Semantic Web 0 (2020) 1–0
IOS Press


https://www.overleaf.com/project/5edf753a37d16c0001f52ec8

# Beyond Querying: A Scalable Architecture for Linked Data Monitoring

Burak Yönyül [a,*], Oylum Alatlı [a], Rıza Cenk Erdur [a] and Oğuz Dikenelli [a]
[a] *Department of Computer Engineering, Ege University, 35100, Bornova, İzmir, Turkey*
*E-mails: burak.yonyul@ege.edu.tr, oylum.alatli@ege.edu.tr, cenk.erdur@ege.edu.tr, oguz.dikenelli@ege.edu.tr*

**Abstract.** Monitoring the data sources for possible changes is an important consumption requirement for applications running in interaction with the web of data. In this paper, MonARCh (Monitoring Architecture for Result Changes) which is a scalable architecture for monitoring the result changes of registered SPARQL queries in the linked data environment has been introduced. Although MonARCh can be comprehended as a publish/subscribe system in the general sense, it differs in how the communication with the data sources are realized. The reason behind this is that the data sources in the linked data environment do not publish the changes on the data. MonARCh provides the necessary communication infrastructure between the data sources and the consumers for the notification of changes. Users subscribe SPARQL queries to the system which are then converted to federated queries. MonARCh periodically checks for updates by re-executing sub queries and notifies users in case of any result change. In addition, to provide scalability MonARCh takes the advantage of concurrent computation of the actor model and a parallel join algorithm for faster query execution and result generation. The design science methodology has been used both during the design and implementation stage and for the evaluation of the architecture.

Keywords: SPARQL, query, monitoring, linked data, scalability, publish/subscribe, pull/push, data set

## 1. Introduction

Linked data concept has been first introduced by Tim Berners-Lee in his technical note [9] where he manifested the basic principles for making the data published on the web to become part of a global information space which is usually referred to as "Web of Data" [12]. Since then the Web of Data has gradually expanded as organizations from different domains have begun to publish their data following the linked data principles. Today, the Web of Data contains a large number of interrelated data sets covering specific domains such as government, media, entertainment, life sciences, geographical places as well as data sets which span over multiple domains such as DBpedia [1, 6, 13, 33] which is all referred as Linked Open Data (LOD) Cloud [2].

Querying alone might not satisfy all of the data consumption requirements of linked data applications. It has become a critical task for applications to monitor changes on the application related part of the Web of Data and react to asynchronous events caused by these changes. Thus, management of data dynamics is an important issue for the continuously expanding and dynamically changing web of data. This requirement is also addressed in a comprehensive survey where the requirements for the linked data consumption process have been identified [30]. In particular, in the survey the authors define this requirement as the ability to periodically monitor a data source and trigger actions when the data source changes (please see requirement 15 in [30]).

In this paper, **MonARCh:** ***Mon****itoring* ***A****rchitecture for* ***R****esult* ***Ch****anges* which is a linked data monitoring architecture have been presented. Since monitoring requests can come from a large number of asynchronous applications, scalability is an important issue in developing such an architecture. MonARCh has been developed based on the actor model to provide scalability. The basic entities to be monitored and the underlying communication mechnanism of the monitoring architecture are the other concerns that constitute the

*Corresponding author. E-mail: burak.yonyul@ege.edu.tr.

linked data perspective in developing such an architecture. In MonARCh, SPARQL [54] queries are the basic entities that are monitored. The underlying communication mechanism is based on a combined pull-push approach. Monitoring infrastructures are usually based on either pull or the push approach. In the pull approach, the consumer monitors the datasets by means of periodic queries. In the push mechanism, the consumer subscribes to the data publisher and then the data publisher acknowledges the changes to subscribers. On the other hand, a combined push-pull mechanism has been employed during development of MonARCh. The pull approach has been used in the "Web of Data interface" and the push approach has been used in the "user applications interface". Using the pull approach, all new datasets are dynamically discovered each time before querying. Applications which need to monitor some specific part of the Web of Data register their requests through SPARQL queries to the monitoring service and then begin to wait for being notified in case of a change. Using combined pull-push approach in this manner constitutes one of the novel sides of the proposed monitoring service.

The monitoring architecture also has the ability of separating a complex query into its sub queries, and executing each one of these sub queries independently based on the change frequencies of their related data sets. The partial results may come either from a newly executed sub query or from the local partial result caches of previously executed sub queries, and are merged using a hash join based algorithm. This capability constitutes the other novel side of the proposed monitoring architecture.

The survey about linked data consumption also introduces and evaluates sixteen tools in the context of linked data consumption [4]. But, none of the tools evaluated addresses the linked data monitoring requirement (please see table 3 and 4 in [4]). As far as is known, MonARCh is the first architecture that addresses linked data monitoring requirement. Actor based scalability, employing combined pull-push approach in the underlying architecture, and the ability to monitor sub queries independently based on the frequency change data sets are the prominent features of MonARCh.

MonARCh has been developed following the design science methodology in [53]. A case study that uses DBpedia, New York Times and a Stock market datasets has been developed. MonARCh has been evaluated also based on the design science methodology. Rest of the paper is organized as follows. In section 2 monitoring requirement for SPARQL queries in linked data environment which constitutes the core idea of this paper has been defined. While Section 3 explains the research methodology used in this work, Section 4 gives an outline about defining the research purpose and asks some questions about the relationship between system and environment. Section 5 and Section 6 elaborate the design and evaluation details of MonARCh respectively. Lastly, Section 7 gives the literature review related to paper and Section 8 summarizes the whole work and proposes some possible improvements.

## 2. Roots of the Linked Data Monitoring Architecture

With the increase of data intensive applications, systems such as services and APIs which open its data for access have become an indispensable part of the information era. Both producers and consumers of this open data may have different requirements. According to the type of information requirement, different techniques have emerged to provide the data flow among producers and consumers. For example in the pull model, consumers query the data sources and acquire information themselves [11, 14, 28]. In the push model, consumers want to get informed when there is a change in the data that they are interested in [11, 14, 28]. The two models can be combined to form a pull-push model as well [28]. Push model comes with a solution which is called publish/subscribe architecture [20]. Publishers propagate their new data and they do not know how it will be used by the consumers (subscribers in this context). Subscribers register to some topics or to some contents and wait for being informed about new data about their interest. A middleware (matching) layer which is the key component of a publish/subscribe (pub/sub) system, seeks for a match between the data sent by publisher and topics/contents registered by subscriber. It can be seen as a bridge and its primary goal is to disseminate the relevant data from publisher to the subscriber.

All of the publish subscribe systems are useful when the data and stores are owned by the system itself which enables alerting the subscribers immediately whenever newly updated data comes. It is not always reasonable to own the data source & service and expect it to publish the updates on its data. The concept of publisher can be seen as a service that proactively disseminates its content to the outside world. This causes another problem which is filtering the relevant contents among all other irrelevant ones for the purpose of delivering to the receiver. Furthermore, a misworking matching mechanism may lead to skipping some relevant data. Most of the data sources have such a reactive structure which they issue their data when requested. Also when there is no broker who is aware of the updates of a data source, it is not possible to find a match between published and subscribed content, and also inform consumers just in time. Thus, an alternative informative mechanism is required to be built in order to deliver updated content. In this case an approach aiming to find changes by querying data sources periodically seems to be a good solution. It is similar to publish/-subscribe architecture but just differentiate on replacing publisher with pull component while embracing subscriber itself. From another point of view this can be seen as a pull/push approach which consequently checks the changes by pulling data and pushing these changes to the subscribers.

The semantic web[10] is a web of data that is readable and processable by both humans and machines. Linked Data[9] resides at the heart of the semantic web which suggests building relationships between the data sources to make the web of data interconnected and more accessible. According to the nature of linked data, data sources are connected to each other with the principle of not to duplicate but to reuse existing ones. Consequently, the semantic web is an environment of humans, agents, interlinked disjoint data sources, and SPARQL endpoints that interact with each other. Changes to the data sources can be described in a predictive way using metadata such as DaDy[24] which provides regularity and information about frequency of updates. In MonARCh, since it is not obligatory for SPARQL endpoints to be publishers, we assume to have no publishers but only subscribers who register to the result changes for SPARQL queries in the linked data environment. DaDy metadata has been applied to the VoID[5] documents of datasets in order to know how often a dataset changes. As previously described, because of the characteristic of semantic web, pull/-push approach has been used in this study and different queries with the same pattern are repeatedly sent to the endpoints of datasets according to the change frequency metadata residing in the VoID documents. If any change has been detected between the previous and current result of a query then changed result is issued to the subscribers.

Since the pull/push approach is decided to be used, a strong messaging mechanism is needed to be built as the backbone of the architecture. On the other hand a large number of queries will be monitored for a result change, therefore the architecture of the system is also required to be scalable, distributed and concurrent. From the programming point of view a thread based approach will perfectly fit as a remedy for this requirement. Under the light of these statements and facts, actor model [26, 27] which deals with concurrent computation comes as the ultimate solution. Actor is the primitive unit of computation that accepts messages through its mailbox and reactively does some computation according to the type of message they receive. Actors are also designed to cooperate with each other by sending messages. Actor model supports fault tolerance by “let it crash” philosophy, which proposes that instead of avoiding every single fault condition, failures should be left to the supervisor who knows how to heal them. With the aforementioned and many other useful features, actor model is a better and model-based replacement of thread programming. Thus, the system has been decided to be built on top of the actor model. Scala[1] is used as the implementation language and Akka[2] toolkit [23] is selected as the actor model implementation on JVM (Java Virtual Machine). Thanks to the cluster sharding property of Akka, the system has become highly scalable and concurrent. Moreover it has high performance and is also resilient to the burst of message workloads which can go up to thousands of CPU bound actors and tens of thousands msg/sec per computer cluster node. Consequently, the proposed system MonARCh is a scalable actor based result change detection and notification tool for SPARQL queries, which is based on the pull/push mechanism differs in some ways from publish/subscribe architecture and works in the linked data environment.

## 3. Research Methodology

The main motivations for this research are to enhance the response ability of applications to changes in the Linked Data and to isolate developers from the details of Linked Data monitoring. These motivations call for the design of an artifact that satisfies these requirements. Therefore, the conducted study can be identified as a design science research and follows the Design Science Methodology in [53]. This methodology consists of four main steps:

[1] https://www.scala-lang.org
[2] https://akka.io

I. Defining the design problem and knowledge questions to be answered by the research.
II. Design cycle which consists of problem investigation, designing an artifact and defining the context it will operate in (artifact and its context are defined as a treatment [53]) and the validation of the treatment defined.
III. Defining the conceptual framework the research is based on and the theoretical generalizations that can be concluded at the end of the research.
IV. Empirical cycle which aims testing the treatment with empirical methods with the following steps:

- problem analysis
- research setup design
- inference design
- validation of inferences against research setup
- research design
- research execution
- data analysis

According to [53] there are four empirical research methods that can be utilized by the empirical cycle. These methods are observational case studies, single-case mechanism experiments, technical action research and statistical difference making experiments. Since the aim of this research is to validate the abstract architecture designed for monitoring linked data changes, single-case mechanism experiments (SCME) method is the most suitable one.

Single-case mechanism experiments (SCME) are conducted with an object of study (OoS), which is a prototype model of the designed artifact placed in a model of the real world context it will operate in. SCME are done in the lab so as to allow the researcher having access to the architecture of OoS, explaining its behaviour in terms of the architecture and validating the artifact design. These explanations are expected to be generalized to the population of all possible implementations of the artifact design and they will be validated by analogical reasoning.

For the conducted study, design science methodology is used for both system design and implementation processes namely design cycle and evaluation design, execution and analysis processes which constitute the empirical cycle. But first, design problem and knowledge questions will be presented in the next section which is the heart of the methodology.

## 4. Design Problem and Knowledge Questions

Before starting a design science research, the design problem which defines the research purpose should be defined. A template for defining design problems in [53] is as follows:

- improve <a problem context>
- by <(re)designing an artifact>
- that satisfies <some requirements>
- in order to <help stakeholders achieve some goals>

When this template is applied to the linked data monitoring problem, the following design problem definition will be formed:

- improve linked data change monitoring capability of applications
- by designing an industrial-scale linked data query monitoring architecture
- that satisfies following requirements:
  * The results of federated SPARQL queries are monitored
  * Different client applications can register new federated SPARQL queries for monitoring
  * Client applications are notified when the result of a query changes
- in order to:
  * Enhance the response ability of client applications to data changes
  * Isolate developers from the details of linked data monitoring

After this step, knowledge questions should be defined. These questions aid in exploring the artifact to be designed in terms of its context and its relationship between the context. There are two kinds of knowledge questions: descriptive and explanatory. Descriptive questions look for answers to what, where, who, when, which by only observation. On the other hand explanatory questions try to get answers to why to learn about causes.

From the linked data and query monitoring perspective, the following descriptive and explanatory knowledge questions are defined for the linked data monitoring artifact:

### 4.1. Descriptive Questions

- Were the expected number of notifications gotten?

- What is the maximum number of observed federated queries?
- How many queries per unit-of-time could be successfully registered?
- How would the performance of the system be affected if the number of subqueries in a federated query increases?
- What are the pros and cons of the join algorithm used in the system and context?

### *4.2. Explanatory Questions*

- What mechanism causes not being able to monitor more queries?
- Why can't more queries be registered per unit of time?
- How do the number of subqueries in a federated query affects the performance of the system?
- How would a change in the query characteristics affect the performance of the system?

A more specific classification of knowledge questions closely related with design science research can be made by exploring the artifact-context relationship. Thus, there are four kinds of questions which can be classified as effect, trade-off, sensitivity and requirements satisfaction questions.

### *4.3. Effect Questions (Artifact X Context)*

Effect questions explore effects produced by the interaction of an artifact with the context. The context consists of a linked dataspace, SPARQL endpoints, DaDy definitions, federated SPARQL endpoints and software developer(s). Effect questions for this interaction are as follows:

- Were the expected number of notifications gotten?
- What is the maximum number of observed federated queries?
- How many queries per unit-of-time could be successfully registered?
- What mechanism causes not being able to monitor more queries?
- Why can't more queries be registered per unit of time?
- How do the number of subqueries in a federated query affects the performance of the system?
- How would a change in the query characteristics affect the performance of the system?

### *4.4. Trade-off Questions (Alternative Artifact X Context)*

Trade off questions search for the phenomena that will happen if alternative artifacts interact with the same context. Trade off questions for alternative linked data monitoring artifacts can be defined as follows:

- How would the artifact be affected if a different join algorithm was used?
- How would the artifact be affected if the actor model was not used?

### *4.5. Sensitivity Questions (Artifact X Alternative Context)*

Sensitivity questions try to find out how an artifact is going to behave in different contexts. For the linked data monitoring artifact they are listed below:

- What happens if the query selectivities change?
- How do the number of subqueries in a federated query affects the performance of the system?
- What happens if a query should be answered by a SPARQL endpoint with no DaDy definition?
- What happens if a query should be answered by a SPARQL endpoint with no VoID definition?
- What happens if query registration requests get more frequent?

### *4.6. Requirements Satisfaction Questions*

Requirement satisfaction questions ask whether results of the interaction between an artifact and a context satisfy the requirements. Satisfaction is not strict but to some degree. Requirement satisfaction questions defined for linked data monitoring artifact can be grouped as follows:

- Does the artifact isolate developers from the details of linked data monitoring? (What assumptions does the artifact make about the linked data monitoring effort required by the developer?)
- How much time passes between the detection of a query result change and notification of the registered applications? Does it satisfy functional requirements?
- How much time passes between query result change and change detection? Does it satisfy functional requirements?

## 5. Design Cycle

The design cycle of a design science project consists of three major processes which are ***problem investigation***, ***treatment design*** and ***treatment validation***. Since the design of the linked data monitoring artifact was already described and elaborated in the prior sections, the design will be briefly explained in terms of design cycle processes in this section.

### *5.1. Problem Investigation*

This process explains which phenomena should be improved and the reason behind the need for that improvement. Problem investigation requires the definition of the potential stakeholders of the proposed artifact, their goals to use it, the conceptual framework of the problem domain, the phenomena present in this domain and results of applying the proposed artifact to the problem inducing phenomena in the domain.

Stakeholders who can be affected by the linked data monitoring artifact are the software developer and users of the developed software. A software developer may have the goal to be able to implement code that monitors required SPARQL queries with the least effort. Though the end user does not use the artifact directly, they have the goal to have software that is able to react to changes in data.

An artifact and its context are composed of some structures which form the conceptual framework. It consists of concept definitions named constructs that explain the structure of context and artifact. Constructs of conceptual framework fall into two categories that are architectural and statistical structures which are described below from the perspective of linked data query monitoring.

The problem domain for the linked data monitoring artifact is linked data space. Therefore, architectural structures of the conceptual framework are:

- linked data space
- SPARQL endpoints
- SPARQL queries
- federated SPARQL queries
- join algorithms
- linked data change dynamics
- DaDy definitions
- pull, push, publish/subscribe, pull/push systems
- actor systems
- actor components of the system: Query Distributor, Sub Query Distributor, Sub Query Executor, Parallel Join Manager, Hash Join Performer.

Statistical structures are variables which can be used in the definition of phenomena in a treatment. For the linked data monitoring artifact they are linked data change dynamics and DaDy definitions of SPARQL endpoints.

As for the phenomena that pose problems for the stakeholders in this domain, the need and complexity of creating a new software that is able to monitor linked data changes and current approaches/solutions should be discussed.

Data on the linked data space changes rapidly. Additionally, a great number of the applications used on a daily basis are data sensitive. That is, they should be aware of data changes and react to these changes in some form depending on their uses. Current approaches to linked data monitoring are limited to monitoring single data sources or dependency on the push mechanisms of data sources. This leaves the responsibility of combining information coming from different data sources to the programmer. If a monitoring solution is not available to the programmer, she may have to use the push mechanism provided by data sources. However, data push is not mandatory for SPARQL endpoints. If the programmer has to use an endpoint that does not support data push, she becomes responsible for periodic data pull from that source too. Since this requires the programmer to develop software in linked data domain which may be different from the application domain, an extra burden is put on him. Additionally, since a new task is loaded over the application, this brings an extra burden to the application too. A more detailed discussion of current approaches to linked data monitoring and the problems they cause were given in section ???, therefore this discussion is kept brief in this section.

### *5.2. Treatment Design*

In this section the architecture and implementation details of MonARCh have been given. The term treatment is used to define a solution for a problem in a context by designing the right interaction between an artifact and the problem context.

In order to design a treatment, requirements and assumptions about the context should be defined. The purpose of the linked data monitoring artifact is to take the periodic data monitoring load from the programmer and the application she develops. Therefore it should satisfy the following requirements:

- The system should be able to monitor the results of federated SPARQL queries at the change frequencies of the SPARQL endpoints involved.
- Different client applications should be able to register new federated SPARQL queries for monitoring.
- Each query should be able to require dynamically interacting with SPARQL endpoints which is completely a new feature to the system.

These requirements make the presence of the following context assumptions necessary:

- To define the periodic querying frequency of the data sources, the DaDy metadata of related SPARQL endpoints should be present.
- In order to be able to monitor data sources continuously, there should be no network problems.
- In order to be able to monitor data sources continuously, SPARQL endpoints should be operational at all the time.

The combination of the requirements, provided that the context assumptions are present, is expected to contribute to the programmer by isolating herself and her code from the complexity of linked data monitoring.

Although there are some available similar treatments for this purpose, they do not serve the full functionality planned to be served with the linked data monitoring artifact. Since these treatments were examined thoroughly in the related work, here they will not be examined in detail.

The new treatment design made in this study has the capabilities of monitoring many queries whose results come from a diverse set of SPARQL endpoints, and combining their results as requested in the federated queries registered by the client applications. This design isolates the programmer completely from the details of linked data monitoring tasks.

System is built on top of the Actor model [27], [26], which has a wide range of implementations mainly aims constructing concurrent systems. The monitoring system which is designed to work in the linked data environment, constantly keeps track of the changes in SPARQL query results. In order to maintain the scalability and concurrency under such a heavy query workload, asynchronous structure of actors is well-suited to the system architecture. Subscribers issue some SPARQL queries to the monitoring engine about their interest in order to get notified about possible changes in future. Monitoring engine identifies <query,endpoint> pairs and distributes the execution work over the cluster. Worker actors do the heavy work by periodically executing a query over its relevant endpoint in an infinite loop according to the estimated update interval of the endpoint. New query result is compared with the previous one to find out if there is any change. SPARQL queries can be both simple or federated, so that in case of federated detecting the change in their result may need for the recalculation of the sub results by performing join operations. For adapting to the distributed and concurrent nature of the actor model, hash join technique is chosen for the generation of federated query results. In the hash-based join method, sub results are disjoint from each other, therefore join operation can be operated concurrently. Hash join algorithm has been implemented based on GRACE [29] and Hybrid [15] methods which are designed for parallel multiprocessor architectures. From the system scalability point of view, all actors are automatically deployed on a computer cluster distributed via a modulo-based simple routing algorithm. System is able to scale up by adding more resources to a node in the cluster, and scale out by adding more nodes to the cluster.

Internal architecture of the system is depicted in Figure-1. There are five actor based components; Query Distributor, Sub-Query Distributor, Sub-Query Executor, Parallel Join Manager and Hash Join Performer. While Query Distributor, Sub-Query Distributor and Sub-Query Executor manage the query monitoring flow, they should be persistent and also have their own regions that control the creation of appropriate actor instances and routing messages to them. On the other hand Parallel Join Manager and Hash Join Performer are result based components and also the results may not be the same and can change over time then these components should not be persistent and recreated for each join operation. When the system receives a query, it is routed to the query distributor actor via its region (step 1 in Figure-1). Query Distributor splits the main SPARQL query into sub-queries if it is federated. Relevant SPARQL endpoints are assigned for each detected sub query. Then these <sub query, endpoints> pairs are sent to the sub-query distributor region which delivers messages to the relevant Sub-query Distributor actors (step 2 in Figure-1). Query Distributor is also responsible for keeping track of results for the sub-queries that will come from Sub-Query Distributors. Because these sub-results are then sent to the Parallel Join Manager to make them joined into one main result (step 8 and 9 in Figure-1). Similarly when sub-query distributor receives <subquery,

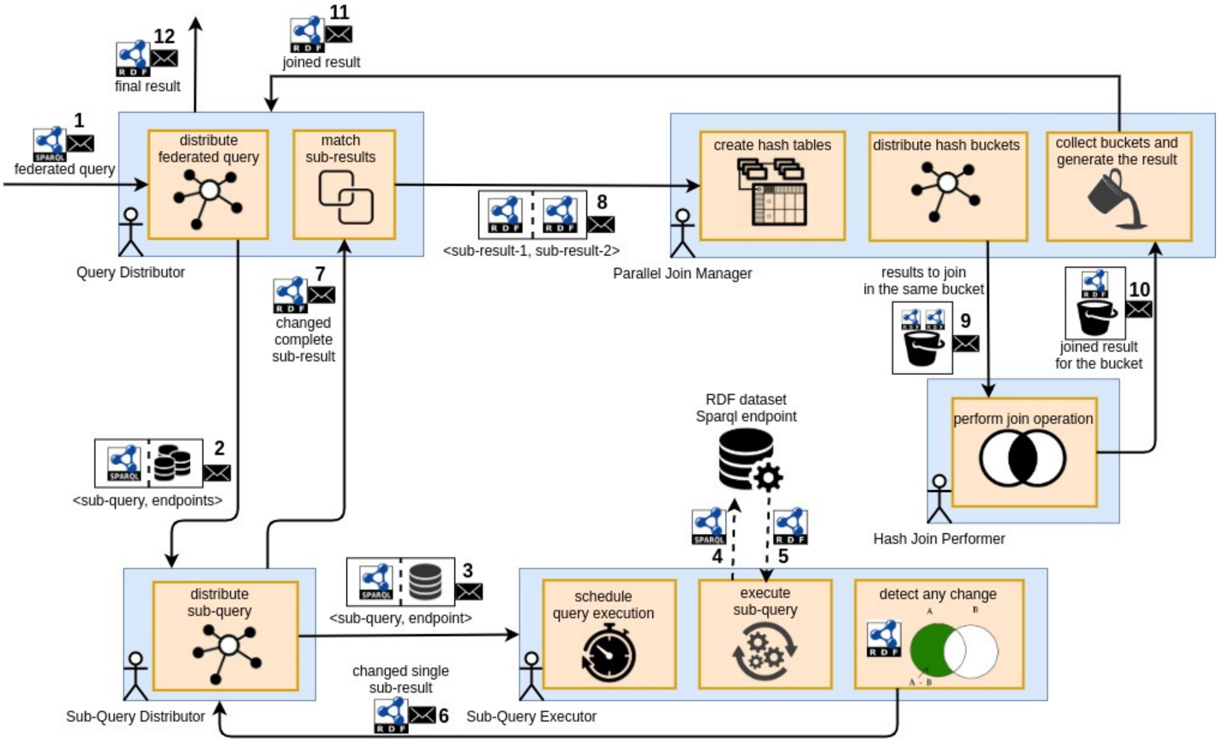


Fig. 1. Query Monitoring Pipeline of the System

endpoints> pair as a message, it sends sub-query to the relevant sub-query executor actor for each endpoint via the sub-query execution region (step 3 in Figure-1), then collects and merges the results coming from them. Finally when the sub-query executor receives a <subquery, endpoint> pair as a message it executes the sub-query against the relevant endpoint. This module is the key component which tries to find the change in the result of a sub-query. For this purpose it schedules itself according to the estimated change interval of the endpoint to re-execute the query and get the new result. New result is compared with the old one to check if there is any change. Once a change has been detected it is propagated back to the Subquery Distributor. Subquery distributor collects the new results of its relevant subquery which may be executed over different datasets.It creates a new merged result for the subquery and propagates back to the Query Distributor. When a new (changed) result is received by the Query Distributor it finds a matching sub result via a common query variable, then sends two sub results to the Parallel Join Manager for the join plan. Parallel Join Manager creates hash tables using the Grace Hash Join algorithm according to a fixed bucket size for both results. For executing the join operation parallelly, buckets of the hashmaps are paired according to keys and each pair is sent to a distinct Hash Join Performer actor. Simple hash join operation is performed by the Hash Join Performer and the joined result is sent back to the Parallel Join Manager. Parallel Join Manager collects joined results for all buckets, merges them, generates the final joined result and sends it back to the Query Distributor. As the last step, when Query Distributor receives the final result for the first time or as a change, it sends the result to the issuer agent of the main SPARQL query.

Actor system is designed to work as a cluster deployed on computer nodes enabling full scalability. Akka toolkit is used as the implementation of actor model on JVM for building the actor based system architecture and cluster management. A sample cluster structure model built using the Akka cluster sharding feature can be seen in Figure-2. Roots of the cluster sharding architecture of Akka are inspired from [25]'s work. Cluster has a coordinator actor that directs an incoming message to the relevant node by communicating them via gossip protocol. Three critical modules of the system are actor regions themselves controlling the creation of entity actors and routing of the

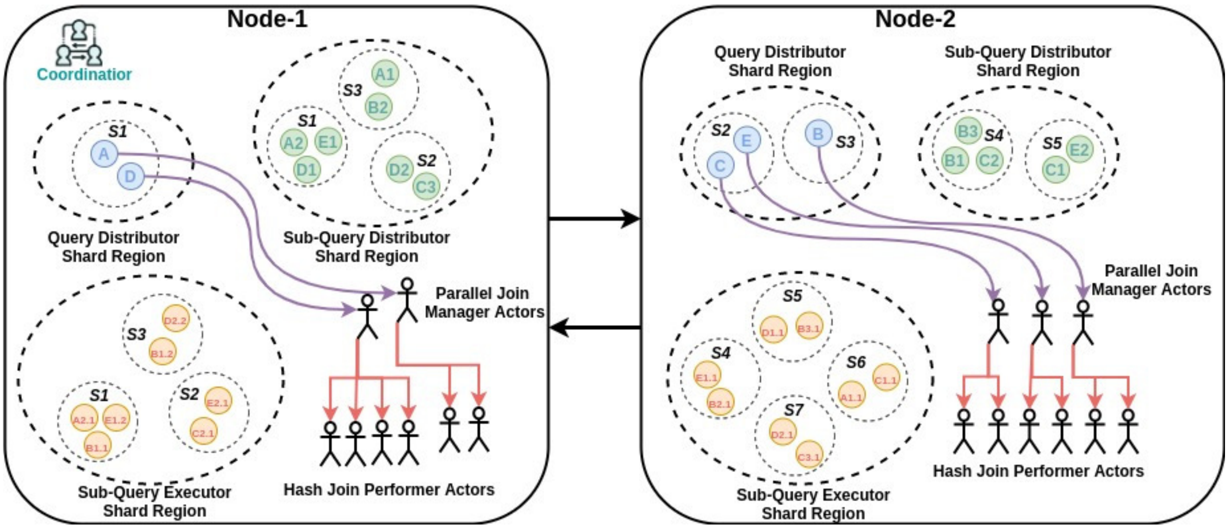


Fig. 2. Sample Cluster Structure Model for the Query Monitoring Environment

messages. Shards are simply groups of actor instances, named entities residing in regions, therefore this structure facilitates moving these entities later on if needed. The mentioned cluster architecture provides location transparency for actors, thus there is no need to know on which node an actor is running. Messages are just sent to a region in any node, then cluster management takes care of the rest, including load balancing of actor deployment and message delivery. Entity actors are shown as letters (some letter with numbers) with colored circles grouped in shards coded with a letter S and shard number. Each node of the cluster has its own shard regions and shards are distributed over the cluster according to the same logical region. There are three shard regions named as; Query Distributor, Sub-Query Distributor and Sub-Query Executor owned by both nodes but managed as the cluster shown in Figure-2. For example a federated query coded with letter B and represented as a blue circle resides in the Query Distributor shard region of Node-2. Its subqueries with relevant endpoints are coded with the same letter and a query number as B1, B2 and B3 represented as green circles residing in Sub-Query Distributor shard region. While entities are logically in the same region they can be located in different nodes. For example B2 is in Node-1 and B1, B3 are in Node-2. Similarly subqueries with its relevant endpoint are coded with the same subquery code, a dot and an endpoint number as B1.1, B1.2, B2.1 and B2.3 residing in the Sub-Query Executor shard region. Parallel Join Manager and Hash Join Performer actors are not managed by cluster, so they exist in the same node with their parent actors. Because both are join relevant actors and a join operation with the same two results is executed only once and never used again, these actors are destroyed after the join operation has been completed. Parallel Join Manager is created by Query Distributor and Hash Join Performer is created by Parallel Join Manager actors.

### 5.3. *Treatment Validation*

Treatment validation is the justification that a treatment developed would contribute to stakeholder goals [53]. Justification requires predicting how the designed artifact will behave in a predefined context. Therefore, an analogical model of the artifact and a model of the context are needed to be defined. The next step is observing the interaction of the artifact and environment models which means an empirical study should be conducted. Thus, discussion about treatment validation is left to subsection 6 where the empirical cycle is examined.

## 6. Empirical Cycle

Evaluation of a research project is named as the empirical cycle in design science methodology. While validating a proposed architecture, the empirical cycle guides researchers by providing incremental and iterative steps to form and consolidate the evaluation. Empirical cycle requires identification of the *"Research context"* first, and it consists of five steps which are; *"Research problem analysis"*, *"Research and inference design"*, *"Validation of research and inference design"*, *"Research execution"* and *"Data analysis"* respectively.

There are two main approaches to empirical cycle implementation namely experimental and observational studies. Researchers are expected to select the approach they will follow according to the characteristics of their research.

The research conducted in this study is experimental. Experimental research is further classified into case based and sample based research. Since the proposed linked data monitoring architecture should be constructed to be validated and there is not a sample present to be observed, this study can be further classified as case based research.

Case based research can be done following one of the three techniques which are single-case experiment, comparative-cases experiment and technical action research. Single-case mechanism experiments are recommended for validation research. Since the aim of the empirical cycle of this research is validation of the model proposed for linked data monitoring, single-case experiment technique (SCME) has been chosen.

In this section, steps of the validation study which is done in accordance with single-case mechanism experiment technique will be discussed.

### 6.1. Research Context

Research context is identified by defining the knowledge goal, improvement goal and current knowledge about the research being conducted.

The knowledge goal of single-case mechanism experiments used in validation studies are validating new technology. In this case, therefore, the knowledge goal is *"validating abstract architecture of the query monitoring system"*.

When specifying the improvement goals of single-case mechanism experiments, credible application scenarios for project results are searched for [53]. Another point-of-view for specifying improvement goals is defining the goal of the engineering effort intended to solve a specific problem [53]. From both points of view, the improvement goal for this SCME can be defined as *"improving linked data change monitoring capability of applications"*.

The current knowledge of a problem context is defined as the related work in the literature relevant to the study being conducted. The literature about the systems based on *"publish/subscribe, pull-based, push-based and pull/push-based architecture"* which was given in Chapter 7 in detail constitutes the current knowledge. The conducted research contributes to the literature by introducing a federated SPARQL query monitoring system which notifies client applications about changes in a linked data environment as fast as possible.

### 6.2. Research Problem Analysis

In validation research the object of study is the treatment which includes the artifact being examined and its context. A research problem is analyzed by defining the conceptual framework of this treatment, knowledge questions about the performance of the artifact in this treatment, and the potential artifact population represented by this particular artifact.

The conceptual framework and knowledge questions about the linked data monitoring artifact have been given in sections 5.1 and 4 respectively. Hence, they are not discussed here again.

A population can be defined as a group of objects that satisfy a population predicate. A population predicate is the generalization and abstraction of all population elements in some respect(s). More specifically a population is all possible implementations of the artifact-context pairs. Thus, the population in this research is "federated query monitoring systems based on the actor model which use parallel join algorithms and notify client applications on any change". The elements of this population can be similar and dissimilar to each other in following ways:

- Actor framework used in the system (Akka, Erlang[3], etc.)
- Parallel join algorithm used in the system (Hash join, sort-merge join, etc.)

### 6.3. Research Design and Validation

A Single-case mechanism experiment requires construction of a validation model which includes a representative sample of the proposed artifact model and the foreseen context for it. After that these two components are observed while interacting with each other which is called the treatment phase. During the treatment phase measurements are taken, so that inferences can be made from the acquired data.

[3]https://www.erlang.org

#### 6.3.1. Constructing the Validation Model

In validation research, the object of study corresponds to a validation model which includes a prototype of the artifact and a model of the context.

Since in this study, the validation method chosen is a single-case mechanism experiment, the object of study is singular, which includes the actor based artifact and its context whose models have been examined in Section 5. This artifact has two main workflows. One of them is responsible for the execution of queries and observation of the changes, the other one is responsible for collecting the sub-results and performing the join operations to produce the main result. To take advantage of the parallel, distributed and scalable architecture of the actor model, which the artifact is built upon, a parallel hash join algorithm has been used. The context in which the artifact is intended to operate is composed of; SPARQL endpoints, clients who issue query monitoring requests, RDF triple stores in which the datasets persist, linked data change dynamics which issue updates to rdf triple stores of datasets and a key-value store to keep statistics about the system. After gathering the necessary data for validation research, abductive and analogical inferences have been used (as indicated in the single case mechanism experiment method) to show the validity of the OoS. This work can be seen as a guide for studies that will use similar architectural design.

#### 6.3.2. Constructing the Sample

The prototype, which is a member of the potential artifact population it represents, has been implemented using the Akka actor system and a parallel hash join algorithm on a two node cluster. Each node on the cluster had Intel Xeon E5 2620 v4 (8 core - 16 thread) processor, 32 GB RDIMM RAM, 1.8 TB 15K SAS Disk with RAID 5 configuration and Ubuntu 18.04 operating system installed on.

Akka toolkit which is an actor model implementation on JVM has a cluster sharding feature which lets the system to work as a cluster on multiple nodes (computers) and provides location transparency for actors enabling easy access to them. Therefore the actor system can make use of multiple JVMs running on different nodes.

To enhance the scalability and support the query monitoring feature; grace hash join algorithm, which is an efficient parallel join implementation in the database literature for query processing purpose, has been implemented.

The artifact context has been sampled as a closed world, which is not open to any outside influences, in the financial domain. Two nodes in the cluster have Virtuoso[4][18, 19] RDF servers installed and running on them that serve as the SPARQL endpoints for DBPedia, Nytimes and Stock datasets. In order to log the acquired experimental data, Redis[5] key value store has been installed on the third node. Queries are issued to the system via a separate client actor system running on a distinct fourth node.

After the construction of the OoS, analytical induction strategy has been used to prove that the designed architecture is able to monitor linked data changes in a scalable manner by executing a planned treatment (case). The treatment prepared for this study is discussed in the next section.

In order to apply analytical induction, a strategy should be chosen from the two options that designing confirming cases and disconfirming cases which are extreme cases produced to explore the boundary conditions for a proposed artifact model. Confirming cases prove reproducibility of similar samplings under similar conditions and disconfirming cases show the limitations and conditions that such a sampling cannot be produced anymore. For this study, designing confirming cases strategy has been chosen, since the main aim is to prove that this scalable artifact design may be used when there is such a need to monitor periodic changes of linked data coming from multiple sources. On the other hand, during the experimental studies a disconfirming case has been encountered too. In this case, queries could no longer be processed and result change notifications could not be made since the akka cluster the prototype was built upon stopped working properly.

Finally, to be able to support abductive and analogic inferences on the data that can be gathered during the experiments, the prototype is designed to log the data related to metrics like query processed per second, total query processed, result (tuple) processed per second, and total result (tuple) processed into the Redis store instance as stated above.

#### 6.3.3. Treatment Design

Validation models are tested against scenarios named treatments. In other words treatments are used to validate an artifact in a simulated context. For constructing

[4]https://virtuoso.openlinksw.com
[5]https://redis.io

scenarios, treatment instruments (components) must be identified first.

The scenario used in this study uses the draft SPARQL query shown in Figure-1 to monitor the changes in the stock value and Nytimes article count data about different companies. While Dbpedia dataset holds general information about the companies including link predicates to their Nytimes conjugate resources, Nytimes dataset has article count, and Stock dataset keeps the stock value of these companies. To construct this scenario, it is supposed to replicate local copies and SPARQL endpoints for the Dbpedia and Nytimes datasets, and create a local Stock dataset with its own SPARQL endpoint. For creating the periodic changes in these datasets, Nytimes and Stock datasets should be updated periodically and continuously by two programs with different periods. Moreover, to keep statistical data about the actor and query counts, measurements of the Akka cluster should be recorded. Finally, to implement this treatment it is required to create users, which are represented by Akka actors, that register their raw SPARQL queries into the prototype cluster and want to be notified about changes in the results of these queries.

When the requirements given above have been taken into consideration, the need for the following treatment instruments would become apparent: local datasets and SPARQL endpoints for Dbpedia, Nytimes and Stock data created on local Virtuoso instances that run on separate nodes, two updater programs for updating the Nytimes and Stock datasets periodically that run on a different node, Akka actors representing clients of the prototype, and finally Redis key value store for recording observational data.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX nytimes: <http://data.nytimes.com/elements/>
PREFIX stockmarket: <http://stockmarket.com/elements/>
SELECT * WHERE {
  <companyURI> owl:sameAs ?nytCompany .
  ?nytCompany nytimes:associated_article_count ?articleCount .
  ?nytCompany stockmarket:stockValue ?stockValue .
}
```

Listing 1: Draft Raw SPARQL Query for System Evaluation

The datasets for this treatment have been taken from the FedBench suite [47], except for the Stock dataset which has been created for this study. However, there are only 500+ companies that satisfy the requirements of the query in Listing-1, it is not enough to test the boundaries of the system in terms of scalability for this setting. Therefore, artificial data about companies will incrementally be inserted into the datasets until reaching to the limit of the system for experimental purposes.

```
PREFIX owl: <http://www.w3.org/2002/07/owl\#>
PREFIX nytimes: <http://data.nytimes.com/elements/>
PREFIX stockmarket: <http://stockmarket.com/elements/>
SELECT * WHERE {
  SERVICE <dbpediaServiceEndointURI> {
    <companyURI> owl:sameAs ?nytCompany .
  }
  SERVICE <nytimesServiceEndointURI> {
    ?nytCompany nytimes:associated_article_count ?articleCount .
  }
  SERVICE <stockServiceEndointURI> {
    ?nytCompany stockmarket:stockValue ?stockValue .
  }
}
```

Listing 2: Draft Federated SPARQL Query for System Evaluation

The SPARQL query given in Listing-1 is converted into the federated query seen in Listing-2 by means of the SPARQL query federator engine present in the prototype architecture. In the scenario, a high number of queries are planned to be registered to the system which are all instances of the single scenario shown in Listing-2. For each query registered, an actor is expected to execute queries against the system periodically. Scaling the system up to handle a big number of queries for this treatment is considered to validate the artifact model.

As for the schedule for registering new queries to the system, the set of all possible queries should not be sent at once but fragmentally. Since actors in the actor system are threads and do some sort of work using CPU and memory resources they need to release the resources after a timestamp passed, for allocating them new actors to use. For the sake of the treatment a schedule should be specified by empirically as trying to split the query set into fragments with a processing delay cost.

#### *6.3.4. Measurement Design*

For validating the object of study, results of the research execution is required to be measured. In this context, the following variables and constructs should be measured:

- Total query count system can monitor
- Total actor count system can have
- Relationship between query count and actor count
- Max query count system can monitor per unit of time
- Max actor count system can have per unit of time
- Max result size system can have per unit of time
- Total result size system can have

- Max memory usage of the system
- Average memory usage of the system
- Maximum CPU usage of the system
- Average CPU usage of the system

To be able to measure these variables and constructs, the data sources should be identified from which these measurements can be taken. The following are the means by which the measurement data given above can be acquired:

- Each actor registers itself to Redis store by increasing the actor count.
- Each query handled by the system is registered to Redis store by increasing the query count.
- Actor system logs include query count, actor count, memory usage and CPU usage.
- Each query has the same sub result and final result size calculated by executing the instance of a query.

To store and serve the acquired data the following components have been added to the system:

- A Redis instance that is deployed as a server for storing and managing measured actor and query counts.
- A middleware actor component that stores query and actor counts into Redis store.
- A middleware actor component that retrieves CPU and memory usage using sigar-loader utility of kamon-io library.
- Logger components of the actor system.
- Federated and sub SPARQL query instances are used to calculate result size.

The frequency of data probing is important. So, the following measurement schedule has been defined:

- A query load consists of a bunch of SPARQL queries will be sent to the system with a time interval.
- System will handle a number of query loads until it reaches current maximum company count.
- Actor system logs

Finally the measured actor and query count data are going to be stored and managed in Redis instance which is deployed as a server.

### 6.4. Inference Design and Validation

There are three steps for making valid inferences from single case mechanism experiments which are description, architectural explanation, and generalization by analogy [53]. In this section inference design of the experiment at each step will be discussed.

#### 6.4.1. Descriptive Inference Design

There are three aspects to descriptive inference design: defining the way words and images should be interpreted, planning the way data obtained during experiments can be summarized, and ensuring the validity of descriptive expressions used [53]. Each will be discussed in the subsections below.

*I. Interpretations of The Terms Used*

Some technical terms used in the paper that may cause confusion are explained in Table-1.

*II. Validity of Data Acquisition, Summarization and Interpretation*

One of the main requirements for the validity of data used in descriptive inference is the purity of data. By purity it is meant that the data should be free of information that can be added by the researcher by means of the observations or the previously gained knowledge [53].

In order to ensure this purity, uncommented data coming directly from the system is decided to be used in this work. Number of monitored actors and queries taken directly from the running Akka cluster instance are recorded to Redis store. When the system has reached monitoring its query and data set count limit, the experiment has been concluded. Meantime, the memory and CPU cycles used by the monitoring system has been retrieved from the operating system logs.

As for the validity of tables and graphs, only the recorded data has been plotted and given without any further comments or calculations. This enhances the reproducibility of the results and interpretations.

#### 6.4.2. Abductive Inference Design

Abductive inference looks for architectural cause-effect explorations for the observed phenomena. In order to ensure the internal validity of the abductive inference, design science methodology defines a checklist under the three main headings causal inference, architectural inference and rational inference [53].

*I. Checking the Validity of Causal Inference*

In order to ensure the validity of causal inference there are four main points needed to be paid attention to [53]: OoS dynamics, treatment control, treatment instrument validity, and measurement influence.

In this study, since there is only one prototype of the proposed artifact, OoS interaction is not possible

Table 1

Interpretations of The Terms Used

| Term | Interpretation |
|---|---|
| Query | In this paper, the term query is used to mean a federated SPARQL query. A federated SPARQL query consists of more than one basic SPARQL query, each aimed at a different SPARQL endpoint. The result of a federated SPARQL query is obtained by combining the results that come for each of these basic queries. |
| Subquery | It is used to mean individual basic queries that retrieve data from a single SPARQL endpoint and constitute a federated SPARQL query. |
| Endpoint | This term may be used instead of SPARQL endpoint. |
| Dataset | It is used to mean an RDF dataset which is stored on an RDF triple store and open to SPARQL queries sent through the SPARQL endpoint it is connected to. |
| RDFStore | This term is used to denote a purpose-built database for storage and retrieval of RDF triples through semantic search. |
| Query Federation Engine | This is used to denote an RDF query federator that parses raw SPARQL queries, selects the data sources related with each of the subqueries, optimizes them and converts them into federated queries. |
| Actor | In this study, the term actor represents Akka actors[6] which are defined as objects that encapsulate state and behavior, communicate by exchanging messages which are put directly into the recipient's mailbox. |
| Cluster | It is used as a term to denote computers that are connected to each other under a common control mechanism and perform the same task. |

which can also affect the inference. When it comes to treatment control, the experimental setup is closed to any interaction coming from outside. Additionally, since the experiment runs on its own without any intervention once started, there is no other risk about treatment control.

As for the treatment instrument validity when run on more powerful computers with higher memory and processor capacities or on a cloud environment, better results may be obtained. Finally, measurement does not interfere with the inner mechanism of the artifact. However, since the processes that take and record measurements run on the same environment as the artifact itself, it may degrade the system performance to a small degree.

*II. Checking the Validity of Architectural Inference*

To ensure the validity of architectural inference all architectural elements and their interactions present in the treatment design have been specified, including the ones in the architectural analysis.

According to design science methodology, the real world case components should completely match the architectural components. All architectural components in the prototype can be used while developing a real life instance of the design. For instance, Akka matches with the actor system component, and Virtuoso SPARQL endpoints match with the SPARQL endpoint component of the context the artifact should operate in.

In the conducted experiment, the same federated SPARQL query pattern for different companies has been used. Though the abstractions used in the experiment match real world cases, only exception to this is the usage of this query pattern. In real world cases many different kinds of query patterns can be monitored using the artifact architecture proposed. However, since the aim of the experiment is to test the performance of the system, this basic query pattern as a unit for measuring the performance of the prototype has been preferred.

### 6.4.3. Analogic Inference Design

A validation model can be generalized to the population it represents if it is able to support valid analogic inference. To decide if the analogic inference to be done is valid, researchers should be certain that the OoS is similar enough to the population it represents [53].

In this case, since the implemented architecture and its components are suitable for use in a real life implementation, it can be said that the OoS represents the population of the artifact design properly. However, there are three details that may not match real life cases.

First one is the query pattern used during the experiment. In a real case, many different query patterns that require querying many different SPARQL endpoints are expected to be seen. In the designed experimental case, the queries tested all share the same pattern and are replied using the same SPARQL endpoints. However, as stated before this setting was chosen in order to see the performance limits of the prototype better. If a heterogeneous set of query patterns have been used, it would be hard to visualize the limits of the OoS. This experimental query pattern is used as a unit of performance.

Second detail of the OoS that may not match real life cases is the configurations of the computers used. In a real life case much more powerful machines or a cloud is expected to be used which will enhance the performance.

The last detail is the change frequencies of the data on the SPARQL endpoints. In the implemented treatment, *2f* per hour for Stock dataset and *f* per hour for Nytimes dataset as change frequencies have been used. In real cases, the frequency of change is much lower [24]. Therefore, in a real life case higher numbers of queries are expected to be monitored, since the system will have to monitor the SPARQL endpoints in longer time intervals.

As a result, it can be said that the analogical inferences made at the end of the experiment are valid.

### 6.5. Research Execution

After the research has been designed, it is executed and the findings are recorded. Even if some unplanned events or behaviours occur during research steps everything relevant with data analysis should be noted.

Object of study is constructed as an actor based SPARQL query monitoring architecture designed to monitor SPARQL queries to be executed over service endpoints of RDF datasets. In order to achieve a distributed and parallel implementation there are some choices like Java threads, map reduce jobs, stream processors, and actors. In this study, the actor model has been chosen among them to build a query monitoring prototype for the object of study. Between the data/services to be monitored like JSON/REST-Service and Tuple/Database-Service etc., RDF/SPARQL-Endpoint pair has been selected as context for the object of study. Constructed object of study has the same architecture as the designed one.

Since the research method in this study is SCME, only one sample has been constructed by implementing proposed architecture with Akka toolkit and loading Dbpedia, Nytimes and Stock datasets over separate Virtuoso service endpoints. During the early implementation of a scalable actor based query monitoring architecture, scalability has been tried to be handled by implementing a separate middleware other than cluster sharding. The middleware was responsible for deploying new actors on discrete nodes by using the remoting feature of Akka and checks if a node has reached its actor limit or not. Actor count was a static number and its purpose was that middleware to raise the load of one node first then jump into another one when predecessors are overloaded. The middleware stores actor counts of nodes and actor references into Redis store, but this leads all the management load to be shifted to a single component which causes a bottleneck. Furthermore a narrow scalability approach managed through only a static actor count even without monitoring resource usage of nodes restricts the full capability and flexibility of the actor model. Therefore it is decided to use the cluster sharding feature of Akka which provides location transparency for actors, manages their distribution and deployment by own. While the aim is to monitor queries, query results are sent between actors as messages. Size of some results may be too large, thus it can be hard to serialize and send through actors running on different nodes with TCP communication. For this reason UDP messaging configuration has been used which suits better for time-sensitive applications and is able to transmit larger messages easily. A two-noded Akka cluster has been set up with CPU parallelism and UDP messaging features. From the context point of view, Virtuoso services have been deployed with Docker[7][37] configuration on two nodes where the cluster has been built upon. Each dataset has been served through separate endpoints and nodes. While Dbpedia dataset is not updated, it has been deployed on the same node with the client application from which query registration request comes from. To make changes on Stock and Nytimes datasets two updater programs have been configured to run continuously as Java threads on JVM. Queries issued by the clients are transmitted to the Akka cluster from a client actor system through cluster client receptionist property of Akka. Updater programs, Redis server and client system have been set up and run on the same machine different from Akka cluster nodes. By this way the query load of the cluster is not affected by any other third party programs or services.

The designed treatment has been executed against the constructed sample as planned. Therefore, a client actor system has been implemented for sending queries to the actor cluster. Unique raw queries have been constructed for each virtual company URI according to the template in Figure-1 and they are converted into federated queries using WoDQA[4] federated SPARQL query engine. Each federated SPARQL query is sent

[7]https://www.docker.com

to the cluster by a distinct client actor via the cluster client receptionist feature of Akka. When too many queries are sent at once, a bottleneck occurs because there will be too many big sized messages trying to be sent through network and trying to be processed at once. If an actor tries to get a message, it waits for UDP driver to be available and to complete transmitting all packages of this message. Therefore when UDP driver is not available for a long time waiting to process remaining messages because of the current transmission work of too many big sized messages, the actor system shuts itself down. Thus queries are needed to be sent gently as reasonable subsets and it should be waited for a while for messages to be sent through network to the destination actors and for actors to do their jobs and release operating system resources.

In order to test the upper limits of MonARCh, RDF resource data and linksets of virtual companies have been generated and inserted into the datasets by a number of 500. Evaluation has been performed iteratively after each data insertion for new company size until system reaches its monitoring boundary. A unique federated query has been generated for each company in stock dataset of current state. For this reason throughout each phase of evaluation, same number of federated queries have been sent with the current company size. Until 2500 company size queries have been sent with 20 percent bunches of total query count and 1 minute delay between each query bunch. That makes Stock dataset updater to run every 5 minutes and Nytimes dataset updater to run every 10 minutes. After 3000 company size near upper boundary of the system, queries are need to be sent with 10 percent bunches of total query count with the same 1 minute delay. Consequently Stock dataset updater runs every 10 minutes and Nytimes dataset updater runs every 20 minutes.

Data sources and measurement instruments are the same as ones in measurement design. After some heuristically executed tests, measurement dynamics have finally been defined as follows:

- Evaluation has been performed iterative and incrementally according to company count.
- Before each iteration 500 new artificial company data and links between datasets have been inserted.
- A query load of %10 company size until 2500 companies and %5 company size after 3000 companies have been sent to the system every minute until system reaches the number of queries same with the company size.
- System metrics such as query execution and notification times, CPU and Memory usages have been collected from actor system logs for each node.

### 6.6. Data Analysis

In this section firstly unprocessed data will be given in the section 6.6.1. After that, the behavior of the system will be analyzed in the light of this raw data in section 6.6.2. In section 6.6.3 the findings will be generalized to the target system population using analogic reasoning. Finally, knowledge questions which are used as a guidance during the study will be answered in section 6.6.4.

#### 6.6.1. Descriptions

There are 3 subqueries in each federated query registered to the system which can be shown at Listing-3, Listing-4, Listing-5.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT * WHERE {
  <companyURI> owl:sameAs ?nytCompany.
}
```

Listing 3: Sub Query of Dbpedia Endpoint

```
PREFIX nytimes: <http://data.nytimes.com/elements/>
SELECT * WHERE {
  ?nytCompany nytimes:associated_article_count ?articleCount.
}
```

Listing 4: Sub Query of NyTimes Endpoint

```
PREFIX stockmarket: <http://stockmarket.com/elements/>
SELECT * WHERE {
  ?nytCompany stockmarket:stockValue ?stockValue.
}
```

Listing 5: Sub Query of Stock Endpoint

According to designed measurement, treatment has been executed and results have been collected. Values of variables and constructs for actor system are listed in Table-2. For each company dataset count per federated query (10000 + 2 x company_size) tuples are received as result. System can monitor up to 3000 query and has failed when company dataset and query size is 3500. Per federated query there are at least 3 query actors permanently executing query, monitoring result change and creating temporary actors for performing join operation to produce final result. While the split count is 100 for producing a bucket map, the tempo-

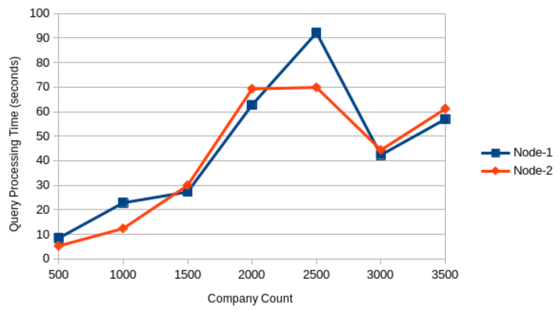


Fig. 3. Average Query Processing Times of Cluster Nodes

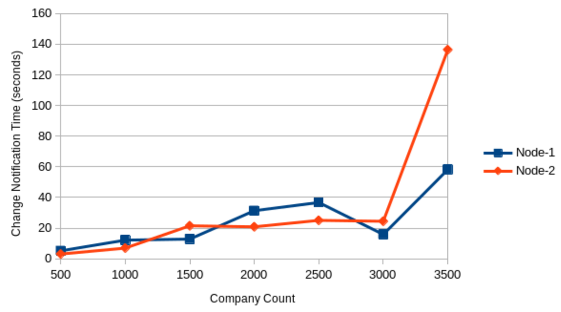


Fig. 4. Average Change Notification Times of Cluster Nodes

rary actor count responsible for one join operation will be 100 at most. On the other hand if the sub query count is n, it needs to be (n-1) join operation required to be done at most. Thus for the applied treatment, there will be (3-1) x 100 = 2 x 100 = 200 temporary actors at most. There are 3 permanent and 200 temporary totally 203 actors created.

On the other hand CPU and Memory system resources used by MonARCh have been monitored and logged by a middleware actor component named MetricsListener. Kamon[8] is a comprehensive monitoring library that works fully integrated with Akka. Moreover it can work with monitoring systems and time series databases such as Prometheus[9] and InfluxDB[10], thus enabling any actor and even non-actor system to lively monitor in detail. In this study just because only metrics that CPU and Memory usage ratios are need to be monitored for now, sigar-loader[11] extension of Kamon library satisfies this requirement. But in future work system can be extended to use all features of Kamon library enabling actor system level monitoring. Metrics monitored by MetricsListener are listed in Table-3 for each iteration of evaluation according to company count per cluster node. Moreover average query execution times for the first notification and result notification times after the detection of changes have been calculated and listed in Table-3.

In the conducted experiment, the Dady[12] change frequencies metadata of the local endpoints are defined in Table-4.


Fig. 5. Average Memory Usage of Cluster Nodes

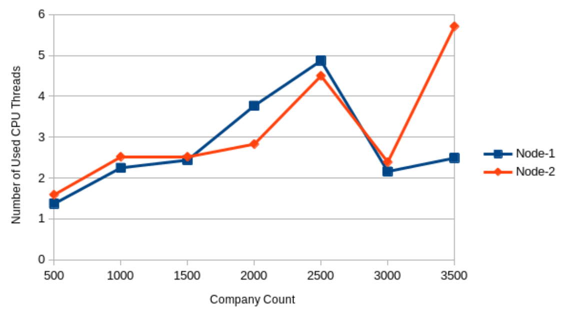


Fig. 6. Average CPU Usage of Cluster Nodes

### 6.6.2. *Explanations*

There are two main observations made during the experiment:

i System can not handle more than 500 query registrations per minute.

ii System can process 2500 federated queries whose results change once every 5 minutes successfully.

[8]https://kamon.io
[9]https://prometheus.io
[10]https://www.influxdata.com/products/influxdb-overview
[11]https://github.com/kamon-io/sigar-loader
[12]http://purl.org/NET/dady#

Table 2
Measurement Results of Actor System

| Dataset and Total Query Count | Result Size per Federated Query | Actor Count per Federated Query | Max Query Count per min | Max Actor Count per min | Max Result Size per min | Total Result Size |
|---|---|---|---|---|---|---|
| 500 | 11000 | 203 | 100 | 20300 | 1.1M | 5.5M |
| 1000 | 12000 | 203 | 200 | 40600 | 2.4M | 12M |
| 1500 | 13000 | 203 | 300 | 60900 | 3.9M | 19.5M |
| 2000 | 14000 | 203 | 400 | 81200 | 5.6M | 28M |
| 2500 | 15000 | 203 | 500 | 101500 | 7.5M | 37.5M |
| 3000 | 16000 | 203 | 300 | 60900 | 4.8M | 24M |
| 3500 | 17000 | 203 | 350 | 71050 | 5.95M | 29.75M |

Table 3
Measurement Results of System Performance for Cluster Nodes

| Dataset Count | Node | Avg Query Processing Time (secs) | Avg Change Notification Time (secs) | Max Memory Usage (GB) | Avg Memory Usage (GB) | Max CPU Usage (processes) | Avg CPU Usage (processes) |
|---|---|---|---|---|---|---|---|
| 500 | 1 | 8.45 | 5.26 | 6.49 | 4.45 | 5.01 | 1.37 |
| 500 | 2 | 5.25 | 3.14 | 2.84 | 2.15 | 7.92 | 1.59 |
| 1000 | 1 | 22.83 | 12.2 | 9.89 | 7.30 | 7.62 | 2.25 |
| 1000 | 2 | 12.36 | 7.02 | 5.41 | 3.66 | 10.75 | 2.52 |
| 1500 | 1 | 27.27 | 12.87 | 7.33 | 4.76 | 12.18 | 2.44 |
| 1500 | 2 | 29.95 | 21.54 | 12.52 | 10.19 | 9.08 | 2.52 |
| 2000 | 1 | 62.74 | 31.41 | 18.57 | 12.92 | 15.18 | 3.77 |
| 2000 | 2 | 69.28 | 20.88 | 9.55 | 7.05 | 14.49 | 2.83 |
| 2500 | 1 | 92.12 | 36.75 | 22.22 | 16.37 | 14.27 | 4.87 |
| 2500 | 2 | 69.88 | 25.09 | 13.31 | 9.49 | 20.49 | 4.50 |
| 3000 | 1 | 42.36 | 15.9 | 14.94 | 8.63 | 9.79 | 2.16 |
| 3000 | 2 | 44.37 | 24.49 | 23.78 | 19.28 | 14.18 | 2.39 |
| 3500 | 1 | 56.97 | 58.22 | 16.06 | 11.27 | 11.13 | 2.49 |
| 3500 | 2 | 61.22 | 136.36 | 24.81 | 19.22 | 15.90 | 5.71 |

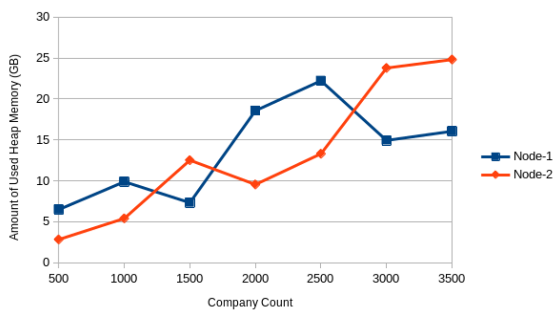


Fig. 7. Max Memory Usage of Cluster Nodes

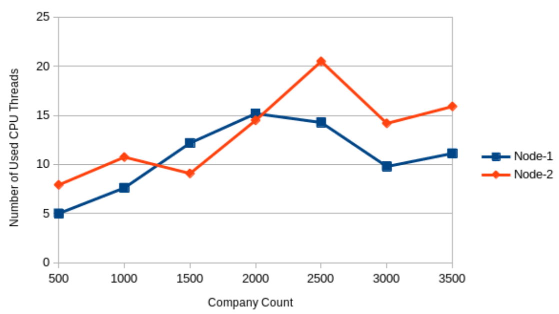


Fig. 8. Max CPU Usage of Cluster Nodes

The first observation can be explained by the physical limits of the system like the memory size and CPU frequency. Although the system can handle 2500 queries at once, since query registration processes require some additional work like the creation of new actors to the system, additional load is put on the memory and the CPU. Actors in the actor system are threads and do some sort of work using CPU and memory

Table 4

Dataset Endpoint Change Frequencies

| **Dataset Endpoint** | **Frequency** | **Explanation** |
|---|---|---|
| Dbpedia | NoUpdate | Data never changes |
| NyTimes | MidFrequentUpdates | Data changes from one a week to a couple of months |
| Stock | HighFrequentUpdates | Data changes once a day or more frequent |

resources. They need to release the resources after a timespan to allow other actors to use the system resources. Therefore, due to instantaneous resource bottleneck, the system can handle only 500 query registrations per minute. With more powerful computers this limit may get higher.

As for the second observation, the system has not been pushed to its limits. It is thought that the system can monitor more queries at the same time since in the real world datasets change less frequently which leaves more time between data changes and lets the system to observe the changes in the results of other queries meantime.

*********************************

Since the time intervals defined in the original DaDy specification are too long for an experimental study, MidFrequentUpdates has been mapped to once every 10 minutes, and HighFrequentUpdates has been mapped to once every 5 minutes. As a result, (1+12500+2500) x 2500 = 37502500  37.5M RDF tuples are processed every 5 minutes after the 6th minute to the end of the experiment which took 6 hours. That means the system processed 7.5M tuples successfully every minute. During the experiment all result change notifications have been delivered to the clients successfully.

The prototype ran on Akka cluster built on top of Node-1 and Node-2 with similarly distributed loads. The load on these nodes were as follows respectively: CPU: %165-memory %28.1, and CPU:%128-memory:%13.6. Node-1 has the role of cluster sharding coordinator which distributes messages to the relevant entities (actors) in the right region, thus it consumes more resources than other cluster nodes.

*********************************

*6.6.3. Analogic Generalizations*

There are three aspects that the designed system may not generalize exactly to real life cases.

First, as stated above, the expected change frequency of real world datasets are much lower. Even the dataset with the highest change frequency is expected to change a few times a day according to the DaDy specification [24]. Therefore the system is expected to monitor a higher number of queries in real life since the change frequency of a dataset gets lower, the frequency of monitoring queries get lower which leaves extra time to monitor other queries to the system.

Second, it can be easily foreseen that the queries that can be registered in a real life implementation of the artifact will have many different queries with different complexities that require a diverse set of SPARQL endpoints to be queried. In the designed treatment a single query pattern with only three SPARQL endpoints is used to evaluate the system properly. Since SPARQL endpoints are part of the context the artifact operates in, this will not bring additional load on the system. Only if the queries to be monitored gets too complex with more than two constantly changing subqueries, the number of queries that can be monitored may get lower. However, even if the number of queries that are to be monitored gets lower, it is expected that the number of subqueries to stay approximately the same.

Lastly, the designed architecture assumes the presence of DaDy metadata to be present for each of the datasets queried. If that data is not present, it may be necessary to query the datasets at the highest frequency defined in the DaDy specification. This may, in turn, degrade the system performance.

*6.6.4. Answers to Knowledge Questions*

*I. Answers to Descriptive Questions*

In section 6.6.1 all descriptive questions have been answered except two questions.

The first one asks the pros and cons of the join algorithm used in the system. In the implemented system, a hash join algorithm called grace join has been used to produce the main result by using intermediate results. Grace hash join splits intermediate results into much smaller buckets according to a mathematical function to produce a hash map and makes use of concurrency to join buckets with the same key simultaneously. Thus dividing a big join operation into smaller jobs and using all cores and processing power of the CPU reduce the join time in many cases. This method fits well to the system architecture based on the actor model. On the other hand when intermediate results are too big to fit into the memory and skewed, hash join performance starts to drop because computation of join operation is done at the client side other than server side where database service operates. For this scenario bind join makes much more sense because it binds one intermediate result of a subquery into another one for both narrowing intermediate result space and passing the computation load to the database as a query execution.

The other question seeks for answer to how do the number of subqueries in a federated query affects the performance of the system?

Federated queries are composed of sub queries each related to one or more SPARQL endpoints. Sub queries mostly share some query variables to tie with each other. Results of two queries which have a common query variable are joined each other by applying a join algorithm. After all sub results have been joined then the main result would be generated. If a federated query has more sub queries, that means more join operation per federated query is needed to be performed. In order to make use of concurrency, MonARCh uses a parallel hash join algorithm to join sub results each other at the same time which yields quick generation of main result. Even if using a parallel join algorithm if there is an increase in the sub query count in a federated query, then more CPU computation is needed to be performed and more memory is needed to be occupied. This may degrade the total query count and max query count per minute metrics in Table-**??**.

*II. Answers to Explanatory Questions*

One in the previous section and rest in section 6.6.2 all explanatory questions except one have been answered.

Question which is left here for the discussion is how would a change in the query characteristics affect the performance of the system?

Triple patterns in the query has the feature named selectivity which affects the result size to be retrieved. A higher selective sub query in a federated query produces less result compared to a less selective one, therefore less computation would be needed to perform the join operation for that query. On the other hand if a sub query is less selective, its result may have many tuples and the computation cost of the join operation to be performed will be high. Thus the selectivity of sub queries will affect the total query count and max query count per minute metrics in Table-**??** directly.

*III. Answers to Effect Questions*

Because all effect questions are included by both descriptive and explanatory questions and they have been answered in sections 6.6.1 & 6.6.2, they will not be discussed here again.

*IV. Answers to Trade-off Questions*

How would the artifact be affected if a different actor model implementation was used?

There are several actor model languages and frameworks which can be available to use as an actor model implementation. Key point is that the actor framework to be mature, robust, well documented, easy to use and resource(computer) friendly. For example; Erlang and Elixir programming languages depend upon the Erlang virtual machine whose threads are much more lightweight compared to Java threads. But on the other hand Erlang and Elixir are both fully functional programming languages which developers are not familiar with, therefore this may affect the development process in a bad way. When the implementation difficulty has been put aside, from the system performance point of view Elixir or Erlang can be seen as a replacement to Akka toolkit. On the other hand there are emerging alternative actor frameworks such as Riker[13] written in Rust programming language , CAF[14] written in C++, Comedy[15] built for Node.js, Orleans[16] built for .net framework. But they all have a less mature multithread mechanism, and have a narrow development community compared to Akka, Erlang and Elixir.

Another trade-off question which is asked at the beginning of the study is how the artifact would be affected if the actor model has not been used.

There are three main reasons for choosing the actor model for concurrency:

i Actor model enforces immutability. When multithreaded applications become mutable this causes many bugs and errors on runtime.
ii Actor systems are easily scalable, which is vital for the designed system.
iii Synchronization is a big problem in concurrent systems. Actors fully encapsulate their behavior and state since they communicate with each other via sending messages. This allows avoiding synchronization and resource allocation problems common in concurrent systems.

These advantages of actor systems over other concurrency models makes them a natural choice for the implemented artifact. If another concurrency model has been chosen, it would be hard to avoid synchronization problems, resource allocation errors and scalability issues.

[13]https://riker.rs
[14]http://actor-framework.org
[15]https://github.com/untu/comedy
[16]https://dotnet.github.io/orleans

*V. Answers to Sensitivity Questions*

Questions related with sensitivity except one have been explained in the previous sections. The question that is left for discussion here asks what would happen if a subquery should be answered using a SPARQL endpoint that has not got a VoID description.

Some federated query engines like SPLENDID[22] and WoDQA use VoID descriptions for constructing federated SPARQL query. In this study VoID descriptions of datasets are used because WoDQA makes use of these metadata for fast query analysis and federated query generation. Since SPARQL endpoints do not need to have a VoID description, an alternative query federation engine which does not based on VoID descriptions like FedX[48], ANAPSID[3], FEDRA[38] or HiBISCUS[46] can be used instead of WoDQA. But this may affect the generation time of the federated query a little. There are some other works about SPARQL query federation which are not mentioned here because they do not have any downloadable source code or binary release in order to be used by any newly developed system. On the other hand VoID descriptions about SPARQL endpoints are used to store DaDy change frequencies. Also if a dataset has no VoID description then a default change frequency can be set to maintain the monitoring job.

*VI. Answers to Requirements Satisfaction Questions*

First requirement satisfaction question is that does the artifact isolate developers from the details of linked data monitoring? (What assumptions does the artifact make about the linked data monitoring effort required by developer)

System simulation and empirical results show that developers can easily monitor their desired subset of linked data by developing agent applications which issue and register SPARQL queries to MonARCh. In this way the monitoring process is handled by MonARCh also developers do not need to know about details such as detection and notification of changes. Developers should be aware of the linked data environment context, SPARQL syntax, actor model and development of actor based applications compatible with the one (Akka) MonARCh based on.

Another requirement satisfaction question asks how much time passes between the detection of a query result change and notification of the registered apps and whether it satisfies functional requirements?

When a query executor actor component in MonARCh detects any change it mostly takes 1 millisecond to notify the agent who registers the related query. For consecutive queries there is a tiny delay like 60 to 80 milliseconds between their result notifications. While queries are sent with the 500 bunches in 5 cycles with a delay of 1 minute between each cycle, bursty response time of 500 queries in a bunch is more important than total processing time. Thus it takes 18 seconds between the first query sent and the last notified result for 500 queries. Results are reasonable even if in a simulated environment and scenario.

Last question in this section seeks for how much time passes between a query result change and change detection, and whether it satisfies functional requirements?

When MonARCh is synchronous with dataset updates it takes from 8 to 18 seconds to pass between the result change of a dataset and the change detection by an actor component. But when this synchronization starts to break, detection of changes may get delayed according to DaDY change frequencies of each dataset. Delayed timespan will be between 0 and DatasetChangeFrequency. Configuring DaDy change frequencies is like a double-edged-blade, while increasing frequency binds more work to the system, decreasing causes some changes to be missed. For this reason change frequencies should be properly specified for datasets in order to balance the system workload and the delay time.

## 7. Related Work

The intended context of the system discussed in this paper is the Semantic Web. Since there is no restriction over data changes in the Linked Data cloud, and data sources are not obliged to inform their users about them, data consumers do not have exact knowledge about updates of data sources. That's the point where the proposed system contributes; it monitors for data changes and informs its clients about these changes. Although the requirements, purpose and structure of this system are very similar to those of publish/subscribe (pub/sub) systems, it differs from pub/sub systems in its change detection mechanism to detect the aforementioned changes in the Linked Data served by SPARQL endpoints.

PubSubHubbub [21], is a web based publish/subscribe protocol that supports both pull and push based approaches. The protocol can be applied with or without the cooperation of the data provider. Data providers that cooperate with a hub notify it about data changes. When the hub gets a change notification, it delivers this notification to the clients subscribed for monitoring those changes. On the other hand, if the data provider does not cooperate with the hub, the hub queries the data provider periodically and notifies the clients whenever there is a change in the data they are monitoring. SparqlPuSH [40] is a partial implementation of the PubSubHubbub protocol for RDF data stores. It implements the push side of the PubSubHubbub protocol that utilizes data provider support. Through the interface it provides, the users register SPARQL queries which they want to monitor on a single RDF data store. System creates a feed about the registered SPARQL queries and registers the feed to a hub. Whenever a change occurs on the RDF triple store, the registered SPARQL queries executed on the data store, feeds related with the SPARQL queries whose results have changed are updated and the hub is notified. Finally, the hub notifies all subscribers of that query about the change.

DSNotify [41] is another publish/subscribe system which can be used in either a pull based or a push based manner, not in a combined style. Users can define a special monitoring area which corresponds to an RDF graph. Whenever a change occurs, the change is registered to the system. According to the preferences of the users, the changes are either sent to the subscribers in a push based manner, or the users can query the changes via http requests in a pull based manner. Like SparqlPuSH, DSNotify can monitor queries on a single RDF endpoint, and does not support federated queries.

SEPA [43] ,which is a publish/subscribe system as well, is based on recent research about smart space applications. It detects data changes by means of its publishers that fully support the SPARQL 1.1 Update language. In addition to that, the study proposes the SPARQL 1.1 Secure Event Protocol and SPARQL 1.1 Subscribe Language that are supported by the publishers and subscribers in the system.

Unlike the systems discussed above, state of the art publish/subscribe systems are scalable and have excellent features and techniques for delivering relevant content to the consumer. In the literature, there are two main approaches available for these systems which are attribute (topic) based and content based.

Topic-based approach narrows search space, that's why systems that follow this approach are both scalable and fast. Among topic based approaches BlueDove [34] is the first one which is designed to work in a cloud computing environment. System has been implemented by extending the source code of Cassandra[17][32] excluding the storage parts which facilitates scalability and elasticity. For handling scalability, the system uses gossip based network at hardware level, and at software level it uses multidimensional partitioning technique to match messages with the related server. System also checks the data skew and adapts itself to changing workloads. SEMAS [36] which is inspired from BlueDove is also an attribute based event matching service designed to work on cloud environments. In this work, authors point out to the skewed real time data dissemination especially in emergency applications. Adapting to change of the burst of event workloads is the main purpose of the system which contains scalable and novel event matching & routing algorithms in its core. Evaluations were performed upon their OpenStack-based testbed by taking into account several viewpoints such as scalability, load balancing, reliability, elasticity and overhead. The topic based systems discussed so far run in cloud environments. In contrast, POLDERCAST [49] runs over a scalable computer network overlay. It is a scalable topic-based publish subscribe service which aims fast dissemination of topics. Novelty of this work is to propagate messages over the network using ring, vicinity and cyclon modules in a scalable manner. Evaluation of the system was performed using open Facebook and Twitter datasets by comparing themselves with Scribe [45] system. Kafka [31], which was first designed as a distributed messaging system for log processing, is capable of much more as a distributed streaming platform and can be accepted as a publish/-subscribe system. The collection of a particular stream of records is defined as a high level abstraction named as a "topic". These published messages and/or streams are then stored in "brokers" each of which is actually a Kafka cluster formed by a set of server nodes. Consumers can subscribe to one or more topics and pull a stream of data from brokers. Kafka is so scalable and fault tolerant that according to the evaluation results its performance far exceeds ActiveMQ[18][50] and RabbitMQ[19][17]. It was first developed by Linkedin company and today is used by thousands of companies in production. POLDERCAST and Kafka are not categorized as cloud-based approaches in their work but both can easily be adapted to work in a cloud environment.

On the other hand content based approach focuses on expressiveness and extends the term scope. Sim-

[17]https://cassandra.apache.org
[18]http://activemq.apache.org
[19]https://www.rabbitmq.com

ilarity metrics and techniques are incorporated into the event matching algorithms to find out how close a content is to a published message, which can be a scalability trade off and lead to system slow down. STREAMHUB [7] is one of the content based approaches aiming high throughput and runs on a public cloud environment. In this work the publish/subscribe engine is split into many logical and operational pieces to support parallel execution on cloud environments. Evaluation was performed on a cluster with 384 cores. According to the results, they achieved a balance between publications and subscriptions with a ratio of 1/100 and scale of 1000. Previous STREAMHUB has been enhanced to become elastic and scalable both in and out as E-STREAMHUB [8]. The system evaluation results showed that it can react and adapt dynamically to changing workloads. Another cloud and content-based pub/sub system is SREM [35] that aims maximizing matching throughput under a big load of subscription requests. They build a distributed overlay SkipCloud using a prefix routing algorithm to achieve low latency. For checking the scalability, reliability, workload balance and memory overhead abilities of the system, a testbed called CloudStack[20] was configured and incorporated into the system evaluation. Unlike these cloud & content based pub/sub architectures [44] presents a semantic pub/sub architecture named SPS to support smart space applications in IoT. Publishers use SPARQL update queries to make changes to the endpoint. The aggregator component of the system uses SPARQL queries to retrieve and store the results of the update query. All query results are stored in an in-memory RDF triple store. It is checked from a key-value lookup table if the updated triples will change the result of a query. Events are detected by a novel algorithm that resides in the SPARQL subscription (SUB) engine. Evaluation of the system was performed against a benchmark of city lighting use case. Results are shown in terms of processing power of subscriptions per second (subs/sec) metric for both simple and complex updates.

## 8. Conclusion and Future Work

In this paper, MonARCh which is a scalable monitoring tool for SPARQL queries in the linked data environment has been presented. Users subscribe SPARQL queries to MonARCh and get notified when a result change occurs related to any subscribed query. It is designed as a pull/push approach proactively executing sub queries over endpoints according to DaDy change frequencies (pull) and constructing the new result and notifying related clients when a change has been detected (push). System has been implemented according to design science methodology by using Akka actor model toolkit and its cluster sharding feature for taking advantage of scalability and concurrency. MonARCh is novel in terms of being the first work that enables scalable SPARQL query monitoring in the linked data environment. On the other hand when both working as a cluster and reaching a high number of query & actor counts have been considered, the empirical results showed that MonARCh is scalable.

In future direction it is planned to extend the architecture enabling to monitor RESTful and multi model database services such as graph, relational, document, key-value, columnar etc. Moreover, the persistence feature of Akka is planned to be included into MonARCh which lets the actor system to self heal when a node or cluster has been shut down.

The detailed linked data consumption survey by Klimek et. al [30] states that learning linked data technologies is a problem for users. Though MonARCH supports developers greatly for query monitoring, it still requires the developers to write SPARQL queries which may pose a problem for the developers not familiar with SPARQL language or linked data tools. There are some machine learning based studies for this problem [39, 42, 51, 52], however none of them are integrated with a tool like MonARCH. Therefore, another direction for further research may be automatized translation of queries given in natural language to SPARQL.

One last future direction for research is defining the change frequencies of data sources whose DaDy definitions are not given. There are some studies that aim analyzing and defining the dynamics of linked data sets [16]. These studies may be combined with MonARCH to define DaDy metadata for data sources with no dynamics information in order to allow MonARCH to monitor them too.

## Acknowledgements

The initial version of the linked data monitoring architecture presented in this article has been developed within the context of a work supported by The Sci-

[20]http://cloudstack.apache.org

entific and Technological Research Council of Turkey (TÜBİTAK) under grant no 111E027.

## References

[1] Dbpedia. https://wiki.dbpedia.org. Accessed: 2020-06-16.

[2] Linked Open Data Cloud. https://lod-cloud.net. Accessed: 2020-06-16.

[3] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. Anapsid: an adaptive query processing engine for sparql endpoints. In *International Semantic Web Conference*, pages 18–34. Springer, 2011.

[4] Z. Akar, T. G. Halaç, E. E. Ekinci, and O. Dikenelli. Querying the web of interlinked datasets using void descriptions. *LDOW*, 937, 2012.

[5] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets. 2009.

[6] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[7] R. Barazzutti, P. Felber, C. Fetzer, E. Onica, J.-F. Pineau, M. Pasin, E. Rivière, and S. Weigert. Streamhub: a massively parallel architecture for high-performance content-based publish/subscribe. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 63–74, 2013.

[8] R. Barazzutti, T. Heinze, A. Martin, E. Onica, P. Felber, C. Fetzer, Z. Jerzak, M. Pasin, and E. Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 567–576. IEEE, 2014.

[9] T. Berners-Lee. Linked data, 2006. http://www.w3.org/DesignIssues/LinkedData.html.

[10] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[11] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Transactions on Computers*, 51(6):652–668, 2002.

[12] C. Bizer, T. Heath, and T. Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global, 2011.

[13] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. Dbpedia-a crystallization point for the web of data. *Journal of web semantics*, 7(3):154–165, 2009.

[14] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *Proceedings of the 10th international conference on World Wide Web*, pages 265–274, 2001.

[15] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 1–8, 1984.

[16] R. Dividino, T. Gottron, A. Scherp, and G. Gröner. From changes to dynamics: Dynamics analysis of linked open data sources. In *CEUR Workshop Proceedings*, volume 1151. CEUR Workshop Proceedings, 2014.

[17] D. Dossot. *RabbitMQ essentials*. Packt Publishing Ltd, 2014.

[18] O. Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[19] O. Erling and I. Mikhailov. Virtuoso: Rdf support in a native rdbms. In *Semantic Web Information Management*, pages 501–519. Springer, 2010.

[20] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[21] B. Fitzpatrick, B. Slatkin, M. Atkins, and J. Genestoux. Pubsubhubbub core 0.4. working draft, pubsubhubbub w3c community group (2013).

[22] O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *Proceedings of the Second International Conference on Consuming Linked Data-Volume 782*, pages 13–24. CEUR-WS. org, 2011.

[23] M. Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[24] M. Hausenblas. Dataset dynamics (dady) vocabulary, January 2010. URL http://vocab.deri.ie/dady.

[25] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, volume 2007, pages 132–141, 2007.

[26] C. Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[27] C. Hewitt, P. Bishop, and R. Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

[28] H. Huang and L. Wang. P&p: A combined push-pull model for resource monitoring in cloud computing environment. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 260–267. IEEE, 2010.

[29] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[30] J. Klímek, P. Škoda, and M. Nečaskỳ. Survey of tools for linked data consumption. *Semantic Web*, 10(4):665–720, 2019.

[31] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[32] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[33] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[34] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei. A scalable and elastic publish/subscribe service. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 1254–1265. IEEE, 2011.

[35] X. Ma, Y. Wang, and X. Pei. A scalable and reliable matching service for content-based publish/subscribe systems. *IEEE transactions on cloud computing*, 3(1):1–13, 2014.

[36] X. Ma, Y. Wang, Q. Qiu, W. Sun, and X. Pei. Scalable and elastic event matching for attribute-based publish/subscribe systems. *Future Generation Computer Systems*, 36:102–119, 2014.

[37] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[38] G. Montoya, H. Skaf-Molli, P. Molli, and M.-E. Vidal. Federated sparql queries processing with replicated fragments. In *International Semantic Web Conference*, pages 36–51. Springer, 2015.

[39] P. Ochieng. Parot: Translating natural language to sparql. *Expert Systems with Applications: X*, 5:100024, 2020.

[40] A. Passant and P. N. Mendes. sparqlpush: Proactive notification of data updates in rdf stores using pubsubhubbub. In *SFSW*, 2010.

[41] N. Popitsch and B. Haslhofer. Dsnotify–a solution for event detection and link maintenance in dynamic datasets. *Journal of Web Semantics*, 9(3):266–283, 2011.

[42] C. Pradel, O. Haemmerlé, and N. Hernandez. Natural language query interpretation into sparql using patterns. In *Proceedings of the Fourth International Conference on Consuming Linked Data-Volume 1034*, pages 13–24. CEUR-WS. org, 2013.

[43] L. Roffia, P. Azzoni, C. Aguzzi, F. Viola, F. Antoniazzi, and T. Salmon Cinotti. Dynamic linked data: A sparql event processing architecture. *Future Internet*, 10(4):36, 2018.

[44] L. Roffia, F. Morandi, J. Kiljander, A. D'Elia, F. Vergari, F. Viola, L. Bononi, and T. S. Cinotti. A semantic publish-subscribe architecture for the internet of things. *IEEE Internet of Things Journal*, 3(6):1274–1296, 2016.

[45] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *International workshop on networked group communication*, pages 30–43. Springer, 2001.

[46] M. Saleem and A.-C. N. Ngomo. Hibiscus: Hypergraph-based source selection for sparql endpoint federation. In *European semantic web conference*, pages 176–191. Springer, 2014.

[47] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *International Semantic Web Conference*, pages 585–600. Springer, 2011.

[48] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *International semantic web conference*, pages 601–616. Springer, 2011.

[49] V. Setty, M. Van Steen, R. Vitenberg, and S. Voulgaris. Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 271–291. Springer, 2012.

[50] B. Snyder, D. Bosnanac, and R. Davies. *ActiveMQ in action*, volume 47. Manning Greenwich Conn., 2011.

[51] T. Soru, E. Marx, A. Valdestilhas, D. Esteves, D. Moussallem, and G. Publio. Neural machine translation for query construction and composition. *arXiv preprint arXiv:1806.10478*, 2018.

[52] N. Steinmetz, A.-K. Arning, and K.-U. Sattler. From natural language questions to sparql queries: a pattern-based approach. *BTW 2019*, 2019.

[53] R. J. Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[54] W. S. working group et al. Sparql 1.1 overview. *World Wide Web Consortium*, 2013.