inSQeLto: a Query Language for Probabilistic Databases

Jochem Groot Roessink
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.grootroessink@student.utwente.nl

ABSTRACT

A Database Management System (DBMS) is a useful tool to store, manipulate and query data. However, it is always possible that the database contains incorrect data. In most DBMSs, there is no direct way to indicate which data might be incorrect. For this reason, probabilistic databases exist, which have the ability to express how correct data is. An example of such a database is DuBio, currently under development at the University of Twente. DuBio has functionality that can keep track of the probability of correctness for its data and this probability can be presented to users when they query some data. This will help in identifying incorrect data which can then be rectified. The current way of querying data in DuBio is only meant to be a temporary solution and some queries can be quite lengthy and complicated. The goal of this research is to design the query language inSQeLto that can be used for probabilistic databases, and DuBio specifically. While the resulting language looks the same as standard SQL, it works differently under the hood. Namely, the inSQeLto queries are mapped onto DuBio queries, and these can perform the probabilistic functionality. Because of this existing SQL queries can now be used to interact with DuBio in a probabilistic way. This language could help spread the use of probabilistic databases. This could lead to a decrease in the amount of incorrect data that is being used, which will have a positive impact on an increasingly digital world.

Keywords

Database Management System, Probabilistic/Uncertain Data, Structured Query Language, Domain-Specific Language

1. INTRODUCTION

A Database Management System (DBMS) is a tool where one is able to store, manipulate and query data. Unfortunately, incorrect data might arise in such a database. Most DBMSs in use right now are deterministic and have no direct way of indicating what data might be incorrect or what the probability of correctness is for certain data. An example of a process where incorrect data may originate is data integration.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
35th Twente Student Conference on IT Jul. 2nd, 2021, Enschede, The Netherlands.
Copyright 2021, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

Data integration is the process of joining multiple database tables together to form just one. When one real-world object is represented in multiple of those source tables it should be represented in the new table just once. To achieve this the system needs to identify which entries in the source tables refer to the same object. This could be done by checking which entries use the same name. However, these names are usually manually filled in by multiple users. This could lead to data for one object being spread over multiple columns or one column containing heterogeneous data [7]. If a user now queries this table they might receive an incorrect answer because of that. Finding out such mistakes after the table is already created can be a difficult and time-consuming job, especially if large tables are used.

A probabilistic database is a DMBS that manages probabilistic (uncertain) data [11]. This type of DBMS can be used to mitigate some of the problems that are present in deterministic databases, like those described above. Besides all the functionality that deterministic DBMSs use, they can also store some extra information for every entry. This information describes in what possible world the corresponding entry may exist. This can then be used to calculate a probability of correctness for that entry. When a user queries a probabilistic database, they can also request the probability of correctness for every entry. This will let them know if some data might be incorrect, after which they could confirm or deny whether multiple entries reference the same object and, if so, which one is correct. This user feedback can help in improving the database [12].

At the University of Twente a new probabilistic database, called DuBio, is currently being developed [10]. DuBio is an extension of PostgreSQL[9], a deterministic DBMS, and adds probabilistic functionality. One goal of DuBio is to be scalable so that the performance of the system does not get drastically worse when it is used on larger databases.

For this research, a Domain-Specific Language (DSL) is designed for probabilistic databases and DuBio [8] specifically. This language is based on Structured Query Language (SQL) [2] and can be used to query and alter a DuBio database, while also allowing for probabilistic behaviour. The name for this language is inSQeLto, a combination of the Italian word ‘incerto’, which translates as uncertain, and SQL.

2. PROBLEM STATEMENT

Currently, since DuBio is an extension of PostgreSQL it can already be queried and altered with regular SQL. However, this is only meant as a temporary solution and most of these queries are lengthy and therefore not user-friendly.

Table 2 shows that queries for relatively simple functionality can be quite complex and long. This is why this project is in place, to design inSQeLto as a new DSL for DuBio. Queries in inSQeLto are mapped onto the current SQL solution in order to be executed by DuBio. Therefore, the goal for this project is to design a language that can be used for interacting with probabilistic databases, and DuBio specifically.

2.1 Research Questions

From the problem statement the following research question arises:

How to design a query language for probabilistic databases, and DuBio specifically, that is intuitive to use for people that have experience in working with SQL or probabilistic databases?

To help answer this research question the following sub-questions are created, which will have to be answered before the main question:

- **SQ1**: How are existing SQL constructs[1] interpreted so that they work for both probabilistic and deterministic tables in DuBio?
- **SQ2**: What new constructs are desired and how are they added to inSQeLto?
- **SQ3**: Does the inSQeLto compiler require a connection to a database or can it run without such a connection, and why?

3. REQUIREMENTS

These are the functional requirements for the design of inSQeLto.

- **R1:** The language should be based on SQL since this will be intuitive to people that have experience in working with DuBio, other probabilistic databases that also use an SQL-based language, or standard databases that use SQL.
- **R2:** Existing SQL commands that do not involve probabilistic behaviour should not be converted.
- **R3:** The language should be able to be extended so that if not all desired functionality is realized for this project, it could still be completed later on.

And the following non-functional requirement is established as well:

- **R4:** Compile time should not be longer than 0.2 seconds. This is because one of the goals of DuBio is to operate fast, even on large databases. A compile time that would take longer than this would hinder that goal. Since the implementation for this project is not intended as a final product, failing this requirement is not a major problem but might expose some problems with the system. These problems can then be taken into account during implementation of the next version.

[1]"Syntactically allowable part of a program that may be formed from one or more lexical tokens in accordance with the rules of a programming language" [4].

4. RELATED WORK

This section discusses several existing probabilistic database systems. All of the systems discussed have their own way of being queried, these are investigated and used for inspiration during the development of inSQeLto. These DBMSs are the following: Trio [13], MCDB [5], MayBMS [3] and MystiQ [1].

Trio.

Trio is developed at Stanford University. Trio has its own query language, the Trio Query Language (TriQL). TriQL inherits most of the existing SQL constructs, together with some extra constructs for specific functionality. While most of the constructs are the same as SQL they are interpreted differently, so that they can be used for probabilistic behaviour. [13].

MCDB.

MCDB is created at the University of Florida together with the IBM Almaden Research Center. MCDB does not have its own query language but uses SQL to interact with the probabilistic data. To achieve this, some functions have been created that allow for probabilistic behaviour [5]. This is also how DuBio works (without inSQeLto).

MayBMS.

MayBMS is jointly developed at Oxford University and Cornell University. Its query language is the aptly named MayBMS Query Language. This query language extends SQL (although the aggregate functions from SQL are disabled) with some additional constructs. The standard SQL constructs can still be used normally and the additional constructs provide the language with probabilistic functionality [3].

MystiQ.

MystiQ is developed at the University of Washington and the American University of Beirut. There are two languages that are used to interact with MystiQ: the MystiQ data modeling language (mDML) and the MystiQ data definition language (mDDL). Just like MayBMS these extend SQL with new constructs that allow for probabilistic behaviour [1].

The one thing these query languages have in common is that they are based on SQL. This already makes all of them valuable for this research since one of the requirements for inSQeLto is that it is based on SQL as well. However, different methods are used: TriQL mostly uses existing SQL constructs but interprets them differently; MCDB still uses SQL with some extra functions; and the MayBMS Query Language, mDML and mDDL extend SQL but perform most probabilistic behaviour with new constructs. These different methods of creating an SQL-based language are all interesting for inSQeLto and are therefore explored during the development.

5. METHODOLOGY

This section explains what methodologies are necessary in order to answer all the sub-questions and, subsequently, the main research question. This methodology is about design science, where a problem is defined and a treatment for the problem is designed, implemented, and finally validated [14].

5.1 Language Design

There are several possible methods for performing certain desired functionality in inSQeLto. Such a method can either be an entirely new construct; an existing SQL construct that is interpreted differently; or an existing SQL construct with calls to probabilistic functions. These methods are compared with each other during user testing.

Two steps are taken to use and test the language with DuBio. First, a parser is created to parse the language into a parse tree, which can then be converted into SQL queries that DuBio can execute. Parsec is used to create this parser, mainly because it is designed to be fast and simple to use [6]. Since it is fast, it can help in passing **R4**. Second, a DuBio test environment is created where test scenarios can be created in order to test the generated SQL queries.

5.2 User Testing for Design Choices

During language design, three different methods for querying probabilistic data are designed. User testing is used to decide which is of these methods stays in the final language. There are two groups of users: group 1 consist of people that have experience in working with probabilistic databases; and group 2 consist of people that only have experience in working with SQL.

The three methods that are designed are the following: method 1 uses the standard SQL SELECT command where the probability is automatically added to the results; method 2 also uses the standard SQL SELECT command but when the probability is desired it has to be called explicitly in the same way one would call a column; method 3 uses a new command that is meant specifically for probabilistic data. The documentation for these methods can be found in Appendix B.

During user testing, a user is faced with a task where they have to query some data from a probabilistic table together with the corresponding probability. They are provided with the documentation for each method and then have to try to complete the task using each method. After they have done this they are asked to give feedback for each method, pick their favourite one and explain why. This feedback helps in deciding what method is best fit for querying probabilistic data. Other questions that the participants were asked were related to minor characteristics of the language like whether the probability should be presented as a percentage or a number between 0 and 1. If one option for these questions is overwhelmingly preferred it is incorporated into the language.

5.3 SQ1

To see to what part of the system the sub-questions are related see Figure 1.

How are existing SQL constructs interpreted?

During implementation of the system and the testing of queries in the DuBio test environment, it becomes clear how each construct should be converted. They should be converted so that when the output is executed, the desired functionality of the original input is performed, and it now works for both probabilistic and deterministic tables.

5.4 SQ2

What new constructs are desired and how are they added to inSQeLto?

When there is no logical existing construct to use for some desired functionality a new construct is created. Just like the method for **SQ1** it becomes clear during implementation of the system and testing in the test environment how these constructs should be converted. While existing constructs are mostly just altered, new ones are in their entirety replaced by existing constructs during conversion. This is done so that they can be executed by DuBio.

Observation	Yes	No
User has experience in working with SQL	8	0
User has experience in working with probabilistic databases	3	5
User was able to complete the task with every method	8	0
User prefers method 1	4	4
User prefers method 2	4	4
User prefers method 3	0	8
User prefers probability represented as a number between 0 and 1 over a percentage	5	2

Table 1. Summarized results of the user testing for the design choices, which was done with eight participants. The 'Yes' column counts the participants that were observed to choose or do what is described in the first column, the 'No' column counts those who did not. If the sum of these columns is less than eight in a row it means the rest of the participants were neutral.

5.5 SQ3

Does the inSQeLto compiler require a connection to a database?

First, queries with probabilistic functionality are tested in the DuBio test environment. After that, it is known whether such queries can be generated without the compiler knowing anything about the database they are used on. If that is not the case the compiler requires a connection with the database.

5.6 Research Question

How to design a query language for probabilistic databases?

When the sub-questions are answered by using their corresponding methodology, they result in the design of a system that is able to compile inSQeLto queries and convert them to SQL queries that can be executed by DuBio. This system together with the methodology for language design creates several options for what the query language can look like. The user testing with people experienced with SQL or probabilistic databases is used to find out which of these options is the most intuitive to them. All of this together results in an answer to the main research question.

6. USER TESTING RESULTS

Table 1 provides a summary of the results of the user testing for design choices. It is apparent that there are eight participants in total, three of which are considered to be from group 1 and five from group 2. All of these participants were able to complete the task with all methods correctly (disregarding some irrelevant mistakes like using a number between 0 and 1 where they were asked to use a percentage). The paragraphs below provide some insight into some of the results.

First of all, none of the participants thought that method 3 was the best. This makes sense since using this method would require users to know which tables are probabilistic in order to know what command to use. So, a command

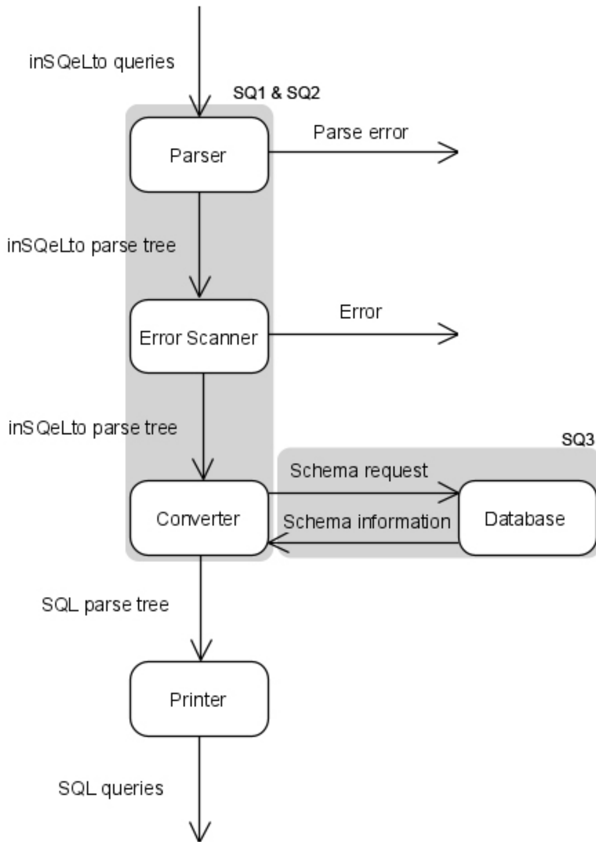

Figure 1. Global description of the system. The gray areas show to which processes the sub-questions are related.

that can be used for both non-probabilistic and probabilistic data was preferred by all.

Furthermore, the initial expectation was that participants from group 1 would prefer method 1, while participants from group 2 would not have a preference. This was expected because it would mean that users would always get the probability of their query results and incorrect data would be recognized more often. The idea was that participants from group 1 would view that as important while participants from group 2 would not know about this. While two out of these three participants from this group indeed preferred this option, the reason for their choice was different. Also, too few participants were used to confirm such an expectation. Overall, the participant's choices for the first two methods were evenly divided.

Finally, most participants preferred using a number between 0 and 1 over a percentage for the probability. Overall, there was no significant difference between the answers of group 1 and the answers of group 2.

7. TECHNICAL CONTRIBUTION

By analyzing what is possible during implementation the following design is created. The first step of the system is to parse the inSQeLto input. After it is parsed the parse tree is scanned for anything that can be parsed but that is not allowed, like more than one FROM clause in a command or a wrong order in a command. If nothing illegal is detected, the next step is to convert the parse tree to a new one. This new parse tree is printed in the final step in order to generate an SQL query that performs the desired functionality. This system is also described in Figure 1.

The resulting language is still very similar to SQL. In fact, constructs that are not (yet) recognized by the inSQeLto compiler are parsed as a string. These strings are not converted in any way and printed again in the result. This makes it possible to run normal SQL queries as well. The compiler is created with the idea in mind, that it can be extended, see section 8.3. When a new construct is added to the compiler, it will be parsed and converted as intended. The implementation can be found at the GitHub page of DuBio [10].

7.1 Grammar

The simplified grammar for inSQeLto is displayed below. Here, *text* is either a single word consisting of letters and numbers or any string as long as it is between double quotes. If *text* is displayed between the single quotes in the grammar it can also be any string. Second, *name* is one or more occurrences of *text*, separated by dots. Furthermore, *number* is any number, *op* is an arithmetic operator (+, -, *, etc.) and *comp* is a comparison operator (>, =, <>, etc.). Finally, *other* can be any string, it is used to parse SQL commands that are not yet recognized by inSQeLto and are not converted.

```
program → command ";" program
program → command

command → "SELECT" values clauses
command → other

values → value "," values
values → value

value → expr
value → cond
value → "(" values ")"
value → "'" text "'"

expr → "(" expr ")"
expr → expr op expr
expr → number
expr → name
expr → text "(" values ")"

cond → "(" cond ")"
cond → "TRUE"
cond → "FALSE"
cond → cond "AND" cond
cond → cond "OR" cond
cond → "NOT" cond
cond → expr comp expr

clauses → clause clauses
clauses → clause | ""

clause → "FROM" tables
clause → "WHERE" cond
clause → "GROUP BY" name
clause → "ORDER BY" name ord
clause → "LIMIT" number
clause → "INTO" name

tables → name "," tables
tables → name

ord → "DESC" | "ASC" | ""
```

7.2 Query Mapping

By analyzing the results of the user testing for the design choices the following query mapping is created. The construct that is already implemented is the SELECT command. During the language design, several possibilities for this command were created. All user testing participants preferred to use a command that can be used for both deterministic and probabilistic behaviour, but it was not made clear whether the probability should be called or not. Eventually, explicitly calling the probability was chosen, mainly because of the computation time for this probability. If a user does not desire the probability they can request the results without adding a call to it and it is not calculated. It was also decided to use a number between 0 and 1 as the probability as opposed to using a percentage.

There are two requirements for a SELECT query to be converted: the probability has to be called and at least one of the tables used has to be probabilistic. If a call to the probability is made but only deterministic tables are used, the call is replaced by the number 1. If both these requirements are passed, two changes happen. First, the table that stores the probability dictionaries is added, as well as a condition that ensures the correct dictionary is used. Second, the probability call is changed by a call to a function that can calculate the probability, taking the selected dictionary and the sentences of the probabilistic tables as arguments. These sentences describe for each entry of a table in what case the entry is correct [10]. If more than one probabilistic table is used the sentences have to be combined with the AND-operator (all sentences have to be true for the combined sentence to be true). If a GROUP BY clause is used in the query, the sentence has to be wrapped in the *agg_or* function. The *agg_or* function combines the sentences of entries belonging to a group with the OR-operator (only one sentence has to be true for the combined sentence to be true).

Table 2 shows some queries before and after they are converted. The first query does not call the probability (*_prob*) so is not converted. The second query does call the probability and this call is replaced by a function call that calculates the probability. The dictionary that stores the probabilities is added to the tables of the query and a condition is added which ensures the correct dictionary is used. Query three also calls the probability but on a deterministic table so this call is replaced by the number 1. The fourth query uses two probabilistic tables, so their sentence attributes have to be combined in the argument for the probability function (so that the entries that are connected both have to be correct). The final query uses the GROUP BY clause so the sentence has to be wrapped in the *agg_or* function.

Input (inSQeLto)	Output (DuBio SQL)
select id, lname from person	SELECT id, lname FROM person
select id, lname, _prob from person	SELECT id, lname, round(prob(_dict.dict, person._sentence) ::numeric,3) AS probability FROM person, _dict WHERE _dict.name = 'mydict'
select id, lname, _prob from person_det	SELECT id, lname, 1 AS probability FROM person_det
select order.oid, customer.name from order join customer on order.pid = customer.pid	SELECT order.oid, customer.name, round(prob(_dict.dict, **order._sentence & customer._sentence**) ::numeric,3) AS probability FROM order JOIN customer ON order.pid = customer.pid, _dict WHERE _dict.name = 'mydict'
select lname, _prob from person group by lname	SELECT lname, round(prob(**sum**(_dict .dict), **agg_or**(person._sentence)) ::numeric,3) AS probability FROM person, _dict WHERE _dict.name = 'mydict' GROUP BY lname

Table 2. Input and output of some inSQeLto queries.

8. VERIFICATION

This section checks for all the set requirements whether they have been met by the system described in section 7.

8.1 R1

The language should be based on SQL.

As of now, only querying is implemented. The existing SQL SELECT construct is used for this, and the calculated probability can be called as one would call a column. So, at this moment inSQeLto qualifies as SQL-based. Functionality that is added in the future could use entirely new constructs. These new constructs should use the same way of calling columns and tables and a similar structure to SQL to keep the language SQL-based. This can also be verified by user testing with people that have experience with SQL.

8.2 R2

Existing SQL commands without probabilistic behaviour should not be converted.

If none of the tables in the SELECT command is probabilistic the command is not converted. One exception to this is that if the probability is called, it is replaced by the number 1 so that the query can still be executed. Other SQL commands are parsed as a string and then printed again without being converted. Therefore, this requirement is passed.

8.3 R3

The language should be able to be extended.

Every part of the system is designed to be extensible. A new construct can be added to the grammar and the parser can be extended to parse that construct. Furthermore, the error scanner can call errors if needed and the converter can make changes to the parsed construct if desired. Finally, the printer can print it to generate the output SQL queries. This ensures that new constructs can be added and that the requirement is passed.

8.4 R4

Compile time should not be longer than 0.2 seconds.

See Table 3 for the compile time measurements for different types of queries. The different parts of the system were tested by measuring the compilation time of differ-

ent queries. While most of these queries easily passed the requirement, some took a lot longer than was deemed acceptable. These queries were the ones that contained a call to a function ("select **concat(fname, ' ', lname)** from person"). It was quickly found that the parser was the limiting factor for the compile time for these queries. It was also noticed that whether typecasting ("select id**::numeric**, fname from person") was enabled in the parser had a significant influence on the compile time.

For queries without function calls that are tested, the compile time always passes the requirement. However, with the current implementation, the compile time grows almost linearly with the number of queries in a command. Luckily, this can be attributed to the fact that internet speed is the limiting factor for these queries and requests to the database are sent one by one for each query. This is confirmed by the fact that a query without tables, which requires no database request, compiles much faster than the same query with tables (see Table 3). It can easily be altered so that only one request has to be sent at the beginning of the compilation. Conversion of the queries does not seem to have a significant effect on the compile time.

The reason that the parser is a lot slower when function calls are made is that it parses values (see section 7.1). A function call is in an expression, but an expression can also be at the beginning of a condition. Since the value is parsed from left to right it first has to check if the expression is a part of a condition before it can be parsed as an expression. This will not take very long for most queries, but the arguments of a function call are values as well. Parsing when there are nested values takes an unacceptable amount of time. When typecasting is added to the parser the compile time gets even slower since the parser now first tries to parse a value with a typecast before it can parse value without one.

Because of these reasons this requirement has been failed. However, the goal for this project was to design the language and the implementation was mostly intended to find out how to convert queries. The requirement was mainly set to find which parts of the system could hinder the performance of the system and how to improve on those in later versions. Sections 9.5.3 and 9.5.4 describe future work that could ensure that a newer version of the system passes the requirement.

ID	Description	Compile time (ms) A	Compile time (ms) B
1	query no tables	9	13
2	query with table	37	41
3	query with probability	40	59
4	query and update command (not parsed)	41	59
5	two queries	72	81
6	three queries and two update commands (not parsed)	104	121
7	same as above but no tables	22	46
8	group by	41	38
9	order by	37	47
10	query with function call	144	718
11	two queries with function calls	460	3799

Table 3. Compile time for different sorts of queries. The A and B columns for the compilation time refer to running with parsing for typecasting disabled and enabled respectively. These values are the average of five runs. The exact query used for each row can be found in Appendix A by their id.

9. CONCLUSION

This research has put in the groundwork for a query language that is designed for probabilistic databases and can be used for DuBio specifically. The conclusions to the research question and its sub-questions answer why some design decisions have been made and to what resulting design these have led. Since inSQeLto is not a finished product yet some further work has to be done, which is described in section 9.5. Overall, this language could help spread the use of probabilistic databases. This could lead to a decrease in the amount of incorrect data that is being used, which will have a positive impact on an increasingly digital world.

9.1 SQ1

How are existing SQL constructs interpreted?

During compilation, the input is parsed into a parse tree. This parse tree can then be converted so that columns, tables, conditions, etc. can be changed, deleted or added. For example, during conversion of the SELECT command, the table of dictionaries is added to the tables, a condition is added so that the correct dictionary is used, and the call to the probability is changed by a call to the probability function that takes a sentence and dictionary as input. After this conversion, the parse tree is printed again and presented to the user.

9.2 SQ2

What new constructs are desired and how are they added to inSQeLto?

New constructs are created for desired functionality for which there is no existing SQL construct that would be logical to use. There are already several ideas for functionality that could have their own constructs, see section 9.5.1. New constructs should be designed to be similar to existing SQL constructs so that **R1** is still passed (see section 8.1). Furthermore, the grammar and parser of inSQeLto are both designed to be extendable. This makes it possible to add these new constructs to the inSQeLto compiler. Just like how existing SQL constructs are converted (section 9.1), the new constructs can also be converted. However, the difference is that while existing constructs are only altered to include the probability for example, new constructs have to be converted to an existing SQL construct. After this conversion the parse tree is printed to SQL. This now only consists of existing SQL constructs that can be executed by DuBio and perform the desired functionality of the new construct.

9.3 SQ3

Does the inSQeLto compiler require a connection to a database?

It is possible to check if a table contains a certain column in SQL. This can be used to see which tables contain a sentence attribute, and are therefore probabilistic [10]. In order to calculate the combined probability of all the tables used in a query, all probabilistic tables could use this sentence column to calculate their probability and the non-probabilistic tables could just use a probability of 1. Unfortunately, all the columns used in an SQL query have to exist, even if they are never called because of the checks that happen. A possible solution could have been to add a command that adds the sentence column to all non-

probabilistic tables before the query and one that deletes them afterwards. But this would be quite cumbersome and might hinder performance as well. Therefore, sending a query to determine which of the tables in a query are probabilistic before the input is converted is a better solution. Knowing this, a connection to the database on compile time might not be necessarily required but is the best solution. This is also supported by the goal that eventually the compiler will run on the same server as the database anyway (see section 9.5.2).

9.4 Research Question

How to design a query language for probabilistic databases?

This conclusion is established by combining the conclusions of the sub-questions. The final design for inSQeLto is a query language that includes all SQL constructs, some of which are converted so that they automatically work for both deterministic and probabilistic functionality and some are left untouched. It can also be extended with new constructs for specific functionality, these new constructs should be similar in format to existing SQL constructs to still consider the language SQL-based. The compiler for inSQeLto does require a connection to a database so that it knows which tables in the database are probabilistic.

9.5 Future Work

This section explains what future work has to be done in order to make inSQeLto a success.

9.5.1 Extension

At this moment, only querying is implemented. It is also possible to use standard SQL commands to achieve other probabilistic behaviour. However, for the language to be complete more probabilistic functionality has to be designed and implemented. Examples of such functionality are:

- A command that can be used to merge multiple tables, while automatically creating sentences for the entries.
- A command that updates the probabilities for a variable in a dictionary.
- Extension of the SELECT command to improve expressiveness.

9.5.2 Integration with DuBio

As of now, the compiler is a loose system that can be installed on the client-side with its own user interface. While this is fine for demonstrating that everything works as a proof of concept, it is only a temporary solution. In the future, it should be running on the server-side together with the rest of DuBio. Just like how SQL commands can be sent to the server on a specific port, this subsystem for inSQeLto should have its own port. When receiving input on this port it should compile and convert it to SQL and send the result to the existing system.

9.5.3 Optimisations

Sometimes parsing can take a long time, mainly because the parser parses the input stream from left to right without any extra information. This input stream can sometimes be parsed as multiple nodes in the grammar, and all of these have to be checked. A possible solution might be to rebuild the grammar so that the parser never needs to look ahead far into the input stream, but this will hinder **R1**. Another solution could be to use a parser that has extra information, like what the input types are for certain functions, this could improve the performance of the parser. An even better solution might be the future work in section 9.5.4 since this uses a parser that is already optimized for SQL.

9.5.4 Use of the PostGreSQL Parser

SQL that is used for PostGreSQL supports quite complex commands. So extending the current system to support all these commands as well can be a time-consuming and complex operation. A better solution might be to use the already existing parser stage from PostGreSQL to parse an input, possibly with some modifications so that newly added constructs can be parsed as well. This should fix the shortcomings of the current system described in section 8.4 as well. Another advantage of doing this is that it can increase performance even more. This is because now the input is parsed, the converted parse tree is printed, and this result is parsed again. By using the PostGreSQL parser, the input only has to be parsed once, which will most likely be a quicker process.

10. REFERENCES

[1] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. Mystiq: a system for finding more answers by using probabilities. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 891–893, 2005.

[2] J. L. Harrington. *SQL clearly explained*. Morgan Kaufmann, Burlington, Massachusetts, 2010.

[3] J. Huang, L. Antova, C. Koch, and D. Olteanu. Maybms: A probabilistic database management system. In *ACM SIGMOD 2009*, June 2009.

[4] International Organization for Standardization. Information technology — Vocabulary. https://www.iso.org/obp/ui/iso:std:iso-iec:2382. Accessed: 25-06-2021.

[5] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 687–700, 2008.

[6] D. Leijen. Parsec, a fast combinator parser. October 2001.

[7] M. Magnani and D. Montesi. A survey on uncertainty management in data integration. *J. Data and Information Quality*, 2(1), July 2010.

[8] I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin. *Domain Engineering*. Springer, Berlin, Heidelberg, 2013.

[9] The PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/about/. Accessed: 28-04-2021.

[10] M. van Keulen. DuBio Wiki. https://github.com/utwente-db/DuBio/wiki. Accessed: 28-04-2021.

[11] M. van Keulen. *Probabilistic Data Integration*. Springer, Netherlands, Feb. 2018.

[12] M. van Keulen and A. de Keijzer. Qualitative effects of knowledge rules and user feedback in probabilistic data integration. *VLDB Journal*, 18(5):1191–1217, October 2009.

[13] J. Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*. Springer, 2008.

[14] R. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, Heidelberg, 2014.

APPENDIX

A. QUERIES

ID	Query
1	select 3
2	select lname from people
3	select id, lname _prob from people
4	select id, lname, _prob from people
5	select fname, _prob from people; select lname from people where id $<$ 3
6	select fname from people; update people set lname = 'doe' where id = 0; update people set fname = 'john' where id = 0; select fname from people; select fname, lname, _prob from people
7	select fname; update people set lname = 'doe' where id = 0; update people set fname = 'john' where id = 0; select fname; select fname, lname, _prob
8	select lname from people group by lname
9	select lname from people order by id asc
10	select lname from people group by lname having sum(id) $>$ 5
11	select fname, round(_prob*100) from people; select lname from people group by lname having sum(id) $>$ 5

B. METHOD DOCUMENTATION

B.1 Method 1

```
SELECT column1 [, column2, etc.]
FROM table1 [, table2, etc.]
[WHERE column [=|>|<|<=|>=] value]
[ORDER BY column [ASC|DESC]]
```

For this method the existing SQL SELECT query is used. If one or more of the tables that are used are probabilistic it will automatically calculate the probability of correctness (as a percentage) for every entry and display it as the last column in the results. This probability can also be called as a column in the WHERE and ORDER BY clauses by the name "_prob".

B.2 Method 2

```
SELECT column1 [, column2, etc.]
FROM table1 [, table2, etc.]
[WHERE column [=|>|<|<=|>=] value]
[ORDER BY column [ASC|DESC]]
```

For this method the existing SQL SELECT query is used. To display the probability of correctness (as a percentage) for each entry, "_prob" has to be called as a column, which can also be used in the WHERE and ORDER BY clauses.

B.3 Method 3

```
PROBSELECT column1 [, column2, etc.]
FROM table1 [, table2, etc.]
[WHERE column [=|>|<|<=|>=] value]
[ORDER BY column [ASC|DESC]]
```

For this method a new selection query is used, PROBSELECT. This query needs to be used if one of the tables is probabilistic. It will automatically calculate the probability of correctness (as a percentage) for every entry and display it as the last column in the results. This probability can also be called as a column in the WHERE and ORDER BY clauses by the name "_prob".