Ontology-based Framework for Integration of Time Series Data: Application in Predictive Analytics on Data Center Monitoring Metrics

Lauri Tuovinen and Jaakko Suutala
Biomimetics and Intelligent Systems Group, University of Oulu, Finland

Keywords: Data Integration, Data Analytics, Time Series Data, Data Center, Domain Ontology, Software Framework.

Abstract: Monitoring a large and complex system such as a data center generates many time series of metric data, which are often stored using a database system specifically designed for managing time series data. Different, possibly distributed, databases may be used to collect data representing different aspects of the system, which complicates matters when, for example, developing data analytics applications that require integrating data from two or more of these. From the developer's point of view, it would be highly convenient if all of the required data were available in a single database, but it may well be that the different databases do not even implement the same query language. To address this problem, we propose using an ontology to capture the semantic similarities among different time series database systems and to hide their syntactic differences. Alongside the ontology, we have developed a Python software framework that enables the developer to build and execute queries using classes and properties defined by the ontology. The ontology thus effectively specifies a semantic query language that can be used to retrieve data from any of the supported database systems, and the Python framework can be set up to treat the different databases as a single data store that can be queried using this semantic language. This is demonstrated by presenting an application involving predictive analytics on resource usage and electricity consumption metrics gathered from a Kubernetes cluster, stored in Prometheus and KairosDB databases, but the framework can be extended in various ways and adapted to different use cases, enabling machine learning research using distributed heterogeneous data sources.

1 INTRODUCTION

Data mining and machine learning applications often require integrating data from multiple sources. Even data of the same type and representing measurements of the same entity may be distributed across different databases, each with its own query interface and syntax. For example, in the domain of data center monitoring, metrics representing usage of computing resources may be stored in one time series database while those representing consumption of electricity are stored in another one. Integrating data from these two sources is necessary, for example, in order to develop models for load prediction, resource allocation and optimization of electricity consumption for the data center.

In a situation like this, data integration can be made more seamless by using semantic technology to build an application programming interface (API) that hides database-specific details from the data user and enables them to express queries on a higher level of abstraction, using domain concepts rather than a particular query language. We have developed such an API to support integration and analysis of time series data in Python. The API was built for an application involving metrics collected from a virtualized data center, based on the Kubernetes platform (Kubernetes, 2021), and stored in Prometheus (Prometheus, 2021) and KairosDB (KairosDB, 2021) databases, enabling the development of artificial intelligence-based sustainable and energy-efficient data centers. However, the API can be extended to work with other database systems and can also be adapted to different use cases.

The API is based on an ontology that defines a set of classes and properties representing databases, queries, metrics and data center components. The ontology thus bridges the domains of time series databases and data center monitoring. Using the ontology classes, the user of the API can compose queries without knowledge of the correct query syntax for a given database or even of which database is supplying a given metric. The ontology thus effectively specifies a semantic query language that

Tuovinen, L. and Suutala, J.
Ontology-based Framework for Integration of Time Series Data: Application in Predictive Analytics on Data Center Monitoring Metrics.
DOI: 10.5220/0010650300003064
In *Proceedings of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2021) - Volume 2: KEOD*, pages 151-161
ISBN: 978-989-758-533-3; ISSN: 2184-3228

abstracts away the differences between different database systems, which in this case are typically substantial since time series databases, unlike relational databases, do not have a unifying standard query language.

When the API is initialized, it retrieves information about the metrics supplied by each database and populates the ontology with individuals representing the available metrics. Once the initialization is complete, the user can start building queries by creating individuals of the query class and setting their ontology properties. When the user executes a query, the query individual is handed over to a query processor, which handles the process of translating the semantic query into the appropriate system-specific query language and parsing the query result into a representation format that is again independent of the format used by the database system. The user – who is a data miner or a machine learning engineer, not a data engineer – thus only needs to specify the data they wish to process, and the API will take care of the details of how to retrieve it. In the future, the API will also provide pre-processing operators that the user can use to prepare data for analysis.

The principal contributions of the paper are as follows:

- An ontology for semantic composition of database queries and other data processing operations using domain concepts;
- An extensible Python software framework enabling the execution of such operations;
- A demonstration of the applicability of the query API in the domain of data center monitoring, providing data for machine learning research in coherent manner.

In the remainder of the paper, we first discuss some essential background information and review related work in Section 2. We then present the problem addressed in the form of requirements in Section 3. The solution is described in Sections 4 and 5, which present the ontology and the software framework, respectively. Section 6 demonstrates the validity of the solution by describing an application where the API is used to solve a real-world data integration problem, and Section 7 shows how the capabilities of the API can be extended beyond the requirements of this original problem. Section 8 discusses the significance of the results, and Section 9 presents our conclusions.

2 BACKGROUND

The development of the data integration API was motivated by the observation that while querying different time series databases providing data on the same entity of interest – in our case, a Kubernetes cluster – is semantically similar, syntactically the queries may differ on a number of levels:

- **Interface:** Different database systems have different APIs via which they accept queries over HTTP.
- **Language:** Different database systems generally implement their own custom query languages.
- **Representation:** Different database systems may use different representation formats for e.g. timestamps.
- **Nomenclature:** The naming of domain concepts is not necessarily consistent across databases; for example, a virtual machine in a Kubernetes cluster might be referred to as a node in one database and as a host in another.

Our ontology was designed to hide all of these differences by establishing a level of abstraction where queries can be specified according to the same syntax regardless of which database they are targeting. The Python framework, in turn, was created in order to provide a practical implementation of the semantic query language defined by the ontology. The idea of translating queries to different platform-specific languages is similar to TSL (Hébert, 2019), but this language operates on a lower, wholly non-semantic level of abstraction. Our API could be said to represent a middle ground between this and ontology-based data access in the traditional sense, in that our ontology models both the domain of time series databases and the application domain proper (albeit on a relatively high level of abstraction).

Ontology-based data access is a relatively established idea where a semantic query expressed in SPARQL is mapped to an equivalent query in SQL, allowing the desired data to be specified in terms of domain concepts rather than database structures (Xiao et al., 2018). More recently, similar ideas have begun to be applied to time series data. SE-RRD (Zhang et al., 2016) is a semantically enhanced time series database for monitoring data, with an ontology that bears some similarity to ours but is more focused on representing the semantics of the data. SE-TSDB (Zhang et al., 2019) builds on this and the paper explicitly mentions mapped queries, but no details are provided on how the queries are prepared and executed.

Two instances of semantic time series query engines where the authors do specify the query language implemented are FOrTÉ (El Kaed and Boujonnier, 2017) and ESENTS (Hossayni et al., 2018). The former of these uses SPARQL, while the latter implements its own domain-specific query language. As far as we know, there are no examples in the literature, at least in the domain of time series data management, of a framework like ours where the ontology itself serves as the query language and queries are prepared by creating ontology individuals and setting their properties, which the framework translates into REST API requests to the databases used.

In the domain of data center management specifically, ontology-based solutions have been proposed to various problems. These include situation analysis (Deng et al., 2013), resource allocation (Metwally et al., 2015), simulation (Memari et al., 2016), container description (Boukadi et al., 2020) and power profiling (Koorapati et al., 2020). There appears to be no previous work in this particular domain where an ontology is used to facilitate data access or integration.

3 REQUIREMENTS

In the original use case for the API, monitoring data for a Kubernetes-based data center is accessible via two time series databases: Kubernetes metrics representing usage of computing resources (e.g. CPU time, memory allocation, disk I/O) are exported to a Prometheus database, while information about the electricity consumption of the data center hardware is stored in a KairosDB database. Data users can retrieve data from these databases over an HTTP connection using their REST APIs. New data entered into the databases is retained for a limited time, so long-term historical data is not available.

The data user in this use case is a machine learning engineer aiming to build predictive models for estimating, anticipating and optimizing electricity consumption in the data center. To obtain training data for the models, the user needs to be able to integrate and synchronize data from the two monitoring databases. From this use case, the following core requirements were derived:

- The user *must* be able to retrieve data for any metric supplied by either of the two databases;
- When querying for metric data, the user *must* be able to specify a time range;
- When querying for metric data, the user *must* be able to specify a Kubernetes node (or set of nodes);
- The user *must* be able to store and reuse queries for periodic retrieval of data;
- The user *must* be able to store retrieved data in a local database;
- The user *should* be able to specify multiple metrics in a single query;
- The user *should* be able to specify aggregator functions to be applied to the data;
- The user *should* be able to specify data transformation operations for further processing of retrieved datasets;
- The user *must not* be required to be familiar with database APIs, query languages or any other system-specific details;
- The user *must* be able to extend the capabilities of the API without modifying its internal logic.

These requirements are summarized in Table 1. In the table, each requirement is given a short reference number; these range from R1 to R10. Additionally, each requirement is classified as either required, meaning that it represents an essential feature, or desired, meaning that the feature is not essential but would be useful to have.

Implementing requirement R1 ensures that the user of the framework is able to retrieve all the data required for the training of the machine learning models. Implementing requirements R2 and R3 ensures that the user can exclude data that is of no interest. Implementing requirement R8 makes it more convenient to retrieve data for multiple metrics, but this can also be achieved by executing multiple queries, so the feature is considered non-essential. Implementing requirement R9 enables the user to e.g. retrieve the growth rate of a counter-type metric, which is often more useful than the raw data, but since such functions can also applied to the data locally after retrieval, this feature is also considered non-essential.

Implementing requirements R4 and R5 ensures that the user has access to a sufficient quantity of the data needed for model training and testing. Since the remote databases do not provide long-term historical data, the user needs to build such a dataset locally by retrieving periodic snapshots of the metrics used. For this purpose, the user needs to be able to reuse queries and to store their results in a local database. Requirement R10 is considered non-essential, but it would be useful if the framework would provide operators for pre-processing the retrieved data for analysis. For example, one such operator could provide the ability to stitch together the periodic snapshots to form a single contiguous time series spanning a longer time range.

Requirement R6 applies to the features represented by all of the other requirements. Implementing the requirement involves building an abstraction layer that hides the system-specific details discussed in Section 2 from the user. This ensures that the user can concentrate on developing the models, since they can access the data they need without having to specify how it is to be retrieved. Finally, requirement R7 enables the user to adapt the API to different use cases; this can always be achieved by modifying the existing code, but this must not be necessary in order to e.g. add support for a new database system. Instead, the API must provide abstract classes that the user can implement to add new capabilities in a modular fashion.

4 THE ONTOLOGY

The ontology was designed using the Protégé tool (Musen and the Protégé Team, 2015). A partial class hierarchy is shown in Figure 1. The hierarchy is organized under four principal top-level classes:

- **KubernetesObject**, representing objects related to the composition of a Kubernetes cluster;
- **MonitoringObject**, representing objects related to the collection and storage of monitoring metrics;
- **RestApiObject**, representing objects related to communicating with the REST APIs of monitoring databases;
- **AnalyticsObject**, representing objects related to local storage and analysis of retrieved metric data.

Additionally, certain subclasses of MonitoringObject and AnalyticsObject share a common superclass. **Database** is the common superclass of all classes representing databases, both those from which metric data is retrieved and those in which it is stored locally. **DataContainer** is the common superclass of all classes representing in-memory containers of metric data, both query results received from remote databases and datasets generated locally by processing query results.

Under the KubernetesObject class, the subclasses of **KubernetesExecutionUnit** are particularly important. These represent objects that are directly involved in the execution of code on the Kubernetes platform: workloads, containers, pods and nodes. These are used in queries to select data generated by a specific node, for example. Other subclasses of KubernetesObject represent Kubernetes namespaces, services and storage volumes.

Under the MonitoringObject class, **MonitoringDatabase** represents remote databases supplying metrics, which in turn are represented by **MonitoringMetric**. MonitoringDatabase has the subclasses **PrometheusDatabase** and **KairosDatabase**, representing different time series database systems, as well as **PrimaryDatabase**. The latter is used to designate a database as the one from which information about the composition of the Kubernetes cluster will be retrieved when the ontology is populated during initialization of the framework. The **DatabaseCredentials** class is used to represent the username and password to be used to access a given database.

Queries to the databases are represented as instances of **DatabaseQuery**, which has several subclasses representing different types of queries: **ClusterCompositionQuery** represents queries for information about the Kubernetes cluster, **MetadataQuery** represents queries for information about the metrics provided by a database, and **MetricValuesQuery** represents queries for metric data. More specific subclasses of these classes are not visible in the figure, but notably, a MetricValuesQuery can be either an **InstantQuery**, which returns data for a given point in time, or a **RangeQuery**, which returns data for a time range. A query may also be a compound query, i.e. consist of multiple subqueries, in which case each subquery is represented by its own DatabaseQuery instance.

A query is built by setting the properties of the DatabaseQuery instance. Some of these are data properties, such as the time instant of an instant query or the range start and end times of a range query. Others, such as the metric to be retrieved and the Kubernetes nodes targeted, are specified as object properties. Aggregator functions (e.g. rate, sum, avg) can be invoked by attaching instances of the corresponding **QueryAggregator** subclasses (not shown) to the query.

When a query is ready for execution, it is handed over to an instance of the RestApiObject subclass **QueryProcessor**. The query processor analyzes the properties of the DatabaseQuery instance and generates a query in the appropriate query language such as PromQL. The generated query is then sent over HTTP to the appropriate API endpoint using the appropriate request method, represented by the classes **RestEndpoint** and **HttpRequestMethod**. Each query type may have a different API endpoint associated with it, represented by specific subclasses of RestEndPoint (not shown).

Details about the query execution are recorded in an instance of the **QueryExecution** class, facilitating reuse of previously created queries. When the

Table 1: Required and desired features of the data integration API.

ID	Requirement	Type
R1	The user can retrieve data for any metric	Required
R2	The user can specify a time range for queries	Required
R3	The user can specify a set of nodes as the query target	Required
R4	The user can store and reuse queries	Required
R5	The user can store retrieved data locally	Required
R6	The user does not need to be familiar with database APIs	Required
R7	The user can extend the capabilities of the API	Required
R8	The user can specify multiple metrics in a single query	Desired
R9	The user can invoke aggregator functions in queries	Desired
R10	The user can specify data transformation operations	Desired

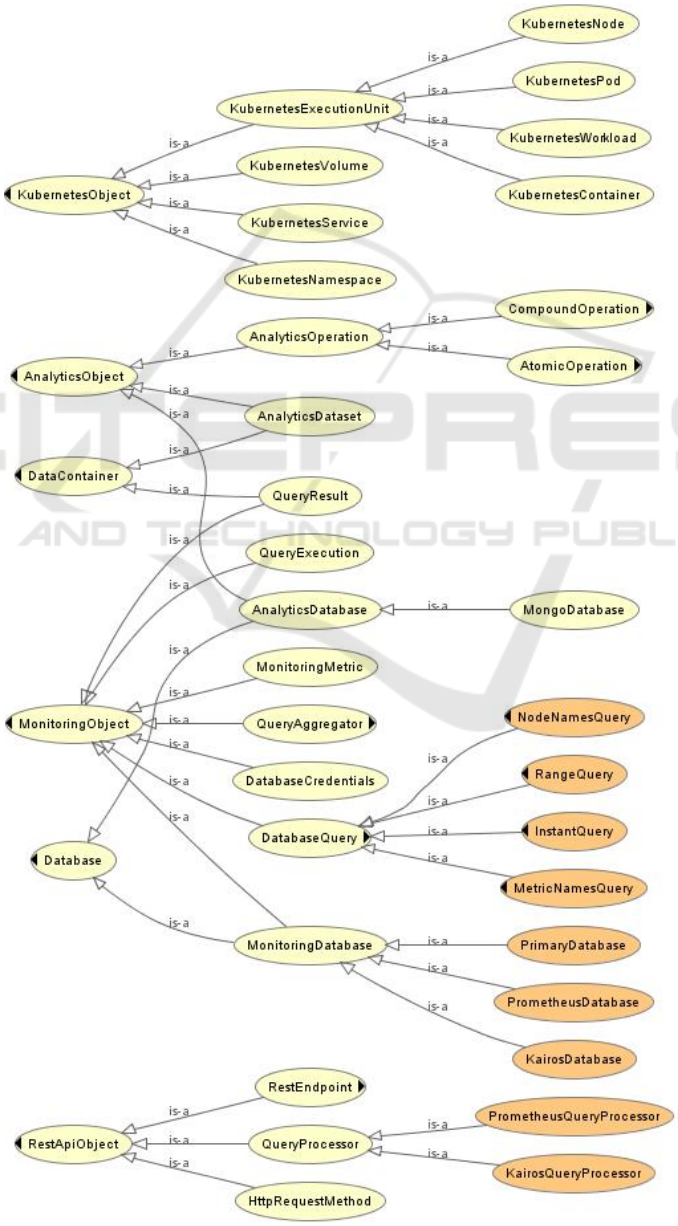

Figure 1: Class hierarchy of the ontology. The top-level classes in the figure have owl:Thing as their immediate superclass.

query processor receives a response from the database REST API, it extracts the data and creates an instance of **QueryResult** representing the response. This is then associated with the QueryExecution instance. The returned data can be stored in the QueryResult instance itself or in a local database, represented by the **AnalyticsDatabase** class. The framework currently supports the use of MongoDB as the local database backend, represented by the **MongoDatabase** subclass.

Locally generated datasets are represented by the **AnalyticsDataset** class, which, like QueryResult, is a subclass of DataContainer. Analytics datasets are derived from other data containers using data transformation operations, represented by the **AnalyticsOperation** class. These are divided into atomic operations and compound operations, as shown in the figure; more specific subclasses of these (of which there are only a few in the current version of the ontology) are not shown.

5 THE FRAMEWORK

The overall architecture of the data integration and analysis system is shown in Figure 2. The Python framework consists of two modules, labeled in the figure as Interface Module and Ontology Module. The ontology is loaded from an OWL file using the Owlready2 Python package (Lamy, 2017). Other key packages on which the framework depends include Requests, which is used to communicate with the REST APIs of the remote databases, and PyMongo, which is used to access the local MongoDB database.

The Owlready2 package treats ontology classes as Python classes, ontology individuals as instances of these classes, and ontology properties as attributes of these instances. Python code can be added to the classes by defining class methods. The package also enables a Python module to be executed automatically when an ontology is loaded by specifying the name of the module as the value of an annotation property. This feature of Owlready2 is used by our API to load the Ontology Module, which defines the following:

- Constants for different query types, HTTP request types, database types and data representation formats;
- In the class DataContainer, methods for manipulating the contents of the container;
- In the class QueryProcessor, abstract methods for formatting queries and parsing query results;
- Two subclasses implementing the QueryProcessor interface, **PrometheusProcessor** and **KairosDBProcessor**;
- Single instances of PrometheusProcessor and KairosDBProcessor, which are added to the ontology as individuals;
- In the class AnalyticsDatabase, abstract methods for opening and closing a database connection and for reading and writing data;
- One subclass implementing the AnalyticsDatabase interface, MongoDatabase.

The Interface Module defines a single class, **KubernetesOntology**, which provides the methods by which the user of the API can build and execute queries on the remote databases. The class constructor takes as argument an IRI string pointing to the location of the OWL file containing the ontology. This is then loaded into memory using Owlready2, which triggers the execution of the Ontology Module as described above.

The KubernetesOntology class provides methods for looking up and creating instances of ontology classes needed in queries. The most important method is *create_query()*, which is used to create a new instance of the DatabaseQuery class. The query is built by setting the data and object properties of this instance; ontology individuals to be used as object property values can be looked up with the general-purpose method *get_individual()* or with more specific convenience methods such as *get_metric()* and *get_node()*. Invocations of aggregator functions can be instantiated using the *create_invocation()* method.

When the properties of the query have been set, the *execute_query()* method is used to run it. The method starts by checking if the query is a compound query; if it is, the method is run recursively for each subquery. If the query is not a compound query, the method first finds the target database of the query, which may be either asserted explicitly or, in the case of metric value queries, inferred from the target metric. From this information, the method then infers the query processor that will handle the query, as well as the endpoint URL, HTTP request method and authentication credentials that will be used to send the query to the remote database.

To obtain the query data in a format that can be sent to the database REST API, the method calls the *format_query()* method of the query processor object. After this, the private method *__send_query()* is called, which in turn calls the appropriate Requests method (get() or post()) and returns the response received from the REST API. The *parse_result()* method of the query processor is then used to extract the relevant data from the response and to create a QueryResult

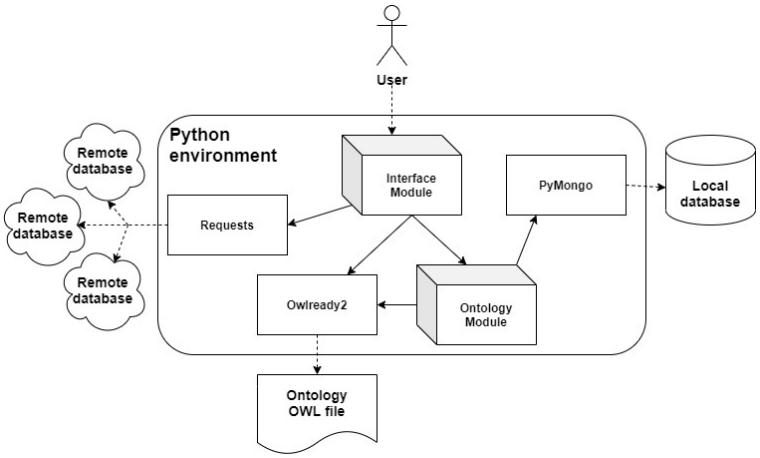

Figure 2: An architectural diagram of the data integration framework and its dependencies. Within the Python environment, 3D boxes denote modules that are part of the framework, while 2D boxes denote other key Python packages used in the implementation of the framework. Solid arrows denote internal dependencies, and dashed arrows represent external dependencies.

instance as a container for this data. The above sequence is illustrated in Figure 3.

The data contained by the query result can be accessed using the *get_content()* and *get_metadata()* methods defined for the DataContainer class (which is a superclass of QueryResult) in the Ontology Module. In the case of metric value queries, the convenience methods *get_values()* and *get_times()* can be used to get the metric values and the corresponding timestamps, respectively. The convenience methods provide an optional parameter that can be used to specify the representation format of the returned data; the timestamps, for example, can be returned as UNIX timestamps (the default), Python datetime objects or ISO-formatted datetime strings.

The data inside data containers can be stored in the container objects themselves, in which case it is represented as JSON-formatted strings and there is no need for an external local database. However, this is a rather cumbersome and inelegant way to store the data, so the Python framework provides the option to specify an instance of AnalyticsDatabase as the local storage backend, in which case data containers will use it to store their data persistently. Once the backend has been set up, everything else works exactly the same way from the user's perspective, because the implementation details of data storage and retrieval are hidden by the DataContainer interface.

6 USING THE API

In our data center monitoring use case, the API is initialized using the following script:

```
1 kube_onto = KubernetesOntology(ONTO_FILE)
2 db_mongo = kube_onto.create_backend(...)
3 db_mongo.open_connection()
4 kube_onto.set_default_backend(db_mongo)
5 db_prometh = kube_onto.create_database(...)
6 db_kairos = kube_onto.create_database(...)
7 kube_onto.get_available_metrics()
8 kube_onto.get_cluster_nodes()
```

On line 1, the main API class is instantiated. On line 2, a new AnalyticsDatabase instance representing a locally installed MongoDB server is created; the arguments passed to the creation method are not shown, but they consist of a unique name by which the database can be referred to, the type of the database and all the information necessary to establish a connection to the database (e.g. host address and port). A connection to the MongoDB server is then opened (line 3) and the database is registered as the default backend for storing data container contents (line 4).

On lines 5–6, two MonitoringDatabase instances representing the remote Prometheus and KairosDB databases are created. The arguments passed to this creation method again include a name for the database, database type and the information required to connect to it. Additionally, there is an optional boolean argument that, if passed and if true, designates the database as the primary database that will provide information about the composition of the Kubernetes cluster. In this case, the first database created (*db_prometh*) is the primary one.

Finally, on lines 7–8, information about the metrics supplied by the databases and about the nodes in the Kubernetes cluster is retrieved and added to the ontology as MonitoringMetric and KubernetesNode instances. Information about metrics is retrieved from both databases, information about nodes from the primary database. To avoid repeating this process later,

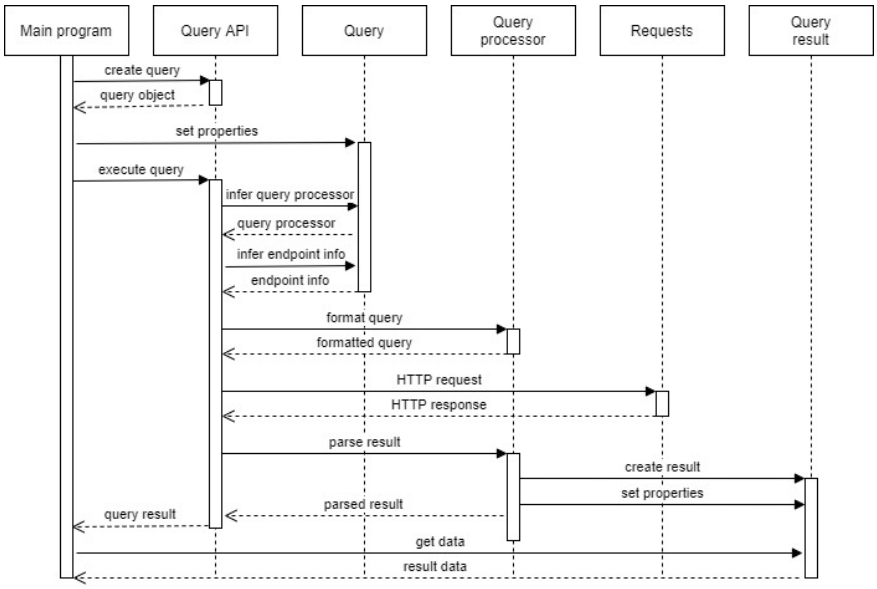

Figure 3: A sequence diagram using adapted UML notation, showing how a query is created and processed using the API.

the modified ontology can be stored persistently using the *save_ontology()* method of the KubernetesOntology class.

With the API thus initialized, the KubernetesOntology class instance can be used to run queries on the remote databases. In the example below, a compound query consisting of two subqueries is used to retrieve CPU usage (supplied by Prometheus) and electric power (supplied by KairosDB) for a single node in the Kubernetes cluster:

```
1  q = kube_onto.create_query(QUERY_TYPE_RANGE)
2  q.hasTargetUnit = kube_onto.get_nodes(
     ["a04r01srv07"])
3  q.hasRangeStartTime = "2021-02-01T12:00:00"
4  q.hasRangeEndTime = "2021-02-01T18:00:00"
5  q.hasResolutionStep = "1m"
6  sq1 = kube_onto.create_query()
7  sq1.isForMetric = kube_onto.get_metric(
     "Prometheus",
     "node_cpu_usage_seconds_total")
8  sq1.invokesAggregator = [
     kube_onto.create_invocation(
     "RateFunction",
     {"sampling": "5m"})]
9  sq2 = kube_onto.create_query()
10 sq2.isForMetric = kube_onto.get_metric(
    "KairosDB", "power")
11 q.hasComponentQuery = [sq1, sq2]
12 q.isCompoundQuery = True
13 result = kube_onto.execute_query(q)
```

On lines 1–5, the main query *q* is created and its properties are set. The type of the query is passed as argument to the query creation method; the constant *QUERY_TYPE_RANGE* is defined in the Ontology Module. The target node is an object property, so its value must be an ontology individual; this is retrieved from the ontology using the *get_nodes()* method, which returns a list of KubernetesNode instances based on their names. The range start and end times and the resolution step of the query are data properties. These master properties are applied to each subquery, so for the subqueries it is only necessary to set the properties that are not the same for all subqueries.

Subquery *sq1* (lines 6–8) retrieves the CPU usage. The target metric is again an object property, and its value is retrieved from the ontology based on the name of the supplying database (given when the database was registered during initialization) and the name of the metric using the *get_metric()* method. This is done similarly for subquery *sq2* (lines 9–10), which retrieves the power. Subquery sq1 additionally applies a rate aggregator to transform the values of the CPU usage seconds counter into a more useful representation of CPU usage; this is done by calling the *create_invocation()* method, which takes as arguments the name of the aggregator function to be invoked and a dictionary containing the invocation parameters. The return value of this method is then associated with the subquery via another object property.

On lines 11–13, the two subqueries are added as components of the main query and the query is executed. The variable *result* will now contain the query result, which is an instance of the QueryResult class. Associated with this object will be two more QueryResult instances, one for each subquery, which can be accessed via an object property. These, in turn, provide access to the data returned by the subqueries,

which the user can retrieve using the methods of the DataContainer class. These use the methods of the MongoDatabase class to retrieve the data from the local MongoDB server, but the user does not need to be aware of this.

Figure 4 depicts a machine learning application that uses the API to retrieve metric data on resource usage and electricity consumption, exported into two separate databases from a Kubernetes cluster (1). The load forecasting component uses the resource usage data to generate predictions of future load (2), and these are then combined with the electricity consumption data by the resource allocation control component (3). This outputs resource management strategies that are handed over to the Kubernetes cluster and used to determine optimal times for starting up and shutting down virtual machines (4).

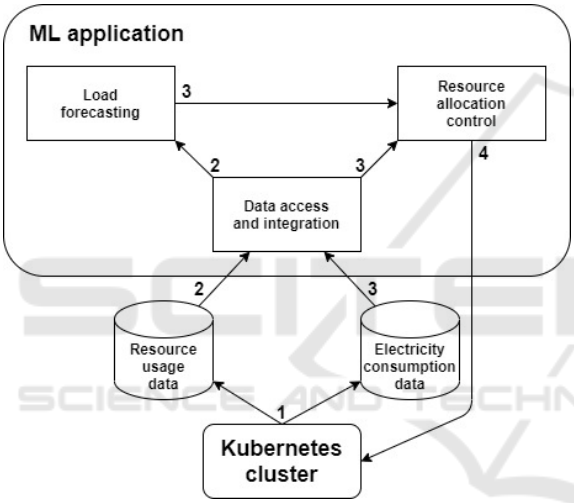

Figure 4: An overview of a machine learning application designed to optimize resource management in a Kubernetes cluster. The application uses the ontology-based API (labeled 'Data access and integration') to retrieve input data from multiple databases.

7 EXTENDING THE API

The API includes several abstract classes designed to be subclassed by the user to extend the capabilities of the API. The most important one of these is the QueryProcessor class. As discussed in Section 5, the API depends on implementations of the QueryProcessor interface to translate between semantic (system-independent) and system-specific representations of queries and query results; each database registered using the create_database() method of KubernetesOntology must have a query processor associated with it before queries can be sent to the database. By default, the create_database() method makes this association automatically by searching the ontology for a QueryProcessor individual whose *hasDatabaseType* property value matches the type of the database.

Implementations of the QueryProcessor interface are required to implement two methods, format_query() and parse_result(). The format_query() method takes as argument the query to be formatted (a DatabaseQuery instance), and additionally the HTTP request method to be used to send the query, since this may affect the formatting. The return value is a dictionary or string that can be passed as an argument to an invocation of Requests.get() or Requests.post(). The parse_result() method takes as argument the HTTP response received, and additionally the type of the query, which affects the parsing. The return value is a QueryResult instance representing the result.

Another abstract class that the user of the API can derive subclasses from is the AnalyticsDatabase class. By creating a new implementation of the AnalyticsDatabase interface, the user can use a database system other than MongoDB as the local storage backend for data containers. Implementations of the AnalyticsDatabase inteface must implement the methods *set_connection_params()*, *open_connection()* and *close_connection()* for connection management, and the methods *write_content()*, *read_content()*, *write_metadata()* and *read_metadata()* for data manipulation.

Users of the API are also intended to be able to create their own subclasses of the AnalyticsOperator class. This aspect of the API is currently under construction, but the general idea of the operators is that they accept one or more data containers as inputs, process their contents and generate one or more data containers as outputs. Implementing these operators as ontology classes enables them to be stored in the ontology and reused in a similar way that database queries can be stored and reused.

To adapt the API to a different application domain, it is first necessary to model the domain semantics as ontology classes and properties. Query processors capable of interpreting these semantics can then be implemented. The existing query processor classes cannot be subclassed for this purpose, but if the same database systems are being used, much of the code of the existing query processors can still be reused.

8 DISCUSSION

The query mechanism of the API satisfies the essential requirements R1–R3, as well as the non-essential requirements R8 and R9. The time range of the query (R2) is specified via data properties of the Database-

Query object representing the query, and the target metric (R1), target nodes (R3) and possible aggregator invocations (R9) via object properties. Multiple target metrics (R8) can be specified by creating a compound query where each of the desired metrics is retrieved by a distinct subquery.

Something worth noting about the query API here is that it is an abstraction of concepts and features that are found in both Prometheus and KairosDB. This means that if a given feature is found in one of the supported database systems but not the other, it is not supported by the API, because if it were, this would involve exposing system-specific implementation details to the user, which we have specifically aimed to avoid. On the other hand, this functionality trade-off means that the API can do something that none of the APIs of the supported database systems can, namely run queries that retrieve data from several different databases as if they formed a single data store with a unified interface.

Reuse of queries (R4) is enabled by the storage of queries as individuals in the ontology. An identifying name can be assigned to each query in the form of a data property value, enabling the user to retrieve the query later. Local storage of retrieved data (R5) is possible directly in the ontology or, using the MongoDatabase implementation of the AnalyticsDatabase interface, on a MongoDB server. By default, the MongoDB server is assumed to be running on localhost, but the MongoDatabase class can also be configured to connect to a remote host.

Requirement R6 is also satisfied, since the queries are expressed using classes and properties defined in the ontology rather than any system-specific query language. To set up the MonitoringDatabase objects, the user needs to know the base URLs of the REST APIs of the databases used and the username and password to be used to access each database. Additionally, the user needs to designate one of the databases as the primary database. The Python framework sets up the API endpoints for different query types and associates the appropriate query processor with each database. Some trade-offs are made here between supporting a specific application domain and making the API as generic as possible, and this is something we intend to address in future work in order to improve the adaptability of the API, with the aim to release it to the community as open-source software.

Requirement R10 is taken into account in the design of the ontology with the AnalyticsOperation class and its subclasses, but the Python framework does not currently implement any logic for these classes. This is another aspect of the API that we are planning to address in future work. The API should provide data transformation operators with wide applicability, such as atomic operators for common pre-processing tasks and a compound operator for chaining multiple atomic operators to be applied as a pipeline.

An example of a useful real-world application that can be implemented using the API is given in Section 6. Although this is just an example, it is worth noting that the API has been designed specifically with the development of machine learning and data mining applications in mind. With tight integration between the ontology and the Python language enabled by the Owlready2 package, the API provides an intuitive syntax for data access where the application developer can specify queries using Python object representations of query targets as described above. The query results are represented in a format that preserves their semantics while making it convenient for the developer to process the retrieved data using the wide array of data analytics libraries available for Python.

This approach distinguishes our API from the systems reviewed in Section 2, which place more emphasis on modeling the semantics of the data in the databases on a detailed level. We have opted to model the domain on a higher level of abstraction, resulting in a lightweight ontology that can be populated automatically with information such as metric and node names and adapted with relatively little effort to support different applications. Building additional levels of detail into the ontology might cost us some of these benefits but bring others, such as more potential for making use of automated reasoning. Therefore another question we intend to explore in future work is what is the most appropriate level of semantic modeling for the purposes of our API.

9 CONCLUSIONS

In this paper we presented a domain ontology and Python software framework designed for accessing and integrating time series data stored in remote databases. The ontology defines a language for specifying queries in terms of system-independent concepts instead of system-specific query language constructs, and the Python framework provides an engine for translating queries and query results between semantic representations and system-specific formats. The ontology and framework together form a time series database API in which queries are built in a modular fashion from Python object representations of ontology elements.

The original motivation for the API is an application where monitoring metrics collected from a Kubernetes-based data center are integrated from two different time series database systems for the purpose of developing predictive models and optimization algorithms. However, the API is designed to be adaptable to different use cases, and users can extend its capabilities by writing new implementations of abstract classes provided for this purpose. In future work we aim to further improve the adaptability of the API and to extend it with data transformation operators that make it more convenient for users to pre-process retrieved data for analysis.

ACKNOWLEDGEMENTS

The work presented in this paper was carried out as part of the ArctiqDC project, with financial support from the European Regional Development Fund via the Interreg Nord program. We would also like to thank Daniel Olsson of RISE Research Institutes of Sweden for providing access to the databases used.

REFERENCES

Boukadi, K., Rekik, M., Bernabe, J. B., and Lloret, J. (2020). Container description ontology for CaaS. *International Journal of Web and Grid Services*, 16(4):341–363.

Deng, Y., Sarkar, R., Ramasamy, H., Hosn, R., and Mahindru, R. (2013). An ontology-based framework for model-driven analysis of situations in data centers. In *2013 IEEE International Conference on Services Computing*, pages 288–295.

El Kaed, C. and Boujonnier, M. (2017). FOrTÉ: A federated ontology and timeseries query engine. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 983–990.

Hébert, A. (2019). TSL: a developer-friendly time series query language for all our metrics. URL: https://www.ovh.com/blog/tsl-a-developer-friendly-time-series-query-language-for-all-our-metrics/, accessed 9 May, 2021.

Hossayni, H., Khan, I., and El Kaed, C. (2018). Embedded semantic engine for numerical time series data. In *2018 Global Internet of Things Summit (GIoTS)*, pages 1–6.

KairosDB (2021). KairosDB documentation v1.2.0 — KairosDB 1.0.1 documentation. URL: http://kairosdb.github.io/docs/build/html/index.html, accessed 9 May, 2021.

Koorapati, K., Rubini, P., Ramesh, P. K., and Veeraswamy, S. (2020). Ontology based power profiling for internet of things deployed with software defined data center. *Journal of Computational and Theoretical Nanoscience*, 17(1):479–487.

Kubernetes (2021). What is Kubernetes? URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, accessed 9 May, 2021.

Lamy, J.-B. (2017). Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. *Artificial Intelligence in Medicine*, 80:11–28.

Memari, A., Vornberger, J., Gómez, J. M., and Nebel, W. (2016). A data center simulation framework based on an ontological foundation. In Marx Gomez, J., Sonnenschein, M., Vogel, U., Winter, A., Rapp, B., and Giesen, N., editors, *Advances and New Trends in Environmental and Energy Informatics: Selected and Extended Contributions from the 28th International Conference on Informatics for Environmental Protection*, Progress in IS, pages 39–57. Springer International Publishing.

Metwally, K. M., Jarray, A., and Karmouch, A. (2015). Two-phase ontology-based resource allocation approach for IaaS cloud service. In *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, pages 790–795.

Musen, M. A. and the Protégé Team (2015). The Protégé project: A look back and a look forward. *AI Matters*, 1(4):4–12.

Prometheus (2021). Overview | Prometheus. URL: https://prometheus.io/docs/introduction/overview/, accessed 9 May, 2021.

Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., and Zakharyaschev, M. (2018). Ontology-based data access: A survey. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 5511–5519.

Zhang, S., Yen, I., and Bastani, F. B. (2016). Toward semantic enhancement of monitoring data repository. In *2016 IEEE Tenth International Conference on Semantic Computing (ICSC)*, pages 140–147.

Zhang, S., Zeng, W., Yen, I., and Bastani, F. B. (2019). Semantically enhanced time series databases in IoT-edge-cloud infrastructure. In *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, pages 25–32.