incorrectly transcribed by a speech–to–text (STT) engine in the case of spoken requests.

We use a custom fuzzy matching approach, adjusted to the observed wrong transcriptions of Whisper, to counteract this problem. One main part of our fuzzy matching algorithm is based on the fact that STT engines try to identify the correct phonemes of spoken words, e.g. in an audio file, in order to map those phonemes to the correct spelling. Therefore it is much more likely, that words are transcribed incorrectly to similar sounding words, as we saw in, e.g., Case 2 (`salary` $\mapsto$ `celery`).

Our *phonetic matching* approach, to find homophones in the database, generates a phonetic key for each word, which tries to represent the most important parts of its pronounciation. This key generation is mainly derived from the known metaphone–algorithm [14].

Hereafter, for every case, mentioned in Subsection 4.2, it will be discussed how our fuzzy matching approach addresses those listed mistranscriptions.

Case 1: Separation of Words by Whisper. The phonetic key generation does not pay attention to any whitespaces, hyphen etc.. Therefore `dno` and `d-n-o` have the same phonetic key, same goes with `f name` and `fname`. If we are not able to find an exact match with a given value in the database, we compare the phonetic keys and pick those with the highest similarity. This reduces the amount of format issues significantly.

Nevertheless, we have to pay attention to those cases, in which word separations differentiate values in the database. For example if there are an `Anna Lyn` and an `Annalyn` in the database. A user may want to find `Anna Lyn`, but Whisper transcribes it incorrectly to `Anna Lin`. All three names have the same phonetic key. As it is more likely that Whisper uses the correct word separations, `Anna Lyn` will be preferred as a match.

Case 2: Transcriptions of Similar or Phonetically Similar Words. Here, several, but not all of the mistranscriptions have the same phonetic key than their origin. `salary` and `celery` have the same phonetic key of `SLR`. `bdate`, `beat it` and `be date` have also the same phonetic key of `BTT`. `franklin t wong (FRNKLNTWNK)` and `franklin t wom (FRNKLNTWM)` or `Pork (PRK)` and `Borg (BRK)` have different phonetic keys. In order to be able to rank the similarities of phonetic keys, we use a custom *Levenshtein Distance*, where we take into account, that some differences of phonetic keys are more likely to occur than others. It is much more likely, that a difference of `P` and `B` in the phonetic keys is based on a mistranscription of Whisper than `P` and `L`. Therefore differences like `P` and `B` contribute to a much lesser degree to the calculated distance than differences like `P` and `L`.

We also pay attention to those characters in the near surrounding of a detected difference. Keys like `NL` from the word `null` and `N` from `no` get a lower distance than `NK` from `nik` and `N`. It is much more likely, that Whisper mistranslates `null` to `no` than it does with `nik`.

These adjustments are based on the pronounciation itself and our testing with Whisper. Of course these rules are dependent of the used language. Right now, we only consider english as spoken language. It would also be possible to deal with other languages, but for every other language, an own phonetic algorithm has to be created.

Case 3: Speech–to–Text for Numbers. In the case of numbers, all spelled out digits are first converted to their corresponding digits. Then, based on the amount of consecutive digits or the given context, it is verified whether this could be a date. If this is not the case, all consecutive digits, which are separated only by whitespaces or similar, are merged. Then we search in the database for numbers that contain the given number, or if the given number contains a number in the database. This gives us the opportunity, to address numbers, which are not known in their entirety. If the user only knows the first few digits of the `SSN` of an employee, it is possible to ask the user if a certain `SSN`, containing the given number, was the `SSN` he meant. If there is more than one match with a high similarity, the user can choose for which of them he wants to get the according result.

It is not recommended to combine this approach with the Levenshtein Distance, because in our testing, Whisper usually did not mistranscribe one digit into another. It was more likely that Whisper translated a `4` into `for`. Hence we consider cases, in which `for` was not changed and cases, in which `for` was changed to `4`. Those results are preferred, that did not change anything.

Case 4: Transcription of Date Values. The usual way for a user to provide dates, is to pronounce the month or year directly instead of spelling it out digit by digit. This works quite well with Whisper. But if just the according digits are mentioned, we verify if a certain set of digits could be a date. The different aspects we consider are the length and the value of the number. From those possible date representations with the wrong format, we create all dates that are eligible, compare them with the values in the database and ask the user if he meant this specific date. `1931 620` for example, can only be `1931-06-20`.

Case 5: Transcription of Addresses. Many approaches discussed so far help us to interpret a misunderstood address the correct way. `berry bellaire tx (BRBLRTX)` and `berry bel air texas (BRBLRTXS)` have very similar phonetic representations, our algorithm will take this kind of mistranslations into account. As we are also looking for matching substrings, it would also be possible to respond to from Whisper transcribed queries like *Give me all employees, who live in berry bel air texas.*

Case 6: Diverse. Abbreviations that can occur in the database are addressed by a simple dictionary, which stores those words and their according abbreviation.

It is not only possible, to compare words according to spelling or pronounciation, it is also possible to compare words according to their meaning. If we have a look at the query *How much does the employee Borg earn*, then `earn` and `salary` have a similar meaning. This can give us the opportunity to interpret a wider range of variations in the formulation of queries in natural language correctly.

5 The Declarative Database Query Language DDQL

In DDQL, the generation of queries is based on declarative concepts from logic programming. The knowledge-based compilation of keyword–queries to Datalog and/or SQL 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.

The declarative programming toolkit Declare [20] and its deductive database system DDbase already have functionality for evaluating database queries formulated using extensions of Datalog. Even hybrid queries including different knowledge representation formalisms are possible in DDbase. Relational databases can be accessed using Datalog or SQL queries and ODBC; for XML processing, a query, transformation and update language FNQUERY is given in [21]. In DDbase, Datalog programs are *evaluated bottom–up* with *stratified fixpoint* computation, and they can – possibly, for simple forms of aggregation – be compiled to SQL queries; often Datalog 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.[1] For non–stratified default negation, DDbase could use *answer set* solvers, cf. [5], if there are no aggregation literals.

5.1 Knowledge–Based Compilation of Queries

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

$$\mathcal{Q}_N \to Q_A \to \mathcal{Q}_F \to \mathcal{Q}_D.$$

The result tables are shown in a graphical interface in Figure 1. First, using Definite Clause Grammars (DCGs, see, e.g., [4, 8]), an annotated query $\mathcal{Q}_A$ is generated. Using, e.g., the following grammar rule in the extended DCG formalism introduced in [19], also the resulting parse tree can be computed:

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

[1] 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, beginning with the lowest.

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. 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. In summary, the annotation $\mathcal{Q}_N \rightarrow \mathcal{Q}_A$ depends on – and can be refined by – the speech–to–text recognition by Whisper and the text–to–text corrections.

Then, the compilation of the annotated queries $\mathcal{Q}_A$ to SQL or Datalog is done by adding suitable *join conditions* 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 SQL, queries usually have the form `select-from-where-group by`: The `select` part specifies the attribute values that we are interested in and can contain aggregations of attribute values. The `from` part specifies the tables that should be used to obtain the answer. The intelligent `where` part describes the join-conditions for the envolved tables. For transforming $\mathcal{Q}_A$ to $\mathcal{Q}_F$ and $\mathcal{Q}_D$, in many cases, 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 databases without given foreign key constraints, join conditions can be inferred from previous SQL queries in the log file: 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 systen and presented in XML to the user and analysed with Prolog.

Obviously, query $\mathcal{Q}_1$ can simply be computed from table `employee`, since `salary` is an attribute of `employee` and `Borg` is an attribute value in this table of the attribute `lname`. Sometimes the word `Borg` could have been entered or spoken wrongly as, e.g., `Bork` with an ending "`k`". Unless corrected by similarity search, the query would find no result. For $\mathcal{Q}_2$, the situation is more complicated. `Research` is an attribute value in the table `department`, and DDQL has to find the suitable join condition `employee.DNO = department.DNUMBER`. Query $\mathcal{Q}_3$ requires the aggregation function `sum` on the `salary` values, and the departments have to be `grouped by name`. 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 is controlling a given project and that is located in a given city, then two different link trees might be used.

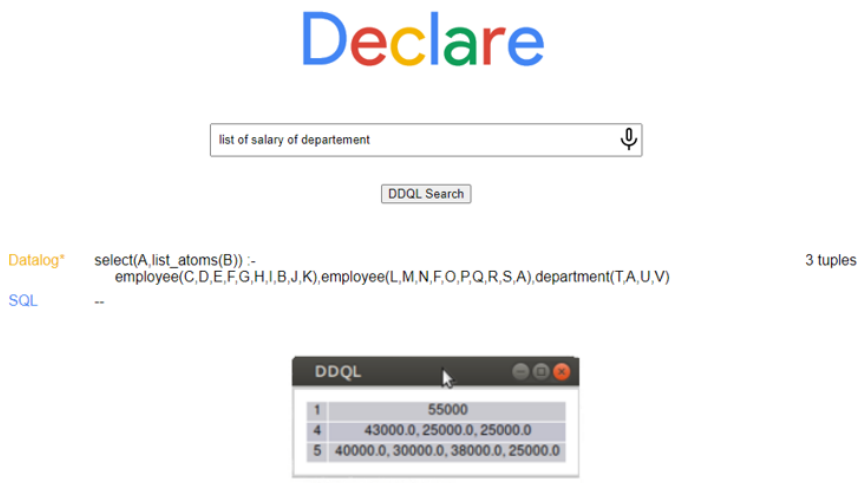

Figure 1 Graphical User Interface of DDQL: Query $\mathcal{Q}_4$.

5.2 The Graphical User Interface

A prototype of the graphical user interface (GUI) of DDQL is shown in Figure 1. The query keywords can be entered by speech after pressing the microphone icon or typed separated by blanks into the input box. Then, the a list of possible search queries in Datalog or SQL is generated, which 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 1 shows such a case with the user-defined aggregation function `list` for the query $\mathcal{Q}_4$. For this example, only one Datalog answer was found, which leads to the result table shown in the figure. Here, a unique answer was returned by DDQL. The join condition between the first and the third Datalog atom of the query is given by the middle atom for `employee`: The two occurences of `F` for the social security number identify the two `employee` atoms, and `A` in the `department` atom ensures the SQL condition `employee.DNO = department.DNUMBER`.

5.3 Integration of Whisper into Declare in a Docker-Based Tool

To integrate Whisper with DDQL, several prerequisites must be met. First, Whisper must be installed. For this, we need a current version of Python together with the PyTorch framework and FFmpeg. For DDQL, we need the Declare developers toolkit, which contains DDQL to be installed along with SWI–Prolog version 7.6.4 and ODBC to access the database. In addition, we need a Python script which was developed to provide the command line interface and connect Whisper and DDQL. To connect DDQL to the desired database, we need to manually create a DSN with the required information. Moreover, the client environment must be able to offer an X display for the graphical component of Declare to work.

Since the installation relies on a delicate balance of versions for all dependencies, the process is somewhat cumbersome. On top of that, several other inconveniences may occur. Whisper seems to have trouble with other installed Python packages. To avoid this, Whisper should be installed in an isolated virtual environment. Another inconvenience could be the

operating system used itself. It can influence the installation of the individual components and can lead to unexpected behavior, since not all parts are tested on every variant of every operating system.

To increase usability as well as other features such as portability and availability, several approaches can be considered. As a first step, we have *dockerized* Declare[2] to conveniently provide a Declare distribution with all its dependencies. To use Whisper we either install it as mentioned in the description above or use an existing Docker image of Whisper. However, we need to install Python to run the script, which needs some small modifications, to provide the CLI for the tool. We also have to create the DSN to connect DDQL with the database. Another approach could be to combine all the required dependencies into a single Docker image, so that only Docker needs to be installed to use the tool via CLI. Then, the database connection can be created inside the Docker container by passing all the required information as environment variables. This maximizes the portability and availability of the program, as we can easily retrieve the Docker image from any machine and run it as needed. To further optimize the usability of this approach, we could integrate the python script into a web server that runs inside this single Docker container. With this step, we are able to install it both on a local machine or on a server to provide the tool to the user with an appropriate web page.

6 Conclusions and Future Work

In this paper, we have shown how queries to relational databases can be answered in keyword–based natural language interfaces using intelligent, cooperative techniques. The concepts mentioned in the query are linked based on contextual background knowledge, mainly from the database schema. 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*.

We have containerized Declare – inluding DDQL – in a docker image and added a *voice recognition* part based on Whisper. In this paper, we have described the techniques used for recognizing the correct queries in the context of the database. DDQL can benefit from interleaving it with Whisper. Later, we are planning to use the voice recognition also for DDQL on mobile devices and the knowledge acquisition could be done with a voice assistant based on a domain–specific language, see [17]. Currently, we are training the Whisper tool on different databases with varying schemata and instances. We would also like to add voice output, i.e. spoken answers, such that the returned table could be read to the user. At the moment the tool Whisper can transcribe 99 languages and we are planning to add multilinguality to DDQL.

Graphical user interfaces (GUIs) might also be acted upon. Then, the GUI is the context; it might also be described in a structured form such as HTML 5 [16] or the XML user interface language XUL [7], which has been used by Mozilla for the firefox web browser; then text matching has to be done on the interface components with reflection over a GUI. In Declare, XUL interfaces can already be interpreted, and we are working on Prolog links in HTML.

[2] `https://hub.docker.com/r/declaredocker/declare`

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 Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, Ilya Sutskever. Robust Speech Recognition via Large-Scale Weak Supervision, 2022. arXiv preprint arXiv:2212.04356. URL: https://arxiv.org/pdf/2212.04356.pdf.

3 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.

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

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

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

7 Christian Schneiker, Dietmar Seipel. Declarative Web Programming with Prolog and XUL. In *Proc. 26th Workshop on Logic Programming (WLP)*, 2012.

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

9 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.

10 Dietmar Seipel, Daniel Weidner, Salvador Abreu. Intelligent Query Answering with Contextual Knowledge for Relational Databases. *10th Symposium on Languages, Applications and Technologies (SLATE)*, 2021.

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

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

13 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 arXiv:1412.5567*, 2014.

14 Lawrence Philips. Hanging on the Metaphone, Computer Language, Vol.7, No.12, 1990.

15 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.

16 Matthew MacDonald. HTML 5: The Missing Manual, 2nd Edition. O'Reilly Media, Inc, 2011.

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 OpenAI. Introducing ChatGPT and Whisper APIs. URL: https://openai.com/blog/introducing-chatgpt-and-whisper-apis.

19 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.

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

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

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

A Appendix

We have used the relational database COMPANY from [11] for exemplifying our approach. The database schema in Figure 2 contains 6 entity/relationship types and 8 referential integrity constraints between them (links given by arrows). Some corresponding database tables have been shown in Section 3 earlier in this paper.

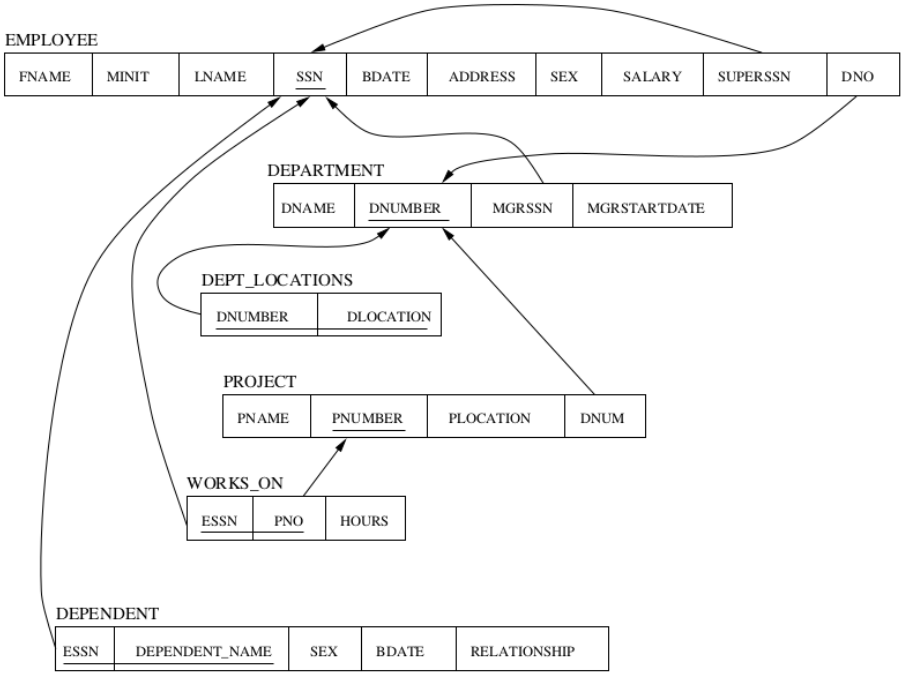

Figure 2 Referential Integrity Constraints for the Relational Database COMPANY.