Proceedings of the Federated Conference on Computer Science and Information Systems pp. 853–862 DOI: 10.15439/2018F218 ISSN 2300-5963 ACSIS, Vol. 15

Model-driven Query Generation for Elasticsearch

Berkay Akdal*†, Zehra Gül Çabuk Keskin*, Erdem Eser Ekinci*, Geylani Kardas†
*Galaksiya Information Technologies, Ege Technopark, 35100, Bornova, Izmir, Turkey
Email: {berkayakdal, zehragulcabuk, erdemeserekinci}@galaksiya.com
†International Computer Institute, Ege University, 35100, Bornova, Izmir, Turkey
Email: geylani.kardas@ege.edu.tr

***Abstract*—Elasticsearch is a distributed RESTful search engine, capable of solving growing number of use cases and can handle petabytes of data in seconds. However, Elasticsearch comes with a complex query language which causes a steep learning curve for the developers and, therefore, creation of queries can be difficult and time-consuming in many cases. Hence, in this paper, we introduce a Domain-specific Modeling Language (DSML), called Dimension Query Language (DQL), to support the model-driven development of Elasticsearch queries. Elasticsearch queries can be automatically generated from DQL models and DQL's IDE is capable of executing these auto-generated Elasticsearch queries on remote repositories. An evaluation of using DQL has been performed at the industrial level with the participation of a group of developers. The conducted evaluation showed that the use of the language significantly decreases the development time required for creating Elasticsearch queries. Finally, qualitative assessment, based on the developers' feedback, exposed how DQL facilitates the development of Elasticsearch queries.**

I. Introduction

ELASTICSEARCH is a distributed RESTful search engine, which is based on Lucene information retrieval software library [1] and is capable of solving growing number of use cases. Many types of searches (e.g. structured, unstructured, geo, metric) can be prepared and combined. It works in clusters, and according to some tests performed by its developers (namely, Elastic Team), it is reported that Elasticsearch can handle petabytes of data in seconds [2].

Elasticsearch differs from classical relational database management systems (RDBMS) in many ways: Elasticsearch's primary database model is a search engine and it stores documents instead of key-values. Each document in Elasticsearch is a JavaScript Object Notation (JSON) object, and hence it does not use Scripted Query Language (SQL). Queries are provided with its own language based on JSON. A given search can be performed not only in a form of a query; filters can also be used for document search which is faster than the queries. Finally, it is schema-free, i.e. two documents of the same type can have different sets of fields [3]

However, such kind of powerful engine comes with a very complex query language which causes a steep learning curve for the query developers. Moreover, there are numerous types of queries and scripts combinable with each other whose creation and use can be difficult and time consuming in many cases.

There exists a tool for visualizing Elasticsearch data, called Kibana, which is also developed by the Elastic Team [4]. It works on top of the content indexed on an Elasticsearch cluster and it can directly connect to an Elasticsearch server to be used for generating visualizations and reports; but again, the users must have prior knowledge about how Elasticsearch works and need to be experienced in dealing with its complex query language.

The paradigm shift introduced by model-driven development (MDD) [5, 6] in which the focus changes from code to models, leverages the abstraction level and promotes the software development for various application domains (e.g. [7-13]). Moreover, domain-specific languages (DSLs) / domain-specific modeling languages (DSMLs) [14-18] which have notations and constructs tailored toward a particular application domain, assist to the developers during execution of MDD processes by providing first a user-friendly syntax for modeling systems (mostly in a visual manner) and then a translational semantics for generating application software and any other artifacts automatically [19].

Abovementioned features and benefits of applying MDD and using DSMLs in other domains conduce toward producing a MDD framework also for Elasticsearch. Hence, in this paper, we introduce a DSML which can be used inside this MDD framework to facilitate the query writing process required for the Elasticsearch. Although many efforts exist in model-driven database processing and query generation (e.g. [20-23]), they do not consider the specifications of Elasticsearch and do not support generating queries, structured according to Elasticsearch which differs from the traditional databases.

Originating from a metamodel of Elasticsearch, which is also derived in this study, the proposed language provides a graphical concrete syntax for modeling queries within its integrated development environment (IDE). Models of the queries, visually prepared in this IDE, are automatically translated into corresponding Elasticsearch structures which are ready to be executed. If the developer requests execution

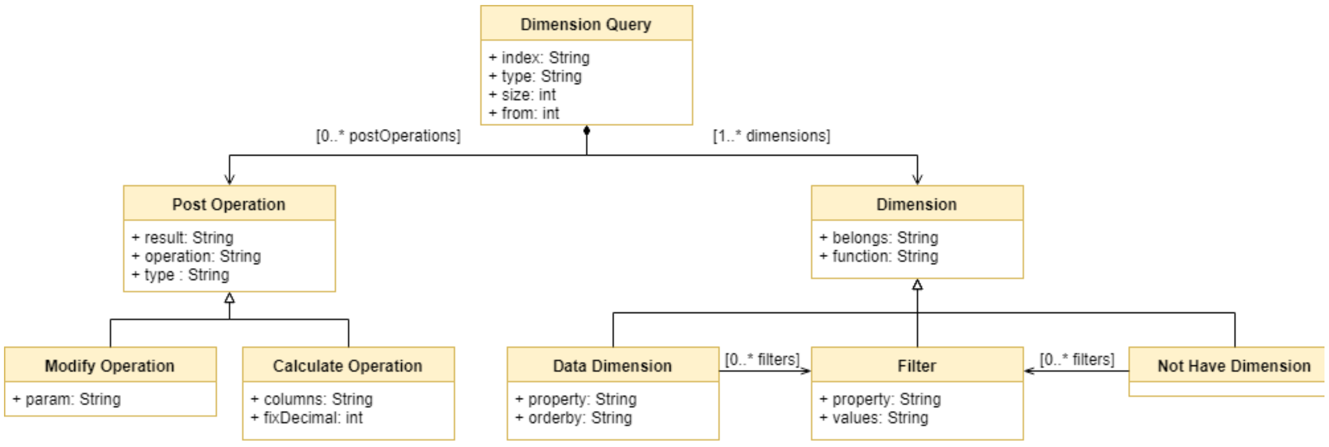

Fig. 1. DQL Metamodel

of these queries, it is also possible to execute those modeled and automatically generated queries on Elasticsearch storages. In this paper, we also discuss the use of this DSML for the industrial applications and give the results of evaluating its use inside a software company specialized for developing commercial Big Data applications.

The rest of the paper is organized as follows: In Section 2, the proposed query language is discussed with including its metamodel, fundamental elements and built-in query transformation process. Section 3 demonstrates the use of the language. Evaluation of the language and results of this evaluation are discussed in Section 4. Related work is given in Section 5. Finally, Section 6 concludes the paper.

II. Dimension Query Language

When we think of the three-dimensional space we are in, every object has coordinates to locate their position and hypothetically, it is possible to list and create reports for each object's or living creature's position on earth. Such report would have three fields for the coordinates linked with the name of the related object or person. To find an entry on the report, we would have needed to know the related entry's name and coordinates.

Mathematics and physics define dimension as the minimum number of points required to know an object's position and velocity on the space they belong. By this definition, we can say that our hypothetical report is a four-dimensional space, containing entries with four dimensions. Originated from this, we named our Elasticsearch query model as Dimension Model (DM) and the proposed Elasticsearch DSML as Dimension Query Language (DQL). In DM, each dimension corresponds to a field of data that must be included within the query.

In the following subsections we first define the fundamental elements and the relations inside DQL which compose the abstract syntax on the DSML for Elasticsearch. Then, we discuss how constraint checks and query validations are performed inside DQL's IDE before automatically transforming prepared query models into Elasticsearch queries. Finally, query transformation process is discussed.

A. Fundamental DQL Elements

Our transformation service (that will be discussed later) accepts Dimension Query (DQ) instances and generates Elasticsearch queries. The users can choose to view the transformed queries or directly execute them to view a table report over the underlying Elasticsearch storage. These DQ instances are created by conforming a metamodel which defines fundamental elements and their relations required for Elasticsearch queries. The metamodel, which leads to the generation of DQL syntax, is depicted in Fig. 1. Elements and properties of the metamodel are written in bold in the following text.

Elasticsearch storages, namely indexes, are a collection of documents that have similar characteristics [24]. Documents belonging to the same index may have relations with each other. On Elasticsearch, there are two types of relations. "Parent-child relation" links two documents by marking one as parent while marking the other one as child. "Nested relation" simply writes the whole document into another one.

On query transformation, one of the required properties is the name of the document on the top level, defining the document without a parent document. This needs to be specified as the **type** field in the **Dimension Query**. Another required field is the **index**, which is the name of the index to execute the query on. Finally, on the **dimensions** field, requested dimensions are expected. Along with these, there are some optional fields that a query can uphold, such as expected result size as **size** and offset as **from**.

B. Dimension Types

Dimensions vary in three types; **Data Dimension**s, **Filter** and **Not Have Dimension**s. Data dimensions are used to represent the fields to be retrieved upon the execution of the query and filter dimensions, by their name and hence they are used to filter the retrieved data based on some conditions. However, independently from its type, each dimension must have two main fields; **function** and **belongs**.

The **function** field is used to specify which operation must be performed on the data. With this field; dimensions can be grouped, summed, counted and their average or percentage over their sum may be calculated. **belongs** field represents the name of the Elasticsearch document of the index in which the dimension data are located.

Data and filter dimensions must also have a field called **property**. This field represents the name of the data to be retrieved or filtered. The data dimensions may also have an additional **orderby** field to indicate which dimension must be used on ordering the query results.

If the query designers want to filter data on certain fields, filter dimensions may be used to meet this requirement. These dimensions will filter the data instead of creating another field on the result set. They are different from the data retrieval dimension by having additionally one field called **values**, indicating the values to apply with the filter. Filters can also be applied to specific data retrieval dimensions as well as they can be applied on the query. When used in this way, they affect only the dimension they are getting applied to.

The filtering criterion does not always have to be based on some values. For example, on a customer-invoice database, we may want to list the customers who did not place an order between some dates. To handle this case, there is an additional type of dimension called “Not Have Dimension”. This dimension can be used to filter certain fields, which does not have any relations to the given Elasticsearch document. Considering the customer-invoice example, let us think we have an index with two documents, customer and invoice, and assume that there is a parent-child relation, customer as parent and invoice as child. Creating a “not have dimension” with “belongs” field as `{belongs: 'invoice'}` will allow us to list the customers without an invoice on the whole Elasticsearch index. However, we may want to see the results based on another filter, such as a date interval. On this case, since the “not have dimension”s can also have filters, we can define a filter and add it to our “not have dimension”.

C. Post Operations

Calculations and value formatting is a common thing to do on report generation. When needed, Post Operations may be used to tinker with the retrieved data and they can be elaborated in two types; one for calculating new data, called **calculate operation**, and one for modifying existing data, called **modify operation**.

A post-operation must have the following fields: **result**, **operation** and **type**. They are common for each operation type, where the **result** field is the name of the data field which the post operation will be affecting. For calculation operations, this field will be used as the calculated field’s name. For modify operations, it is the name of the field on which the modification operation works on. **operation** field is the name of the operation to process, such as sum, divide, absolute, floor, ceil. Finally, the **type** field is the indicator of the type of the post-operation itself. It can be whether "*calculateOperation*" or "*modifyOperation*".

Calculate operations have two more additional fields. The first one the "*columns*" which holds the dimensions involved in the calculation operation and the second one is the "*fixDecimal*" which is to specify the number of the digits that should be displayed if the calculated value is a decimal number. Usable calculate operations are essential arithmetic functions; sum, subtract, multiply, divide.

Modify operations have only one additional field called "*param*"; which is the additional required parameter(s) to apply the operation and may not always be necessary. Modify operations, which can be used by the developers, are listed below:

Fix Decimal: To limit the number of the decimals to show of a decimal number. The param field must be the number of the digits.

Floor: To get the floor value of a decimal number.

Ceil: To get the ceiling value of a decimal number.

Abs: To get the absolute value of a number.

Replace: To replace some specific values of a dimension in the result set. A serialized JSON array string, which has objects as elements containing "from" and "to" values, is required as the "param" field.

D. Constraints and Query Validations

Our motivation is to simplify the query generation for the Elasticsearch without needing to know its query formulation details. However, there are also lots of syntactic controls and additional semantic constraints which should also be taken into account while writing Dimension Queries. Based on the metamodel elements and their relations discussed in the previous section, a modeling environment has been developed to use language constructs and features of DQL. Query developers may use our DSML’s graphical syntax and all required constraint checks and hence query validations can be realized automatically according to the Elasticsearch specifications.

Fig. 2 shows a screenshot taken from web-based IDE of the proposed Elasticsearch DQL. On the left side of the screen, the indexes on the system are listed with a combo box. After an index gets chosen, its metadata are shown directly under the index selector. The users then can start to create a DQ by simply dragging and dropping the fields they want to include in the query. The DQ gets automatically updated on the backstage each time the user drops a new field, updates or removes an existing one. The query can be tracked from the query panel at the top side of the screen dynamically.

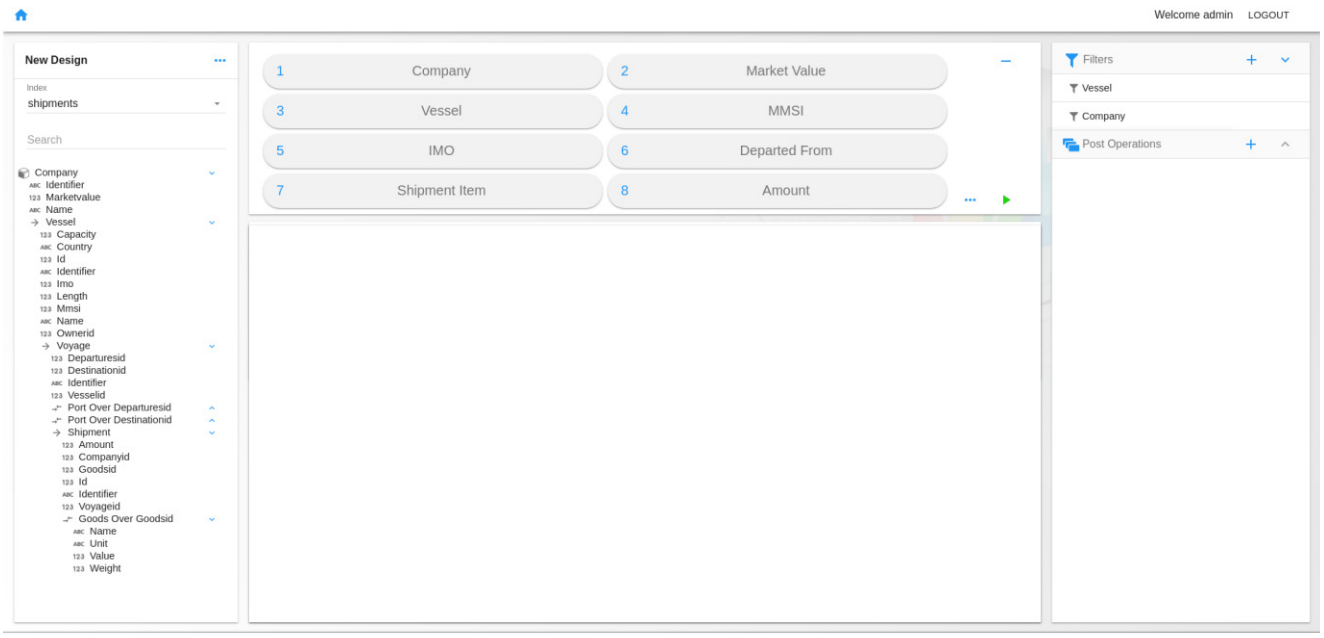

Fig. 2. IDE for the proposed Elasticsearch DSML

Filters and post operations are listed on the panel at the right side of the screen. Users can create new filters and post operations by clicking the add button (+) near them and can include filters to the query by simply dragging and dropping them into either on a data dimension or into the query directly.

After the users finish choosing the fields, the DQ can be sent to the backing server in order to be transformed to the Elasticsearch query. At this point, the users can choose to simply view the transformed query or the results generated with the execution of the transformed DQ.

There are lots of constraints needed to be followed while writing an Elasticsearch query. To be able to generate valid, executable Elasticsearch queries, we have also put some constraints on DQL and hence the IDE warns the user or prints an error message if a constraint gets violated.

Filters are for filtering data; they do not cause a field to be included within the result set. Therefore, filters cannot contain filters. Appending filters to other filters will have no effect on the generated query.

Since they affect the set of all results, the "not have dimension"s can only be used within the query, not within other dimensions.

Fields from different documents with parent-child relation cannot get queried without performing an operation over them. Because, for a member of the parent document, it is possible to have more than one value on the child document and it will not possible to create a result set without making some groupings on the parent document.

Mathematical processes such as number formatting, numerical calculations and digit rounding, can only be performed on numeric dimensions as well as date format operation can only be performed on date dimensions.

Applying group function to a dimension causes an aggregation to get started on Elasticsearch query. On Elasticsearch queries, when an aggregation gets started, all remaining fields must be included into that aggregation in some way. So, when the grouping function is applied over a dimension in the DQ, all remaining dimensions need to have a function value.

Each dimension of the query must be unique. If there is more than one dimension created with the same field of the same document -having also the same function-, one of the dimensions must have a filter different than the other one's filters at least. Semantic definitions on DQL, make all above constraint checks possible inside the IDE.

E. Query Transformation Stage

There are four stages of a query transformation which are all automatized within DQL's IDE. First one is called the **Reducing** stage where the dimensions in the DQ get inspected and grouped by their respective nested documents on the Elasticsearch index. By doing this, it is possible to make fewer aggregations on the Elasticsearch query, therefore, it increases query execution performance by preventing same nested documents to get aggregated over and over. The requirement for two dimensions to be grouped is that they must be in the same document on Elasticsearch and they must have exactly same filters getting applied to them.

After dimensions are reduced, they get **Sorted** according to their documents and the relations between their documents on the index.

Two rules are applied while sorting the dimensions:

1) For multiple dimensions on the same document, if the document has a nested object, its own dimensions have priority than the nested object's dimensions. For instance,

considering the metadata given in Fig. 3, dimensions of Document A have a higher priority than dimensions of the Nested Object B. Likewise, Document C over Nested Object C1 and Nested Object C1 over Nested Object C1.1 have priorities.

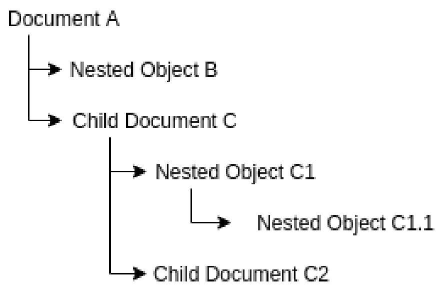

Fig. 3. Sample Index Metadata

2) Finally, calculation dimensions like sum, average, percentage, and count have the lowest priority so they take place at the end of the sorted dimensions list.

Analyzing stage is the one where the Elasticsearch query gets started to be created in pieces. On this stage, each dimension is converted to proper Elasticsearch query fragment and gathered up on a temporary list. This phase is crucial because, during aggregation generation, it is decided whether the aggregations will be linked with a nested relation or parent-child relation. “Not have dimension”s also will be included to the query on this stage.

The final stage of the query transformation is the **Generation** stage. On this stage, the query at hand is already has been reduced (for optimization), sorted and analyzed. The dimensions have been converted to aggregation blocks and relations between these aggregation blocks have been determined.

The list that's holding the aggregation blocks get iterated and linked with respect to their flags set from the previous stage of query transformation. Aggregation link is established with respect to the metadata of the index. That means, aggregation blocks will be linked to the others either as siblings or children according to their dimensions' "belongs" field.

III. USE OF DQL DURING QUERY GENERATION

The Dimension Queries may be grouped into four main types corresponding to the types of the result sets they will generate upon the execution of the transformed Elasticsearch queries. This section will briefly explain these types. For a better comprehension, queries are represented in their textual notation during the following discussion, which are achieved automatically by using DQL and its graphical modeling environment.

In the first type, the query aims at getting direct results without making any grouping or filtering. If that’s the case, there is only one constraint: As stated on the constraints section, the fields on the query must be on the same Elasticsearch document.

When the created query is in this type (see Fig. 4), dimensions must have three main fields. Function field of the dimensions must have a static value of “*include*” (shown in lines 6, 9, 12). Additionally, **orderby** field (line 5) may be added to one dimension to sort the results.

The corresponding translation (see Fig. 5) may seem simple because the translated query is obviously small and easy to write. However, the real power of the Dimension Queries, comes to stage when groupings and functions get involved with the query.

If the created query aims to fetch fields from different related documents (see “belongs” properties in Fig. 6 on lines 4, 7 and 10), the second type of query comes in. This type of query groups fields so different documents may be included in the result set. Again, on this type of query (Fig. 6), there is only one constraint: If a grouping, namely aggregation, starts with a dimension, all remaining dimensions must have a function applied on them.

```
01  { index:"indexName",
02    type: "doc",
03    dimensions: [
04     {property: "prop_1",
05      belongs: "doc", orderby: "asc",
06      function: "include" },
07     {property: "prop_2",
08      belongs: "doc",
09      function: "include" },
10     {property: "prop_3",
11      belongs: "doc",
12      function: "include"}
13    ],
14    from: 0,
15    size: 50 }
```

Fig. 4. Dimension Query without Groupings

```
{ from: 0,  size: 50,
  query: { bool: { disable_coord: false,
      adjust_pure_negative: true,
      boost: 1 }
  },
  _source: {
    includes: ["prop_1","prop_2","prop_3"],
    excludes: [ ]
  },
  sort: [{property_1.sort: {order: "asc"}}]
}
```

Fig. 5. Transformed Elasticsearch Query without Aggregations

```
01  {index: "indexName",
02    type: "topLevelDocName",
03    dimensions: [ {
04      property: "prop_1", belongs: "doc_1",
05      orderby: "asc", function: "agg"
06    }, {
07      property: "prop_2", belongs: "doc_2",
08      function: "sum"
09    }, {
10      property: "prop_3", belongs: "doc_2",
11      function: "sum"
12    } ],
13    from: 0, size: 50}
```

Fig 6. Dimension Query with Functions

While grouping the results, users may also want to do some calculations to see the summarized result. For example, let us consider a report to list the total invoice price for each customer. To do that, first an aggregation needs to be set up on customers ("function" field on line 5 in Fig. 6) then another summarization can be used on other dimensions. The reason for this aggregation requirement is, Elasticsearch needs to make some groupings to calculate summarized results such as sum and avg. Functions can only be applied when a grouping gets applied to the query.

Dimensions on this type of query must have same fields as the ones within the previous query, except the function field values may be "*agg*", "*count*", "*sum*", "*avg*" and "*percentage*" instead of "*include*" (see lines 5, 8 and 11 in Fig. 6). Fully generated Elasticsearch query for this DQ type can not be shown here due to space limitations. However, aggregations part of the generated query can be seen in Fig. 7.

```
...
{aggregations: {
 prop_1_doc_1_agg: {
 terms: { field: "prop_1.keyword",
  missing: "null", size: 2147483647,
  min_doc_count: 1, shard_min_doc_count:0,
  show_term_doc_count_error: false,
  order: { _term: "asc" } },
  aggregations: {
   prop_2_doc_2_sum: {
    children: { type: "documentName_2" },
     aggregations: {
      prop_2_doc_2_sum: {
       sum: { field: "prop_2" } } } },
      prop_name_3_doc_2_sum: {
        children: { type: "doc_2" },
        aggregations: {
         prop_3_doc_2_sum: {
          sum: { field: "prop_3" } } }
} } } } }
...
```

Fig. 7. An excerpt from transformed Elasticsearch Query with Functions

If the users want to filter their data, they may create filter dimensions. Different usages of different filters are given in Fig 8. Applying a filter directly to the query is the case when the filters will be inserted within the *query* field of the transformed Elasticsearch query; thus affecting the whole result set as mentioned before. The filter (lines between 17 – 20 in Fig. 8) on this sample is for listing the results by the prop_3 field of the doc_2 with a value greater than 1000.

Applying filter to specific dimensions (lines between 13 – 14 in Fig. 8) will cause sub-query blocks to be created and inserted as filters to related aggregations. When this happens, the filters will be applied on only the related aggregation and, if there is any, to its sub aggregations.

Finally, the usage of "not have dimension" is shown between the lines 16 – 19 in Fig 8. In this example, a sample filter has been added to the "not have dimension". Before the "not have" filter gets applied, its inner filter will be applied first to narrow down the results.

An excerpt from the generic filter of the generated Elasticsearch query is given in Fig 9. On the transformed query, **prop_2** on line 4 is the name of the field to which the filter applies. It corresponds to the **property** field's value on the dimension query (see Fig. 8, line 14). The field called **from** (Fig 9, line 5) is the value of the **value** field previously indicated in Fig. 8, line 19. Finally, the **type** field in the line 14 (Fig. 9) is the name of the Elasticsearch document, containing the related fields. It corresponds to the **belongs** field (Fig. 8, line 18) in the Dimension Query filter.

```
01  {index: "indexName",
02    type: "topLevelDocName",
03    dimensions: [ {
04      property: "topLevelDocName",
05      belongs: "prop_1",
06      orderby: "asc", function: "agg" },
07     {property: "doc_1",
08      belongs: "prop_2", function: "avg" },
09     {property: "doc_1",
10      belongs: "prop_3", function: "avg",
11      filters: [ {
12        property: "doc_1", belongs: "prop_4",
13        function: "lt", values: [ 3000 ]}] },
14     {property: "prop_2", belongs: "doc_1",
15      function: "gt", values: [ 1000 ] },
16     {belongs: "doc_2", function: "nothave",
17      filters: [ {
18        property: "prop_4", belongs: "doc_2",
19        function: "gt", values: [ 10000 ]}]
20    } ],
21    from: 0, size: 50}
```

Fig. 8. Dimension Query with Dimension Filters

```
01  ...
02  {"has_child": { "query": { "bool": {
03    "must": [ { "range": {
04     "prop_2": {
05      "from": 1000, "to": null,
06      "include_lower": false,
07      "include_upper": true,
08      "boost": 1.0
09     } }
10    } ],
11    "disable_coord": false,
12    "adjust_pure_negative": true, "boost": 1.0
13   } },
14   "type": "doc_1",
15   "score_mode": "sum", "min_children": 0,
16   "max_children": 2147483647,
17   "ignore_unmapped": false, "boost": 1.0
18  } }
19  ...
```

Fig. 9. An excerpt from transformed Elasticsearch Query's Generic Filter

Depending on the function of the filter in the DQ, Elasticsearch query filter properties have different usages. On Fig 9. line 5, **to** field is used as the upper limit when the DQ function is either *less than or range.* On the given example **include_upper** field is *true* since it is a *greater than* filter and the upper value limit is infinity. **include_lower** field acts like the same when the filter function is *less than*. The same fields get used when the filter is *less than or equal to* and *greater than or equal to*.

```
01  ...
02  {"aggregations": {
03   "prop_1_topLevelDocName_agg": {
04    "terms": {
05     "field": "prop_1.keyword",
06     "missing": "null",
07     "size": 2147483647,
08     "min_doc_count": 1,
09     "shard_min_doc_count": 0,
10     "show_term_doc_count_error": false,
11     "order": { "_term": "asc" }
12    },
13    "aggregations": {
14     "prop_2_doc_1_avg": {
15      "children": { "type": "doc_1" },
16      "aggregations": { "prop_2_doc_1_avg": {
17        "avg": { "field": "prop_2" } },
18   "prop_3_doc_1_avg_filters_prop_4_lt_4000":
19        { "filters" },
20   "prop_3_doc_1_avg_filters_prop_4_lt_4000":
21        { "avg": { "field": "prop_3" } }
22  } } } } } }
23  ...
```

Fig. 10. An excerpt from transformed Elasticsearch Query's Aggregations

Part on the aggregations included in the same transformed query is given in Fig. 10. Aggregation names (bold texts on lines 3, 16 and 20 in Fig. 10) are generated by combining **property**, **belongs** and **function** fields on dimensions. The dimension specific inner filter (Fig. 11) is inserted in place of the bold *filters* text in line 19 of Fig. 10. The transformed "not have dimension" is given in Fig 12. Note that the inner filter is nearly the same as the one in Fig. 10. The *must not* keyword in line 2 determines the purpose of the filter.

```
01  {"filters": { "filters": [ {
02   "bool": { "filter": [ { "range": {
03    "prop_4": {
04     "from": null,
05     "to": 4000,
06     "include_lower": true,
07     "include_upper": false,
08     "boost": 1.0
09     } } } ],
10     "disable_coord": false,
11     "adjust_pure_negative": true,
12     "boost": 1.0
13    }
14   } ],
15   "other_bucket": false,
16   "other_bucket_key": "_other_"
17z } }
```

Fig 11. An excerpt from transformed Elasticsearch Query's Inner Filter

```
01  ...
02  {"bool": { "must_not": [ { "has_child": {
03   "query": { "bool": { "filter": [ {
04    "range": { "prop_4": {
05      "from": 10000, "to": null,
06      "include_lower": false,
07      "include_upper": true,
08      "boost": 1.0
09      } } } ],
10     "disable_coord": false,
11     "adjust_pure_negative": true,
12     "boost": 1.0 } },
13    "type": "doc_2",
14    "score_mode": "sum",
15    "min_children": 0,
16    "max_children": 2147483647,
17    "ignore_unmapped": false,
18    "boost": 1.0
19    } } ],
20   "disable_coord": false,
21   "adjust_pure_negative": true,
22   "boost": 1.0 } }
23  ...
```

Fig. 12. An excerpt from transformed Elasticsearch Query's "Not Have" Filter

IV. Evaluation

An evaluation of using DQL has been performed at the industrial level with the participation of a group of developers from Galaksiya Information Technologies (http://galaksiya.com/). Galaksiya is a software company, located in Izmir, Turkey and its business domain mainly consists of Big Data and its applications. In some of their software solutions, the developers in the company recently started to work on Elasticsearch and related data storage.

At the beginning of the evaluation, we have determined the logistics as the target domain and created a logistics database for our case study. The main reason for choosing that domain is logistics datasets are very large in volume thus making them hard to query. Considering an end-user scenario to create a report over a logistics dataset to view the latest activities around the world, we have created a sample database by using the most active 51 ports on the world, 82 random selected shipping company names, distributed to 19 random countries. Each company on the dataset has random amount of ships with a total of 5000. The dataset has around 4750 auto-generated voyages with randomly selected goods. Total number of goods in the system is 50000, again all randomly generated.

Fig. 2 also shows a DQL instance model prepared for this evaluation. In the query panel residing at the upper middle of the IDE, there exist model items correspond to the required data dimensions in the query, namely Company, Market Value, Vessel, MMSI, IMO, Departure Port, Shipment Item and Amount. On the right panel under the filters, the Vessel and the Company are the defined filters which can be used in the query. Once dropped on a dimension or to the query, they will be transformed into filters within the related dimension or into a filter dimension depending where they are being applied.

As being an instance of DQL, the created query aims at listing the amount of the goods on each shipment with the information of departure ports, vessel details and company information. In addition, the same query model leads to prepare the query results inside a report grouping the data by the companies, vessels and departure ports.

For the qualitative assessment of DQL usage, five software developers became volunteer and agreed on being an evaluator. All of these evaluators has B.Sc. in computer science / software engineering and two of them are M.Sc. students in computer related fields at the time of this evaluation performed. Evaluators possessed the experience of developing software in industrial scale considering Big Data and/or Linked Open Data applications for different business domains (3 years on the average). Although they were skilled with creating database queries and working with data storages, they had no or very little knowledge on the query language required for Elasticsearch. After a brief introduction of DQL and its IDE, the evaluators were requested to create the same report given in Fig. 2. Upon completion their modeling session, a questionnaire including the following open-ended questions was given to the evaluators and their responses were gathered:

1. How does DQL and its IDE make writing Elasticsearch queries easier?
2. Did you encounter any difficulties while modeling queries and creating reports with using DQL? If any, please provide your suggestions to fix them.
3. Do you think DQL is easy to learn and use?

All the evaluators agreed on the biggest advantage of DQL that it eliminates the syntax errors which may be encountered while creating a query since there is no query writing process. They also agreed that the use of the DQL removed hardcoding the Elasticsearch queries hereafter. And most of them indicated that it is possible to create Elasticsearch queries without writing a single line of code. One of the evaluators stated that DQL's graphical syntax is comprehensive enough to cover all Elasticsearch domain and accompanying IDE helped them for determining and visualizing the details of queries from scratch. Some of the evaluators found the model panel residing on the left side of the DQL IDE (see Fig. 2) very helpful by means of dynamically showing the whole data model pertaining to the query under development.

For the second question, some of the evaluators stated that applying a filter to the report directly or using it separately on dimensions is a little bit confusing at the first time but after using the editor for a while, it gets simpler. Based on the feedbacks gained from the evaluators, visual concrete notations required for query modeling and organization of them inside the IDE were also re-arranged since some of the evaluators found the arrangement of these components a bit complicated.

Finally, for the last question, everyone agreed that even end-users with no knowledge on Elasticsearch would be able to use DQL for Elasticsearch query design and implementation after a small training. Most of the evaluators confirmed that there is no need to know any kind of syntax and programming (or querying) language for a person to use DQL and its IDE. They also added that little knowledge about basic query logic is enough for a user before using DQL. However, all of the evaluators also answered the third question by indicating there is still a learning curve to get used to the DQL editor, but with a short training session it becomes easy to learn and design reports.

In order to measure whether the use of DQL speeds up query creation, each evaluator's query design with using DQL has been recorded. The evaluators completed the generation of the Elasticsearch query required for the above logistics case study around 30 mins. on the average. The evaluators were also requested to create the same query again but this time without using DQL. They just used Elasticsearch query syntax and the result was amazing: It took around 6.5 hours to complete writing the same query on the average. Although that measurement was achieved from only a single case study, we believe that the speedup gain obtained with using DQL in here is promising since the experts in the company confirmed that the query handled in here is complex enough comparing with the exact queries created in their commercial applications for datasets which are almost same size with the logistics dataset used in the case study.

V. Related Work

Like other domains, in order to master the problems of creation, management and evolution of databases and querying on these databases, the researchers investigate the ways of applying MDD principles and/or proposing the use of DSLs / DSMLs. For instance, MDSheet is proposed in [25] for model-driven engineering of spreadsheets. End-users can build sheets within MDSheet framework via its tool. The framework is enriched with a model-driven query language [20] which supports most of the SQL standards. The language is also structured as a DSL and the related MDE framework is integrated with Google Query function in [26]. Similarly, FDL [27] is a description language for spreadsheets, which is empowered with visualization and analysis tool for constructing the separation between the input of formulas and the output of calculation results. Although these studies provide a good MDD framework for spreadsheets, it gets very difficult to extract information on a single potentially large matrix in an effective way inside spreadsheets and this deficiency may cause spreadsheets a weak alternative for databases, especially the ones as being Elasticsearch storages.

Ristic et al. [21] define a model-driven database reverse engineering mechanism through a chain of model-to-model transformations. These transformations are applied between physical database schema and generic relational schema. A similar model-to-model transformation approach is followed in [22] for automatically achieving a form type data model again from a generic database schema. Hence, the form type specification represents a platform independent prescription model of both future screens and report forms which can be generated later for a complete application. Popovic et al. [23] propose a DSL, called IIS*CFuncLang, to specify application-specific functionalities of business applications for different domains at the platform-independent model level. The DSL enables modeling the system to be developed and generalization of the required executable codes is realized via some model transformations. Hence, specifications defined with the DSL can be converted into executable PL/SQL program codes. These studies bring valuable MDD solutions on database processing, query generation and reverse engineering of databases. However, the metamodels, the transformations and the DSLs defined in these studies do not consider the Elasticsearch engines and hence, deriving an MDD framework for generating queries, structured according to Elasticsearch specifications on various query types, is not covered in these studies.

Research on Elasticsearch has been recently emerged due to novelty brought into query structures. Kononenko et al. [3] discuss how Elasticsearch differs from the traditional relational databases and give some concrete applications of using Elasticsearch queries. In addition, they give their assessment on the strengths and the weaknesses of Elasticsearch for querying new software repositories. Elasticsearch's inverted index capabilities are used in [28] for implementing an optimized intelligent search algorithm. Query optimization with using this algorithm is employed in the retrieval of medicine data. Finally, a social media analysis system is introduced [29] in which features of Elasticsearch are used on analyzing Big Data. Two ways of giving Twitter data as input to Elasticsearch are defined and their performances are compared by means of consuming hardware resources and the capacity of processing tweets. Our work contributes to the research on Elasticsearch by introducing a DSML and its supporting IDE which can be used to facilitate and expedite the creation of Elasticsearch queries by following a MDD process.

VI. Conclusion

A DSML, called DQL, for supporting MDD of Elasticsearch queries has been introduced in this paper. Based on the derived metamodel of Elasticsearch queries, a graphical concrete syntax is provided for query modeling inside the IDE of the language. All required constraint checks and query validations are automatically performed on the models prepared inside this IDE and Elasticsearch queries are generated from these models. Furthermore, IDE is capable of executing these auto-generated Elasticsearch queries on remote repositories and creating reports covering the execution results. The conducted evaluation showed that the use of the language significantly decreases the development time required for creating Elasticsearch queries. Finally, qualitative assessment, based on the developers' feedback, exposed how DQL facilitates the development of Elasticsearch queries.

In the future work, our aim is to extend DQL's coverage on different types of Elasticsearch use cases. Also, we plan to enrich DQL's query modeling environment with improved visualization components especially for reporting Elasticsearch results. In order to determine how these new components enable more feasible query generation, the evaluation performed on DQL will be improved and experiment settings will be structured with including some sort of hypothesis testing as being considered in similar efforts like [30-32].

Acknowledgment

We would like to thank software developers from Galaksiya Information Technologies for their cooperation and valuable feedbacks. This work was supported by Yaşar Group (http://yasar.com.tr/en/).

REFERENCES

[1] A. Bialecki, R. Muir, G. Ingersoil. 2012. "Apache Lucene 4", in *Proc. SIGIR 2012 Workshop on Open Source Information Retrieval*, Portland, Oregon USA, pp. 17–24.

[2] Elasticsearch BV. 2014. "Elasticsearch - The Heart of the Elastic Stack", available at: https://www.elastic.co/products/elasticsearch (last access: July 2018)

[3] O. Kononenko, O. Baysal, R. Holmes, M. W. Godfrey. 2014. Mining modern repositories with elasticsearch. In *Proc. 11th Working Conference on Mining Software Repositories (MSR 2014)*, Hyderabad, India, pp. 328–331, DOI: 10.1145/2597073.2597091.

[4] Elasticsearch BV. 2015. "Kibana - Your Window into the Elastic Stack", available at: https://www.elastic.co/products/kibana (last access: July 2018)

[5] B. Selic. 2003. The pragmatics of model-driven development. *IEEE Software* 20: 19-25, DOI: 10.1109/MS.2003.1231146

[6] J. Poruban, M. Bacikova, S. Chodarev, M. Nosal. 2014. "Pragmatic Model-Driven Software Development from the Viewpoint of a Programmer: Teaching Experience", in *Proc. 3rd Workshop on Model Driven Approaches in System Development (MDASD@FedCSIS'14)*, Warsaw, Poland, pp. 1647–1656, DOI: 10.15439/2014F266.

[7] M. Brambilla, J. Cabot, M. Wimmer. 2017. *Model Driven Software Engineering in Practice, Second Edition*, Morgan & Claypool, DOI: 10.2200/S00751ED2V01Y201701SWE004

[8] J. Whittle, J. Hutchinson, M. Rouncefield. 2014. The state of practice in model-driven Engineering. *IEEE Software*, 31(3):79-85, DOI: 10.1109/MS.2013.65.

[9] G. Kardas. 2013. Model-driven development of multi-agent systems: a survey and evaluation. *The Knowledge Engineering Review*, 28(4): 479-503, DOI: 10.1017/S0269888913000088

[10] S. Mustafiz, X. Sun, J. Kienzle, H. Vangheluwe. 2008. Model-driven assessment of system dependability. *Software & Systems Modeling*, 7(4): 487-502, DOI: 10.1007/s10270-008-0084-1.

[11] H. B. Saritas, G. Kardas. 2014. A model driven architecture for the development of smart card software. *Computer Languages, Systems & Structures*, 40(2): 53-72, DOI: 10.1016/j.cl.2014.02.001.

[12] A. Harbouche, N. Djedi, M. Erradi, J. Ben-Othman, A. Kobbane. 2017. Model driven flexible design of a wireless body sensor network for health monitoring. *Computer Networks*, 129(2): 548-571, DOI: 10.1016/j.comnet.2017.06.014.

[13] F. Erata, C. Gardent, B. Gyawali, A. Shimorina, Y. Lussaud, B. Tekinerdogan, G. Kardas, A. Monceaux. 2017. "ModelWriter: Text & Model-Synchronized Document Engineering Platform", in *Proc 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, Urbana-Champaign, Illinois, USA, pp. 907-912.

[14] M. Mernik, J. Heering, A. Sloane. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4): 316-344, DOI: 10.1145/1118890.1118892.

[15] M. J. Varanda Pereira, M. Mernik, D. da Cruz, P. Rangel Henriques. 2008. Program Comprehension for Domain-specific Languages. *Computer Science and Information Systems*, 5(2): 1-17, DOI: 10.2298/CSIS0802001P.

[16] I. Lukovic, M. J. Varanda Pereira, N. Oliveira, D. da Cruz, P. Rangel Henriques. 2011. A DSL for PIM specifications: Design and attribute grammar based implementation. *Computer Science and Information Systems*, 8(2): 379-403, DOI: 10.2298/CSIS101229018L.

[17] T. Kosar, M. Mernik, J. Gray, T. Kos. 2014. Debugging measurement systems using a domain-specific modeling language. Computers in Industry, 65(4): 622-635, DOI: 10.1016/j.compind.2014.01.013.

[18] B. Bryant, J-M. Jezequel, R. Lammel, M. Mernik, M. Schindler, F. Steinmann. 2015. "Globalized Domain Specific Language Engineering", in *Globalizing Domain-Specific Languages*. B. Combemale, B. Cheng, R. France, J-M. Jezequel, B. Rumpe (eds). *Lecture Notes in Computer Science*, 9400: 43-69, DOI: 10.1007/978-3-319-26172-0_4.

[19] G. Kardas, B. T. Tezel, M. Challenger. 2018. Domain-specific modelling language for belief-desire-intention software agents. *IET Software*, DOI: 10.1049/iet-sen.2017.0094.

[20] J. Cunha, J. P. Fernandes, J. Mendes, R. Pereira, J. Saraiva. 2013. "Querying model-driven spreadsheets", in *Proc. 2013 IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2013)*, San Jose, CA, USA, pp. 83-86, DOI: 10.1109/VLHCC.2013.6645247.

[21] S. Ristic, S. Aleksic, M. Celikovic, V. Dimitrieski, I. Lukovic. 2014. Database reverse engineering based on meta-models. *Central European Journal of Computer Science*, 4(3): 150-159, DOI: 10.2478/s13537-014-0218-1.

[22] S. Ristic, S. Kordic, M. Celikovic, V. Dimitrieski, I. Lukovic. 2016.. "A Model-to-Model Transformation of a Generic Relational Database Schema into a Form Type Data Model", in *Proc. 4rd Workshop on Model Driven Approaches in System Development (MDASD@FedCSIS'16)*, Gdansk, Poland, pp. 1577–1580, DOI: 10.15439/2016F408.

[23] A. Popovic, I. Lukovic, V. Dimitrieski, V. Djukic. 2015. A DSL for modeling application-specific functionalities of business applications. *Computer Languages, Systems & Structures*, 43: 69-95, DOI: 10.1016/j.cl.2015.03.003.

[24] Elasticsearch BV. 2015. "Elastic Stack and Product Documentation", available at: https://www.elastic.co/guide/index.html (last access: July 2018)

[25] J. Cunha, J. P. Fernandes, J. Mendes, J. Saraiva. 2012. "MDSheet: A framework for model-driven spreadsheet engineering"., in *Proc. 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, pp. 1395-1398, DOI: 10.1109/ICSE.2012.6227239.

[26] J. Cunha, J. P. Fernandes, J. Mendes, R. Pereira, J. Saraiva. 2015. "Design and Implementation of Queries for Model-Driven Spreadsheets", in *Central European Functional Programming School*. V. Zsok, Z. Horvath, L., Csató (eds). *Lecture Notes in Computer Science*, 8606: 459-478, DOI: 10.1007/978-3-319-15940-9_13.

[27] Y. Horry. 2017. Financial information description language and visualization/analysis tools. *Computer Languages, Systems & Structures*, 50, 31-52, DOI: 10.1016/j.cl.2017.05.005.

[28] C. Bhadane, H. A. Mody, D. U. Shah, P. R. Sheth. 2014. Use of Elastic Search for Intelligent Algorithms to Ease the Healthcare Industry. *International Journal of Soft Computing and Engineering*, 3(6), 222-225.

[29] P. P. I. Langi, Widyawan, W. Najib, T. B. Aji. 2015. in Proc. 2015 International Conference on Information, Communication Technology and System (ICTS 2015), Surabaya, Indonesia, pp. 181-186, DOI: 10.1109/ICTS.2015.7379895.

[30] F. Haser, M. Felderer, R: Breu. 2016. Is business domain language support beneficial for creating test case specifications: A controlled experiment. *Information and Software Technology*, 79, 52-62, DOI: 10.1016/ j.infsof.2016.07.001.

[31] A. N. Johanson, W. Hasselbring. 2017. Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Software Engineering*, 22(4), 2206-2236, DOI: 10.1007/s1066.

[32] T. Kosar, S. Gaberc, J. C. Carver, M. Mernik. 2018. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering*, DOI: 10.1007/s10664-017-9593-2.