A Novel Approach to Aggregation Processing in Natural Language Interfaces to Databases

Abhijeet Gupta and **Rajeev Sangal**
Language Technologies Research Centre
International Institute of Information Technology - Hyderabad, India
abhijeet.gupta@research.iiit.ac.in and sangal@iiit.ac.in

Abstract

In aggregations, a function is applied to a set of values or entities in a database to yield a single value. Databases use a limited set of inbuilt functions to perform aggregations, hence, restricting NLIDB systems in processing domain based complex aggregations. In this paper, we introduce an aggregation processing framework, which can handle different types of aggregation operations in a natural language query, including direct quantitative as well as indirect qualitative aggregations, and those which combine quantifiers or relational operators with aggregations. Equally importantly, this is done as a separate layer independent of the processing capability in SQL, database query language. The framework has two distinct stages of processing. In the first stage, aggregations are performed on a data stream, and in the second stage the data stream can be further filtered, if required, on the result obtained in the first stage. With our approach, we have achieved an accuracy of 98.1% in processing aggregations and dealt with certain issues arising due to aggregations in SQL queries.

1 Introduction

A Natural Language Interface to Databases (NLIDB) system is an NLP application which accepts a query in natural language (NL) from the user and generates a SQL query from it. The SQL query retrieves the result from a database as a stream of elements and sends it back to the user. The database is a RDBMS which is a tabular representation of a domain's entities and their relations. These entities and relations have properties which are represented in the form of columns of the tables. These properties can represent quantitative or qualitative aspects of an entity. The data is stored in the form of rows in the tables. In the work done by Gupta et al. (2012), a dependency based syntactic parse of the NL query is generated to identify the lexical relations of the elements of the query. Thereafter, using the Computational Paninian Grammar (CPG) framework a semantic dependency parse is generated to identify the domain elements and their relations. This semantic parse helps in generating the SQL query of the given NL query. In this paper, we extend their work further by adding aggregate processing to the NLIDB system.

The need for aggregate processing arises because in a NL query the user may request for data which is derived by extracting and processing over a group of values. In such a scenario, the result (which is a single value output) is the outcome of computation over a group of values on certain criteria defined through a function.

For example: (1) 'List the average marks in the English course'. In this example, computing the result requires the retrieval of the set of marks awarded in the English course and then computing the average over the retrieved set.

This process of computation over a group of values by a function to produce a single value of more significant meaning or measurement is called *aggregation*. The type of aggregation is specified by lexical terms in a natural language. The terms frequently occur as modifiers with nouns, like 'average' in sen-

tence (1). The modifier is called an *aggregate*. In an utterance, aggregation may be specified on values directly, as in 'marks' in sentence (1), or on entities indirectly (e.g.'average students'). In indirect aggregate application, the aggregation is to be done on an attribute of the implied object.

The functions which are implied by aggregates are called *aggregate functions*. Conventionally, aggregations in NLIDB are handled by inserting the name of the aggregate functions in the SQL query directly or through predefined templates. The SQL query thus generated has the relevant aggregation applied on the database element on which computation will be performed. In this approach, the aggregate function to be applied, its triggering criteria and other conditions are pre-determined and integrated into the SQL query (Figure 1). Therefore, the result fetched from the RDMBS has the aggregations as well as other SQL conditions already computed before the answer is sent to the user.

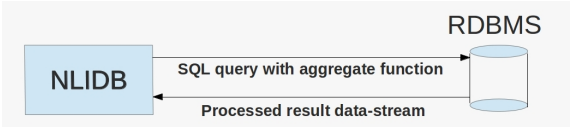

Figure 1: Conventional Aggregate Processing

This approach however suffers from the following drawbacks:

(a) Current RDBMS allow a limited number of inbuilt aggregate functions to be processed through the SQL query (MySql, 2013; Oracle, 2010) . Hence, the NLIDB system can process only those aggregates for which corresponding inbuilt aggregate functions exist in the RDBMS. This limitation reduces the ability of the system to process additional domain based aggregations, complex aggregations or quantifiers.

(b) Aggregates can also be qualitative in nature (e.g.'good students'). Qualitative aggregates represent a *quality* of an element which can be either explicitly present in the RDBMS as an attribute or otherwise implicitly derivable from one of its attributes. These aggregates are more complex than quantitative aggregates which can be resolved by a direct mapping to a corresponding aggregate function. Resolving qualitative aggregates, on the other hand, requires the identification of appropriate attributes and the criteria which help in deriving that quality from the attribute. Since the current techniques are limited to direct mapping of aggregates to RDBMS aggregate functions (Elmasri and Navathe, 2010), hence, qualitative aggregations are not processed.

(c) Aggregates may also appear in combination with quantifiers (e.g.'most average students') or relational operators (e.g.'greater than average marks'). In such occurrences, the aggregate is computed first and the quantifier or relational operator is applied on the results of the aggregation. Hence, a multi-level pass is required over the SQL data stream. In our approach, such cases can be handled.

(d) Processing of aggregates in current RDBMS requires the generation of nested SQL queries or complex joins. A nested query can be present in the SELECT clause, FROM clause or WHERE clause of the root SQL query, depending on the grouping of elements required in the SQL data stream and the computation sequence (Hellerstein, 1997). Because of the variations in the occurrence and thereby the generation techniques of a SQL query with nesting or joins, it is difficult (Li et al., 2006) and costly (Chaudhuri, 1998) to design specific predefined mechanisms which will encompass all of these variations.

The main contribution of our work is the computation of aggregates occurring in the NL query through an Aggregate Processing Layer (APL), independent of the RDBMS, applied to the result data stream before it is sent to the user (Figure 2). The APL gives us the ability to define and process a greater number of domain based as well as user-defined aggregations, a task easily achievable by people without the technical expertise of RDBMS, programming, domain schema or natural language translation. Most importantly, this approach gives us the ability to process qualitative aggregates in the NL query since the APL allows for a multi-pass processing which is essential for such operations.

The structure of the paper is as follows. In Section 2, we discuss issues in aggregation and their solutions through our approach. We also discuss aggregation, its constituents and aggregation types in detail. In Section 3, we describe our methodology of processing aggregates. In Section 4, we review the results of our approach and discuss specific aggregation aspects successfully resolved through it. Sec-

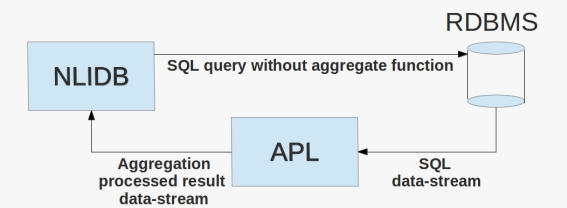

Figure 2: An RDBMS independent Aggregation Processing Layer (APL)

tion 5, concludes our study and mentions the possible future work based on the results and discussion in Section 4.

2 Our Approach

A related earlier work (Gupta et al., 2012) successfully deals with generating SQL query from a given NL query without aggregations. The semantic information (elements in the domain and their relations) are identified and maintained in a CPG based dependency structure, which is called the Domain Semantic Tree (DST). The domain elements of the DST are mapped to an ER-schema graph along with the attribute-value pair constraints. The SQL query is generated from the shortest path identified using the Minimal Spanning Tree algorithm.

In our approach, if the NL query contains an aggregate, it is marked for further processing in the DST. However, the SQL query generation module builds the SQL query without any aggregate functions implied on the domain elements. In other words, the aggregates are identified and marked but not processed initially by the NLIDB system. Therefore, the query generation process (as mentioned above) remains the same with or without aggregates.

We introduce aggregate processing as a layer between the NLIDB system and the database. We first generate the SQL data stream without the implied aggregations by the NL query and send it to the APL. The APL contains the aggregate processing modules which perform the necessary computations and filtering on the data stream elements. The data stream elements that pass through the APL constitute the final result which is then sent to the user. By using this approach, we completely avoid the need of creating pre-defined nesting templates and procedural SQL modules.

This layer has two distinct stages of processing. The first stage performs the aggregation on the data stream through aggregate functions. For efficient processing of data in the APL we have formulated our own aggregate functions that correspond to the aggregate. The second stage, if required, further refines the data stream by filtering it on the basis of additional domain based criteria applicable to the aggregate being processed. Only those stream elements pass through the filter which correlate with the results of stage one as well as satisfy the conditions of stage two. These aggregate specific filter conditions are stored in aggregate rules which help in initializing the filter at run-time.

2.1 Aggregations

Aggregation is the process of computation over a group of values by a function to produce a single value. It may be implied by both quantitative as well as qualitative aggregates.

For example: 'the best student in the Maths course' may imply 'a student who has scored the maximum marks in the Maths course'. Here, finding *maximum*(marks) is an aggregation without which 'best' cannot be computed.

Since RDBMS are restricted to dealing with a limited number of aggregates, the current NLIDB systems too do not provide for the computation of a wide range of aggregations which are present in the natural language such as *best, farthest, highest, good, bad, worst,* etc. Such aggregations require a multi-staged processing wherein not only the aggregation but the qualitative aspect has to be determined and computed as well.

Before we move on to the processing methodology, we briefly describe the elements of aggregates which make the aggregation computing possible and accurate through our approach.

2.2 Aggregation Elements

The primary elements of aggregation are the Aggregate Head, the Aggregate Scope (Scope-Key) and the Aggregate-Key. Every aggregate has exactly one Head and Scope-Key. These elements together help in the identification of the attribute, the Aggregate-Key, on which the aggregate is applied. For better understanding we explain each of these elements with an example:

(2) ‘Show the average students in the NLP course which is offered by the CS department.’

In this example, a query is posed by the user to see the list of students who are average in the NLP course which is offered by the CS department. In our domain, ‘average student in a course’ implies that the marks scored by a student are equal to the average marks computed for the NLP course. Hence, the aggregate *average* is applicable on ‘marks in the NLP course’. Thereafter, every student with marks equal to the computed average is listed as an ‘average student’.

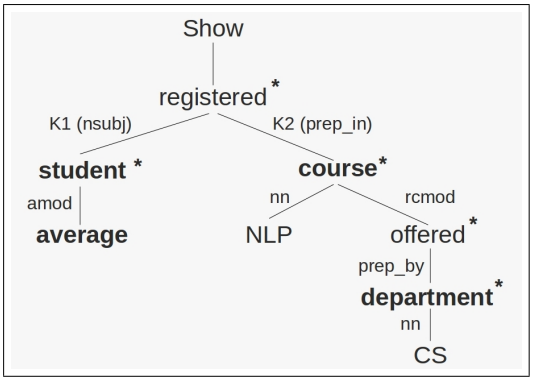

Figure 3: Domain Semantic Tree of sentence (2) showing the identified domain schema elements (*), CPG relations (K1,K2) and Aggregation Elements (in bold)

2.2.1 Aggregate Head

The Aggregate Head (AH) is an entity or attribute of the domain schema on which the aggregation is implied. It is the parent of the aggregate in the syntactic tree.

For example, in sentence (2): the aggregation is implied on the *student* entity since the modifier *average* is modifying the noun ‘students’. This makes *students* the AH. The same is also reflected in the Domain Semantic Tree in Figure(3).

However, in some of the instances, the AH may be present in the sub-tree of the aggregate as well.

For example: (3) ‘What is the average of marks awarded in the NLP course?’. Here, the aggregation is implied on the *marks* attribute, making it the AH. However, in the semantic parse, the ‘marks’ occurs as a prepositional modifier of ‘average’.

The AH is used for establishing the Aggregation Scope.

2.2.2 Aggregate Scope

The Aggregate Scope (AS) can be defined as the area of influenced elements of the AH. The AS establishes which elements of the NL query are directly or indirectly involved in the aggregation.

In sentence (2), we can observe two entities which are influenced by the AH *student*, namely, *course* and *department*. This is because the query has two dependencies in it: (a) The students are registered in the NLP course. (b) The NLP course is offered by the CS department. Hence, from (a), we infer that *course* is directly involved with *student*; From (b), the entity *department* becomes transitively involved with the *student* entity as well. Therefore, *course* and *department* constitute the Aggregation Scope.

Once the scope is established, we identify the Scope-Key element and Sorting Elements.

- **Scope Key:** The Scope-Key is the primary element of the scope and a crucial element for aggregation computation. The Scope-Key serves two purposes: a) The Scope-Key, in combination with the AH, helps in the identification of the Aggregate-Key on which the aggregation is computed; b) For each unique element of the Scope-Key, the corresponding Aggregate-Key value are extracted, ultimately giving us the value set on which the aggregation will be computed.

  In Sentence (2), based on the nearness (distance) of the scope elements with the AH in the DST (see Figure 3), the *course* entity is selected as the Scope-Key. This makes all the other remaining elements of the AS , namely *department*, as Sorting Elements.

- **Sorting Elements:** The remaining elements of the AS other than the Scope-Key constitute the Sorting Elements of the scope. Sorting Elements perform the function of the GROUP BY clause in SQL. They help in the grouped order display of the result data stream and do not directly affect the aggregation computation as such.

2.2.3 Aggregate Key

The Aggregate-Key is the attribute of the AH or Scope-Key element on which the aggregate will be

computed by the application of a relevant aggregate function. The Aggregate-Key is identified using the AH along with the Scope-Key.

From the Aggregate Rule 'average-1' in Table 1, the AH *student* and the Scope-Key *course* together lead to the identification of *marks* as the Aggregate-Key and *average()* as the aggregate function.

It is observed that in some NL queries the AH is explicitly mentioned as an attribute of the element (e.g.'average of the marks', in Sentence (3)). In such a scenario, the Aggregate-Key also functions as the AH and is used to identify the scope.

2.2.4 Aggregate Rules

These contain the necessary information about an aggregate which is required for its computation on the domain elements. Thus, each rule contains the mapping of the aggregate to its aggregate function as well as the user-defined filter conditions for the range of an indirect aggregate, restrictive conditions for the quantifiers and comparative conditions for the relational operators which may occur as with the aggregate.

From sentence (2), the identification of the AH *student* gives us two domain rules for *average* (see Table 1 and Table 2). Average-1 is selected as the Aggregate Rule and Average-2 is discarded for the given query based on the distance of the AS elements *course* and *department* with the AH *student*.

Elements	**Values**
Aggregate Head	student
Scope-Key	course
Aggregate-Key	marks
Aggregate-Function	average()
Filter Condition(s)	-

Table 1: Aggregate rule for 'average' - 'Average-1'

Elements	**Values**
Aggregate Head	student
Scope-Key	department
Aggregate-Key	cgpa
Aggregate-Function	average()
Filter Condition(s)	-

Table 2: Aggregate rule for 'average' - 'Average-2'

Once the correct Aggregate Elements and Aggregate Rules are identified, we have all the necessary information required to process the aggregate. .

2.3 Aggregation Types

Every aggregate is computed through an aggregate function which is a common feature for all aggregates. However, despite this similarity, the overall processing of aggregates differs significantly. This happens because of the difference in the domain based filtering conditions and the linguistic category the aggregate belongs to.

Based on these varied features, we divide aggregation into the following types:

2.3.1 Direct Aggregations

The aggregations in this type are primarily quantitative. This type implies a straight-forward one-step application of the corresponding aggregate function on the extracted set of values. No further processing is required.

For Example: (4) 'Show the maximum marks scored in NLP.'

Here, the aggregate 'maximum' gives a mapping to the aggregate function *max()*. Computing this aggregate is a single-step process which involves the application of *max()* on the attribute *marks*, giving us the single highest value from the set of all marks awarded to students in NLP.

2.3.2 Indirect Aggregations

These aggregations represent qualitative aggregations. Qualitative aggregations are aggregations which also represent a quality of the entity under consideration. In these aggregations the results are computed on a certain range. Qualitative aggregations have 2 distinct stages of processing:

(a) Identification of the quality and the domain based conditions of that quality. In some cases, the conditions implied could also have numerical ranges as a part of processing.

(b) Aggregation computation of the extracted set of values by the aggregate function which corresponds to the aggregation and thereafter application of the conditions which represent the quality under consideration (identified by (a)).

For example: (5) 'Who are the good students registered in the NLP course.'

In this example, the aggregate 'good' is defined in the domain as students with 30% marks above the

average marks scored in the NLP course. Hence, 'good' has two features in its Aggregate Rule: a)Range: more than 30% marks above the average of marks; b)Aggregate Function: *average()* computed on the Aggregate-Key *marks*. We first compute the average of marks scored in the course. Thereafter, we identify all students who have scored 30% marks or greater than the average marks computed. Thereby, giving a list of 'good students' in English.

2.3.3 Aggregations with Quantifiers

In this type of aggregates we have a quantifier (some, few, most, etc.) which directly modifies the aggregate. In other words, a filter is created based on the conditions imposed by the quantifier. This filter is applied on the result data stream on which the aggregate has already been computed.

2.3.4 Aggregations with Relational Operators

In these aggregates, a relational operator (greater than, less than, etc.) modifies the aggregate. Similar to 2.3.3, the relational operator creates a filter which is applied to the results of the aggregation it is modifying.

In both 2.3.3 and 2.3.4, the aggregate types can be direct or indirect. Both types re-filter the results of aggregation stored in an intermediate data stream to execute the conditions imposed by the quantifier or the relational operator.

Types 2.3.2, 2.3.3 and 2.3.4 can be termed as *complex aggregations*. With these aggregates, not only do we have to perform aggregation but also post-processing, through the aggregate filter, on the results of the aggregation and the SQL stream. For indirect aggregates the post-processing is done to give a range to the aggregation value computed. For aggregates with quantifiers and relational operators the post-processing involves an intersection of the original SQL stream with the aggregation result. The conditions of this intersection are determined by the quantifier or relational operator applied in the query.

3 Aggregation Processing

In our study, which focuses on aggregate processing, we divide the NL queries into two categories: a) queries with aggregates; b) queries without aggregates. The SQL query generation process in both the categories remains the same by not taking the aggregate into account during semantic processing, thus, generating the SQL query without the aggregate. This approach generates a data stream of all the necessary elements of the NL query, implicitly and explicitly mentioned. On this stream of elements the APL performs aggregation.

Aggregation processing involves pre-processing of aggregation elements to initialize the Aggregate Structure. Thereafter, the APL uses this structure's information to initialize its aggregation processing and filtering modules.

3.1 Aggregate Structure

The Aggregate Structure is a knowledge structure containing all the required and derived information from the DST and Aggregate Rules for performing aggregation. It has the following constituents:

P={ AE[AE_H,AE_{SK},AE_{AK},AE_{SE}],
$AG_f()$,
DC[DC_{RC},DC_Q,DC_{RO}]
}

Where, from sentence (2),

a) AE[] represents all the aggregate elements such as, the AH (AE_H), Scope-Key (AE_{SK}), Aggregate-Key (AE_{AK}) and sorting elements (AE_{SE}) i.e. the entities and attributes *student, course, marks and department* respectively.

b) $AG_f()$ is the aggregate function which corresponds to the aggregate and performs the aggregation. In this example, *average()*.

c) DC[] contains the additional domain specific conditions, namely, range criteria (DC_{RC}) in case of indirect aggregates, the quantifier restrictions (DC_Q) or relational operator criteria (DC_{RO}) in case of aggregates with quantifiers and relational operators respectively.

This structure initializes the APL by providing the necessary aggregate function in the first stage to compute the aggregate and domain based aggregation conditions in the second stage to create the filter.

3.2 Initializing the Aggregate Structure

First, we identify the Aggregate Head (AH), the Aggregate Rules and the Aggregate Scope (AS) from the DST. Thereafter, from the selected aggregate rule, we derive the correct aggregate function and other possible filtering criteria. Hence, by using the

aggregate elements and the aggregate rule we initialize the Aggregate Structure.

In case more than one rule is identified then we disambiguate by comparing the distance of the Scope-Key of each aggregate rule with the AH. If the ambiguity remains, then we take clarification of the intent of the user through a paraphrased generation of the aggregation elements.

3.3 Aggregation Processing through APL

The process of aggregation through the APL can be a 3 pass process on the data stream D_{str}, depending upon the complexity of the aggregation. The 1st pass corresponds to the first stage of APL, namely, aggregate computation. The 2nd and 3rd pass correspond to the filtering criteria applied on the intermediate result of the 1st pass which is then used to further refine the data stream. Each pass uses the information from the initialized Aggregate Structure (P) which is relevant to its processing.

If the aggregation type is Direct, the result is a single-value output which is sent to the user after the 1st pass. However, Indirect aggregations require a 2nd pass to process ranged criteria and for Aggregations with quantifiers and relational operators a 3rd pass has to be made to further apply restrictive or comparative conditions respectively. The data stream D_{str} is sorted on the Scope-Key values before the processing begins.

1st Pass - Aggregation on the data stream D_{str}:

- For each unique value of the Scope-Key in D_{str}, the set of values from the Aggregate-Key are sent to aggregate function retrieved from P($AG_f()$) and the result R_1 is computed.

2nd Pass - Filtering based on aggregation range P(DC_A):

- An intermediate range RG is computed by applying P(DC_{RC}) on R_1.

- For each unique value of the Scope-Key in D_{str}, those values of the Aggregate-Key which lie within the range RG are selected and stored in an intermediate record-set R_2.

3rd Pass - Filtering based on quantifier restriction P(DC_Q) or relational operator condition P(DC_{RO}):

- For Quantifiers: P(DC_Q) is applied on R_2 to create a record-set R_3, restricting the number of rows as specified through P(DC_Q)

- For Relational Operators: An intersection is done between D_{str} and R_2 with intersection condition provided by P(DC_{RO}), stored in a record-set R_3.

Based on the aggregation type - R_1, R_2 or R_3 is sent back to the user. Since the SQL query does not include the GROUP BY clause, rows of R_2 or R_3 are grouped by the values of the Sorting Elements from P(AE_{SE}).

4 Results

To assess the accuracy of our approach in a real-world scenario we conducted our experiments on a real-time database of a research institute. The database belongs to the academic domain containing detailed information of various academic entities (students, courses, faculty, etc) and the academic processes related to them.

For evaluation we have used 2 metrics: a) Precision: The number of NL queries correctly mapped to a SQL query, divided by the number of NL queries the NLIDB system answers.; b) Recall: The number of NL queries answered by the system, divided by the total number of NL queries.

Three MS students with detailed knowledge of the database schema were selected to create a query set of 180 NL queries covering all possible aggregations that can occur over the elements of the database schema. The query set was created with 45 queries of each aggregation type. For each NL query, a correct (expected) SQL query was prepared by an expert to cross-validate the results.

Since our approach focuses on aggregate processing, we set our evaluation criteria by way of two questions:

(1) What is the prevalence of NL queries having a correct semantic parse, therefore, a correct SQL query ?

A question is considered to be correctly answered if the NLIDB system generates one or more SQL queries which matches with the SQL query prepared by our expert. From Table 3, we can see that the NLIDB system has a precision of 92.3% and recall of 87.77%. On further evaluation, the reason for er-

Queries	Answered	Correct	Semantic Parse Error	SQL Error
180	171	158	9	5

Table 3: Result of NL query execution on NLIDB (Precision 92.3%)

roneous semantic parse was found to be the inability of the semantic parser to find the entities within the schema accurately. The problem in SQL generation was incorrect path generation leading to incorrect entity mapping with the database schema.

(2) Which out of the correctly parsed NL queries can be further processed by the Aggregation Processing Layer accurately ?

Table 4 shows that domain based aggregation processing, which includes complex aggregations as well, gives an accuracy of 98.1%. The Aggregation Processing Layer was able to process Direct aggregations, domain based Indirect aggregations and quantifiers *without mistakes* from the given data stream. The errors in aggregations with relational operators were due to the errors in computing the intersection of the data stream with the intermediate result stream. Interestingly, from Table 4, we see that NL queries with quantifiers and relational operators could be the reason for incorrect semantic parses and SQL query generation since they have a lower number of correct parses as compared to other aggregation types.

The results show that through our approach we can successfully process domain based complex aggregations while maintaining a relatively high accuracy which is comparable to some of the state-of-the-art systems like PRECISE (Popescu et al., 2003) and Mooney's learning NLI (Tang and Mooney, 2001).

4.1 Observations

By performing aggregation independent of the RDBMS, we are able to deal with certain important issues that most conventional RDBMS and NLIDB systems do not resolve:

4.1.1 Aggregation on Entities

In RDBMS, aggregations are applicable only on attributes of an entity (Elmasri and Navathe, 2010) and not on the entity itself i.e. only on the columns of a table. However, with queries in NL, a person may imply aggregation on an entity as well. Aggregation, in such cases too, refers to some attribute of the entity but NL creates a level of abstraction in the utterance which has to be resolved.

For example: (6) 'Who is the tallest student in the Engineering department'.

Here, 'tallest student' implies 'student with the maximum height'. This transfers the aggregation from the *student* entity to the *height* attribute. Hence, when the 'tallest student' has to be listed we find the student with the maximum height amongst all the students in the Engineering department.

In our approach, we maintain such implications (as mentioned in the above example) with the help of the Aggregate Head, Aggregate-Key and Aggregation Rules. Hence, we are able to successfully deal with aggregation on entities.

4.1.2 Complex Aggregations

Those aggregations which require domain based predicates in addition to the aggregate function are called Complex Aggregations. They represent qualitative aggregates generally, sometimes occurring with quantifiers or relational operators.

For Example: (6) 'Show a few good students registered in the English course?' Here, the interpretation of the indirect aggregation, 'good students', is the same as in sentence (5) of Section 2.3.2. However, the quantifier 'few' further restricts the number of 'good students' according to the domain based criteria applicable on the quantifier.

RDBMS systems cannot process such complex aggregates without external functions. However, the APL manages the domain predicate and quantifier processing by obtaining the required information from the Aggregate Rules and integrating them in the aggregate filter, which is the second stage of aggregate processing in our system.

4.1.3 Null Results with nested queries

RDBMS systems execute nested queries in a hierarchical manner from the innermost to the outermost query and the result is carried forward to the immediate outer level query. If this contains a NULL value and is passed to an aggregate function in the outer query as input then the query execution terminates with an error exception (Klug, 1982).

Aggregation Type	NL Queries	Correct Semantic Parse and SQL	Correct Aggregation Processing
Direct	45	41	41
Indirect	45	42	42
With Quantifiers	45	39	39
With Relational Operators	45	36	33
Total	180	158	155

Table 4: Result of Aggregation Processing on queries with correct semantic parse and SQL (Precision 98.1%)

For Example: In sentence (6), if the *height* attribute were to contain NULL values, the query in RDBMS would terminate with no results.

In our approach, we do not create nested queries for aggregation. We compute aggregation by passing the data stream through a custom aggregate function which discounts the rows having NULL values. Thus, the framework does not crash.

4.1.4 Run-time complexity of the nested queries

In some nested queries, the WHERE clause of the inner query block contains a join predicate with reference to a relation of the outer query block. Because of this, the inner query block is processed once for each row of the outer relation which satisfies all simple predicates of the outer relation (Kim, 1982).

For Example: (8) 'Show the student name with maximum marks in the city from which they gave the examination'. This generates the following query where S and SP refer to the same student relation.

```
SELECT  S.name
FROM   student as S
WHERE_S.marks=(SELECT MAX(S.marks)
          FROM student as SP
          WHERE SP.city=S.city)
```

For 4.1.4, our approach remains efficient since the aggregation processing is done outside the RDBMS. There are no re-iterations. In the first stage, the aggregate 'maximum', identified by *max()*, is computed. Then the stream is filtered for student names in the second stage.

To optimize 4.1.3 and 4.1.4 in RDBMS, the next best approach is to create queries by INNER/OUTER joins which try to reduce the nested query to an equivalent canonical form. However, this method gives errors in certain aggregates (e.g.*count*) (Steenhagen et al., 1994), which can be solved by creating a temporary table containing the aggregate functions and joins that they represent. However, automation of nested and join query creation at run-time is a complex process as compared to our approach.

4.2 Limitation

Our approach, presently, is not designed to handle NL queries having multiple aggregates.

For example: (9) 'Who are the good students in the CS department with poor grades in the NLP course?' In such questions, an intersection of the results of aggregation on two separate data streams, namely, 'good students in the CS department' and 'students with poor grades in the NLP course' has to be done. This requires the APL to run twice (once for each stream) before the intersection, which is presently not facilitated.

5 Conclusion and Future Work

In this paper, we present a novel approach to handle aggregations occurring in a NL query. Our key contribution is the development of an aggregation framework that can deal with the direct quantitative as well as domain based complex aggregations efficiently. By creating a framework independent of a database, we've made the system robust, scalable and easily customizable by non-experts while ensuring a high accuracy at the same time. In addition to this, the framework aids in removing the pre-defined automated process of nested SQL query generation for aggregations. To the best of our knowledge, ours is the first formal approach towards dealing with a large variety of simple and complex aggregations with results guaranteeing a high accuracy in their resolution.

Future works include extending the framework to deal with multiple aggregations in an NL query and creating a hybrid statistical aggregation framework to transition from domain based to generalized aggregate processing.

References

Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43.

Ramez Elmasri and Shamkant Navathe. 2010. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition.

Abhijeet Gupta, Arjun Akula, Deepak Malladi, Puneeth Kukkadapu, Vinay Ainavolu, and Rajeev Sangal. 2012. A novel approach towards building a portable nlidb system using the computational paninian grammar framework. *Asian Language Processing, International Conference on Asian Language Processing*, pages 93–96.

Joseph M. Hellerstein. 1997. The case for online aggregation: New challenges in user interfaces, performance goals, and dbms design.

Won Kim. 1982. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469.

Anthony Klug. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, pages 699–717.

Yunyao Li, Huahai Yang, and H. V. Jagadish. 2006. Constructing a generic natural language interface for an xml database. In *In EDBT*, pages 737–754.

2013. Mysql 5.0 reference manual : http://dev.mysql.com/doc/refman/5.0/en/group-by-functions.html.

2010. Oracle database sql language reference, 11g release 1 (11.1).

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*, IUI '03, pages 149–157.

Hennie J. Steenhagen, Peter M. G. Apers, and Henk M. Blanken. 1994. Optimization of nested queries in a complex object model. In *In Proc. of the Int. Conf. on Extending Database Technology (EDBT*, pages 337–350. Springer-Verlag.

Lappoon R. Tang and Raymond J. Mooney. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *ECML*, pages 466–477.