Bootstrapping Natural Language Querying on Process Automation Data

Xue Han[1], Lianxue Hu[1], Jaydeep Sen[2], Yabin Dang[1], Buyu Gao[1], Vatche Isahagian[3], Chuan Lei[4], Vasilis Efthymiou[4], Fatma Özcan[4], Abdul Quamar[4], Ziming Huang[1], Vinod Muthusamy[3]
1. IBM Research AI, China, 2. IBM Research AI, India,
3. IBM Research AI, Cambridge, US, 4. IBM Research AI, Almaden, USA

***Abstract*—Advances in the adoption of business process management platforms have led to increasing volumes runtime event logs, containing information about the execution of the process. Business users analyze this event data for real-time insights on performance and optimization opportunities. However, querying the event data is difficult for business users without knowing the details of the backend store, data schema, and query languages. Consequently, the business insights are mostly limited to static dashboards, only capturing predefined performance metrics. In this paper, we introduce an interface for business users to query the business event data using natural language, without knowing the exact schema of the event data or the query language. Moreover, we propose a bootstrapping pipeline, which utilizes both event data and business domain-specific artifacts to automatically instantiate the natural language interface over the event data. We build and evaluate our prototype over datasets from both practical projects and public challenge events data stored in Elasticsearch. Experimental results show that our system produces an average accuracy of 80% across all data sets, with high precision (91%) and good recall (81%).**

***Keywords*-Natural Language Querying; Process Automation; Workflow; Business Automation Insights**

I. Introduction

Business process frameworks have become a cornerstone for enterprise operations including loan origination, loan validation, invoice management, and insurance claims processing [1]. Companies are deploying applications that automate their business processes with reduced cost and better customer experience. These applications generate high-volume event logs[1], which contain rich information about multiple stages of process workflows, such as their assignment, their sequence of executions, and past and current status. These event logs in turn are used by business intelligence (BI) systems to generate reports or dashboards summarizing key performance indicators (KPI) and other metrics [2].

Given the popularity of business process applications, there is a greater need for business users to analyze event logs to get operational and business insights. However, accessing and querying event logs come with non-trivial challenges. Event logs are usually opaque to business users because they follow a specific structure, which is tightly coupled with the process workflow and log structure. Also, the event logs from process automation can be stored in various backend data stores, such as Elasticsearch, HDFS, or relational databases. All these backends only support a very specific set of query languages with which non-technical users are not familiar. Hence, the access and usage of event logs by non-technical business users are greatly limited, even though they have prevalent need for insight generations from these event logs. Today, the insight generation is mostly limited to a set of static dashboards which are predefined by technical analysts to measure a very specific set of business KPIs.

[1]We use event log and event data interchangeably in this paper.

Recent advances in natural language understanding and processing, as well as the advent of chatbots, have fueled a renewed interest in Natural Language Interfaces to Databases (NLIDB) [3]. These NLIDB systems [4], [5] allow non-technical users to explore the database using natural language queries (NLQs) without knowing specific query languages or the exact schema of the underlying data. The core challenges of building an NLIDB system are understanding the semantics of the natural language query, i.e., the user intent, and then generating the correct query corresponding to the intent to retrieve the data from the backend store. Unfortunately, none of the existing systems are specifically designed for querying process automation data.

Challenges. To fill the gap between process automation data (i.e., event logs) and the NLIDB systems, we chose to use a state-of-the-art NLIDB system ATHENA [5] which has a modular end-to-end pipeline for querying databases, thus making it suitable to customize for applications. However in order to instantiate ATHENA over process automation data we still need to tackle the following challenges. First, ATHENA uses domain ontologies to understand a domain. Hence, it is imperative to produce an ontology that captures the relevant domain semantics of process automation. Second, process automation data are typically stored in NoSQL backend data stores, which provide flexible schemas for the storage and retrieval of data. Hence we need to extend ATHENA, to support the query language used in the NoSQL backend. Third, like most of the NLIDB systems, instantiating ATHENA also involves non-trivial amount of manual effort, which makes domain adaptation cumbersome. Therefore, we need a solution to automatically generate the required system configurations to instantiate ATHENA over the process automation data.

Contributions. In this paper, our contributions can be summarized as follows.

- We extend the state-of-the-art NLIDB system ATHENA [5], which utilizes domain ontologies to capture the domain semantics, with a new query translator to support querying process automation data stored in Elasticsearch.

- We introduce an ontology generation method that derives a domain ontology from the process automation data and further enriches the ontology with business specific terminologies and vocabularies from domain artifacts.
- We design a bootstrapping pipeline that utilizes the ontology derived from process automation data and automatically instantiates each component in ATHENA with domain-specific configurations, which enables easy domain adaptation and automated setting up of ATHENA.
- We evaluate our system using publicly available event logs from BPI'2017 challenge [6] and Statechart [7], as well as three other running applications used in practice. Experimental results show that our system produces an average accuracy of 80% across all data sets, with high precision (91%) and acceptable recall (81%). We also do a thorough user study which confirms that our system can effectively support a wide range of queries for insight generation and so is perceived as a novel and useful application over process automation.

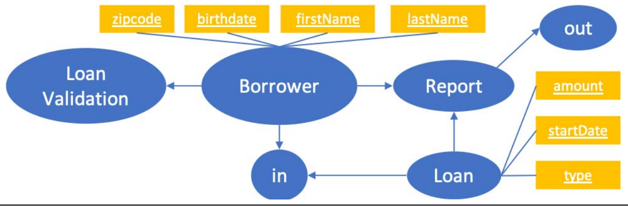
II. Background

We chose to use a state-of-the-art system NLIDB system ATHENA as the backbone over which we build our auto-ingestion pipeline to bootstrap natural language querying engine for process automation data. ATHENA has a modular architecture of sequential connected components. This allowed us to treat each of ATHENA's component as a service and develop our auto-ingestion pipeline as an orchestrator over them. In this section, we briefly go through the individual components of ATHENA - which are (1) Ontology (2) Query Interpretation (3) Query Translation. [2]

A. Ontology

The domain ontology is a central piece in ATHENA, which captures a semantic description of the domain in terms of the entities and their relationships relevant to the domain.

Figure 1 A Snapshot of Loan Validation ODM Process Ontology



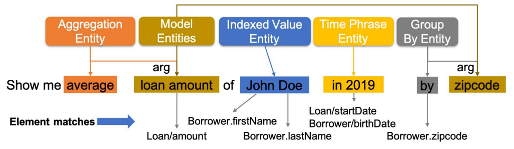
[2]We were granted access to ATHENA's code base by ATHENA authors, which we have reused in our current work.

Figure 1 shows a snapshot of the ontology derived from event data generated by ODM process automation for loan validation. Some of the concepts included in this ontology are *Borrower* and *Loan*. Each concept has a set of properties. For example, the concept *Borrower* has the properties *firstName*, *lastName*, *birthdate*, and *zipcode*. The concepts *in* and *out* correspond to the input and output of ODM decision services for loan validation. The edge between *Borrower* and *in* signifies that there is a relation between those two concepts, since *Borrower* information is used as an input to ODM decision services.

B. ATHENA – Query Interpretation.

Given a natural language query (NLQ), ATHENA parses and annotates the query to produce evidence that one or more ontology elements have been referenced in the input. A lookup index is created over the back-end data to detect mention of data instances too (e.g. a Person name like "John"). Moreover, ATHENA also employs a set of annotators to detect signals of specific operations like aggregation, time phrase mention, group by, etc. to unambiguously identify the role of each relevant word in the target query clause. Figure 2 is an example query along with its annotations.

Figure 2 NLQ Annotations (Best Viewed in Color)



After the evidence annotations, ATHENA uses a Steiner Tree-based algorithm [5] to produce the optimal join paths between the individual ontology elements as referred by the NL query. It produces an interpretation tree (*ITree*) for the given NLQ. This (*ITree*) and the clause level evidence annotations are used to produce a unique OQL (Ontology Query Language) query that represents the NL query intent. OQL is syntactically similar to back-end query languages such as SQL, except that OQL is agnostic to backend stores, and is defined over the domain ontology. We show the OQL query corresponding to the NLQ in Figure 2 below.

```
SELECT AVG(oLoan.amount), oBorrower.zipcode
FROM Loan oLoan, Borrower oBorrower
WHERE oLoan.startDate >= 01-01-2019,
      oLoan.startDate <= 31-12-2019,
      oBorrower.firstName = “John”,
      oBorrower.lastName = “Doe”,
      oBorrower → report_borrower → report_loan =
      oloan,
GROUP BY oBorrower.zipcode
```

C. ATHENA – Query Translation

The OQL query is expressed against the ontology in terms of concepts, data properties and their relationships.

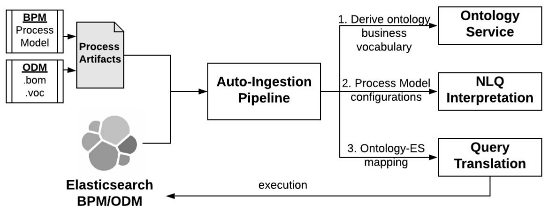
The query translator utilizes a pre-defined mapping between these ontology elements and the elements of the physical schema in which the data is stored to generate the target language query. The query translation process at a high level considers the various clauses of the OQL query like *select*, *aggregation*, *where*, *groupby* etc. and translates them into an appropriate query structure in the target query language that provides an equivalent query functionality. In this paper, we propose an elaborate algorithm to produce OQL-DSL translator (in Section III-D).

III. System Design

A. System Overview

Figure 3 shows the overall system architecture of our auto-ingestion pipeline for instantiating ATHENA components over BAI event data (in JSON) stored in Elasticsearch.

Figure 3 System Architecture



Our auto-ingestion pipeline derives an ontology from the event data (in JSON) stored in Elasticsearch and processes specific artifacts like BPM process models, ODM .bom/.voc files etc.. It uploads the derived ontology into the ontology service so that the ontology can be accessible by the query interpretation and the query translation components. Second, the auto-ingestion pipeline infers more process model specific configurations relevant for natural language querying and passes this as process specific configurations to the query interpretation component. Third, the auto-ingestion pipeline also derives the mapping between the ontology elements and their corresponding paths as JSON keys in the event data stored in Elasticsearch. This completes the bootstrapping and instantiation of each individual components in ATHENA. At query time, user queries are translated into DSL queries and executed on BAI events stored in Elasticsearch to retrieve the results.

In the following sections, we specifically focus on the main contributions of this paper as (1) Ontology generation from event data (2) Extracting process specific information from process models/artifacts and (3) Algorithm to translate an OQL query produced by ATHENA which is back-end agnostic to an executable DSL query on Elasticsearch.

B. Ontology Generation

Ontology generation happens in two phases. First we analyse the BAI event data structure to extract the schema of the back-end data and use it to model the concepts and relations in the ontology. Next, we also utilize the domain specific artifacts to semantically enrich the ontology for understanding process automation specific details. In this section, we describe the ontology generation process from the BAI event data (i.e., JSON documents) stored in Elasticsearch. There are two primary challenges for deriving ontology from the BAI event data.

•**Schema discovery.** Each JSON document can be viewed as a set of key-value pairs. Each key represents a particular field name, and the value can be either a scalar such as a string, integer, etc., or a JSON object (implying a nesting within the JSON document), or a JSON array of values (recursively defined). Note that the structure of event JSONs may have the same, disjoint or overlapping schema depending on its origin and domain. We extract the individual structures from each JSON document, in the form of a schema tree where individual nodes represent JSON objects, arrays or literals, and each root to leaf path represents a unique path in the tree. Individual schema tree paths are then merged in the from of a data guide [8] to represent the overall schema of the data set, containing heterogeneous JSON structures. This schema is then used to derive an ontology. This helps us in discovering a schema that covers the structure of the entire set of event data.

•**Concept and relationship inference.** The second challenge in ontology generation is to identify the concepts, the properties associated with each concept and the relationships between these concepts. Algorithm 1 describes the ontology generation process from the event JSON. Intuitively, it applies the following rules on each path of the schema guide.

1) Path $A.b$: Concept A, Property b of A.
2) Path $A.B.c$: Concept A, Concept B, Property c of B, Relation $A \rightarrow B$.

While creating a relationship between two concepts, we also consider the type of the new concept. If the new concept is a JSON object, the cardinality of the relationship is $1:1$. If the new concept is a JSON array, the cardinality is $1:m$, where m is the length of the JSON array. We skip these details in Algorithm 1 for simplicity. The JSON object associated with the new concept is traversed recursively to expand the ontology with that JSON object as the root (Line 10). The algorithm keeps track of the JSON keys traversed for reaching the new concept, and maintains a mapping between JSON key paths to ontology concepts (Line 9). This mapping is used for query translation as described in Section III-D. Figure 4 shows the ontology generated from the schema tree extracted across all JSON documents in the event data collected for odm-loan validation domain(more on it in Section IV).

C. Process Specific Configurations

The process specific artifacts differ depending on the underlying process automation model. We discuss the usage of process specific artifacts for both BPM and ODM processes.

•**BPM:** Process model is an important artifact which contains information about various activities and how they are sequenced in a process life cycle. Figure 5 shows BPM process model of Travel Pre-approval. We utilize the process

Algorithm 1: Ontology Generation

```
   Input: JSONObject $rootJSON$, Concept $rootConcept$, String $keyPath$
   Output: Ontology $O$, Map[Concept, String] $mapping$
1  $O.addConcept(rootConcept)$
2  foreach $Key \in rootJSON.keys$ do
3      if $rootJSON.get(key)$ is Literal then
4          $rootConcept.addProperty(rootJSON.get(key))$
5      if $rootJSON.get(key)$ is JSONObject | JSONArray then
6          Concept $newKeyConcept$ = new Concept()
7          $rootConcept.addRelation(newKeyConcept)$
8          String $newConceptPath$ = $keyPath$ + "." + $key$;
9          $mapping.put(newKeyConcept, newConceptPath)$
10         Ontology $subOntology$ =
             DeriveOntologyFromJSON($rootJSON.get(key)$,
             $newKeyConcept$, $newConceptPath$)
11         $O.include(subOntology)$
12 Return $O$, $mapping$
```

Figure 4 Ontology Generation for ODM-LoanValidation

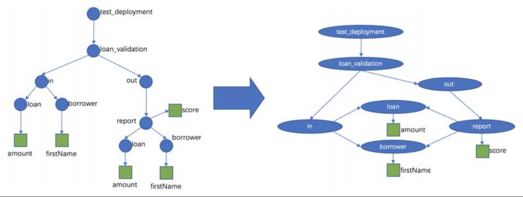
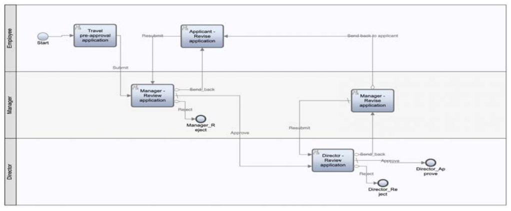


Figure 5 BPM Process Model for Travel Pre-approval



model to extract the following information:
(1) Activity Names: We extract the different activity names from the process model to index for natural language querying. From Figure 5 the activity names are *Travel pre-approval application, applicant-revise application* and so on.
(2) Activity Sequence: The BPM process model also captures the sequence in which these activities can occur in a process life cycle. We use the sequence edges between activities to topologically sort the activities and maintain an ordered list of pre-defined activities. This is useful for answering queries asking for events before or after a specific activity.
(3) Roles: The BPM process model also has a role information associated with each activity. For example, as seen in Figure 5, a *Manager* role performs *Manager-review application* and *Manager-revise application*. The mapping from role to activity helps in answering natural language queries involving roles such as "what are the active events for manager roles".

•**ODM** : We have worked with running ODM processes instantiated from IBM ODM decision services [9] where .bom [10] and .voc [11] files are key domain artifacts. A .bom file describes the domain in terms of entities and their attributes, whereas a .voc file defines the business vocabularies associated with those domain elements. We parse the .bom file to identify the concepts and their attributes along with their datatype. We augment the ontology derived from event data for any missing concept, attribute or its datatype. Furthermore, we exploit the natural language phrases represented as *verbalization examples* [12] associated with each concept and property in the artifact as synonyms for the corresponding concepts and properties in the derived ontology. These synonyms enrich the business-specific terminologies in the ontology, and enhance the recall of natural language query interpretation.

D. DSL Translator

In this section we start with an OQL query produced by ATHENA and design an algorithm to map the specific OQL query clauses (e.g., SELECT, WHERE, FROM, GROUPBY, ORDERBY, HAVING, etc.) to the appropriate DSL constructs with equivalent functionalities. Since OQL is a back-end agnostic query language which only captures the natural language query intent, we believe the algorithm we develop for OQL-DSL translation is easily extendable to any other source query language.

The algorithm first maps OQL.Aggregation clauses to the corresponding aggregation functions supported in DSL. If the aggregation is also associated with a HAVING clause, the algorithm translates it into a *bucket selector* in the DSL query. The GROUPBY and SELECT clauses are translated into group by buckets using the DSL query construct *Term*. In case of non-aggregation queries, the SELECT clause is mapped to the *_source* JSON array, which is a DSL construct used to return the specific sub-fields from each document. The ORDERBY clause is mapped to the *_sort* function which is a DSL query construct for sorting, to produce sorted JSON using *sortJSON* as the argument. In case the ORDERBY has an aggregation as its argument, the *sortJSON* is pushed inside the aggregation. Finally, the algorithm puts all the DSL query constructs created from the OQL clauses together into a final DSL query (*DSLQuery*) and uses the mapping file to map field names in the DSL query to its fully qualified JSON path. the The final DSL query corresponding to the OQL query presented in Section II-B is shown in Figure 6.

E. System Implementation as Services

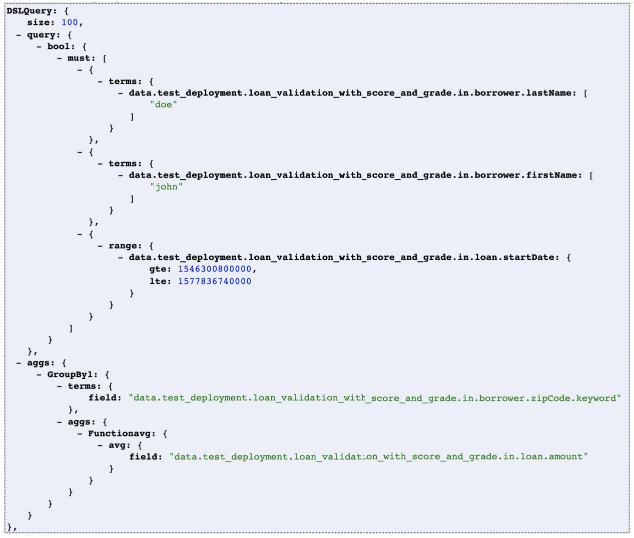
In this section, we focus on the implementation of our prototype system. We implement each component in Figure 3 as a separate service. RESTful API is used for the interaction between services. We provide details on each service signature (RESTful API) and their expected behavior in the end-to-end pipeline. Below, we list the services in sequence of how they should be used.

Step 1:GET /deployment/set_ES_endpoint. This service is used to set the details about the Elasticsearch (ES) store back end where the event data is stored. This service has the following parameters: (*a*) ES url, (*b*) ES index name, (*c*) Elasticsearch user credentials, and (*d*) an assigned

Algorithm 2: OQL-DSL Translation

```
    Input: OQLQuery oql, Ontology O, Map[Concept,keyPath] mapping
    Output: JSONObject DSLQuery
1   JSONObject
      $DSLQuery, queryJSON, aggClauseJSON, GroupByTermJSONs, sortJSON$
      $\leftarrow \{\}$
2   JSONArray $sourceJSONArray \leftarrow \{\}$,
3   foreach AggregationClause agg $\in$ oql.getAggregations() do
4       String $dslaggfunc = getDSLAggregationFucntion(agg.getAggType)$
5       $aggClauseJSON.add(createDSLAggrJSON(dslaggfunc, agg.getProperty))$
6       if $\exists$ OQL.Having on agg then
7           JSONObject $bucketSelectorJSON \leftarrow$
              $createBucketSelectJSON(dslaggfunc,$
8           $comparison.op, comparison.value)$
9           $aggClauseJSON.put($"$bucket\_selector$"$, bucketSelectorJSON)$
10      if $\exists$ OQL.OrderBy OB with agg then
11          $sortJSON \leftarrow$
              $createSORTJSON(dslaggfunc, OB.getTopCount())$
12          $aggClauseJSON.put($"$bucket\_sort$"$, sortJSON)$
13  foreach WhereClause comparison $\in$ OQL.Where do
14      if comparison is Non-numeric filter then
15          $queryClauseJSONs.add(createTERMJSON(comparison.property,$
16          $comparison.op, comparison.value))$
17      if comparison is Numeric filter then
18          $queryClauseJSONs.add(createRangeJSON(comparison.property,$
19          $comparison.op, comparison.value))$
20  $queryJSON.put($"$must$"$, queryClauseJSONs)$
21  foreach OQLClause selGB $\in$ OQL.GroupBy $\cup$ OQL.Select do
22      if OQL has Aggregation then
23          $GroupByTermJSONs.add(selGB.getProperty())$
24      $sourceJSONArray.add(selGB.getProperty)$
25  if GroupByTermJSONs not Empty then
26      $GroupByTermJSONs.put($"$aggs''$, aggClauseJSON)$
27      $aggClauseJSON \leftarrow GroupByTermJSONs$
28  foreach OrderByClause OB $\in$ OQL.OrderBy do
29      $sortJSON \leftarrow$
          $createSORTJSON(OB.getProperty(), OB.getTopCount())$
30  $DSLQuery.put($"$query''$, queryJSON)$
31  $DSLQuery.put($"$aggs''$, aggClauseJSON)$
32  $DSLQuery.put($"$\_source''$, sourceJSON)$
33  $DSLQuery.put($"$\_sort''$, sortJSON)$
34  $DSLQuery \leftarrow mapJSONPaths(DSLQuery, mapping)$
35  Return DSLQuery
```

Figure 6 A generated DSL example



```
DSLQuery: {
    size: 100,
  - query: {
      - bool: {
          - must: [
              - {
                  - terms: {
                      - data.test_deployment.loan_validation_with_score_and_grade.in.borrower.lastName: [
                            "doe"
                        ]
                    }
                },
              - {
                  - terms: {
                      - data.test_deployment.loan_validation_with_score_and_grade.in.borrower.firstName: [
                            "john"
                        ]
                    }
                },
              - {
                  - range: {
                      - data.test_deployment.loan_validation_with_score_and_grade.in.loan.startDate: {
                            gte: 1546300800000,
                            lte: 1577836740000
                        }
                    }
                }
            ]
        }
    },
  - aggs: {
      - GroupBy1: {
          - terms: {
                field: "data.test_deployment.loan_validation_with_score_and_grade.in.borrower.zipCode.keyword"
            },
          - aggs: {
              - Functionavg: {
                  - avg: {
                        field: "data.test_deployment.loan_validation_with_score_and_grade.in.loan.amount"
                    }
                }
            }
        }
    }
},
```

OntologyID. The assigned *OntologyID* is used as a key for future reference.

Step 2: POST /deployment/upload_artifact. This service is used to provide the domain artifact in JSON which combines the information from .bom/.voc files. This service is optional. Namely, if the end user has no specific business vocabulary glossary or equivalent JSON for the domain, this service can be skipped.

Step 3: GET /deployment/generate_config. This service is called after an end user has already provided the credential to Elasticsearch (Step 1) and uploaded the artifact in JSON (Step 2). The service processes the above information and execute the following sequentially: (*a*) generates an ontology; (*b*) generates synonyms and configurations for the derived ontology for query interpretation; (*c*) generates a mapping between the ontology and the schema of JSON data stored in Elasticsearch for query translation; (*d*) starts the ontology service at a designated port ONT_PORT; (*e*) uploads the ontology generated from (*a*), along with configurations from (*b*) to the ontology service started in (*d*).

Step 4: GET /deployment/start_components. Once the domain-specific setup is finished in terms of deriving the ontology for NLQ interpretation and generating the mapping between the ontology and the schema of JSON data needed for query translation, this service starts each component in the following sequence: (*a*) generates the configuration file with Elasticsearch access credentials[3] and other details as needed by the query translation; (*b*) starts the query translation service at a specific port EX_PORT; (*c*) configures the query translation service using the configuration generated in (*a*); (*d*) starts the NLQ interpretation service at a specific port INT_PORT.

Note that query translation service is pre-configured to use the ontology service running in ONT_PORT (as started in Step 3(d)). Similarly, the ATHENA interpretation service is pre-configured to use the ontology service running in ONT_PORT and the ATHENA query translation service running in EX_PORT (as started in Step 4(b)). After Step 4, the ATHENA entry point is running at port INT_PORT and can be used for natural language querying by end users.

IV. Experiments

In this section, we present a comprehensive evaluation of our proposed system through experiments on various benchmarks and also an elaborate usability study.

A. Data Sets

We evaluate our prototype system on event logs from 2 publicly available and 3 internal processes which are practically in use inside our organization.

BPI Challenge 2017 [6]: We used the data released as part of BPI 2017 Challenge. It contains events pertaining to loan applications and offers created for these applications. The data set has three different types of events, of which we choose *application* and *offer* events. Since the structure of event logs are different for application and offer events, we

[3]Note that the Elasticsearch credentials are needed for executing the DSL queries translated from OQL queries.

store the event logs for each of them into separate indices in Elasticsearch. Hence, we consider them as two separate domains in our experiments.

Statechart [7]: It has software event logs obtained through running a process mining program over a data dump. The event log contains method-call level events describing a program run. The life-cycle information in this log corresponds to methods call (start) and return (complete), and captures a method-call hierarchy.

BPM process on case activities. We collected event data from a BPM process deployed to track case activities. The data set mainly includes different type of activities for loan application processing cases. The process has various tasks such as submit, evaluate, decide, etc. Also each task has a duration and a status (e.g. waiting, complete etc).

BPM process on Travel Pre-Approval. We collected event data from a BPM process deployed to automate travel pre-approval process. It has information about travel request applications and various activities including manager approval, application revise, director approval etc. .

ODM service on loan validation. These event logs are generated from ODM decision running on Loan Validation. The decision service takes information about borrower and loan request as input, and generates a report containing the input information as well as a decision on the status (i.e., approved or rejected) of the loan.

The statistics of the above data sets with respect to the derived ontologies is provided in Table I. BPI challenge data had all event information logged at the JSON root level, hence they are single concept domains.

Domain	Concept	Property	Relation	Query
BPI-Loan Application	1	11	-	44
BPI-Loan Offer	1	17	-	43
BPI-Statechart	2	26	1	43
BPM-Case Activity	2	16	1	56
BPM-Travel Pre-approval	3	57	2	46
ODM-Loan Validation	10	235	10	57

Table I: Data Set Statistics

B. Experimental Setup

Users and query benchmark. As we are the first one to propose a natural language querying interface over process automation data, we also propose benchmarks of natural language queries which we curated by collecting queries from real users. We collaborated with real users who volunteered to experiment with our system by asking natural language queries to it. The users were from different profiles as mentioned below:
(1) Business Users: They are the end users of process automation technology including BPM, ODM and other workflow automation tools.
(2) Process Automation Developers: They are the developers who build and customize the process automation tools for specific domains and services.
(3) NLI Experts: They are experts in using Natural Language Interfaces (NLI) but not specifically familiar with process automation technology space.

A total of 4 business users, 5 developers and 3 NLI experts took part in our experiments. The business users and developers had prior experience in using process automation applications and also event logs across various use cases. For NLI experts, we briefed them on process automation applications. All the users were granted access to all the artifacts related to our experimental domains including the actual BAI data, the textual summaries, as well as the domain artifacts like the BPM process model or the .bom and .voc files for ODM service. We had shown them some example NLQs of different categories that our system can handle. During the experiments, the users were encouraged to test more natural language queries which they found interesting or insightful business process management. We collected these queries to create the benchmark against which we have evaluated our system. [4].

Evaluation metrics. We use the following metrics to evaluate the performance of our prototype system.
Accuracy: # NLQs with correct answers / # NLQs asked to the system
Precision: # NLQs with correct answers / # NLQs answered by the system
Recall: # NLQs with correct answers / # NLQs that should have produced correct answers
F1 score: 2 $\times$ Precision $\times$ Recall / (Precision + Recall)

C. Results and Discussion

The result produced by our prototype system is considered correct only if it produces the correct DSL query, and consequently returns the data requested by the NLQ.

Domain	Accuracy	Precision	Recall	F1 Score
BPI-loan Offer	70.21%	93.5%	70.73%	0.80
BPI-Loan Application	80%	81%	75%	0.779
BPI-Statechart	78.26%	92.11%	83.33%	0.875
BPM-Case Activity	80.35%	93.02%	83.33%	0.879
Travel Pre-approval	86.95%	95.12%	90.69%	0.928
ODM-Loan Validation	84.21%	95.34%	85.41%	0.90

Table II: Effectiveness of Our NLI Prototype

Effectiveness. Table II provides the evaluation results for the query benchmark covering different business processes. In Table II, our prototype achieves the highest accuracy of 86.95% accuracy on the Travel Pre-approval data set. That is also the business process which achieves the highest recall. This can be attributed to having the ontology enhanced with the business terminologies found in the BPM process model. In fact all the internal domains such as BPM case activity or

[4]Derived ontologies and the benchmark of queries available here: https://github.com/jdpsen/Benchmark_for_NLQOnBAI

ODM Loan validation also attain higher recall and accuracy owing to the availability of corresponding BPM process models or ODM vocabulary (.voc) files. This in general suggests that having process specific data helps in more accurate natural language querying on the event data.

For the publicly used external processes such as BPI-Loan offer, Loan application and statechart, they did not have any process artifacts beyond the event logs. As seen in Table II, our system still attains a decent accuracy in the range of 70%-80%. The average precision across all queries in our benchmark is quite high (91%) and with around 81% average recall our system can still handle a wide variety of queries expected against such business processes without access to any process specific artifacts.

Discussion and Error Analysis: Table II records a high precision throughout all the domains. This signifies, in most of the cases the QA system is confident about the answer it produces. This correlates with the fact that ATHENA is a state-of-art NLIDB system and the proposed DSL translator in this paper is designed to handle most of the common query intents like filters, aggregations, grouping, ordering etc. The marginal drop in precision is seen for queries where user is asking about complex queries which involves nesting like "show me zipcodes with more amount of loans approved than average " or queries involving mathematical operations like " what is the percentage of loan applications are for housing loans ". We do not yet handle such queries and just retrieve the relevant even logs as an answer, which is counted as wrong during precision.

The absence of process specific artifacts is more seen in the recall, where the system fails to answer some of the queries due to the lack of event data understanding. This is expected because these artifacts like .voc file for ODM or the process model for BPM indeed enhances the ontology and enables the QA system to understand a larger scope of queries. For example to answer a query like " list those ICML travel applications which are past manager approval activity" - the QA system needs to know what is the next scheduled activity after manager approval and it gets that information only when the process model is available to create an ordered list of activities (as we discussed in Sec III-C). This is reflected in Table II where the recall values recorded for our internal test domains with access to process specific artifacts is higher than the publicly available benchmarks with no process specific artifacts. Even then with the accuracy, precision and recall values observed on these publicly available event logs validates that the QA system can still perform reasonably well with ontology derived only from event logs data.

Usability analysis. Along with the performance evaluation of our NLI system, we also conduct a user study to measure the impact and value of our proposed system with real users. We follow an established metric measuring the quality of user experiences [13] with a User Experience Questionnaire (UEQ), consisting of six metrics: attractiveness, perspicuity, efficiency, dependability, stimulation and novelty.

Usability study was performed using the same set of users as described in Section IV-B Each participant is given the definitions of these metrics as detailed in [13], and is requested to fill out the UEQ based on their user experience with our prototype system on a scale of [-3,+3].

Figure 7 Results of the UEQ. Range from -3 (horribly bad) to 3 (extremely good)

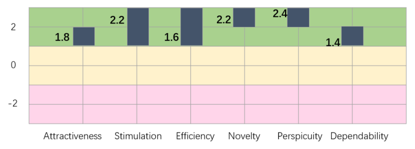


Figure 7[5] shows the results of the user study. All the metrics received a score more than 1.0, which indicates a positive overall experience across all dimensions. The UEQ result also points to some important takeaways that are listed below.

•**Novelty.** Novelty receives the 2nd highest score after perspicuity as 2.2. This implies that the application of an natural language interface over process automation event data is really seen as a novel application. With the bootstrapping framework in place, the ease of using the system also results in a high score for perspicuity.

•**Dependability and effectiveness.** A dependability score of 1.4 coupled with efficiency score of 1.6 means that business users find the query results returned by our system dependable. The high simulation score of 2.2, along with an attractiveness score 1.8 further confirm the user's interest and its value in providing useful insights for business processes.

V. Related Work

Process Automation. Process automation has driven the digital transformation in enterprises, with the initial focus on using BPM frameworks to streamline processes for governance, conformance, and regulations [1]. Tools such as PROM [14] are very much needed for discovery, conformance, and performance monitoring using trace replay capabilities. In recent years, robotic process automation (RPA) has emerged as the next wave of automation, focusing on automating repetitive tasks with minimal changes to legacy software [15]. RPAs have been applied to different application domains including accounting[16], auditing [17] and banking [18]. Most recently, there are attempts to infuse business processes with AI to further automate complex tasks, reduce cost, and provide better customer experience [19]. The BPM literature is rich in machine learning solutions to gain insights on clusters of process traces [20], to predict outcomes [21], and to recommend decisions [22]. Deep learning models, including those from

[5]Presented in the same format as [13], where the length of bar stands for the range of user scores and the printed score is the average of user scores.

the NLP domain, have also been applied [23]. However, the capability of accessing and querying the process automation data has been largely ignored. None of the above systems are equipped with a natural language interface for business users.

Natural Language Interface Systems. Natural language interfaces to databases (NLIDBs) are meant to support NLQs over structured data stored in relational databases. At the core of a NLIDB system lies its ability to understand/interpret a user query expressed in natural language. Typically, an NLIDB system follows either an entity-based approach [4], [5], recognizing the different entities involved in a query using either a database schema or an ontology, or a machine learning-based approach [24], [25], classifying a user query into one of the possible query templates. Machine learning-based approaches often suffer from lack of domain-specific training data to learn complex query classes, and join paths among multiple tables. Seq2SQL [24] can only handle single table select and project queries without joins. Although the rule-based systems such as ATHENA [5] have been shown to achieve better results than the machine learning-based approaches, none of these systems are specifically designed for querying process automation data.

VI. Summary and Future work

In this paper, we introduce a bootstrapping pipeline which can instantiate a natural language querying interface over process automation event logs in Elasticsearch. We proposed novel algorithms for utilizing process specific artifacts and event logs to derive an ontology for querying event logs. Also we propose a novel translation algorithm to take a back-end agnostic OQL query represented in terms of ontology and translate it to an executable DSL query over Elasticsearch.We conduct experiments on six different process automation sources of event data. Experimental results show that our system produces an average accuracy of 80% with high precision (91%) and good recall (81%). The results from the user study reveals the system is perceived as a novel, dependable and efficient system that can handle the long tail of queries that business users may be interested in. In our future work, as we mentioned in IV, we hope to support complex queries which require to combine the answers from different domains. In addition, how to utilize the process model artifact and answer queries requiring the sequence information of events is another research interest.

References

[1] W. Van Der Aalst *et al.*, "Process mining manifesto," in *BPM*. Springer, 2011.

[2] S. Negash and P. Gray, "Business intelligence," in *Handbook on decision support systems 2*. Springer, 2008, pp. 175–193.

[3] F. Özcan, A. Quamar, J. Sen, C. Lei, and V. Efthymiou, "State of the art and open challenges in natural language interfaces to data," in *ACM SIGMOD Tutorial*, 2020, accepted.

[4] F. Li and H. V. Jagadish, "Constructing an Interactive Natural Language Interface for Relational Databases," *PVLDB*, pp. 73–84, 2014.

[5] D. Saha, A. Floratou, K. Sankaranarayanan *et al.*, "Athena: an ontology-driven system for natural language querying over relational data stores," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1209–1220, 2016.

[6] "BPI Challenge 2017," https://www.win.tue.nl/bpi/doku.php?id=2017:challenge, Dec. 2017.

[7] "Statechart Workbench and Alignments Software Event Log," https://doi.org/10.4121/uuid:7f787965-da13-4bb8-a3fd-242f08aef9c4, Aug. 2018.

[8] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in *VLDB'97*. Morgan Kaufmann, 1997, pp. 436–445. [Online]. Available: http://www.vldb.org/conf/1997/P436.PDF

[9] I. D. Automation, "Model, manage and automate repeatable business decisions across your enterprise," 2020, online; accessed 2020.

[10] "Business object model (bom) - ibm knowledge center," 2020, online; accessed 2020.

[11] "vocabulary for odm ibm knowledge center," 2020, online; accessed 2020.

[12] I. K. Center, "verbalization ibm knowledge center," 2020, online; accessed 2020.

[13] M. Schrepp, A. Hinderks, and J. Thomaschewski, "Construction of a benchmark for the user experience questionnaire (ueq)," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 4, pp. 40–44, 06 2017.

[14] B. F. Van Dongen *et al.*, "The prom framework: A new era in process mining tool support," in *International conference on application and theory of petri nets*, 2005, pp. 444–454.

[15] W. M. van der Aalst, M. Bichler, and A. Heinzl, "Robotic process automation," 2018.

[16] K. C. Moffitt, A. M. Rozario, and M. A. Vasarhelyi, "Robotic process automation for auditing," *Journal of Emerging Technologies in Accounting*, vol. 15, no. 1, pp. 1–10, 2018.

[17] D. Fernandez and A. Aman, "Impacts of robotic process automation on global accounting services," *Asian Journal of Accounting and Governance*, vol. 9, pp. 123–132, 2018.

[18] A. Stople, H. Steinsund, J. Iden, and B. Bygstad, "Lightweight it and the it function: experiences from robotic process automation in a norwegian bank," *Bibsys Open Journal Systems*, 2017.

[19] D. A. S. Rao and G. Verweij, "Sizing the prize: What's the real value of AI for your business and how can you capitalise?" *PwC Publication, PwC*, 2017.

[20] P. Nguyen, A. Slominski, V. Muthusamy, V. Ishakian, and K. Nahrstedt, "Process trace clustering: A heterogeneous information network approach," in *In SIAM SDM*, 2016.

[21] D. Breuker, M. Matzner, P. Delfmann, and J. Becker, "Comprehensible predictive models for business processes." *MIS Quarterly*, 2016.

[22] F. Mannhardt, M. De Leoni, H. A. Reijers, and W. M. Van Der Aalst, "Decision mining revisited-discovering overlapping rules," in *CAiSE*, 2016.

[23] N. Tax, I. Verenich, M. La Rosa, and M. Dumas, "Predictive business process monitoring with lstm neural networks," in *CAiSE*. Springer, 2017, pp. 477–492.

[24] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *arXiv preprint arXiv:1709.00103*, 2017.

[25] P. Utama, N. Weir, F. Basik *et al.*, "An end-to-end neural natural language interface for databases," *arXiv preprint arXiv:1804.00401*, 2018.