Linked-data cache with provenance and optimistic execution of linked-data queries

by

I-Hsun Liao

Thesis
Presented to the Faculty of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Bachelor of Science in Computer Science: Turing Scholar Honors

The University of Texas at Austin
May 2016
Abstract

Linked-data cache with provenance and optimistic execution of linked-data queries

I-Hsun Liao, B.S. Computer Science
The University of Texas at Austin, 2016

Supervisor: Daniel P. Miranker

Abstract: Diamond is a linked-data query system based on the Rete-match algorithm. In this paper we introduce the architecture of the Diamond cache. The Rete-match, originally intended for the implementation of forward-chaining rule systems, is able to incrementally and non-monotonically evaluate queries. Thus, we introduce the optimistic execution of linked-data queries. That is, when possible, dereferenced URI’s are concurrently fetched from cache and dereferenced. The query system quickly computes results from cached contents. Subsequently, triples returned by dereferenced URI’s are compared with cached copies. If the cached copies are current no action need be taken. Then, we introduce eventual execution, which is an extension of optimistic execution. In eventual execution, if the aforementioned cached triples have become stale, the current version of the triples are forwarded to the Rete network and the query result updated accordingly. The cache is built using a conventional triple-store. The cache contents are modeled using the standard open provenance model. The expectation is that in future versions of the system the provenance of individual triples may be integrated into the behavior of the cache replacement algorithm.
# Contents

1 **Introduction**  
   1.1 Motivation .......................................................... 1  
   1.2 Contribution ....................................................... 3  
   1.3 Levels of Caching ................................................ 3  
   1.4 Section Overview ................................................ 4  

2 **Background**  
   2.1 RDF ................................................................. 5  
   2.2 SPARQL .......................................................... 6  
   2.3 Standard Open Provenance Model .............................. 7  
   2.4 Rete match ..................................................... 7  

3 **Related Works** .......................................................... 10  

4 **Provenance Cache**  
   4.1 Main Purpose ...................................................... 12  
   4.2 Provenance Information ............................................. 12  
   4.3 Use for Provenance Information .................................. 12  

5 **Levels of Cache**  
   5.1 Live ............................................................ 14  
   5.2 Cached ............................................................ 14  
   5.3 Optimistic Cache .................................................. 14  
   5.4 Eventual cache .................................................... 14  

6 **Calculating difference between two triple sets**  
   6.1 Separate blank triples and non-blank triples ............. 15  
   6.2 Calculate difference between non-blank triple sets .......... 15  
   6.3 Calculate difference between blank triple sets ............ 16  
      6.3.1 Separate triple set into subgraphs .......................... 17  
      6.3.2 Match subgraphs ............................................ 18  
      6.3.3 Find differences between matched subgraphs ........... 19  
      6.3.4 Tying all three steps together .............................. 19  

7 **Evaluation**  
   7.1 Comparing live and cached .................................... 21  
   7.2 Comparing all four levels of cache .......................... 22  

8 **Conclusion**  
   8.1 Accomplishments .................................................. 25  
   8.2 Future work ..................................................... 25  

References ................................................................. 26  

Appendix I ................................................................. 28  

Appendix II ................................................................. 30
# List of Figures

1. Example RDF graph .......................................................... 2
2. Example SPARQL query .......................................................... 2
3. Diamond Architecture Diagram ............................................. 3
4. RDF triple ........................................................................ 5
5. An RDF graph with two URIs and multiple triples ....................... 6
6. Query for RDF graph ........................................................... 6
7. Example SPARQL query for Rete network ................................. 8
8. State of the Rete network after a triple is inserted ..................... 9
9. State of Rete network after multiple triples are inserted ............. 9
10. Provenance cache model ...................................................... 13
11. Pseudocode for equality of RDF triples .................................. 16
12. Pseudocode for removing the same triples from two triple sets .... 16
13. Blank node identifier comparison ......................................... 17
14. Pairwise matching of blank triples sets .................................. 17
15. Group nodes pseudocode ..................................................... 18
16. Matching subgraphs ........................................................... 19
17. Fedbench queries execution time ........................................... 21
18. Fedbench speed up ............................................................. 22
19. CityBench queries execution time ........................................ 23
1 Introduction

In recent years, the Internet has connected vast number of documents in the form of websites and hyperlinks. Web crawlers index these websites so that when users search for information, they can direct users to relevant pages. However, all the information stored on the web is documents. More recently, the web also connects structured data. The set of practices for publishing and connecting structured data on the web is called Linked Data. The adoption of linked data has created a web of data where all kinds of information, such as facts about people, places, scientific publications, and proteins, get stored. Similar to the web of documents, this web of data also requires tools to give access for people or machines. Linked data browsers allow users to browse one datum and jump to other linked data connected to it. Linked data query engines take user queries and crawls through the web of data to find data matching the query.

In other words, linked data is “about using the Web to create typed links between data from different sources.” Linked data are data published on the web that is machine-readable with meaning defined explicitly. It can link to external datasets, and can let external datasets be linked to it.

1.1 Motivation

When traversing the web of linked data, one often starts from a URI. This URI contains references to other links and information about the URI itself. This information comes in the form of an Resource Description Framework (RDF) graph. An small example can be seen in Figure 1. In addition, RDF will be explained in detail in Section 2.1. All RDF graphs are directed and labeled and composed of sets of edges known as triples. The act of retrieving the graph associated with a URI is called dereferencing. From now on, we will say triples were dereferenced from a URI, where triples refer to the edges of a RDF graph. Since the speed of Internet is nowhere as fast as the speed of local processing, dereferencing URIs is often the most time-consuming process when traversing linked data.

When evaluating a query, in an effort to reduce the time used for dereferencing, we decided to build a cache to store the previously dereferenced triples. As an example, the dereference of a URI, u, yields two triples, a and b. The cache will store all three pieces of information, u, a, and, b, after the dereference. The next time u is scheduled to be dereferenced, the cache will return a triple set \{a, b\} as the result without dereferencing u again. This will significantly reduce the time spent on dereferencing URIs.

If we build a linked cache as described previously, cached data may become stale. Assume the same u from before. If u changes its triple set to \{a, c\} during the time between the first
and second dereference, the cache will have the wrong triples for \( u \). In this case, we need some kind of protocol to dereference \( u \) again and update the cache.

Other than speeding up linked data traversals, a linked data cache may also give semantics about the data we are traversing. For example, if an URI has been dereferenced repeatedly in a period of time, one may be able to find out more information about this data source. If dereferencing the URI always returns the same result as its cached counterpart, then the data is very likely to be static and vice versa. Using this information, a machine may decide on its own to continue to cache data from this source or not.

The linked data network, like a database, can be queried using a language called SPARQL (SPARQL Protocol and RDF Query Language). An example can be found in Figure 2. This query attempts to find everyone who knows someone that lives in Austin. After executing this query on the RDF graph in Figure 1, one will get variable \( x \) as \( \text{http://.../i-hsun} \) and \( y \) as \( \text{http://.../rick} \). Intuitively, these two URIs are the answer because they match the pattern described in the \textbf{WHERE} clause. We will explain SPARQL in finer detail in Section 2.2.

```sparql
SELECT ?x WHERE {
}
```

Figure 2: Example SPARQL query

Executing a query on a linked data network is a common action to explore linked data. A query often contains several URIs to be dereferenced. If we can give semantics to these URIs as described in the previous paragraph and link these URIs to the query, then we can give semantics to the query. In order to make this linkage, we created a cache with provenance information (detailed in Section 4).
1.2 Contribution

Diamond is a linked-data query system based on the Rete match algorithm. Given a SPARQL query, Diamond will execute the query and return a set of results. The complete architecture of Diamond can be seen at [12] or Figure 3. The research described by this thesis is the in-memory cache that was added onto the system. Diamond discovers data by dereferencing each resource it came across in the query. A resource, by convention, will be a URI. Therefore, by dereferencing a URI, Diamond receives the RDF document that contains a set of triples that exist in the URI. The job of the cache is to store these triples and reference them to a specific URI. This thesis describes, implements, and compares four types of cache execution: live, cached, optimistic, and eventual.

All types of cache will be a direct extension of the open standard provenance model. By following this model, the cache will store a lot more information than necessary to perform the basic task. However, this will eventually help give semantics to URIs and queries that are executed with the help of this cache. We will discuss how we utilized provenance in the caches and how it will advance the caching.

1.3 Levels of Caching

Live simply means no cache will be used in the execution of a query (the data is live). Cached means the cache is warm or already populated with previously dereferenced triples. For example, if dereferencing $u1$ gives back two RDF triples: $\langle s1, p1, o1 \rangle$ and $\langle s2, p2, o2 \rangle$, 
then cache will store the information. Everytime \( u1 \) is dereferenced Diamond will first check if \( u1 \) exist in the cache. If it does, the cache will return \( \langle s1, p1, o1 \rangle \) and \( \langle s2, p2, o2 \rangle \).

An optimistic execution of the cache means given a query, it will first be executed with the use of a cache like before and return the result (Diamond is “optimistic” that the data in URIs have not changed since the last dereference). Then, Diamond will dereference every URI without using the cache. Diamond will calculate the difference between the two sets of triples for each URI: the triples inside the cache and the newly dereferenced triples. If there is any difference Diamond will notify the user, and the execution ends here. If the execution is an eventual one, Diamond will do all of optimistic execution. Further, using the difference between two triple sets, Diamond will update its cache and Rete network, which returns the correct result of the query efficiently.

Implementing optimistic and eventual cache required us to know the difference between the cached RDF triple set and the newly dereferenced triple set. Finding this difference is more complex if the triples contain “blank node.” Blank node is a subject or object in an RDF triple such that it only denotes the existence not the identifier. Unfortunately, blank nodes do exist in our test dataset. Therefore, we will discuss, in depth, the complications of finding difference between two triple sets when blank node is included and how we solved it.

Using each level of cache, we executed multiple test queries and measured the execution times. From these times, we compared each type of cache and discussed its qualities.

1.4 Section Overview

Section two will talk about the background regarding RDF, provenance model, and Rete network. Section three will mention related work from the literature. Section four will describe the structure of the cache, including its provenance properties. Section five will talk about the four levels of cache in depth. Section six will discuss how we find the difference between two triple sets. Section seven will introduce our evaluation plan and show the result of the evaluation. Section eight will conclude the research and talk about future directions.
2 Background

2.1 RDF

While the primary language used to present documents in the web is HTML, RDF (Resource Description Framework) is used to represent linked data. Linked data uses RDF to literally link data together. Berners-Lee [2] has provided general principles when publishing linked data:

1. Use URIs as names for things
2. Use HTTP URIs so people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)
4. Include links to other URIs, so that they can discover more things.

Abstractly, RDF is a graph. The graph is composed of sets of edges, known as triples. Each triple (edge) has three elements: subject, predicate, and object as shown in Figure 4. By convention, an edge is defined by two vertices, therefore subject and object will always be a vertex, and predicate is a label on the edge connecting the vertices. As suggested by Berners-Lee, each of these elements should be represented by HTTP URIs. In addition, the object can also be a literal such as a string or number.

An example of a small RDF graph can be seen in Figure 5. Suppose one tries to dereference http://dbpedia.org/isbn978. The dereference will yield one triple (http://dbpedia.org/isbn978, sameAs, http://semanticweb.org/isbn978). Now to make the traversal more complete, the person dereferences the http://semanticweb.org/isbn978. This time it yielded two triples: (http://semanticweb.org/isbn978, title,"Programming the Semantic Web") and (http://semanticweb.org/isbn978, author,"Toby Segaran").

Aside from being URIs, the subject and object of a triple can be a blank node. A blank node is neither a URI nor a literal. It has no global identity, and its identifier is only valid in the local scope. If a blank node has identifier _:`b1 in graph G, there might be another...
A blank node denotes the existence of a thing without saying anything about the name of it. Suppose there exist a triple $\langle \_:b1, p, o \rangle$ in graph $G$, then the blank node with identifier $\_:b1$ tells us there exist something relating to $o$ using $p$. There may or may not be a URI reference to this thing, but it does exist somewhere.

### 2.2 SPARQL

Much like a database, RDF graphs can be queried. The query language is called SPARQL Protocol and RDF Query Language (SPARQL). Most SPARQL query consists of a set of triple patterns called basic graph pattern. A triple pattern looks similar to an RDF triple except each of the subject, predicate, or object of a triple pattern may be a variable. A variable in SPARQL starts with a ? followed by an identifier. The main job of evaluating a SPARQL query is to find variables that matches the graph pattern specified by the query.

```
SELECT ?x ?y WHERE {
  ?x :sameAs ?y.
  ?y :title "Programming the Semantic Web".
  ?y :author "Toby Segaran"
}
```

The SPARQL query in Figure 6 tries to find out information regarding the RDF graph in Figure 5. The query contains two clauses. The `WHERE` clause has three triple patterns that make up a connected graph pattern. The `SELECT` clause signifies the client wants to learn the values of variables $x$ and $y$ such that they match the graph pattern in the `WHERE` clause.
Using the information from Figure 5, $x$ will be http://dbpedia.org/isbn978 and $y$ will be http://semanticweb.org/isbn978. There can be multiple distinct values of $x$ and $y$ that matches this graph pattern. SPARQL will return all qualified value as the result set.

2.3 Standard Open Provenance Model

The cache is built based on the Open Provenance Model (OPM). The provenance of an object represents its origins. Provenance contains descriptions of activities that involved in creating or producing the object. There are many uses of provenance. One example would be understanding how data was collected so that it can be used in a meaningful way. Here the “object” being created or produced is clearly the data. The provenance may be the source of where the data is collected and the time of when it is collected.

The standard open provenance model contains three types: entity, agent, and activity (see the top part of Figure 10). According to Moreau and Missier [13], an entity is a physical, digital, conceptual, or other kind of thing with some fixed aspects; entities may be real or imaginary. Put simply, an entity is an object. This object can be anything from URI to dereferenced RDF triples. An entity can be derived from one form to another. For example, a derivation can be a transformation from linked data graph to relational database.

An activity acts upon the entity over a period of time. It can be consuming, processing, transforming, modifying, relocating, using, or generating entities. [13] In general, an activity has two actions for entities. It can utilize or generate entities. For each action there maybe a duration, hence there can be a start and end time for an activity. The generation of an entity means the entity is produced and was not available before generation. Usage of an entity is the beginning of utilizing it. Entity must not have been used before the usage.

Finally, an “agent is something that bears some form of responsibility for an activity taking place, for the existence of an entity, or for another agent’s activity.” [13] An agent can be associated with an activity. This means the agent is responsible for the occurrence of the activity. An entity can also be attributed to an agent. When an entity is attributed to an agent, the entity was generated by an activity that is associated with the agent.

2.4 Rete match

The Rete match is the primary algorithm used, by Diamond, to solve queries. Rete match contains a network of filter nodes and memory nodes. [12] Thus, we call this system the Rete network. Memory nodes can be thought as a storage for solutions to a subset of a query. Solutions that come from unary operators are stored in as alpha memory and solutions that come from binary operators are stored in as beta memory. We are going to explain the how Rete network operates with the following query (Figure 7).
SELECT ?age WHERE {
}

Figure 7: Example SPARQL query for Rete network

Execution of a query starts with dereferencing a URI from the query. For simplicity, dereferencing http://cs.utexas.edu/miranker returns a set of triples, and one of those triples is ⟨http://cs.utexas.edu/miranker, :knows, http://web.ing.puc.cl/arenas⟩. After inserting this triple in Rete network, it will be filtered by triple pattern test and get stored in the corresponding alpha memory (Figure 8). The existence of ⟨http://cs.utexas.edu/miranker, :knows, http://web.ing.puc.cl/arenas⟩ in one of the alpha memory means there might be more data from http://web.ing.puc.cl/arenas. Again for simplicity, dereferencing http://web.ing.puc.cl/arenas returns a set of triples including ⟨http://web.ing.puc.cl/arenas, :age, "28"⟩. Figure 9 shows the state of Rete network after the insertion of this set of triples. The beta memory resulting from the join of two alpha memories contains triples that match the SPARQL query, thus beta memory contains the solution to this query.

A quanta of information that passes through Rete network is called token. A plus token adds information to Rete network. In contrast, minus token removes information. In the above example, all triples were inserted as plus token, resulting in more information in Rete network. A minus token can also be inserted to remove information. Continuing the above example, if http://web.ing.puc.cl/arenas increased its age to 29, one may need to insert a minus token of ⟨http://web.ing.puc.cl/arenas, :age, "28"⟩ and insert a plus token of ⟨http://web.ing.puc.cl/arenas, :age, "29"⟩.
Figure 8: State of the Rete network after a triple is inserted. [12]

Figure 9: State of Rete network after multiple triples are inserted and solution is found. [12]
3 Related Works

Although this is a first use of using provenance in a cache for linked data, there have already been results for building cache for linked data and linked data with provenance.

Hartig [8] discussed the potential benefit of a cache in a linked data query execution system. Similar to that of Diamond, Hartig’s cache also stores dereferenced data from a URI. The first benefit is possible reduced execution time by reducing the amount of time required to execute a query. [8] Query execution time is primarily affected by the number of URIs that need to be dereferenced and the amount of time that it takes to dereference the URI. A cache that stores dereferenced data from a URI can greatly reduce the latter factor. The experiment Hartig did verified the hypothesis. The “higher number of URI look-ups needed . . . increased the amount of delays caused by these look-ups” resulting in significantly longer execution time. In 47 of the 115 test queries the difference was larger than 30 seconds.

A second benefit is potential increase in result completeness. A single query with initially no knowledge on the topic may have unreachable resources. A sequence of queries can accumulate the data on this topic and reveal extra results. For example, suppose a query, q2, wants to find out the name of the people that person p knows. However, the dataset that resides in p is not complete and does not describe person m that p knows. Perhaps, a previous query, q1, that asked about m found out that p knows m and cached the result. By executing q1 before q2 and with the help of a cache, we now find out an extra person that p knows. 53 out of the 115 queries tested showed increase in result completeness. These queries create scenarios that are similar to the one described.

Williams and Weaver [17] took advantage of HTTP primitive caching to reduce execution time of SPARQL queries. HTTP caching works as follows: suppose a client wants to execute a query, it would send the query to the server with a http GET. In return, the server will send back a response with a Last-Modified validator and a date in the header, indicating when the result were last modified. In the future, if the client wants to execute the same query again, it can add If-Modified-Since header with the previously obtained date. If the query result has not changed since the date, the server can simply return some text indicating the result had not been modified. A server that executes SPARQL queries usually contains a database for storing RDF data. Williams and Weaver made modification to the database indexes to determine the modification time of data relevant to a query.

Hartig [7] also created a provenance model for storing provenance regarding creation of data. The model they came up with differs slightly from the standard Open Provenance Model [6], but also has the the three main types, which they call actor, execution, and arti-
fact. They correspond to agent, activity, and entity for OPM respectively. Some information that is stored in Hartig’s provenance model include data creation time, source data, data creator, etc. One use of this model would be to give authenticity to the data being created.

Patni et al. [15] built a framework that contains linked sensor data and their provenance. These linked sensor data contains data from about 20,000 weather stations in the United States. The data originated from MesoWest, which is a project of the Department of Meteorology at the University of Utah. Each weather stations contain sensors that measure temperature, wind speed, humidity, etc. Provenance regarding these data includes “location of the sensor, the time when the observations were taken by the sensor and the sensor observation values.” [15] After data collection, provenance information are stored in the Virtuoso RDF store, an open source triple store. [14] The provenance model by Patni et al. differs from Hartig [7] or OPM [6], but still contains the three basic types: data, agent and process, corresponding to entity, agent, and activity in OPM respectively.
4 Provenance Cache

4.1 Main Purpose

The linked data cache stores dereferenced triples for URIs. It is built using Sesame, a general purpose triple store that exist in memory. [5] For each URI, it stores a list of RDF triples dereferenced by the URI. In the context of the cache model, a dereference is the act of giving the cache a URI and receiving a set of RDF triples as the result. A set of triples can be represented as a graph because triples are very likely to be interlinked.

4.2 Provenance Information

The linked data cache we built on Diamond is a direct extension of the Open Provenance Model (OPM). All problems solved by Diamond start from a SPARQL query. Usually this query contains several URIs to be dereferenced. Therefore, the existence of this query prompts the activity of dereference. According to the OPM, the query is an agent (see Figure 10). After making a dereference, the cache returns a set of triples represented by a graph as described in the previous paragraph. The query takes responsibility of the existence of this graph, making the graph an entity.

Not only did the activity of dereference have a connection with the agent, it also has a relationship with multiple entities. In OPM, an activity can generate or utilize entities. In our provenance cache model, there is another entity that is the URI that the cache dereferences. Unsurprisingly, this URI is “used” by the activity of dereference, completing the utilization part of the relationship between activity and entity. Dereferencing a URI also “generates” a graph of triples, completing the generation relationship.

To perform its basic functions, the cache has to store the URIs and graphs of dereferenced triples. To be a provenance cache, it also needs to store the query where the URI is contained, and make a link between them. Trivially, the query is stored in the form of a string called “Query Text.” In addition, the cache also stores the amount of time it took to dereference a particular URI, denoted by “Start” and “End” in the provenance cache model.

4.3 Use for Provenance Information

Provenance links the query with dereferenced triples and the original URI. This allows for semantics to be given to the query based on the result of its dereferenced URIs. An example would be the dynamism of the query. If all of the URIs for a particular query contains a different set of triples for a period of time, one can assume that this query has rather
dynamic result. The next time Diamond has to dereference a URI for this query, it does not need to store the dereferenced result into the cache as the result will likely change soon. Conversely, if all URIs in a query represent static data, Diamond should make sure it stores the dereferenced result into the cache. Note this optimization will not be possible if there is no linkage between individual URI and their query.
5 Levels of Cache

5.1 Live
Live means the data is live. There is no cache to store dereferenced triples or provenance information. Diamond is used as is as described by Miranker et al. [12]

5.2 Cached
A provenance cache as described in Section 4 is used to store various information during an URI’s dereference. One of the most important pieces of information, among other provenance, that a cache stores are the triples corresponding to a dereferenced URI. When the cache is empty, it is described as cold. When it already contains dereferenced triples from an URI, it is considered as warm. A warm cache means Diamond does not have to dereference the URI again to get its triples, because it is stored in the cache. In this level, Diamond assumes the information stored in cache is always up to date. In a cached execution, Diamond will utilize a warm cache.

5.3 Optimistic Cache
In this level, the same provenance cache is used. When Diamond is ordered to execute a query, it first assumes the data in the cache, if available, is up to date. It returns whatever it gets to the client. If Diamond used the cache’s data to complete the execution, it will dereference every URI encountered during the execution again, ignoring the cache. If the dereferenced triples are different from the cached triples, it will calculate the difference between the two different triple sets. The process of calculating the difference is detailed in Section 6. Diamond will report to the client if there are any difference. Using the information, the client will know if the previously given result is up to date.

5.4 Eventual cache
An eventual execution of cache is a direct extension of the optimistic cache. Eventual cache starts off by doing everything an optimistic cache does. After finding the difference between cached and newly dereferenced triple sets, Diamond will update the Rete network and the cache with the most recent data. Finally, the updated Rete network will produce a new result and return that to the client.
6 Calculating difference between two triple sets

One of the hardest problem in this research is to determine the difference between two triple sets. The difficulty will be explained shortly in Section 6.3. One of the sets comes from dereferencing a URI through the cache. The other will come from dereferencing a URI through the web. Differences between the two dereferenced sets indicate if the contents of the cache is stale. If Diamond is using an optimistic cache, it can use this information to let the client know if the result they receive is current. In the case of eventual cache, the difference of the sets is required to update the cache and Rete network.

Define a blank triple to be an RDF triple that contains at least one blank node. Further, let’s define a blank triple set as a triple set such that it only contains blank triples. For the rest of the algorithm, let $C$ be a triple set that is obtained by dereferencing a URI, $U$, using the cache. Let $R$ be a triple set that is obtained by dereferencing a URI, $U$, through linked data.

6.1 Separate blank triples and non-blank triples

Separate the triples in $C$ into two subsets: one that is a blank triple set, $C_b$, and one that does not contain blank triple, $C_n$. Do the same for $R$ to obtain $R_b$ and $R_n$.

6.2 Calculate difference between non-blank triple sets

Using the variables defined previously, this means to calculate difference between $C_n$ and $R_n$. First, we define how to determine if two non-blank triple are equivalent. Recall an RDF triple consists of subject, predicate, and object. Further, subject and predicate is identified by a URI, while object is identified by a URI or literal. Therefore, two triples are equal if the identifiers for their subject, predicate, and object are equal. Comparing identifiers is a trivial string comparison. The pseudocode for triple comparison is shown in Figure 11.

Next, we define the difference of two triple sets to be all the triples in $C_n$ that are not in $R_n$ and all the triples in $R_n$ that are not in $C_n$. This is easily found by removing all the triples in $C_n$ that also exist in $R_n$ and vice versa. The pseudocode is shown in Figure 12.

Let $M_n$ be $\text{remove\_all}(C_n, R_n)$ and $P_n$ be $\text{remove\_all}(R_n, C_n)$. Each of the triples in $M_n$ only exist in the cache and no longer exist in the actual data source. Each of the triples in $P_n$ only exist in the actual data source and are unknown to the cache. If this is an optimistic cache and either $P_n$ or $M_n$ is non-empty, Diamond will notify the client that the
Algorithm 1: triple\_equality

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>s ← t1.subject.equals(t2.subject);</code></td>
</tr>
<tr>
<td>2</td>
<td><code>p ← t1.predicate.equals(t2.predicate);</code></td>
</tr>
<tr>
<td>3</td>
<td><code>o ← t1.object.equals(t2.object);</code></td>
</tr>
<tr>
<td>4</td>
<td><code>return s and p and o</code></td>
</tr>
</tbody>
</table>

Figure 11: Pseudocode for equality of RDF triples

Algorithm 2: remove\_all

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>foreach i ∈ s1 do</code></td>
</tr>
<tr>
<td>2</td>
<td><code>foreach j ∈ s2 do</code></td>
</tr>
<tr>
<td>3</td>
<td><code>if triple\_equality(i,j) then</code></td>
</tr>
<tr>
<td>4</td>
<td><code>s1.remove(i);</code></td>
</tr>
<tr>
<td>5</td>
<td><code>break;</code></td>
</tr>
<tr>
<td>6</td>
<td><code>end</code></td>
</tr>
<tr>
<td>7</td>
<td><code>end</code></td>
</tr>
<tr>
<td>8</td>
<td><code>end</code></td>
</tr>
<tr>
<td>9</td>
<td><code>return s1</code></td>
</tr>
</tbody>
</table>

Figure 12: Pseudocode for removing the same triples from two triple sets

cached result is incorrect.

If this is an eventual cache and either \( P_n \) or \( M_n \) is non-empty, the cache should then turn each of the triples in \( P_n \) into a plus token and insert them into the Rete network. In addition, the cache should associate each triple in \( P_n \) to \( U \). \( M_n \) should be treated the same as \( P_n \) except minus token should be created from its triples and the cache should disassociate each triple in \( M_n \) to \( U \).

6.3 Calculate difference between blank triple sets

In this part we are concerned in finding the difference between \( C_b \) and \( R_b \). Contrary to normal triples, we cannot compare the equality of blank triples using string comparisons. Recall that the identifiers of blank nodes are only valid locally. For example, in Figure 13 assume \( a \) is obtained by dereferencing \( U \) using the cache and \( b \) is obtained by actually dereferencing \( U \). The blank nodes in \( a \) and \( b \) are equivalent, however, because of different blank node identifiers, they would be marked as different by string comparison.

Since simple string comparison does not suffice, we tried to match each triple in \( C_b \) to another in \( R_b \). A match between two triples denotes their equivalence. A triple in \( C_b \) is matched to another triple in \( R_b \) if their non-blank nodes are equivalent. Figure 14 shows an example. Triple 1 is matched to triple \( a \) because they have the same predicate and object
Figure 13: When comparing blank triples, $a$ and $b$ are considered to be equal because they have the same predicate and object. It is fine for blank node identifier to be different because blank node identifiers are only valid on the local scope.

Figure 14: Pairwise matching of blank triples sets

and triple $a$ precedes triple $b$. Similarly, triple 2 is matched to triple $c$ because they have the same predicate and object. However, this matching is semantically incorrect. From the matching, one can infer that $:_b1$, $:_b2$, and $:_b3$ are equivalent. In contradiction, $:_b2$ and $:_b3$ are different blank nodes in $R_b$.

In order to do a semantically correct matching of triples, we need to match blank nodes in the triples instead of triples in each set. We separate each graph into multiple subgraphs such that each subgraph is connected by a common blank node. Then, we match these subgraphs. We describe this process in detail in the following paragraphs.

6.3.1 Separate triple set into subgraphs

A triple set can be viewed as a graph, where each triple represents an edge. This step involves separating the triple set into multiple subgraphs such that each subgraph is connected. Put simply, we will group the triples by same blank nodes.

In this research, we will represent the subgraphs using a hash map. The hash map will map a blank node to a set of triples where each triple has this blank node as one of its nodes. In a latter step (Section 6.3.2), we repeatedly access contents of the hash map.
making constant get time favorable. The pseudocode is in Figure 15.

---

**Algorithm 3: group_nodes**

Input : A set of RDF triples $t$
Output: A map from blank node to triple sets

1. Map(RDFNode, Set(RDFTriple)) $\text{subgraphs}$;
2. foreach $\text{triple} \in t$ do
   3. // Get blank node
      RDFNode $\text{blank}$;
      4. if isBlankNode($\text{triple}.\text{subject}$) then
         5. $\text{blank} \leftarrow \text{triple}.\text{subject}$;
      6. else
      7. $\text{blank} \leftarrow \text{triple}.\text{object}$;
   8. end
   9. // Insert into map
   10. if $\text{blank} \notin \text{subgraphs}$ then
      11. Set(RDFTriple) $\text{temp}$;
      12. $\text{temp}.\text{add}(\text{triple})$;
      13. $\text{subgraphs}.\text{put}(\text{blank}, \text{triple})$;
   14. else
      15. $\text{subgraphs}.\text{get}(\text{blank}).\text{add}(\text{triple})$;
   16. end
17. return $\text{subgraphs}$

---

Figure 15: Pseudocode for grouping triples with the same blank node. We check if $\text{blank}$ is not in $\text{subgraphs}$ by comparing blank nodes’ identifiers. Here this is valid because $t$ contains triples from the same dereference, meaning all the triples come from the same scope.

Assume $\text{isBlankNode}$ to be some function that determines if a node in a triple is blank. Since a blank node identifier always starts with “_:” in RDF, this is what Diamond checks for in the function.

We use $\text{group_nodes}$ on both $C_b$ and $R_b$. Assume $\text{group_nodes}(C_b)$ is $C_b.\text{subgraph}$ and $\text{group_nodes}(R_b)$ is $R_b.\text{subgraph}$. Continuing from the example in Figure 14, Figure 16 shows what $C_b.\text{subgraph}$ and $R_b.\text{subgraph}$ would look like.

### 6.3.2 Match subgraphs

The next step is to create a match between elements in $C_b.\text{subgraph}$ and $R_b.\text{subgraph}$. The purpose of matching subgraphs is to find the differences between similar subgraphs. For each subgraph, $c$, in $C_b.\text{subgraph}$ we tried to find another subgraph, $r$, in $R_b.\text{subgraph}$ such that $r$ has the maximum number of triples that is the same as $c$. Let the maximum number of triples equivalent be called $M_c$. Equivalence of triples here compares all the nodes that are non-blank. For example, $\langle _:\text{123}, p_1, o_1 \rangle$ is considered equal to $\langle _:\text{456}, p_1, o_1 \rangle$, but $\langle _:\text{123}, p_1, o_2 \rangle$ is not equal to $\langle _:\text{123}, p_1, o_1 \rangle$. A subgraph can only be matched to one other subgraph.

In Figure 16, subgraph 1 has one triple in common with subgraph $a$. In contrast, subgraph
1 has two triples in common with subgraph $b$. Therefore, 1 is matched with $b$.

We tried to match subgraphs with the greatest similarity, because this will create the least difference. In an eventual cache, we need to update Rete network and the cache after finding a difference. The fewer differences we find, the less work Diamond has to do to update, leading to faster runtime.

If a subgraph, $c$, in $C_{b\text{\_subgraph}}$ has $M_c$ of zero, then it does not get matched to anything. Since it no longer exists in the current data source, it will be added as a minus token to Rete network. If a subgraph, $r$, in $R_{b\text{\_subgraph}}$ does not get matched to anything after all the subgraphs in $C_{b\text{\_subgraph}}$ has finished matching, $r$ is new data and will be added as a plus token to Rete network.

6.3.3 Find differences between matched subgraphs

After we find a match for every similar subgraphs, it is time to find the differences. Without loss of generality, assume $c$ in $C_{b\text{\_subgraph}}$ and $r$ in $R_{b\text{\_subgraph}}$ were matched. From $c$, remove all the triples that also exist in $r$. The remaining triples only exist in cache and should become minus tokens to Rete network. Similarly, from $r$, remove all the triples that also exist in $c$. The remaining triples are new and unknown to Diamond. They should become plus tokens and added to Rete network.

6.3.4 Tieing all three steps together

Using previously defined variables, $C_{b\text{\_subgraph}}$ and $R_{b\text{\_subgraph}}$ are the grouped triples for cache and real data, respectively. To finish steps 6.3.2 and 6.3.3, pass them as input to calculate_blank_difference as represented in the Appendix I. A list of minus and plus tokens will be returned after the function is run. These will be inserted in Rete network and
used to update the cache.
7 Evaluation

We compare the query execution time of Diamond using each of the four levels of cache. Making measurement of the query execution time is the most direct way to see if the cache is working. Additionally, we also like to evaluate the overhead of using an optimistic and eventual cache. After all, their value may decrease if the overhead is too high.

7.1 Comparing live and cached

We use the execution time of multiple queries to compare the performance of Diamond under live and cached. We tested eleven Fedbench BGPs (basic graph patterns) from Olaf and Ozsu [9]. As mentioned by Olaf and Ozsu, the original Fedbench BGPs are outdated, and we used the updated version provided by them. Each query was run three times and the average of the times is recorded. The result are shown in Figure 17:

![Fedbench query execution time comparison between live and cached](image)

Figure 17: This graph compares the query execution between live and cached on all 11 Fedbench queries. Cached always requires much less time than live.

As can be seen, a warm cache (cached) is nearly three orders of magnitude faster in query execution time than live execution. This is expected because computer memory runs faster than network speed. We also calculated how much the cache had sped up query execution. As presented in Figure 18 on average we achieved more than 1000 speed up.
### 7.2 Comparing all four levels of cache

To measure the performance of optimistic execution of cache, we need some kind of dynamic data. Unfortunately, we could not find any service that provides dereferenceable URI with dynamic data. We decided to simulate our own dereferenceable dynamic data service using a dataset from CityBench [1]. CityBench datasets are the various streams collected by sensors around the city of Aarhus, Denmark. These sensors include data regarding parking, traffic, weather, pollution, etc. To simulate dynamic data, we read the data from the streams and modified the RDF file for each URI with the newly read data just before we need it.

To get a better perspective of optimistic execution, we decided to look at its performance overhead. We measure what is the average time it takes to return the correct result for a query? If we change the data in RDF for every URI dereference, the execution will always need to dereference every URI again to get the correct answer. This defeats the purpose of optimistic execution, which takes advantage of the fact that the data is not always changing, but sometimes changing.

The stream of data provided by CityBench recorded data in periods of time. For example in AarhusWeatherData0, measurements are recorded on the hour, at twenty minutes, and at fifty minutes. Other sensors may record measurements in five minute periods. In order to test the system in a timely manner, we decided to simulate that at every dereference, one minute passes. Therefore, if the temperature of AarhusWeatherData0 starts at time 00:00:00, we would have to make twenty dereferences to get to the next value. Using this method, we measure the average time each type of cache take to return the correct result of a query. We calculate the average time by executing a test query from CityBench 60 times,

<table>
<thead>
<tr>
<th></th>
<th>Live avg (s)</th>
<th>Cached avg (s)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>fedbench 1</td>
<td>87.173</td>
<td>0.071</td>
<td>1225.782</td>
</tr>
<tr>
<td>fedbench 2</td>
<td>20.566</td>
<td>0.042</td>
<td>488.224</td>
</tr>
<tr>
<td>fedbench 3</td>
<td>224.987</td>
<td>0.148</td>
<td>1520.382</td>
</tr>
<tr>
<td>fedbench 4</td>
<td>1445.889</td>
<td>0.591</td>
<td>2445.854</td>
</tr>
<tr>
<td>fedbench 5</td>
<td>10.060</td>
<td>0.029</td>
<td>351.453</td>
</tr>
<tr>
<td>fedbench 6</td>
<td>1194.264</td>
<td>0.485</td>
<td>2462.418</td>
</tr>
<tr>
<td>fedbench 7</td>
<td>2.953</td>
<td>0.022</td>
<td>132.471</td>
</tr>
<tr>
<td>fedbench 8</td>
<td>732.595</td>
<td>0.317</td>
<td>2310.892</td>
</tr>
<tr>
<td>fedbench 9</td>
<td>40.453</td>
<td>0.086</td>
<td>471.093</td>
</tr>
<tr>
<td>fedbench 10</td>
<td>135.820</td>
<td>0.120</td>
<td>1135.534</td>
</tr>
<tr>
<td>fedbench 11</td>
<td>2528.785</td>
<td>4.734</td>
<td>534.152</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td><strong>1188.932</strong></td>
</tr>
</tbody>
</table>

Figure 18: The chart shows the speed up from live execution time to cached execution time. On average, there is more than 1000 fold speed up.
or one hour of sensor data and taking their average.

Ali et al. [1] described multiple queries across the CityBench dataset. We chose three queries for our evaluation. These queries use the same dataset. Using the same dataset did not limit the broadness of the test, however, as the dataset contains data of hundreds of sensors. Each of these queries also utilized different sensors’ data bringing variety into the evaluation. They are queries 2, 3, and 6 found in Section 3.3 in [1] and repeated in Appendix II. The comparison of query execution time for all types of cache is displayed in Figure [19].

![All caches average execution time comparison](image)

**Figure 19:** The graph compares the query execution time for all levels of cache on three CityBench queries. Cached is still considerably less than all other caches. Optimistic and eventual have very close times and both take longer time than live.

As expected, cached is still magnitudes faster than live. Also as expected, optimistic and eventual are slower than live, because they have to dereference every URI in the query again after they return the result by dereferencing using the cache. Finally, although it may not look like it because of the color, eventual cache is slightly slower than optimistic on all queries. This is also expected since eventual cache has to update the cache and Rete network and return the actual result to the client. What is unexpected is the quickness in updating and calculating the actual result. This process is always done in the matter of milliseconds on all queries. This shows the efficiency of the Rete network acting on small changes.

The big time gap between live and optimistic and eventual is likely to be due to a multi-threading problem. Because of the complexity of the implementation, we did not get parts
of optimistic and eventual cache to run in multiple threads. Specifically, the calculations of difference of two triple sets and update of cache and Rete network are done in a single thread. The calculation of two triple sets also includes dereferencing each URI again to get the most current information. Although the act of dereferencing is still multithreaded, the overall function of finding the difference is not. Therefore, it is as if we are dereferencing each URI in a single thread, greatly increasing the execution time.

Another reason maybe that Diamond takes a long time to calculate the difference between triple sets. We think this is less likely because all the calculations are done in memory and there were not many blank nodes in these test queries.
8 Conclusion

8.1 Accomplishments

We created a cache to store dereferenced data from linked data. Further, this cache is based on the standard open provenance model to create further functionalities. Using this cache, we proposed four different levels of caching: live, cached, optimistic, and eventual. Each of them have advantages and disadvantages, and it really depends on the dataset one is searching through to determine the best level. However, if anything, the most important lesson we learned is that network delay is significant larger than local memory processing, and a cache can effectively neglect this network delay.

While implementing optimistic and eventual cache, we ran into the problem of finding differences between graphs that contains blank nodes. We made a large effort to calculate an accurate difference to update Rete network and the cache efficiently.

8.2 Future work

Provenance information on a cache means we linked query with its possibly multiple URIs. In addition, the query is also linked with the URI’s dereferenced RDF triples. Provenance cache will allow us to give semantics to each query based on the behavior of its URIs. A semantic given to a query maybe the data it is querying are quite static, so it is better to cache its URI’s contents. It may also be the data changes unexpectedly with no obvious trends. It may be better to use an eventual cache for this query, since we don’t know when the data will change, but in the meantime the data is static enough for using the cache.

Our method of calculating blank node differences were created with some degree of arbitrariness. For example, we started matching from cached subgraphs to the real subgraphs. Does it matter if we start matching from real subgraphs instead? Other than finding maximum similarities, what other method could we have used to match the subgraphs? Should we even separate graphs into subgraphs and match them? Semantically, there is not a single correct solution to this problem. Thus, various implementations should be built and evaluated.
References


Appendix I

This pseudocode returns minus and plus tokens for Rete network upon inputting two maps of blank node to set of triples. This is the last step to calculating difference in two blank triple sets. Assume a map has two elements: key, in this case a blank node, and a value, in this case a set of triples which represents a connected graph.
Algorithm 4: calculate_blank_diff

**Input**: Two maps, c and r, from blank node (RDFNode) to RDF triple sets (Set(RDFTriple))

**Output**: Two RDF triple sets representing the triples to be added and deleted

1. Set(RDFTriple) minusTokens;
2. Set(RDFTriple) plusTokens;
3. foreach entry_c ∈ c do
   // Initialize variables
   4. c_subgraph ← entry_c.value;
   5. RDFNode maxElement;
   6. max ← 0;
   7. Set(RDFTriple) sameTriplesCache;
   8. Set(RDFTriple) sameTriplesReal;
   9. foreach entry_r ∈ r do
      if entry_r has not been matched then
         10. r_subgraph ← entry_r.value;
         11. maxForEntry ← 0;
         foreach c_triple ∈ c_subgraph do
            foreach r_triple ∈ r_subgraph do
               if c_triple’s non-blank nodes equals r_triple’s corresponding non-blank nodes then
                  14. maxForEntry ← maxForEntry + 1;
                  15. sameTriplesCache.add(c_triple);
                  16. sameTriplesReal.add(r_triple);
                  break;
               end
            end
         end
         if maxForEntry > max then
            22. max ← maxForEntry;
            23. maxElement ← entry_r.key;
         else
            26. sameTriplesCache ← null;
            27. sameTriplesReal ← null;
         end
      end
   end
   32. if max = 0 then
      33. minusTokens.addAll(c_subgraph);
   else
      // Minus
      35. remove_all(c_subgraph, sameTriplesCache);
      36. minusTokens.addAll(c_subgraph);
      // Plus
      37. r_subgraph = r.get(maxElement);
      38. remove_all(r_subgraph, sameTriplesReal);
      39. plusTokens.addAll(r_subgraph);
      Mark maxElement as matched;
   end
   // Add leftovers to plus tokens
   42. foreach entry ∈ r do
      if entry.key has not been matched then
         45. plusTokens.addAll(entry.value);
      end
   end
48. return (minusTokens, plusTokens)
Appendix II

All FedBench queries come from Hartig and Özsu. [9]

Prefixes

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX dbowl: <http://dbpedia.org/ontology/>
PREFIX dbprop: <http://dbpedia.org/property/>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>  
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
PREFIX gn: <http://www.geonames.org/ontology#>  
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>  
PREFIX swrc: <http://swrc.ontoware.org/ontology#>

FedBench 1

SELECT ?paper ?p ?n WHERE {
    ?paper swc:isPartOf 
}

FedBench 2

SELECT ?proceedings ?paper ?l WHERE {
    ?proceedings swc:relatedToEvent  
}

FedBench 3

SELECT ?paper ?p ?x ?n WHERE {
    ?paper swc:isPartOf
FedBench 4

  ?proceedings swc:relatedToEvent
}

FedBench 5

SELECT ?a ?n WHERE {
  ?a rdf:type dbowl:Album.
  ?a foaf:name ?n.
}

FedBench 6

  ?film owl:sameAs ?x.
  ?x foaf:based_near ?y.
}

FedBench 7

SELECT ?c ?x ?n WHERE {
}
FedBench 8

SELECT ?drug ?id ?s ?o ?sub WHERE {
  ?drug drugbank:drugCategory
  ?drug drugbank:casRegistryNumber ?id .
  ?drug owl:sameAs ?s .
  ?s foaf:name ?o .
  ?s dct:subject ?sub .
}

FedBench 9

SELECT ?x ?p WHERE {
  ?x dct:subject
  ?p dbprop:name "Luiz Felipe Scolari"@en .
}

FedBench 10

SELECT ?n ?p2 ?u WHERE {
  ?p2 owl:sameAs ?n .
}

FedBench 11

  ?x dbowl:birthDate ?d .
}
All CityBench queries come from Ali et al. These queries are written in Continuous SPARQL (C-SPARQL), which differs from SPARQL that Diamond uses. We had to modify these queries in order to execute them. The modified queries have the same basic structure.

**CityBench 2**

```
SELECT ?obId1 ?obId2 ?obId3 ?obId4 ?v1 ?v2 ?v3 ?v4
FROM <http://127.0.0.1:9000/WebGlCity/RDF/SensorRepository.rdf>
FROM stream
  <http://www.insight-centre.org/dataset/SampleEventService#AarhusWeatherData0>
  [range 3s step 1s]
FROM stream
  <http://www.insight-centre.org/dataset/SampleEventService#AarhusTrafficData158505>
  [range 3000ms step 1s]
WHERE {
  #?p1 a <http://www.insight-centre.org/citytraffic#Temperature>.
  #?p2 a <http://www.insight-centre.org/citytraffic#Humidity>.
  #?p3 a <http://www.insight-centre.org/citytraffic#WindSpeed>.

  {?obId1 <http://purl.oclc.org/NET/ssnx/ssn#observedProperty> ?p1.
  ?obId1 <http://purl.oclc.org/NET/sao/hasValue> ?v1.
  ?obId1 <http://purl.oclc.org/NET/ssnx/ssn#observedBy>
    <http://www.insight-centre.org/dataset/SampleEventService#AarhusWeatherData0>.

  {?obId2 <http://purl.oclc.org/NET/ssnx/ssn#observedProperty> ?p2.
  ?obId2 <http://purl.oclc.org/NET/sao/hasValue> ?v2.
  ?obId2 <http://purl.oclc.org/NET/ssnx/ssn#observedBy>
    <http://www.insight-centre.org/dataset/SampleEventService#AarhusWeatherData0>.

  {?obId3 <http://purl.oclc.org/NET/ssnx/ssn#observedProperty> ?p3.
  ?obId3 <http://purl.oclc.org/NET/ssnx/ssn#observedBy>
    <http://www.insight-centre.org/dataset/SampleEventService#AarhusWeatherData0>.
}
```
CityBench 3

SELECT ?obId1 ?obId3 ?v1 ?v3 ((?v1+?v3)/2) as ?avgCongest
FROM <http://127.0.0.1:9000/WebGlCity/RDF/SensorRepository.rdf>
from stream <http://www.insight-centre.org/dataset/SampleEventService
#AarhusTrafficData182955> [range 3s step 1s]
from stream <http://www.insight-centre.org/dataset/SampleEventService
#AarhusTrafficData158505> [range 3s step 1s]

WHERE {
  ### bind((?v1+?v3)/2) as ?avgCongest
  ### bind((?v2+?v4)/2) as ?avgEstimatedTime

  {?p1 a <http://www.insight-centre.org/citytraffic#CongestionLevel>.
  ###UNION{?p1 a <http://www.insight-centre.org/citytraffic#EstimatedTime>.
  {?p3 a <http://www.insight-centre.org/citytraffic#CongestionLevel>. }
  ###UNION{ ?p3 a <http://www.insight-centre.org/citytraffic#EstimatedTime>.}

  
  } }
SELECT ?obId1 ?obId2 ?lat1 ?lon1 ?lat2 ?lon2
from <http://127.0.0.1:9000/WebGlCity/RDF/SensorRepository.rdf>
from stream <http://www.insight-centre.org/dataset/SampleEventService
#AarhusParkingDataKALKVAERKSVEJ> [range 3s step 1s]
from stream <http://www.insight-centre.org/dataset/SampleEventService
#UserLocationService> [range 3000ms step 1s]

WHERE {

    {?obId1 a ?ob.
    ?obId1 <http://purl.oclc.org/NET/sao/hasValue> ?v1.
    ?obId1 <http://purl.oclc.org/NET/ssnx/ssn#observedBy>
        <http://www.insight-centre.org/dataset/SampleEventService
#AarhusParkingDataKALKVAERKSVEJ>.
    }

    {?obId2 a ?ob.
    ?obId2 <http://purl.oclc.org/NET/sao/hasValue> ?v2.
    ?obId2 <http://purl.oclc.org/NET/ssnx/ssn#observedBy>
    }
}