Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica - Scienza e Ingegneria - DISI
Corso di Laurea Magistrale in Ingegneria Informatica

SPARQL Event Processing Architecture: analisi e ottimizzazione delle prestazioni

Tesi di Laurea in Calcolatori Elettronici M

Relatore:
Prof. Luca Roffia

Correlatore:
Cristiano Aguzzi

Autore:
Andrea Ferrari

Sessione III
Anno Accademico 2019/2020

Indice

1 Abstract

Il SEPA (SPARQL Event Processing Architecture) si pone come middleware tra un endpoit SPARQL e i suoi clienti, offrendo vari servizi, in particolare la possibilità di registrarsi a delle sottoscrizioni. Il SEPA dunque implementa il design pattern publish-subscribe o anche chiamato Pub-Sub, che permette ai clienti di sottoscriversi designando il loro interesse a un certo sottoinsieme di conoscenza. In questo caso, il meccanismo di Pub-Sub si basa su richieste SPARQL dove ogni sottoscrizione indica l'interesse a quella parte della conoscenza che viene delimitata da una query SPARQL (query di sottoscrizione). Ogni pubblicazione degli aggiornamenti dei dati è rappresentata da una update SPARQL. Fornire questo servizio di sottoscrizioni, nel contesto semantico in cui il SEPA si pone, risulta molto costoso in termini prestazionali. L'obiettivo di questa tesi è quello di analizzare l'attuale meccanismo Pub-Sub, ricercare e implementare una prima parte di una possibile ottimizzazione per poi condurre ulteriori analisi, con lo scopo di realizzare un algoritmo che possa essere utilizzato per ottimizzare i tempi di gestione delle sottoscrizioni del SEPA, fornendo anche delle metriche sull'algoritmo stesso. Lo svolgimento dell'attività di tesi si può suddividere in quattro fasi. La prima fase è stata di studio sul SEPA e di approfondimento delle tecnologie coinvolte. Nella seconda fase è stato implementato un programma per eseguire dei benchmark in grado di valutare le attuali prestazioni del SEPA ed ottenere così i primi dati significativi sull'algoritmo nativo che gestisce le sottoscrizioni. La terza fase è stata un alternarsi tra studio, progettazione, implementazione e test sul nuovo algoritmo. La quarta e ultima fase si è occupata di riordinare i dati e consolidare il nuovo algoritmo. Al termine della tesi, i dati raccolti hanno confermato la validità dell'algoritmo, dimostrando come questo possa ridurre i tempi necessari alla gestione del meccanismo publish-subscribe.
La soluzione offerta dal SEPA è stata sviluppata e sperimentata all'interno del progetto Europeo H2020 SWAMP, Smart Water Management Platform [15], che rappresenta il primo caso d'uso reale di questa tecnologia. Ad oggi la base di conoscenza della piattaforma SWAMP contiene più di 20 milioni di triple, provenienti da fonti eterogenee, come servizi di previsioni meteo, sensori in campo e database esterni. Questa esperienza ha permesso di capire meglio le criticità in termini di prestazioni e ha fornito in questo senso le linee guida per questo lavoro di tesi.

2 Introduzione

Ai giorni d'oggi, Internet of Things (IoT) è ampiamente riconosciuto per il suo grande potenziale ma la sua usabilità, soprattutto in ambito commerciale, è frenata da un rilevante problema di frammentazione. Il Web of Things (WoT) cerca di contrastare la frammentazione dell'IoT, rendendo più facile la progettazione e la realizzazione di applicazioni distribuite senza la necessità di padroneggiare l'enorme varietà di tecnologie e standard IoT. Il WoT mira ad offrire l'accesso agevolato ad azioni, eventi e dati esposti da servizi e gemelli digitali di sensori e attuatori. Nell'ambito del WoT, l'interoperabilità a livello di informazione può essere ottenuta sfruttando le tecnologie e gli standard del Web Semantico. In questo contesto, ogni cosa è identificata da un URI che può essere dereferenziato per ottenere una descrizione interpretabile dalla macchina. Gli URI vengono anche utilizzati come identificatori di descrittori semantici per l'interoperabilità tra fornitori e consumatori di servizi. Le entità designate da URI possono essere descritte sotto molteplici punti di vista, come descrittori riguardo al loro comportamento, alle loro interazioni, al loro schema e formato di rappresentazione dei dati, alle configurazioni di sicurezza e protocolli di comunicazione. L'insieme di tutti questi dati viene spesso definito come base di conoscenza.
Ci sono molti settori che possono trarre vantaggio dal Web of Things, soprattutto quelli in ambito smart, come gli impianti intelligenti nell'industria, nelle abitazioni, nelle città, nell'agricoltura e nella sanità.
SPARQL Event Processing Architecture (SEPA) si pone come potente strumento per la gestione della conoscenza e degli eventi in ambito WoT, facendo riferimento allo standard RDF e al protocollo SPARQL.
Poniamoci ora in un generico scenario molto complesso in cui è presente un grande ammontare di dispositivi a capacità di calcolo limitate, in ambito WoT. Si è interessati alla gestione della totalità delle informazioni che i dispositivi, che potrebbero essere sensori, stanno raccogliendo. Sono presenti anche molti dispositivi invece interessati solo all'analisi di una parte dei dati o di una correlazione tra essi. Tutti questi clienti sono autonomi e devono prendere decisioni in tempo reale. In accordo con il modello del SEPA viene scelto RDF per rappresentare tutti i dati e viene utilizzato SPARQL come linguaggio di interrogazione della base di conoscenza. Definito un punto di accesso alla base di conoscenza, tutti i clienti interessati ad una sua ben definita parte, dovrebbero richiedere con strategia polling i nuovi dati tramite una query. Così facendo non solo è probabile che un buon numero di volte la richiesta sia superflua, ma è anche molto probabile che i dati in risposta siano parzialmente già noti al cliente richiedente. Questo è un enorme spreco di risorse, che comprometterebbe l'efficienza in un contesto con tanti clienti a basse prestazioni e in cui le risposte e le reazioni devono essere tempestive. Il SEPA tra i servizi offerti, permette la propagazione delle nuove informazioni tramite sottoscrizioni e notifiche, eliminando così tutti i problemi della strategia a polling. In questo modo tutti i clienti che sono interessati ad una sotto parte della base di conoscenza potranno registrare la loro query, che rappresenta il filtro per ottenere la conoscenza desiderata, per poi essere di conseguenza notificati con le rispettive nuove informazioni, ovvero i cambiamenti del risultato della query dovute all'alterazione della base di conoscenza.
La gestione di un meccanismo di sottoscrizioni basato su query SPARQL ha però portato molta della complessità del trattamento delle informazioni sul SEPA, che all'arrivo di un nuovo dato rappresentato da una update SPARQL, dovrà oltre a propagare le nuove informazioni direttamente sulla base di conoscenza, capire quali sono le nuove informazioni per ciascuna delle sottoscrizioni per poi notificarle. Potenzialmente ogni sottoscrizione

potrebbe essere interessata a dati diversi.
Attualmente il SEPA fornisce questo meccanismo con un approccio molto diretto e affidabile, ma che di conseguenza è limitante prestazionalmente.
L'obiettivo di questa tesi è quello di studiare, analizzare e ottimizzare questo meccanismo, per rendere il SEPA più performante nella gestione delle sottoscrizioni. Potrebbero venir sviluppati algoritmi molto performanti che però non sono in grado di gestire la totalità delle casistiche, oppure algoritmi che aumenteranno le performance solo in alcuni scenari con la possibilità di diminuirle in altri. Una soluzione mista che a seconda delle necessità sfrutti un algoritmo o un altro potrebbe essere l'ottimale.

3 Tecnologie e standard

In questo capitolo vengono riassunte le principali tecnologie utilizzate, soffermandosi sulle peculiarità rilevanti per il lavoro svolto.

3.1 RDF

Resource Description Framework (RDF), concordato e sviluppato dal W3C, definisce il formato per la rappresentazione dei dati e lo scambio di informazioni sul Web. Si basa su metadati strutturati e consente l'interoperabilità tra applicazioni che condividono informazioni machine-understandable.
RDF è un modello standard per lo scambio dei dati sul Web, estende la struttura dei collegamenti utilizzata nel Web, tramite gli URI identifica univocamente le entità e le loro relazioni.
La struttura cosi descritta, basata sui collegamenti, porta alla formazione di una rete di grafi. Un nodo di un grafo può essere un entità identificata tramite URI, un litterale o a sua volta un grafo. Anche i grafi sono identificati tramite URI. Un litterale è la rappresentazione in caratteri del valore di un dato come un numero, una stringa o una data. Viene definita tripla l'insieme di due nodi e la relazione che li lega, solo il nodo di partenza ha la necessità di essere un URI, il nodo destinatario della relazione potrà essere un URI o un litterale. Se però il legame tra i due nodi è bidirezionale il nodo destinatario dovrà essere un URI. La tripla è anche vista come la composizione di un soggetto, un predicato e un oggetto. Solo l'oggetto può essere un litterale, mentre soggetto e predicato devono essere entità rappresentate da URI. La visualizzazione grafica a grafo è il modello mentale più semplice per rappresentare RDF [4].

3.2 SPARQL1.1

SPARQL Protocol and RDF Query Language (SPARQL) è un linguaggio di interrogazione per basi di dati RDF o per sistemi middleware nei quali la base di conoscenza è rappresentata tramite RDF. SPARQL non si limita all'interrogazione di un database ma può interrogare più archivi RDF contemporaneamente, questo è tecnicamente possibile perché SPARQL è più di un semplice linguaggio di query, è anche un protocollo di trasporto basato sul protocollo HTTP, è quindi possibile accedere a qualsiasi endpoint SPARQL tramite un livello di trasporto standardizzato.
Vengono divise le richieste di interrogazione in query per effettuare le richieste di ottenimento dei dati, in update per effettuare modifiche sulla base di conoscenza, quest'ultima è derivata dalla definizione della prima.
Si parla di Graph Store, indicando il contenitore di grafi RDF gestiti da un unico servizio, contenente uno slot (senza nome) che contiene un grafo predefinito, denominato Default-Graph, e zero o più slot per grafi il cui nome è specificato. Le operazioni possono indicare i grafi su cui lavorare o possono fare affidamento su un grafo predefinito. Un Graph Store può conservare copie locali di grafi RDF definiti altrove sul Web e modificare quelle copie indipendentemente dal grafo originale.

3.2.1 Query

La specifica "SPARQL 1.1 Query Language" [9] definisce la sintassi e la semantica del linguaggio di query SPARQL, composta da parametri obbligatori e opzionali, supportando i concetti di aggregazione, sottoquery e negazione con la possibilità di utilizzare espressioni per la valutazione e la generazione di valori. I risultati delle query SPARQL possono essere espressi come set di risultati o come grafi RDF. Esistono alcune forme diverse per le richieste di tipo query: Select, Construct, Ask e Describe.
In SPARQL una variabile viene formata dal carattere di punto interrogativo "?" seguito dal nome della variabile stessa. Una variabile può indicare un soggetto, un oggetto, un predicato o un grafo. Le variabili possono essere utilizzate anche come contenitori per valori temporanei, per operazioni più complesse. Per identificare un URI viene semplicemente racchiuso tra parentesi angolari, "<" e ">". Per identificare un grafo viene utilizzata la parola chiave *"graph"* seguita dal URI del grafo e da un modello racchiuso tra parentesi graffe. Tra la parola *"graph"* e il pattern è anche possibile utilizzare una variabile al posto dell'URI. Le triple vengono separate da punti se riportate nella forma estesa soggetto, predicato, oggetto. Nel caso di più triple con lo stesso soggetto, esiste la forma contratta che permette dopo aver riportato la prima tripla per intero, di riportare le altre triple con stesso soggetto omettendolo e separandole con punti e virgola.
La richiesta di select è la più comune, facendo un caso generale e semplice, permette di ottenere ennuple definite dalla clausola principale select, che vengono formate a partire dalle quadruple della clausola principale che possono appartenere a uno o più grafi, se specificati dalla clausola *"from"*, filtrandole per i modelli presenti nella clausola *"where"*.
Il risultato di una query SPARQL1.1 consiste in un insieme di "bindings" che consistono in coppie di variabili e valori, esiste anche la possibilità di rappresentare questi risultati in JSON.

3.2.1.1 Construct

La richiesta di una query di tipo Construct, ottiene i set risultanti da ciascuna soluzione della sequenza risolutiva della query, ne sostituisce le variabili, combina le triple restituite per poi generare un singolo grafo RDF, risultato dell'unione di tutti set.
Le triple non valide, come una tripla formata da un litterale in posizione di soggetto o predicato, oppure una tripla con una variabile non legata, non verranno incluse nel grafo restituito.
Data una query di tipo Construct con annessa clausola where, il suo risultato sarà dunque l'unione di tutti i set di triple risultanti dalla clausola where, costruiti seguendo il modello che viene definito dalla Construct stessa, escludendo la presenza di triple illegali.

3.2.1.2 Ask

La richiesta di una query di tipo Ask, viene usata per verificare se un modello di una query abbia oppure non abbia una soluzione. La Ask non restituisce ulteriori informazioni su possibili soluzioni, si limita a restituire un booleano che asserisce solo la presenza o meno della soluzione.
Una delle sue applicazioni più comuni è, ad esempio, venire usata per testare la presenza di una o più triple nella base di dati, richiedendo l'esecuzione di una Ask il cui modello è l'insieme delle triple su cui ci si interroga per la loro presenza sulla base di dati. La

Ask restituirà una affermazione se tutte le triple sono presenti, restituirà una negazione se almeno una delle triple è inesistente.

3.2.2 Update

SPARQL 1.1 Update [8] è un linguaggio di aggiornamento per i grafi RDF, utilizza una sintassi che deriva da SPARQL Query Language.
Vengono fornite operazioni per aggiornare, creare e rimuovere i grafi RDF di un Graph Store, contenitore mutabile di grafi RDF. Le richieste di update possono essere suddivise nelle seguenti tipologie:

- UpdateDataInsert, per la sola aggiunta di triple esplicite (non da risolvere, senza variabili) su uno o più grafi.
- UpdateDataDelete, per la sola rimozione di triple esplicite su uno o più grafi.
- UpdateDeleteWhere, per la sola rimozione di triple su uno o più grafi potendo definire una clausola where per poter selezionare le triple da rimuovere tramite dei modelli.
- UpdateModify, per aggiungere e rimuovere triple, con la possibilità di dichiarare una clausola where.
- UpdateClear, per rimuovere tutte le triple da un grafo specificato.
- UpdateDrop, per rimuovere tutti i grafi indicati e le loro triple.
- UpdateCopy, per inserire tutti i dati del grafo indicato nel grafo di destinazione, il grafo di destinazione verrà prima svuotato.
- UpdateAdd, per inserire tutti i dati del grafo indicato nel grafo di destinazione, mantenendo i dati già presenti nel grafo di destinazione.

3.3 Lavorare con i grafi

Data una base di dati composta da più grafi ed un endpoint che la gestisca, ogni volta che si eseguono operazioni di lettura o scrittura, possono essere coinvolti uno o più grafi. Si definisce DefaultGraph il grafo senza contesto, un contenitore che non è realmente un grafo. In questo caso le query o le update eseguite sul DefaultGraph lavoreranno su triple nel senso stretto della parola, dato che non sono riferite ad alcun grafo e quindi sono composte solo da soggetto, predicato ed oggetto, non esiste nessuna relativa quadrupla.
In caso di update e query con grafo o grafi esplicitati, i dati sono invece organizzati in quadruple composte dalla tripla e dal grafo a cui appartiene. Si definisce Data Set l'insieme dei grafi coinvolti in un'operazione.
Nelle query è possibile imporre il grafo o l'insieme dei grafi da cui si vogliono estrarre le triple tramite le clausole *"from"* e *"from named"*. La clausola *"from"* restringe le possibili triple, che corrisponderanno al modello della query, alle sole contenute all'interno del grafo selezionato dalla clausola, limitando dunque il Data Set. Nel caso di una query con un grafo come variabile, la clausola *"from named"*, restringerà il dominio di quella variabile al solo grafo selezionato dalla clausola o ai soli grafi selezionati, se presenti più clausole *"from named"*. Le due tipologie di clausole *"from"* e *"from named"*, possono essere combinate per poter, ad esempio, selezionare delle quadruple i cui grafi sono incognite ma

i cui domini sono ristretti da una o più clausole *"from named"* e le cui triple sono legate ad altre triple presenti nel grafo esplicitato dalla clausola *"from"*.
Riportiamo un esempio concreto (listing 1) in cui si vogliono tutti i nodi soggetto $?s$ in relazione con i loro grafi $?g$, limitando il dominio di $?g$ ai soli grafi $< g2 >$ e $< g3 >$ tramite la clausola *"from named"*. Come modello viene imposto che il soggetto $?s$ esista in una tripla nella quale il predicato sia $< p >$ seguito dall'oggetto $?o$ e sia legata alla tripla $?o < attribute >$ "*colorito*". Quest'ultima deve trovarsi all'interno del grafo $< g1 >$ vincolato dalla clausola *"from"*.

```
SELECT ?s ?g
FROM <g1>
FROM NAMED <g2>
FROM NAMED <g3>
{
  GRAPH ?g {
    ?s <p> ?o .
  }
  ?o <attribute> "colorito" .
}
```

Listing 1: Esempio query con from e from named.

Nel caso di una query senza alcun grafo definito, questa lavorerà su quello che viene chiamato ActiveGraph che può essere definito a discrezione dell'endpoint, per esempio alcuni endpoint lo implementano con l'unione della totalità dei grafi, altri lo fanno corrispondere al DefaultGraph.
Per quanto riguarda le operazioni di update, queste adottano lo stesso approccio. Nello specifico vengono impiegate parole chiave differenti, usando *"with"* per definire il grafo su cui effettuare l'update ed esistono anche le clausole *"using"* e *"using named"*, analoghe delle clausole *"from"* e *"from named"* nelle query.

4 SPARQL Event Processing Architecture

Figura 1: SEPA.

Il SEPA (SPARQL Event Processing Architecture) [19][20] è un'architettura decentralizzata Web-based progettata per supportare lo sviluppo di servizi e applicazioni distribuite, dinamiche, context-aware e interoperabili. L'architettura consente il rilevamento e la notifica delle modifiche sul Web dei dati mediante un meccanismo di pubblicazione e sottoscrizione basato sul contenuto, supportando a pieno il linguaggio SPARQL1.1 sia per richieste di update che per richieste di query.
L'architettura si basa sul protocollo W3C SPARQL 1.1 e introduce i protocolli SPARQL 1.1 Secure Event [23] per implementare comunicazioni protette e SPARQL 1.1 Subscribe Language [24] come mezzo per trasmettere ed esprimere richieste e notifiche di sottoscrizione. L'implementazione di riferimento dell'architettura offre agli sviluppatori un design pattern per uno sviluppo di applicazioni modulare, scalabile ed efficace [13].

4.1 Struttura interna

Considerando il SEPA come una scatola nera possiamo elencare i suoi ingressi e le sue uscite:

- UpdateRequest, richiesta asincrona, ricevuta con lo scopo di poter eseguire una update secondo SPARQL1.1 sulla base di dati.
- UpdateResponse, risposta asincrona alla richiesta UpdateRequest.
- QueryRequest, richiesta asincrona, ricevuta con lo scopo di poter eseguire una query secondo SPARQL1.1 sulla base di dati.
- QueryResponse, risposta asincrona alla richiesta QueryRequest.
- Subscribe, richiesta asincrona, ricevuta da un cliente per poter stabilire una sottoscrizione.
- SubscribeResponse, risposta asincrona verso il cliente, per poter confermare la sottoscrizione.
- Notification, notifica in uscita con i relativi cambiamenti, effettuati da una UpdateRequest, alla base di dati a cui la sottoscrizione è interessata.
- Unsubscribe, richieste asincrona in ingresso per poter chiudere la sottoscrizione.
- UnsubscribeResponse, risposta asincrona verso il cliente, per poter confermare la chiusura della sottoscrizione.

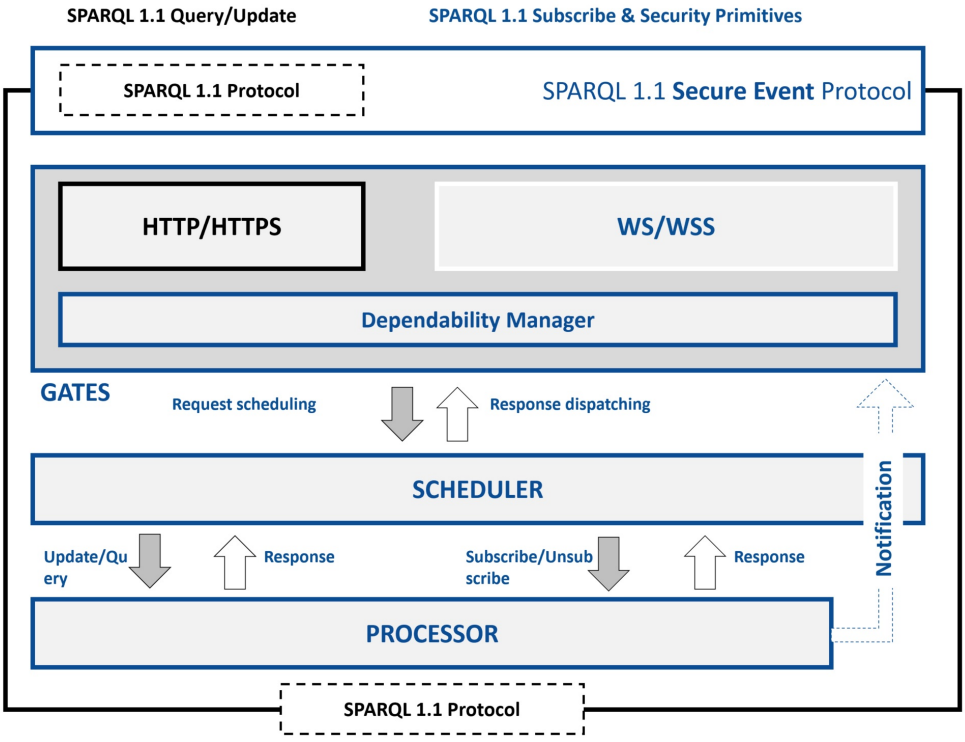

Figura 2: Filiera della gestione di una richiesta da parte del SEPA.

Una richiesta Subscribe contiene una richiesta di query SPARQL1.1, che sarà utilizzata come selettore della conoscenza d'interesse del cliente. A tal scopo, il cliente richiede di voler stabilire una sottoscrizione basata su una query (query di sottoscrizione) per poter essere notificato tutte le volte che verranno aggiunte o rimosse triple o quadruple di suo interesse. All'arrivo di una Subscribe, il cliente riceverà il set completo dei risultati della query di sottoscrizione. Sono possibili più Subscribe contemporanee appartenenti allo stesso cliente. Le notifiche, Notification, relative alla stessa sottoscrizione sono differenziate da un indice incrementale.
Possiamo suddividere la struttura del SEPA in gates, scheduler e core. Generalmente una richiesta viene servita seguendo la filiera in figura 2. La richiesta attiva prima il livello di gates, per passare al livello di scheduler e giungere al livello di core (o processor), dopo di che la risposta generata, come le eventuali notifiche, ripercorreranno tutti i livelli a ritroso, richiedendo la schedulazione per poi giungere al livello di gates dove verranno inoltrate ai destinatari [11].

Il livello gates implementa il protocollo SPARQL1.1 SE (figura 3) gestendo le richieste per lo scheduler, le risposte alle richieste asincrone e le notifiche. L'implementazione fornisce sia comunicazioni HTTP (s) che WS (S) entrambe con supporto al protocollo Transport Layer Security (TLS). Il protocollo WS viene utilizzato per le sottoscrizioni, dunque per le notifiche. Si trova in corso di svolgimento l'aggiunta del supporto ad altri protocolli per il pub-sub quali MQTT[17], CoAP[3] e Linked Data Notification[14].
Il livello scheduler implementa i meccanismi e le policy di schedulazione delle richieste che sono accodate in una coda FIFO, con la possibilità di selezionarne la capienza massima.

Lo scheduler implementa anche il meccanismo di assegnazione delle risposte e delle notifiche formulate dal livello core, che dovranno essere reindirizzate verso l'opportuno gate appartenente al livello gates.
Il livello core, anche detto processor, è un insieme di processori logici. Esiste una tipologia di processore logico per ogni tipologia di richiesta, come possiamo vedere in figura 2, vi è un processore per le richieste di query, Query processor, e un processore per le richieste di update, Update processor. Queste due tipologie di processori logici però da soli non sono in grado di gestire le sottoscrizioni. Esiste perciò un processore logico, Subscription Processing Unit Manager (SPUManager), che può usufruire degli altri due processori logici e che gestirà un Subscription Processing Unit (SPU) per ogni sottoscrizione.

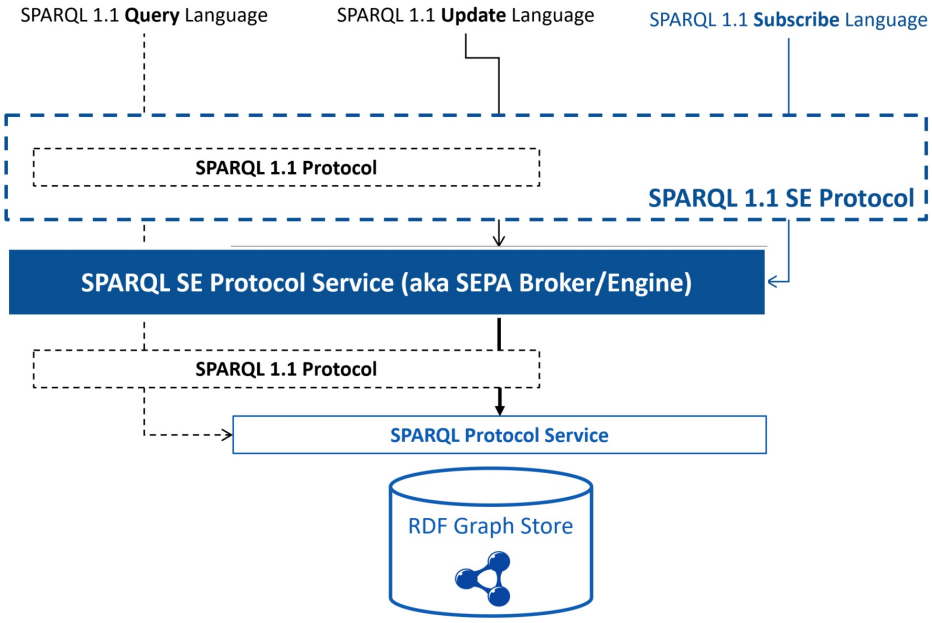

Figura 3: SEPA, protocolli dell'architettura pub-sub.

4.2 Processamento delle update e gestione delle sottoscrizioni

Limitandoci solamente al livello processor, tramite l'apposito processore viene generata una SPU dedicata per ogni sottoscrizione. Ogni richiesta di update passa attraverso l'SPUManager che si occupa della gestione di tutte le SPU e che con dei metodi di filtraggio attiverà solo le SPU che ritiene coinvolte nell'operazione di modifica della base di dati scaturita dalla update. Le SPU che passano il filtro, si attivano incaricandosi di ottenere i bindings da aggiungere e i bindings da rimuovere che dovranno essere mandati come aggiornamento alla loro sottoscrizione.
All'arrivo di una nuova richiesta di sottoscrizione, viene subito eseguita la relativa query i cui risultati vengono restituiti al cliente. L'architettura manterrà una copia dei risultati, per ricavare all'arrivo di una futura update gli aggiornamenti che ne conseguiranno, così da poterli poi notificare alle sottoscrizioni.

Prendendo tutte le sottoscrizioni, in generale, ognuna di esse è interessata a un sottoinsieme della base di conoscenza dell'endpoint, delineato dalla query di sottoscrizione.
L'architettura deve quindi propagare la nuova conoscenza scaturita dalla update alle sole sottoscrizioni interessate, tenendo conto che ogni sottoscrizione potrebbe necessitare della notifica di bindings da aggiungere o da rimuovere che appartengono a triple effettivamente aggiunte o rimosse alla base di conoscenza o che potrebbero necessitare dell'aggiunta o della rimozione per effetto collaterale. Definiamo "modifica per effetto collaterale" il caso in cui la modifica, ad un risultato di una sottoscrizione a causa dell'esecuzione di una update, non si poteva predire dalla sola analisi della update.
L'algoritmo che nativamente processa le update, dopo averne eseguita una, effettua per ogni sottoscrizione la relativa query per poi confrontare la nuova risposta con i risultati anteriori alla update, i quali sono tenuti in memoria locale. Il confronto tra i due stati, prima e dopo l'attuazione della update viene poi inoltrato alla sottoscrizione, sotto forma di bindings che risultano da aggiungere e di bindings che risultano da rimuovere, in modo che il cliente possa aggiornare la sua conoscenza locale allo stato attuale di quella remota.
Una prima ottimizzazione che è stata effettuata è l'aggiunta di un filtro che attiva le sole SPU le cui query di sottoscrizione interessano almeno uno dei grafi coinvolti nell'azione di update. Questo primo filtro sarà oggetto di studio e analisi nel capitolo 5.2.
Esponiamo le varie casistiche di inoltro verso le sottoscrizioni, degli eventuali bindings da aggiungere e rimuovere a causa del processamento di una update, sia per effetto diretto che per effetto collaterale, spiegando nel dettaglio cosa si intende per effetto collaterale tramite un esempio. Consideriamo per semplicità un solo grafo e solo sottoscrizioni che sono interessate a quest'ultimo, che hanno quindi passato il filtro basato sui grafi. Il grafo "<example>" è popolato con quattro triple, per poter ricostruire l'esempio è possibile utilizzare la update al listing 2.

```
INSERT DATA {
      GRAPH <example> {
                <s0><p0><o0>.
                <s1><p1><o1>.
                <s2><p2><o2>.
                <s3><p2><o3>.
        }
}
```

Listing 2: Update per costruire la base di conoscenza.

```
% Query di sottoscrizione di S0
SELECT ?s FROM <example> WHERE {
        ?s <p2> ?o. }
% Query di sottoscrizione di S1
SELECT ?s FROM <example> WHERE {
        ?s ?p <o1>.
        bind( if(exists {<s5> <p3> <o5>},<p1>,<p>) as ?p)
}
% Query di sottoscrizione di S2
SELECT ?o FROM <example> WHERE {
        ?s <p0> ?o.
        bind( if(exists {<s3> <p2> <o3>},<s0>,<s>) as ?s)
}
% Query di sottoscrizione di S3
SELECT ?p FROM <example> WHERE {
        ?s <p1> ?p. }
```

Listing 3: Query di sottoscrizione.

Le sottoscrizioni attive sono quattro $S0, S1, S2$ e $S3$, le cui query di sottoscrizione sono presentate al listing 3. Ipotizziamo che tutte siano già state stabilite e che quindi siano sincronizzate con i dati presenti in remoto, di seguito raffiguriamo in forma tabellare lo stato corrente della base di dati sotto forma di triple e lo stato locale delle sottoscrizioni, i risultati alle loro query, sotto forma di elenco di variabili e rispettivi valori.

Base di conoscenza		
?s	?p	?o
<s0>	<p0>	<o0>
<s1>	<p1>	<o1>
<s2>	<p2>	<o2>
<s3>	<p2>	<o3>

Bindings sottoscrizioni		
Sottoscrizione	Variabile	Valore
S0	?s	<s2>
S0	?s	<s3>
S2	?o	<o0>
S3	?o	<o1>

Simuliamo l'arrivo della update, presente al listing 4, che scatenerà l'aggiornamento per tutte e quattro le sottoscrizioni.

```
DELETE {
  GRAPH <example> {
            ?s ?p ?o.
          }
}INSERT {
  GRAPH <example> {
            <s4><p1><o4>.
            <s5><p3><o5>.
          }
}WHERE {
    <s3> ?p ?o.
    ?s ?p ?o.
}
```

Listing 4: Update che causerà l'aggiornamento per le sottoscrizioni.

Nello specifico la update aggiungerà alla base di conoscenza due triple e ne rimuoverà una, comportando l'aggiunta e la rimozione diretta e indiretta di risultati sulle sottoscrizioni. Riportiamo in forma tabellare gli stati a posteriori della update, sia per la base di conoscenza sia per la propagazione alle sottoscrizioni degli aggiornamenti relativi, evidenziando in rosso la conoscenza rimossa e in verde la conoscenza aggiunta.

Base di conoscenza		
?s	?p	?o
<s0>	<p0>	<o0>
<s1>	<p1>	<o1>
<s2>	<p2>	<o2>
<s3>	<p2>	<o3>
<s4>	<p1>	<o4>
<s5>	<p3>	<o5>

Bindings sottoscrizioni		
Sottoscrizione	Variabile	Valore
S0	?s	<s2>
S0	?s	<s3>
S2	?o	<o0>
S3	?o	<o1>
S1	?s	<s1>
S3	?o	<o4>

Esaminiamo ogni singola sottoscrizione focalizzando l'attenzione sul cambiamento dei bindings da prima a dopo l'esecuzione della update:

- S0, prima della update possedeva due risultati (?s,<s2>) ed (?s,<s3>), è stata poi aggiornata con il risultato da rimuovere (?s,<s3>), quest'ultimo era legato ad una tripla che è stata effettivamente rimossa dalla base di conoscenza dalla update "<s3><p2><o3>".

- S3, possedeva solamente il risultato (?o,<o1>). Dopo la update è stata aggiornata con il nuovo risultato (?o,<o4>), quest'ultimo risultante da una tripla che è stata effettivamente aggiunta alla base di conoscenza dalla update "<s4><p1><o4>".

- S1, la sua query di sottoscrizione, allo stato della conoscenza anteriore all'esecuzione della update, non ha prodotto risultati. In questo caso l'esecuzione della update ha scatenato l'aggiornamento della conoscenza della sottoscrizione per effetto collaterale. La sottoscrizione è stata aggiornata con un nuovo risultato (?s,<s1>) che però non deriva da una tripla direttamente aggiunta dalla update, ma bensì fa parte della tripla "<s1><p1><o1>" che esisteva prima della update e che continua ad esistere anche dopo la sua esecuzione. La vera modifica che ha scatenato l'aggiornamento è stata l'aggiunta della tripla "<s5><p3><o5>".

- S2, prima dell'esecuzione della update, aveva un unico risultato (?o,<o0>). Le modifiche alla base di conoscenza apportate dalla update hanno scatenato un aggiornamento per S2 che ha portato alla sua rimozione dai risultati correnti.

Abbiamo così dimostrato come sia impossibile ricostruire gli aggiornamenti per i risultati da aggiungere e rimuovere per le sottoscrizioni solamente a partire dalla update e dai suoi effetti sulla base di conoscenza. Esistono dunque casistiche per le quali i risultati di una query di sottoscrizione cambiano per l'effetto collaterale della modifica sulla base di conoscenza da parte di una update.
Rappresentiamo in figura 4 l'esempio prima descritto, mostrando la relazione tra le triple aggiunte e rimosse dalla update sulla base di dati e i bindings aggiunti e rimossi alle sottoscrizioni.
A partire da questo esempio, si possono costruire altre casistiche più complesse, in cui vengono combinate le quattro presentate. In oltre esiste anche l'eventualità che i bindings non siano il soggetto, il predicato oppure l'oggetto di una tripla, come invece abbiamo visto in questo esempio. I bindings ottenuti da una query possono anche essere i risultati di un'espressione svolta all'interno della query stessa e quindi non riconducibili a elementi di una tripla. Un'altra possibilità è quella di una query che produce come binding un URI di un grafo. L'esempio mostrato è dunque una parte di tutte le eventualità.

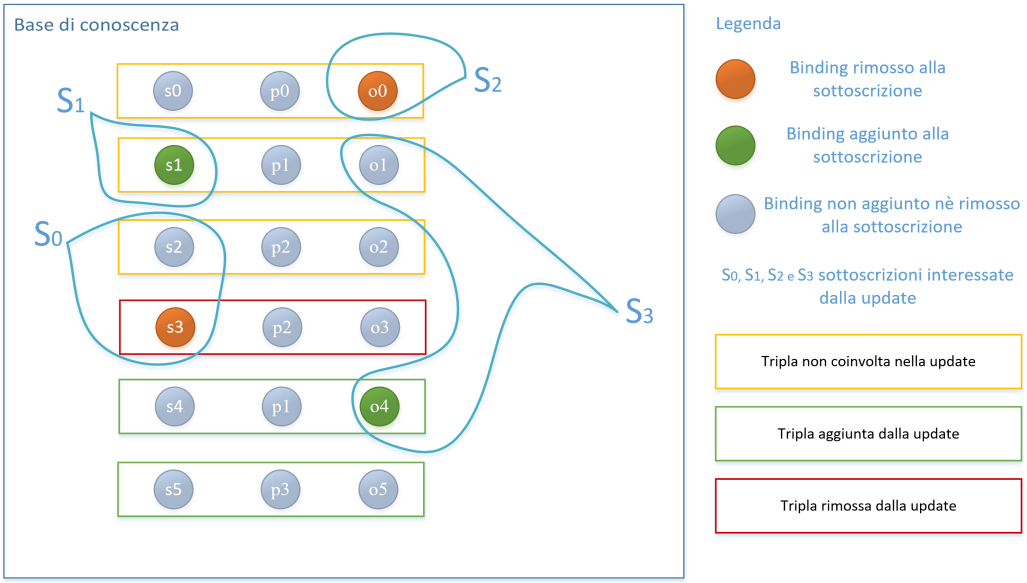

Figura 4: Effetti della update sulle sottoscrizioni S_0, S_1, S_2 e S_3.

4.3 Modello di sviluppo delle applicazioni

Il SEPA viene affiancato dal modello di sviluppo per applicazioni PAC (Producer Aggregator Consumer) e da vari strumenti quali il JSAP (JSON SPARQL Application Profile) e la Dashboard per agevolarne l'implementazione.

4.3.1 Produttore, aggregatore e consumatore

Nell'ambito di un utilizzo pratico, il SEPA può essere visto come l'entità che fornisce i servizi per la memorizzazione, la gestione e la navigazione della conoscenza di uno scenario applicativo. I produttori, gli aggregatori e i consumatori sono invece i clienti di questi servizi.
PAC (Producer Aggregator Consumer) è un modello di progettazione per lo sviluppo di applicazioni che si basano sull'utilizzo del SEPA, realizzato per fornire un'implementazione generale e trasparente di un possibile cliente che si interfaccia con il SEPA.
Il produttore, concettualmente è un mero generatore di conoscenza, che non interroga nessuno e non rende conto a terzi, produce triple o quadruple da inviare tramite update verso il SEPA. Un potenziale produttore risiede in un qualsiasi tipo di sensore, che quando ha una nuova lettura del dato, lo formatta e lo inoltra alla base di conoscenza tramite il SEPA.
Il consumatore è il duale del produttore, interessato solo alla raccolta di dati provenienti dal SEPA, non è un suo compito fornire ulteriori triple o quadruple indirizzate alla collettività. Sfrutta il concetto di sottoscrizione, per essere sempre aggiornato sui dati che lo interessano. Non sono previste update in uscita da un consumatore. Un esempio potrebbe essere quello di un attuatore che a seconda dello stato della conoscenza agisce.
Un aggregatore è un'estensione di un consumatore, che nel caso ricevesse una notifica di un nuovo dato è anche in grado di eseguire una update. Può quindi raccogliere dati, aggregarli, manipolarli ed estrarre nuova conoscenza da condividere con eventuali altri

attori.
Queste tre tipologie di clienti del SEPA, prevedono anche l'utilizzo dei file di configurazione JSAP, in modo che la parte di logica applicativa racchiusa dallo SPARQL delle query, update e sottoscrizioni alla base di questi tre attori, possa essere modificata senza dover dipendere dall'implementazione degli attori stessi.
Ne è un esempio concreto il SEPA-Chat, un'applicazione che simula una chat tra utenti basata proprio su produttori, aggregatori e consumatori. Base di partenza di uno dei tester realizzati, descritto nella sezione 5.2.

4.3.2 JSAP

JSON SPARQL Application Profile [13], viene utilizzato per descrivere un'applicazione che utilizza il SEPA come architettura per il processamento di eventi e la gestione della conoscenza.
JSAP assume il JSON come formato predefinito di rappresentazione ed è predisposto per seguire il modello PAC spiegato nella sezione 4.3.1.
Il ruolo per cui è nato JSAP è quello di raccoglitore di descrittori e regole per il modello PAC, ma può essere utilizzato anche su modelli differenti. Racchiude l'insieme delle update SPARQL1.1 e le query di sottoscrizione, che verranno utilizzate dall'applicazione che si vuole definire. Specifica i parametri di protocollo da utilizzare. Può contenere gli elenchi dei prefissi e dei grafi del dominio applicativo, una sezione dedicata ad eventuali estensioni e una dedicata all'autenticazione.
Un file di configurazione JSAP può essere visto come la carta di identità di un'applicazione.
In molte applicazioni la logica delle update, come quella delle query, una volta definita rimane la stessa, mentre a cambiare sono le entità o i valori coinvolti, le triple e le quadruple che li rappresentano. Risulta dunque molto comodo poter utilizzare lo stesso scheletro di una richiesta cambiando solo i componenti di una quadrupla o tripla. Per questo motivo le liste di update e di query necessarie all'applicazione, che sono definite in un file JSAP, offrono anche un meccanismo denominato ForcedBindings. Questo concetto permette di stabilire delle variabili che a tutti gli effetti seguono il formato delle variabili SPARQL, che verranno però utilizzate e risolte dall'applicazione, in modo completamente trasparente per il SEPA e per l'endpoint.
Quando la variabile viene definita sotto l'attributo di "forceBindings" non presenta il classico punto interrogativo che precede il nome della variabile stessa, ma quando viene utilizzata all'interno della richiesta ha lo stesso formato di una variabile secondo la definizione data da SPARQL1.1.
Facciamo riferimento alla update descritta nella parte di un file JSAP mostrata al listing 5, che deriva dal JSAP di un'applicazione di messaggistica. La update in oggetto ha lo scopo di archiviare la conferma dell'arrivo di un messaggio scambiato tra due utenti.
Si può osservare una lista di ForcedBindings, ognuno dei quali è composto da un nome, da un tipo e da un valore. Il nome, seguendo il formato JSON, è il punto di accesso per le altre proprietà "type", "value" e opzionalmente "datatype". Il valore (value) è il default che verrà assegnato alla variabile prima che la richiesta venga inoltrata al SEPA, in mancanza di una diversa assegnazione a run-time. Il tipo demarca la tipologia del valore che la variabile è destinata ad assumere, rispettando i tipi definiti da SPARQL1.1. Il tipo è indicato con gli attributi "type" e opzionalmente, se "type" corrisponde ad un "literal", vi è anche la possibilità di definire il "datatype", specializzazione del tipo "literal".
A linea 11 del listing 5, viene definita la variabile con nome "dateSent" che potrà contene-

re valori di tipo "literal" che rispettano il formato definito da "xsd:dateTime". Il valore a default, in questo caso è "2020-12-12T00:00:00", una data specifica che rispetta il formato delineato da "datatype". Le variabili successive sono predisposte per contenere l'entità messaggio, il suo contenuto e le entità che sono coinvolte come mittente e destinatario nell'azione di inviare il messaggio.
Viene così messa a disposizione la struttura base per effettuare update che hanno in comune tutti i costrutti principali, ma con la possibilità di impostare a run-time i valori delle triple tramite i ForcedBindings.

```
"STORE_SENT": {
  "sparql": "
      INSERT DATA {GRAPH <http://wot.arces.unibo.it/chat/log/> {
          ?message schema:text ?text;
          schema:sender ?sender;
          schema:toRecipient ?receiver;
          schema:dateSent ?dateSent
      }}
  ",
  "forcedBindings": {
    "dateSent": {
      "type": "literal",
      "value": "2020-12-12T00:00:00",
      "datatype": "xsd:dateTime"
    },
    "message": {
      "type": "uri",
      "value": "chat:ThisIsAMessage"
    },
    "text": {
      "type": "literal",
      "value": "A message to be stored"
    },
    "sender": {
      "type": "uri",
      "value": "chat:IAmASender"
    },
    "receiver": {
      "type": "uri",
      "value": "chat:IAmAReceiver"
    }
  }
}
```

Listing 5: Esempio ForcedBindings di un file JSAP.

4.3.3 Dashboard

La Dashboard [5] è un applicativo in Java che permette l'esecuzione delle richieste di query, update e sottoscrizioni al SEPA attraverso un'interfaccia utente.
L'applicazione è in grado di caricare i file JSAP da cui leggere relative update e query seguite dai loro ForcedBindings, permettendone la manipolazione e l'esecuzione.
Lo scopo principale della Dashboard è quello di immettersi tra il livello applicativo di un eventuale applicazione basata sull'uso del SEPA e quello descrittivo di un file JSAP, nato per essere usato nelle applicazioni che seguono il modello PAC.
Risulta un ottimo strumento per poter creare, correggere ed elaborare i file JSAP, permettendo di testare e analizzare le singole richieste anche modificandole a run-time. Può

anche registrare sottoscrizioni a partire da una qualsiasi query descritta nel file di configurazione. La Dashboard può essere vista come un simulatore di consumatori, produttori e aggregatori.
Inoltre offre una visione globale della base di conoscenza, permettendone la navigazione tra i grafi e tra le loro triple.
Durante molteplici fasi di studio per l'ottimizzazione del SEPA, la Dashboard è stata utilizzata per poter fare una prima valutazione sugli effetti delle update in presenza di sottoscrizioni e soprattutto per generare i file JSAP di tester e benchmark creati appositamente per le fasi di analisi più approfondite.

5 Analisi delle prestazioni

Per poter realizzare l'ottimizzazione di un'architettura, come primo requisito servono delle metriche su cui basarsi per quantificare le performance e capirne al meglio i fattori correlati. Inoltre è necessario che durante l'ottimizzazione si verifichino e si mantengano le qualità di correttezza, completezza e affidabilità. Abbiamo basato tutta la fase di ottimizzazione su metriche temporali legate alla quantità di tempo necessaria all'architettura a gestire e smaltire le richieste dei clienti. In secondo luogo sono stati monitorati altri fattori come quantità di memoria sfruttata, lo stato delle code interne dell'architettura e il numero di SPU.
Sono stati creati tre ambienti di test diversi per poter analizzare e valutare l'architettura con un focus diverso: ChatTest, LumbTest e UpdateTest. Il primo, ChatTest, mira allo studio prestazionale macroscopico, il secondo LumbTest a uno studio specifico sull'algoritmo nativo e sull'algoritmo implementato per l'ottimizzazione. L'ultimo, UpdateTest, deriva dai test effettuati nel secondo ambiente di test, utilizza le stesse update per effettuare dei test puramente funzionali. L'ambiente di test UpdateTest consiste in un file Junit contenente un test per ogni tipologia di update trattata, con lo scopo di accertare il corretto funzionamento dell'algoritmo di ottimizzazione.
Le soluzioni trovate, riferendoci all'ottimizzazione, si basano sulle richieste di Ask e sulla Construct di SPARQL1.1 che non sono spesso usate in modo articolato, pertanto presentano alcune problematiche di compatibilità da parte degli endpoint, che di conseguenza compromettono le caratteristiche di correttezza, completezza e affidabilità del SEPA. Sono dunque stati considerati due endpoint diversi, non solo per la raccolta di metriche, ma anche per poter analizzare queste incompatibilità e gestirle.
Tutti i risultati dei test riportati nella tesi fanno riferimento, se non diversamente esplicitato, alla seguente configurazione, in cui tutti gli attori coinvolti del test come il tester, il SEPA e gli endpoint, eseguivano all'interno della stessa macchina fisica che poteva contare su 16GB di RAM con una frequenza fissata a 3200Mhz e un processore "AMD Ryzen 9 3900X".
In questo capitolo verranno descritti tutti gli strumenti utilizzati per le fasi di analisi e verrà trattata l'analisi, tramite il ChatTest, delle prestazioni del SEPA. La fase di analisi dell'algoritmo di ottimizzazione è riportata al capitolo 6.2.

5.1 Strumenti

A seguire la descrizione degli strumenti usati per condurre le analisi prestazionali sul SEPA.

5.1.1 RDF endpoint

Il SEPA si pone tra i clienti e l'endpoint che racchiude la base di conoscenza RDF, il SEPA è compatibile con diversi endpoint tra cui Virtuoso e Blazegraph, scelti per i test che accompagnano questa ottimizzazione.
Virtuoso è una piattaforma moderna costruita su standard aperti che sfrutta la potenza dei collegamenti ipertestuali per aggredire le difficoltà nel raccogliere e manipolare dati. Virtuoso è uno degli endpoint più utilizzati per gestire dati eterogenei RDF e convertirli in grafi di conoscenza, accessibili utilizzando linguaggi di query e update come SPARQL1.1 e SQL, tramite qualsiasi applicazione o servizio che supporti ad esempio HTTP, ODBC o JDBC [22].

Per i test è stata utilizzata la versione di Virtuoso fornita tramite Docker, che è una piattaforma aperta per lo sviluppo, la spedizione e l'esecuzione di applicazioni. Consente di disaccoppiare le applicazioni dall'infrastruttura in modo da poter distribuire agevolmente il software. Fornisce metodologie per la spedizione, il test e la distribuzione rapida del codice. Docker offre la possibilità di creare pacchetti ed eseguire un'applicazione in un ambiente isolato chiamato container. L'isolamento dei container fornisce un ambiente perfetto per il test, dato che limita fortemente le influenze dovute all'hardware della macchina e dalle altre applicazioni che eseguono sulla stessa [6].
Il secondo endpoint utilizzato è Blazegraph, un database a grafo ad alte prestazioni che supporta Blueprints, API RDF e SPARQL1.1. Pensato per coprire tutte le esigenze, dalla piccola applicazione fino ad applicazioni con 50 miliardi di collegamenti tra nodi RDF su una singola macchina. Possiede un'estrema facilità d'installazione, è presentato in un singolo archivio jar, che non necessità di particolari configurazioni [2].
Inoltre il SEPA al suo interno utilizza la libreria di Jena per effettuare la lettura e la conversione delle richieste SPARQL1.1. Esistono anche alcuni endpoint basati sulla libreria Jena, come Apache Jena Fuseki, pertanto in parte è come se stessimo testando anche questi endpoint, oltre al fatto che grazie a Jena il SEPA ha integrato un livello di validazione delle richieste di update e query, dato che tutte le richieste passano tramite questo livello (figura 5). Anche la futura ottimizzazione utilizzerà la libreria Jena, in particolar modo per analizzare le richieste di update e per validare le "Internal query" come le Ask e le Construct che verranno generate dall'algoritmo di ottimizzazione.

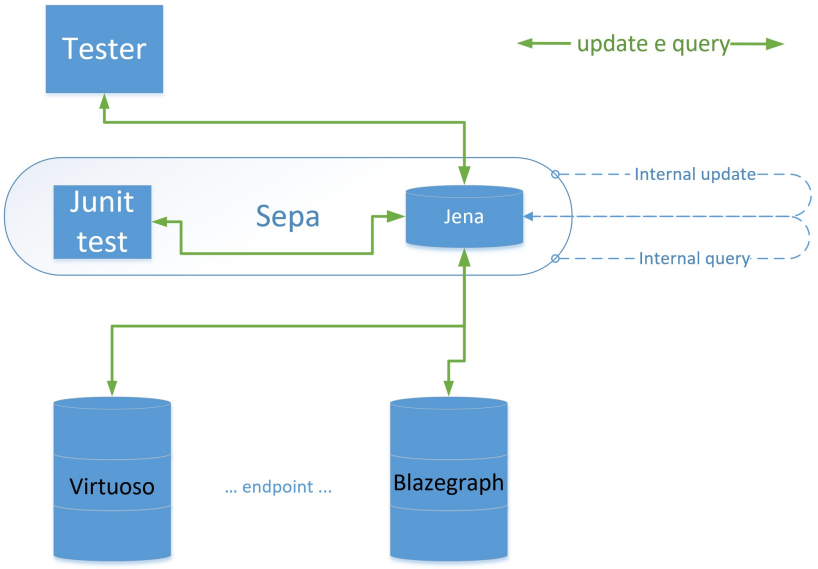

Figura 5: Flusso query e update durante i test.

5.1.2 LUMB

LUMB è un progetto che nello specifico racchiude un'ontologia, un generatore di conoscenza RDF basato su quest'ultima e delle query SPARQL1.1 inerenti. Il tutto con lo scopo di fornire uno strumento per poter testare i motori d'inferenza su base di dati RDF. Il benchmark della Lehigh University è stato sviluppato per facilitare la valutazione dei repository di Web semantico in modo standard e sistematico. Il benchmark ha lo scopo di valutare le prestazioni di tali repository rispetto alle query su un ampio set di dati che si impegnano su una singola ontologia realistica [16]. Consiste in un'ontologia che rappresenta un dominio universitario con dati sintetici personalizzabili e ripetibili. Possiede una serie di query di test e diverse metriche delle prestazioni [10].
Indichiamo le classi dell'ontologia sottolineandole e indichiamo con il corsivo le proprietà. Il generatore di quadruple basate sull'ontologia fornisce una base di conoscenza che è costituita da N, numero configurabile, University e ciascuna rispetta i seguenti vincoli:

- esistono $15 \sim 25$ Departments *subOrgnization* University
- $7 \sim 10$ FullProfessors *worksFor* Department
    - di cui 1 FullProfessors *headOf* Department
- esistono $10 \sim 14$ AssociateProfessors *worksFor* Department
- esistono $8 \sim 11$ AssistantProfessors *worksFor* Department
- esistono $5 \sim 7$ Lecturers *worksFor* Department
- ogni Faculty possiede $1 \sim 2$ *teacherOf* Courses
- ogni Faculty possiede $1 \sim 2$ *teacherOf* GraduateCourses
- i Courses tenuti dalle facoltà sono disgiunti a coppie
- esistono $10 \sim 20$ ResearchGroups *subOrgnization* Department
- UndergraduateStudent : Faculty $= 8 \sim 14 : 1$
- GraduateStudent : Faculty $= 3 \sim 4 : 1$
- ogni Student possiede *memberOf* Department
- $1/5 \sim 1/4$ degli GraduateStudents sono stati scelti come TeachingAssistant rispetto a un Course
    - tutti i GraduateStudents scelti come TeachingAssistant per un Course sono a coppie diversi.
    - $1/4 \sim 1/3$ degli GraduateStudents sono stati scelti come ResearchAssistant
- 1/5 degli UndergraduateStudents ha un *advisor* Professor
- ogni GraduateStudent ha un *advisor* Professor
- ogni UndergraduateStudent possiede $2 \sim 4$ *takesCourse* Courses
- ogni GraduateStudent possiede $1 \sim 3$ *takesCourse* GraduateCourses

- ogni FullProfessor possiede $15 \sim 20$ *publicationAuthor* Publications
- ogni AssociateProfessor possiede $10 \sim 18$ *publicationAuthor* Publications
- ogni AssistantProfessor possiede $5 \sim 10$ *publicationAuthor* Publications
- ogni Lecturer possiede $0 \sim 5$ Publications
- ogni GraduateStudent è co.autore di $0 \sim 5$ Publications con qualche Professors
- ogni Faculty possiede:
    - un *undergraduateDegreeFrom* University
    - un *mastersDegreeFrom* University
    - un *doctoralDegreeFrom* University
- ogni GraduateStudentha un *undergraduateDegreeFrom* University

L'ontologia contiene 243 assiomi logici, 43 classi, 25 proprietà definite per relazioni con IRI come oggetto e 7 proprietà definite per relazioni con letterali come oggetto. Con la figura 6 si può rappresentare l'interconnessione e la complessità dell'ontologia, rappresentando le classi con blocchi rettangolari e con frecce le relazioni tra esse [15].

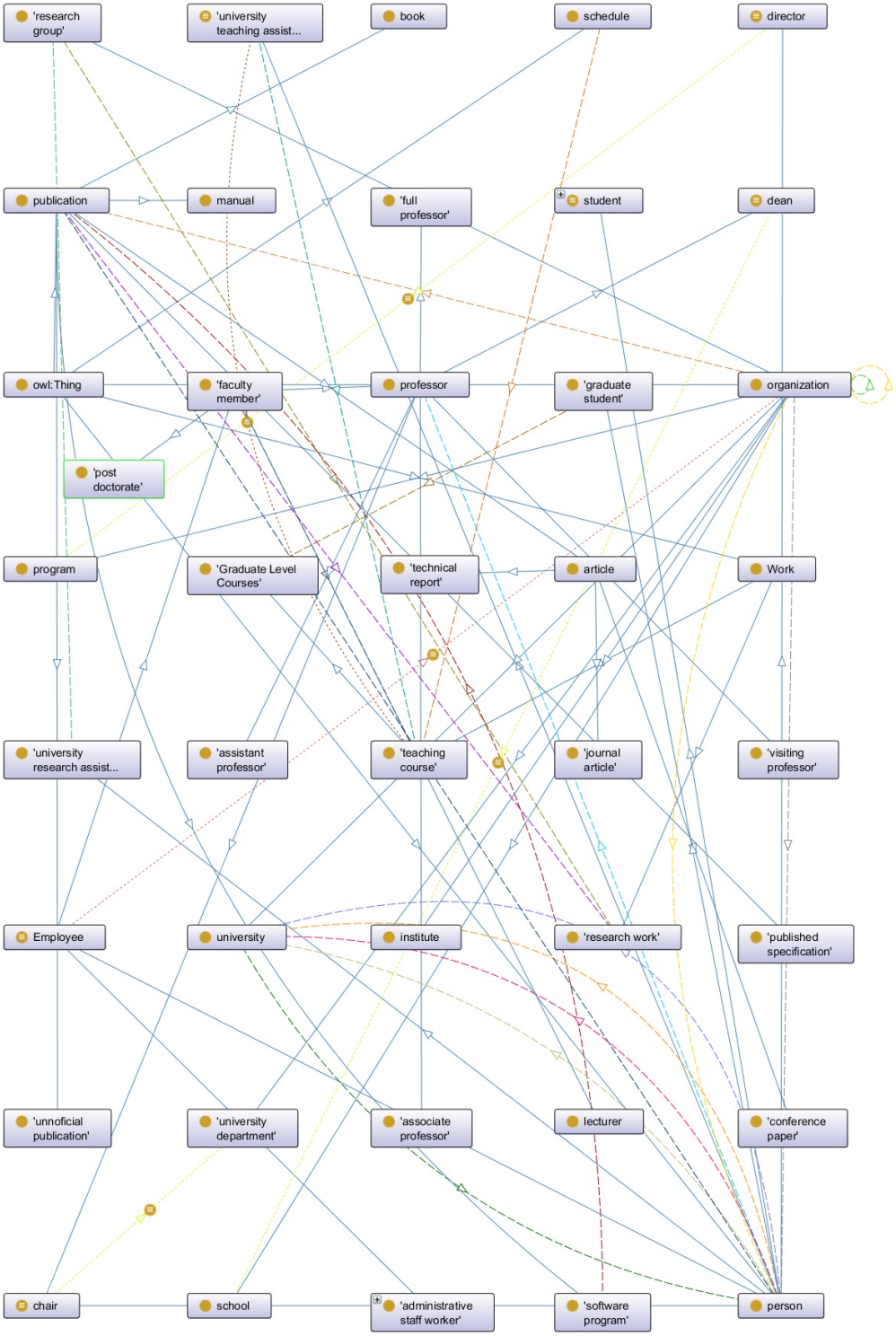

Figura 6: Schema a grafo delle classe dell'ontologia LUMB.

5.1.3 JMH

JMH (Java Microbenchmark Harness) è un toolkit che aiuta a implementare correttamente i microbenchmark Java. JMH è stato sviluppato dalle stesse persone che implementano la Java virtual machine, dunque un team che conosce molto bene l'ambiente Java e la sua JVM, fondamentale per poter evitare o tener in considerazione tutti gli automatismi della macchina virtuale che provocano overhead nell'esecuzione del codice e quindi possono alterare le metriche dei test.
I Microbenchmark hanno lo scopo di misurare le prestazioni di una piccola parte di un'applicazione più grande, questo risulta difficile in Java dato che esistono molte ottimizzazioni che la JVM o l'hardware sottostante possono applicare al componente durante la sua esecuzione. I microbenchmark male implementati possono quindi far credere che le prestazioni del componente siano migliori di quanto saranno nella realtà. Scrivere un microbenchmark Java corretto in genere comporta l'impedimento delle ottimizzazioni che la JVM e l'hardware possono applicare durante l'esecuzione del microbenchmark [12].
JMH essendo in grado di isolare porzioni di un applicazione Java per impedirne alla JVM di applicare ottimizzazioni che portino a tempi non coerenti con l'ambiente reale di esecuzione, può anche essere utile per ottenere metriche temporali più chiare, in un ambiente di test su lungo periodo, come nel caso di un benchmark di stress test. Durante l'esecuzione di uno stress test creato in Java, le tempistiche potrebbero essere influenzate con conseguenti errori nel campionamento dei tempi, utilizzando JMH è possibile ridurre queste impurità.

5.2 Benchmark

Una delle esigenze era quella di riuscire a testare il sistema nel suo complesso, con una visione a scatola nera del SEPA e di tutti i suoi meccanismi. Serviva un ambiente di test che mettesse in difficoltà il sistema con un numero di richieste molto elevato, in modo da poter raccogliere il tempo totale di questa simulazione senza entrare nei dettagli, senza campionare i tempi dei singoli livelli o singoli algoritmi del SEPA.
ChatTest è il primo ambiente di test a cui il SEPA con l'algoritmo nativo è stato sottoposto, possiamo classificarlo come benchmark, dato che è stato costruito appositamente per studiare un caso d'uso problematico che mette sotto sforzo l'algoritmo cuore del SEPA, per la gestione delle sottoscrizioni, sfruttando diversi scenari graduatamente sempre più impegnativi.
Il ChatTest deriva dal progetto SEPAChat[21] che aveva lo scopo di mostrare in un caso reale l'utilizzo del SEPA e dei suoi agenti (aggregatori, produttori e consumatori, spiegati al paragrafo 4.3.1). Il progetto originale, consisteva (come mostrato in figura 7) nell'esempio di una chat in cui veniva usato il SEPA come nodo centrale per la gestione dei servizi.

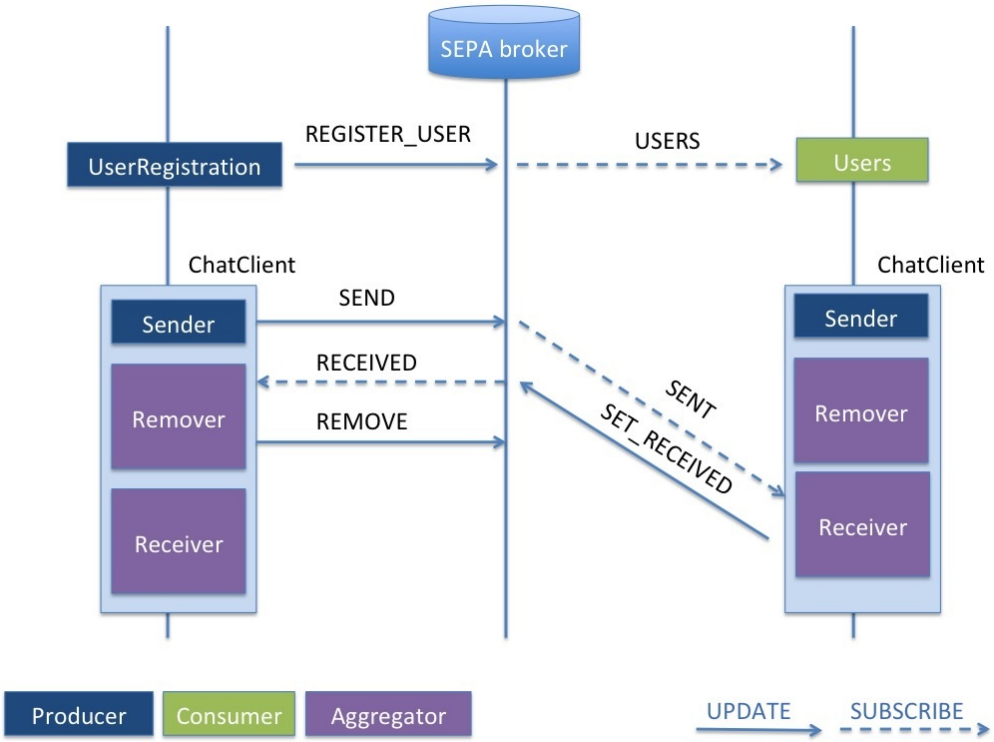

Figura 7: Schema progetto SepaChat.

Ogni cliente simulato possedeva:

- Un produttore per richiedere la registrazione dell'utente.
- Un consumatore per potersi sottoscrivere e ricevere le notifiche relative alla gestione degli account come registrazione o eliminazione di un account.
- Un produttore per poter spedire messaggi ad altri utenti, l'invio del messaggio verso il SEPA è asincrono.
- Un aggregatore per potersi sottoscrivere ed essere notificati all'arrivo di nuovi messaggi, per poi in automatico confermare la ricezione del nuovo messaggio, tramite una richiesta asincrona verso il SEPA.
- Un aggregatore per potersi sottoscrivere ed essere notificati delle conferme di ricezione dei messaggi, per poi in automatico rimuovere il messaggio e la sua conferma di ricezione dalla base di dati, tramite una richiesta asincrona verso il SEPA.

Il progetto originario permetteva già l'osservazione qualitativa del comportamento del SEPA in una simulazione realistica, in cui N utenti si scambiavano M messaggi.
Per mettere in difficoltà il SEPA serviva:

- Aumentare il rapporto richieste su tempo per creare congestione e sovraccaricare il livello di scheduler del SEPA.

- Aumentare il numero di clienti e di conseguenza il numero di sottoscrizioni, ricordando che ogni sottoscrizione possiede una sua SPU che verrà gestita dal livello core.
- Aumentare il numero di grafi RDF coinvolti.

ChatTest aggiunge JMH per poter eseguire il benchmark lato cliente in un ambiente isolato, in modo che la JVM non possa influire sull'esecuzione del benchmark, in modo che i tempi raccolti siano reali e privi di fluttuazioni. SEPAChat utilizza un unico grafo per tutta la simulazione, sapendo che la quantità di grafi è un ottimo fattore di stress per il SEPA, ChatTest aggiunge anche la possibilità di utilizzare più grafi RDF. Viene aggiunta la possibilità di creare un grafo per ogni stanza virtuale di chat, ad esempio si possono racchiudere in grafi diversi i meccanismi di messaggistica di ogni coppia di utenti, oppure dedicare un grafo per una chat di gruppo, il tutto mantenendo sempre la stessa logica applicativa ed il flusso di messaggi del progetto originale.

5.2.1 Considerazioni sui grafi

Una delle prime ottimizzazioni effettuate sul SEPA consiste in un filtro basato sui grafi coinvolti da una richiesta di update e i grafi d'interesse delle singole SPU (o sottoscrizioni).
Nello specifico, avendo N SPU, ogni SPU ha una relativa query di sottoscrizione che interessa M grafi. Con il filtro sui grafi, all'arrivo di una generica update che punta a modificare triple su X grafi, verranno attivate solo le n SPU che hanno almeno un grafo in M che esiste in X, con $n \leq N$.
In questo modo più grafi verranno utilizzati dall'applicazione, minore sarà il tempo impiegato dal SEPA per riuscire a gestire le sottoscrizioni.
Qualitativamente possiamo descrivere due casi d'uso d'esempio che si trovano agli estremi tra loro, riguardo l'ottimizzazione aggiunta dal filtro sui grafi. Per esempio supponiamo di avere N SPU ognuna con query di sottoscrizione su un grafo distinto. All'arrivo di un'update su un singolo grafo verrà attivata al più una SPU. Considerando il caso duale, in cui le N SPU hanno query di sottoscrizione, anche se diverse, interessate allo stesso grafo, all'arrivo di un'update che tende a modificare triple relative a quel grafo, verranno attivate tutte le SPU. In quest'ultimo estremo, il tempo necessario al SEPA per soddisfare tutte le sottoscrizioni non cambia a seconda che sia o meno implementato il filtro sui grafi.
Il primo scopo del ChatTest è quello di evidenziare gli effetti del filtro basato sui grafi, nei tempi necessari per la gestione di un'operazione di update che scatenerà delle notifiche ai clienti interessati.
Il benchmark ChatTest permette la creazione di specifici scenari di test, ottenendo il tempo complessivo della simulazione degli scenari. Per poter osservare la differenza nei tempi nell'utilizzare oppure non utilizzare il filtro sui grafi, si sono considerati 14 scenari. Il grafico in figura 8 mostra chiaramente il distacco nei tempi dovuto all'utilizzo del filtro sui grafi. Gli scenari utilizzati, sono tutti basati su un ambiente a molteplici grafi, variando tra loro per numero di utenti coinvolti e numero di messaggi scambiati tra coppie di utenti. Dati n utenti il numero di grafi utilizzati sarà pari a $N = C(n, k)$ con la combinazione di n elementi a gruppi di $k = 2$ senza ripetizioni:

$$N = C(n, k) = (n!/(k!(n-k)!)) \tag{1}$$

Per ogni coppia di utenti si stabiliscono sei sottoscrizioni, per ogni utente coinvolto nello scambio: una per la query "received", una per la query "sent" e una per la query "users", come mostrato a figura 7. Dunque il numero di sottoscrizioni contemporanee in uno scenario è pari al numero di conversazioni moltiplicate per sei. Dato che il numero di conversazioni è uguale al numero di grafi possiamo ottenere il numero di sottoscrizioni come $S = N * 6$.
A parità di grafi utilizzati, gli scenari cambiano per numero di messaggi inviati da ciascun utente. Definiamo m come il numero di messaggi che ogni utente deve inviare ad ognuno degli altri utenti.
Prendendo ad esempio una coppia di utenti A e B, l'utente A dovrà inviare m messaggi all'utente B e riceverne altrettanti m dallo stesso utente B, questo per ogni coppia di utenti presenti nella simulazione. Ogni comunicazione tra coppie di utenti è racchiusa in un grafo, che quindi dovrà gestire nel corso dell'esecuzione di un singolo scenario $m * 2$ messaggi.
Il numero complessivo dei messaggi creati e gestiti dal benchmark durante un singolo scenario è pari a M:

$$M = (N * 2 * m) \tag{2}$$

Descrizione degli scenari:

Scenario	Utenti (n)	Grafi coinvolti (N)	Sottoscrizioni contemporanee (S)	Messaggi che ogni utente invia ad ogni altro utente (m)	Messaggi complessivi (M)
1	2	1	6	5	10
2	2	1	6	10	20
3	2	1	6	20	40
4	2	1	6	40	80
5	5	10	60	5	100
6	5	10	60	10	200
7	5	10	60	20	400
8	5	10	60	40	1000
9	6	15	90	5	150
10	6	15	90	10	300
11	6	15	90	20	600
12	6	15	90	40	1200
13	10	45	270	5	450
14	10	45	270	10	900

I dati relativi a questo test sono stato ottenuti utilizzando un'istanza di Virtuoso all'interno di un container Docker come endpoint.

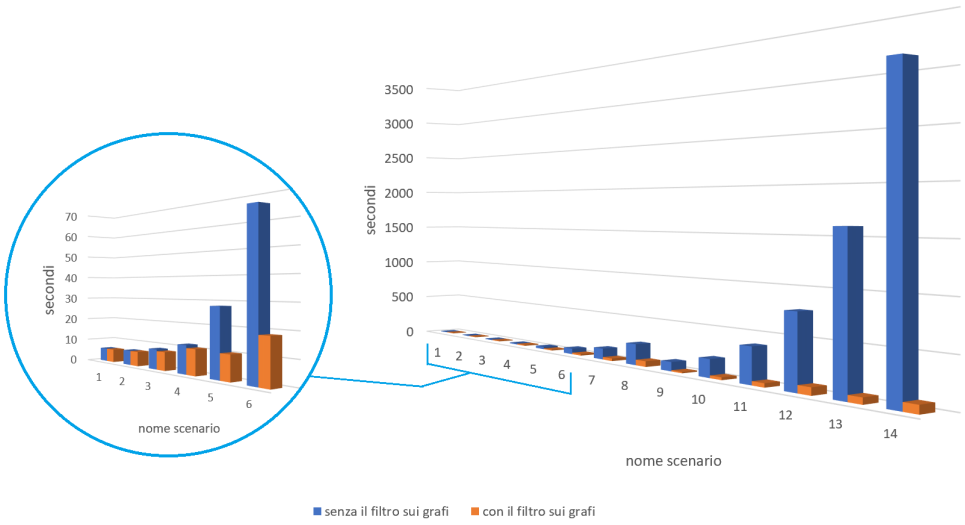

Figura 8: Risultati benchmark ChatTest, studio del filtro sui grafi.

Come ci aspettavamo, all'aumentare dei grafi il distacco nei tempi aumenta, diventando estremamente evidente nello scenario 14 in cui l'esecuzione dell'intero benchmark impiega più di 51 minuti per simulare 10 utenti che a coppie si scambiano 10 messaggi. L'intero scenario genera e gestisce un totale di 900 messaggi. Possiamo osservare che non è lo scenario con più messaggi complessivi, dato che lo scenario 12 ne possiede 1200, infatti il fattore dominante è il numero di sottoscrizioni. Il numero di sottoscrizioni influenza molto di più il tempo necessario all'esecuzione dello scenario rispetto al numero di messaggi inviati, specialmente nel caso dell'esecuzione del benchmark ChatTest senza il filtro basato sui grafi.
Sempre considerando lo scenario 14, il tempo necessario all'esecuzione del benchmark in presenza del filtro sui grafi, è poco più di 85 secondi. Lo scenario 14 mostra quanto il filtro sui grafi sia importante, facendo cambiare ordine di grandezza alla metrica temporale, da minuti a secondi necessari per l'esecuzione dello stesso scenario. Ricordiamo che nonostante il benchmark sia basato su un caso reale di implementazione di un sistema di messaggistica, è anche stato pensato e costruito appositamente sul concetto dell'utilizzo di un numero molto elevato di grafi, infatti lo scenario 14 utilizza 45 grafi distinti. Ci aspettavamo queste grandi differenze nei tempi, dato che senza il filtro nello scenario 14 vengono attivate tutte le volte, tutte e 270 le sottoscrizioni.
Possiamo concludere questa fase di analisi sui risultati effettivi dell'applicazione del filtro sui grafi, riportando che lo scenario 12 con filtro attivo, è riuscito a gestire ben 1200 messaggi in 80 secondi, ricordando che la gestione dello scambio di un messaggio non consiste in una semplice richiesta asincrona da mittente a destinatario, ma vengono gestiti più stati, come possiamo osservare dalla figura 7, il messaggio viene spedito, depositato, inoltrato, poi una conferma viene ricevuta, depositata, inoltrata e per finire tutte le informazioni che sono state fin ora depositate sull'endpoint riguardanti il messaggio vengono

rimosse, tutto questo mediamente eseguito in 15 millisecondi per messaggio considerando lo scenario 12 con filtro attivo.
Risultati del test in forma tabellare, le ultime due colonne riportano i valori dell'esecuzione scenari in millisecondi:

Scenario	Utenti (n)	Messaggi che ogni utente invia ad ogni altro utente (m)	Senza filtro sui grafi [ms]	Con il filtro sui grafi [ms]
1	2	5	5828	5802
2	2	10	6358	6332
3	2	20	8577	7869
4	2	40	11861	10721
5	5	5	28042	10372
6	5	10	65173	18481
7	5	20	125940	29023
8	5	40	236309	62710
9	6	5	101541	13645
10	6	10	198559	28876
11	6	20	388805	43226
12	6	40	781314	80398
13	10	5	1599510	66228
14	10	10	3103475	85286

5.2.2 Valutazione dell'attuale algoritmo di notifica

Utilizzando il benchmark ChatTest, si è cercato di misurare anche un altro distacco temporale, la differenza di tempo tra un'esecuzione del ChatTest basata su un singolo grafo e un'esecuzione basata su più grafi, a parità di logica applicativa, numero di clienti, messaggi scambiati ed interazioni.
L'obiettivo è quello di ottenere i tempi di esecuzione al variare del numero di grafi coinvolti.
La logica dell'organizzazione degli scenari, per quanto riguarda numero di utenti e di messaggi, è la stessa del test precedente (studio del filtro sui grafi 5.2.1). Il SEPA rimarrà sempre con la stessa configurazione, il filtro sui grafi implementato nel SEPA rimarrà sempre attivo. Preso uno scenario, il fattore che influirà sui tempi di esecuzione e sul quale si vogliono prendere le due misure da comparare, è l'utilizzo di un singolo grafo o l'utilizzo di molteplici grafi all'interno delle richieste SPARQL1.1, a parità dello scopo della logica applicativa.
Riferendoci al test precedente in un singolo scenario, il numero di grafi coinvolti era costante, quello che cambiava era l'utilizzo o meno del filtro sui grafi. In questo caso, come abbiamo detto, il filtro è sempre attivo, ma sarà il benchmark che cambierà utilizzando un singolo grafo oppure utilizzando un grafo per ogni comunicazione tra due utenti (quest'ultima modalità è quella utilizzata nel test precedente).
Per calcolare il numero di messaggi totali del singolo scenario, utilizzare la formula (2).
Nella figura 9 sono rappresentati i dati raccolti di tutti i 16 scenari e di seguito riportiamo in forma tabellare tutti i dati degli scenari, sia per la variante a singolo grafo che per la variante a grafi multipli, insieme ai relativi valori dei tempi di esecuzione.

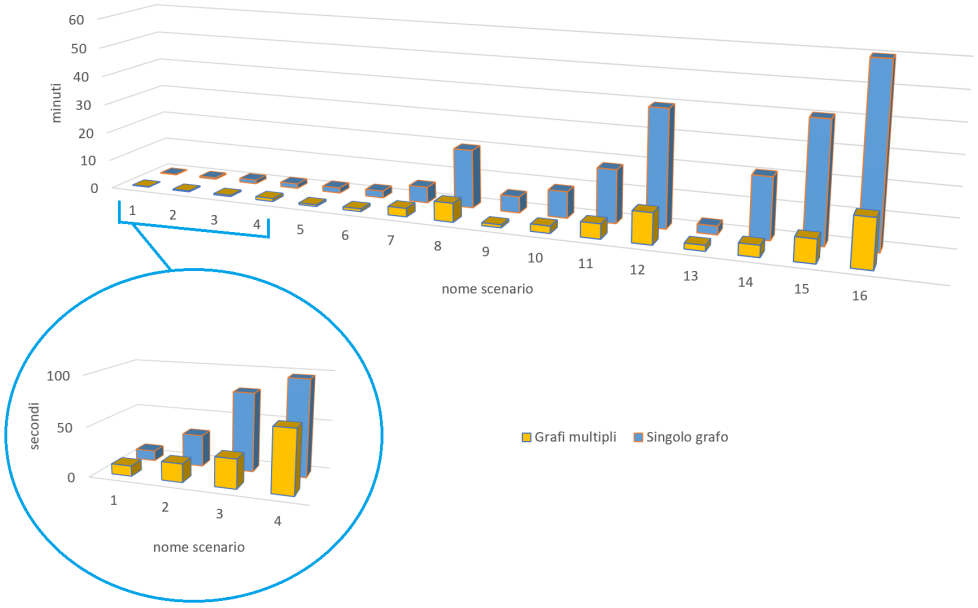

Figura 9: Risultati benchmark ChatTest, singolo grafo vs grafi multipli.

Scenario	Utenti (n)	Messaggi che ogni utente invia ad ogni altro utente (m)	Messaggi totali (M)	Numero grafi (Singolo grafo)	Singolo grafo [ms]	Numero grafi (Grafi multipli)	Grafi multipli [ms]
1	5	5	100	1	10405	10	10364
2	5	10	200	1	32147	10	18512
3	5	20	400	1	79767	10	29072
4	5	40	800	1	96831	10	62649
5	5	5	450	1	110986	45	36228
6	5	10	900	1	143338	45	85323
7	5	20	1800	1	327947	45	174574
8	5	40	3600	1	1195105	45	386178
9	12	5	660	1	333697	66	59710
10	12	10	1320	1	543342	66	146849
11	12	20	2640	1	1087904	66	303745
12	12	40	5280	1	2380183	66	630838
13	15	5	1050	1	174044	105	115883
14	15	10	2100	1	1244956	105	235150
15	15	20	4200	1	2428528	105	470225
16	15	40	8400	1	3598399	105	981428

Dai risultati di questi benchmark, possiamo notare come il numero di sottoscrizioni (che nel nostro caso cresce con il numero di utenti) sia molto più gravoso del numero di messaggi, osservazione visibile anche dai test precedenti. Nello specifico prendiamo in considerazione gli scenari $8, 11$ e 14, in cui viene dimezzato il numero di messaggi (passando da 40 a 20 e infine a 10) e viene incrementato di poco il numero di utenti (che passa da 10 a 12 e infine a 15). Questi scenari mantengono circa le stesse tempistiche tra loro, rispetto agli altri.
La seconda osservazione è a favore dell'ambiente per cui è stato pensato il SEPA, il WoT. Quest'architettura è stata pensata per uno scambio di informazioni che sia più continuo e povero di triple rispetto a uno scambio di informazioni molto verboso e raro, in più in campo WoT si ha un numero di entità abbastanza elevato ed è consigliato lavorare su grafi differenti, racchiudendo ogni entità in un suo grafo che possa contenere le triple di suo interesse e pertinenza. Quindi lo scenario per cui il SEPA è consigliato, preferisce un numero di grafi elevato, un numero di triple ridotto, con la possibilità di aggiornamenti continui a distanze ravvicinate di tempo.
I risultati di questo benchmark mettono in evidenza la maggiore agilità del SEPA nella gestione delle sottoscrizioni su un numero elevato di grafi rispetto a quella su un singolo grafo.
Da questi test possiamo dedurre che l'ottimizzazione deve incentrarsi sullo smaltimento rapido delle sottoscrizioni, poiché questo risulta essere il fattore di latenza dominante.

6 Ottimizzazione delle prestazioni: l'algoritmo AR

Considerando che, come dimostrato nel capitolo 4.2, non sia sempre possibile ricostruire gli aggiornamenti dei bindings da aggiungere e rimuovere per le sottoscrizioni, solamente a partire dalla update e dai suoi effetti sulla base di conoscenza, per poter ottimizzare l'algoritmo nativo si agisce sul filtraggio delle SPU da attivare all'arrivo di una update e nello snellire il processamento della query di sottoscrizione, nel caso delle SPU attivate.
In entrambi i casi è necessario manipolare la richiesta di update allo scopo di estrarre le triple che questa andrà ad aggiungere e rimuovere dalla base di conoscenza. Infatti, il protocollo SPARQL 1.1 non prevede alcun messaggio di risposta a fronte di una richiesta di update e quindi non è previsto conoscere quelli che sono stati gli effetti dell'update. La mancata conoscenza delle triple effettivamente aggiunte e rimosse da una update preclude la possibilità di costruire algoritmi diversi da quello attualmente implementato che consiste nel ripetere la query di sottoscrizione e confrontare i risultati ottenuti con quelli precedentemente memorizzati prima dell'update.
Definiamo N il numero di sottoscrizioni, $t(u)$ il tempo necessario all'esecuzione dell'update, $t(q_i)$ il tempo di esecuzione della query di sottoscrizione i e $t(c_i)$ il tempo necessario al confronto tra lo stato precedente e successivo alla update per ottenere i risultati da inoltrare come aggiornamento alla sottoscrizione i con $i \in [1 - N]$.
T_1 è una semplificazione del tempo necessario all'algoritmo nativo e risulta:

$$T_1 = t(u) + \sum_{i=1}^{N} (t(q_i) + t(c_i)) \tag{3}$$

Per ridurre T_1, l'algoritmo AR fornirà i dati per diminuire N applicando un ulteriore filtraggio delle sottoscrizioni interessate. Tali dati potrebbero essere utilizzati anche per semplificare le query delle sottoscrizioni riducendo i tempi $t(q_i)$. L'algoritmo AR per produrre i dati necessari introdurrà un overhead su $t(u)$, bisognerà quindi essere in grado di guadagnare più tempo di tale overhead.
Lo scopo dell'algoritmo AR qui presentato è quello di ottenere le quadruple aggiunte e rimosse relative all'esecuzione update SPARQL1.1 sulla base di conoscenza.

6.1 Algoritmo AR

Facendo riferimento direttamente agli effetti di una update sulla base di dati, ignorando per ora il suo effetto sulle singole sottoscrizioni che dovrebbero poi essere notificate, il risultato finale si può rappresentare come un insieme di quadruple, o triple, aggiunte ed un insieme di quadruple, o triple, rimosse dalla base di dati. Nel corso di questo progetto le quadruple aggiunte alla base di dati vengono denominate anche "added" e le rimosse "removed", da qui il nome dell'algoritmo AR (Added Removed, Aggiunte Rimosse). Queste triple vengono definite come globali proprio perché interessano la totalità della conoscenza, coinvolgendo zero o più sottoscrizioni.
Possiamo rappresentare il processo di estrazione delle added e removed dal punto di vista insiemistico (figura 10) definendo l'insieme $X = \{\alpha_0, \alpha_1, \alpha_2, \Omega_0, \Omega_1\Omega_2\}$ come lo stato del endpoint, denominato KB o base di conoscenza, anteriore alla update. L'insieme $Y = \{\beta_0, \beta_1, \beta_2, \beta_3, \Omega_0, \Omega_1, \Omega_2\}$ come lo stato a posteriori dell'esecuzione della update, dove $\alpha_i, \beta_i, \Omega_i \forall i$ sono triple o quadruple. Possiamo notare che l'insieme $X - Y = \{\alpha_0, \alpha_1, \alpha_2\}$ rappresenta le triple rimosse dalla update e l'insieme $Y - X = \{\beta_0, \beta_1, \beta_2, \beta_3\}$ rappresenta quelle aggiunte. L'insieme rappresentante le added e removed è dunque

$G = (X - Y) \cup (Y - X) = \{\alpha_0, \alpha_1, \alpha_2\, \beta_0, \beta_1, \beta_2, \beta_3\}$.
I due insiemi $X - Y$ ed $Y - X$ non si possono definire come disgiunti, dato che SPARQL1.1 permette update in cui una tripla rimossa dalla clausola Delete può essere presente anche nella clausola di Insert, di conseguenza togliendo e rimettendo la medesima tripla nella base di conoscenza.

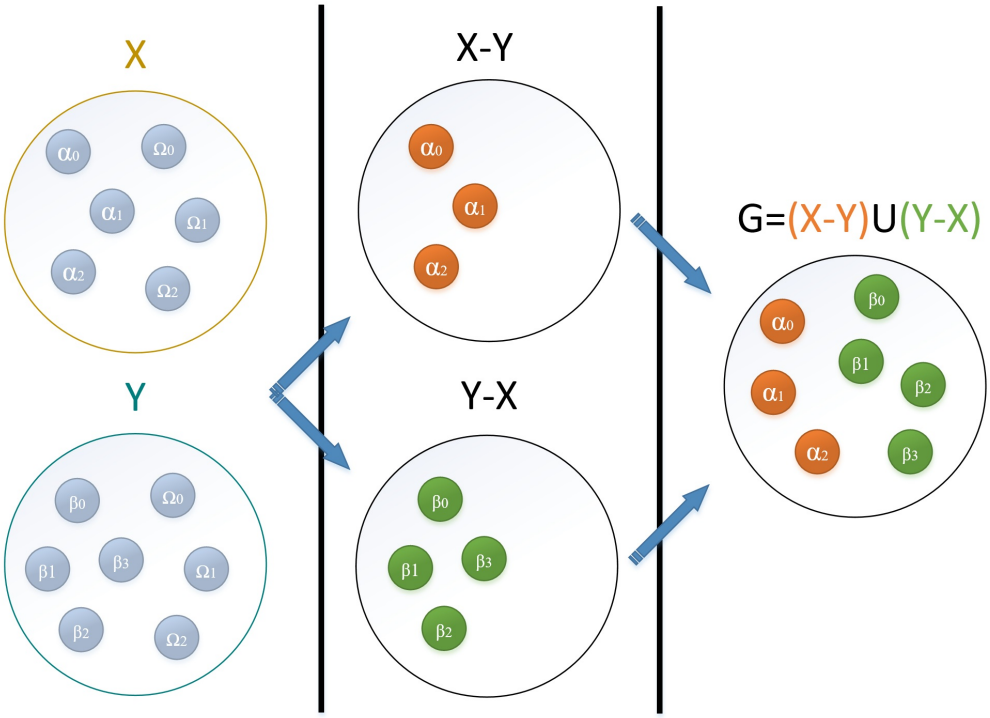

Figura 10: Vista insiemistica del processo di estrazione delle added e removed.

L'operato dell'algoritmo AR per ottenere le triple dell'insieme G si può suddividere in quattro fasi.
La prima fase consiste in un'analisi della update per ricondursi alla sua tipologia che è stata classificata nelle seguenti classi: UpdateDataInsert, UpdateDataDelete, UpdateDeleteWhere, UpdateModify, UpdateClear, UpdateDrop, UpdateCopy oppure UpdateAdd. Nel caso la update sia di tipo UpdateDataInsert, UpdateDataDelete, UpdateDeleteWhere oppure UpdateModify vengono estratte le informazioni riguardanti i grafi coinvolti, le triple esplicite e non (ricordiamo che per quadrupla o tripla esplicita si intende una quadrupla o tripla priva di variabili) e se presente viene estratta anche la clausola *"where"*. In seguito verranno generate una o più richieste di SPARQL1.1 di tipologia Construct. Le richieste di Construct sono necessarie a risolvere la clausola *"where"*, per quindi passare dalle eventuali triple non esplicite a triple esplicite, definendo "triple non esplicite" un Triple Pattern che possiede almeno una variabile.
Vengono costruiti al più due modelli di Construct per ogni grafo interessato, una per la conoscenza che si vuole aggiungere al grafo e l'altra per la conoscenza da rimuovere. In presenza della clausola *"where"* nella update originaria, la si accoda a tutti i modelli di

Construct appena costruiti. Le rimanenti tipologie di update (UpdateClear, UpdateDrop, UpdateCopy ed UpdateAdd), lavorano esclusivamente a livello di grafo e quindi sarebbe controproducente ricavarne le added e removed, in quanto sarebbero l'intero elenco di triple dei grafi coinvolti Queste vengono perciò ignorate, lasciando comunque opportunità future per il loro trattamento.
La seconda fase si occupa dell'esecuzione delle eventuali richieste di Construct precedentemente generate. Per ogni Construct eseguita viene analizzato il modello risultante estraendone le quadruple esplicite, raggruppandole per il loro grafo di appartenenza. Le quadruple ottenute da questa fase prendono il nome di "candidate added e removed" e vengono rappresentate tramite una lista di grafi con annesse due liste di triple, una per le triple che la update tenta di aggiungere ed una per le triple che tenta di rimuovere. Se nessun grafo è esplicitato nella update, la lista di grafi conterranno un solo elemento, il DefaultGraph con le annesse liste di triple. Se è presente la clausola *"with"* il DefaultGraph verrà sostituito dal grafo indicato dalla clausola. Le richieste di Construct eseguono anche una prima validazione delle triple candidate come added e removed, dato che vengono restituite solo triple che rispettano il modello della Construct stessa.
La terza fase si occupa della validazione delle candidate added e removed che la update mira a manipolare. Lo scopo è quello di capire se una tripla che la update vorrebbe aggiungere alla base di dati non sia già presente e quindi debba effettivamente risultare come tripla da aggiungere oppure esista già e quindi debba essere scartata. Allo stesso modo avviene per le triple che la update cerca di rimuovere, se non risultano nelle base di dati prima dell'esecuzione della update, devono essere scartate in quanto non verrebbero effettivamente rimosse dalla update. Una volta scartate le triple non valide si ottiene il vero insieme di added e removed o insieme G in figura 10. Questa fase prende anche il nome di fase di "ASKs", nome che deriva dalla richiesta SPARQL1.1 Ask utilizzata per validare una singola tripla o quadrupla. L'algoritmo esegue una richiesta Ask per ogni tripla candidata ottenendo così conferma o meno della sua esistenza sulla base di dati a priori dell'esecuzione della update. Vedremo in seguito che questa fase è stata ottimizzata in modi diversi in quanto l'esecuzione di una richiesta per ogni tripla risultava troppo costosa.
L'ultima fase si occupa della costruzione di una update alternativa alla update originale. Ottenendo le added e removed ci si può ricondurre ad una update, che chiamiamo UpdateInsertDelete, che consiste in una update con sole quadruple esplicite e senza clausola *"where"* dunque richiedente un tempo di esecuzione da parte dell'endpoint minore o uguale rispetto alla update originaria. La UpdateInsertDelete risulta al più la concatenazione di una update di tipo UpdateDataInsert e una update di tipo UpdateDataDelete. Non in tutte le casistiche sono necessariamente presenti entrambe le update.
Facciamo un esempio pratico mostrando i passaggi più importanti dell'algoritmo AR, prendendo come riferimento la update di tipologia UpdateModify (listing 6), il cui intento è quello di rimuovere tutte le triple che come oggetto e predicato possiedono "ub:memberOf <http://www.unibo.it>" all'interno del grafo "<http://lumb/for.sepa.test/workspace/defaultgraph0>", per poi inserire nel medesimo grafo una nuova tripla esplicita.

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
DELETE {
  GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0>  {
    ?s ?p ?o
  }
} INSERT {
  GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0> {
    <http://www.unibo.it/Student_B0> ub:memberOf <http://www.unibo.it> .
  }
} WHERE {
  ?s ?p ?o .
  ?s ub:memberOf <http://www.unibo.it>
}
```

Listing 6: Esempio UpdateModify.

Otteniamo le seguenti due Construct, ricordando che il grafo trattato è uno solo.

```
% construct delle triple da eliminare
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
CONSTRUCT {?s  ?p  ?o} WHERE {
?s  ?p  ?o ;
     ub:memberOf <http://www.unibo.it>
}

% construct delle triple da aggiungere
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
CONSTRUCT {
    <http://www.unibo.it/Student_B0> ub:memberOf  <http://www.unibo.it> .
} WHERE {
      ?s  ?p   ?o ;
      ub:memberOf  <http://www.unibo.it>
}
```

Listing 7: Esempio di richieste Construct.

L'output sarà una lista di triple per ogni Construct. Le Construct vengono usate per risolvere le variabili ed ottenere una lista di triple esplicite, nell'esempio corrente la conoscenza che si vuole rimuovere non è rappresentabile a priori con una lista di triple esplicite, dato che nella update originale è espressa tramite le variabili "?s ?p ?o" che rappresentano l'insieme delle triple che devono essere risolte dalla update stessa.

Osservazione: nella Construct per le triple da aggiungere (listing 7), possiamo notare che la clausola *"where"* è superflua, dato che serviva solo a risolvere il modello per le triple da rimuovere, mentre le triple da aggiungere erano già esplicite.

Eseguite tutte le richieste di Construct sull'endpoint e ottenute le quadruple che rispettano il modello della update, si esegue una richiesta di Ask per ogni quadrupla, venendo così a sapere se la quadrupla esiste già nella base di dati oppure no.
Se una quadrupla che dovrebbe essere aggiunta secondo la Construct risultasse, tramite Ask, già presente sulla base di dati, verrà rimossa dalla lista delle aggiunte. Allo stesso modo, se una quadrupla da rimuovere secondo la Construct non dovesse esiste sulla base di dati, verrà scartata e non dovrà dunque essere considerata come quadrupla da rimuovere.

Rimanendo sempre sullo stesso esempio, la richiesta Ask, per la quadrupla risultante dalla construct delle triple da aggiungere sarà:

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
ASK  {
    GRAPH  <http :// lumb/for.sepa.test/workspace/defaultgraph0> {
        <http :// www.unibo.it/Student_B0> ub:memberOf   <http ://www.unibo
    .it>
    }
}
```

Listing 8: Esempio di una richiesta Ask.

La terza fase giunge al termine, con la lista delle added e removed dalla quale viene generata la nuova update, UpdateInsertDelete, che ha gli stessi effetti sulla base di conoscenza dell'endpoint di quelli della update originaria. In questo caso la richiesta UpdateInsertDelete è formata dalla concatenazione di una UpdateDataDelete e una UpdateDataInsert. La richiesta UpdateInsertDelete (listing 9) deriva dalla lista di quadruple da rimuovere presenti in added e removed e la richiesta UpdateDataInsert deriva dalle analoghe quadruple da aggiungere. In questo modo la richiesta avrà solo quadruple esplicite e potrà essere sostituita alla update originaria per poi essere inoltrata all'endpoint. La UpdateDataInsert se confrontata con la update originale, avrà bisogno di meno risorse per il processamento da parte dell'endpoint.

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
DELETE DATA {
     GRAPH  <http :// lumb/for.sepa.test/workspace/defaultgraph0> {
        <http :// www.unibo.it/Student_A0> ub:memberOf   <http ://www.unibo
    .it>
    }
};
INSERT DATA {
    GRAPH  <http :// lumb/for.sepa.test/workspace/defaultgraph0> {
        <http :// www.unibo.it/Student_B0> ub:memberOf   <http ://www.unibo
    .it>
    }
}
```

Listing 9: Esempio di una richiesta UpdateInsertDelete.

Possiamo formulare il tempo necessario all'esecuzione dell'algoritmo AR come la somma dei tempi necessari alle quattro fasi. La prima e l'ultima fase hanno tempi trascurabili rispetto alle altre due fasi. Il tempo necessario all'algoritmo risulta essere riconducibile dal numero di richieste SPARQL1.1 necessarie per la seconda e terza fase, pari a:

$$T_2 = \sum_{i=1}^{N} t(Construct_i) + \sum_{j=1}^{M} t(Ask_j) \tag{4}$$

Si è per tanto cercato di diminuire il più possibile T_2 riducendo N ed M.

6.1.1 Ottimizzazione della Construct

Prendendo in esame una Construct trattata dall'algoritmo, questa viene generata per poi essere inviata e risolta dall'endpoint. Non potendo ottimizzare i tempi di risoluzione della richiesta da parte dell'endpoint e sapendo che i tempi di generazione della Construct non sono ingenti, possiamo concentrare la soluzione sui tempi di invio della richiesta e ricezione del risultato, parliamo dunque di overhead HTTP.
Ogni richiesta inoltrata all'endpoint ha una buona parte del costo dovuta alla trasmissione tramite il protocollo HTTP. L'algoritmo tratta due Construct per ogni grafo presente in ogni richiesta di update. Nei casi di update con N grafi una buona parte del tempo viene dunque impiegata all'overhead HTTP per inviare M richieste di Construct, dove $M \in [N, 2 * N]$ dato che ogni grafo può avere triple da aggiungere e/o da rimuovere.
Si è pertanto cercato di ottimizzare le Construct, generando ed eseguendo una sola richiesta cumulativa. Questo non ha avuto successo, dato che il protocollo SPARQL1.1 non prevede la concatenazione tramite ";" di Construct come invece è previsto per le normali query.

Osservazione: il SEPA è stato pensato per l'ambiente WoT, è dunque inusuale l'utilizzo di richieste di update che riguardano un grande numero di triple o di grafi. La norma è la modifica di un numero basso di triple che fanno riferimento a un solo grafo. Per cui, restando in questo ambito la mancata ottimizzazione, di cui sopra, incide relativamente poco.

Non potendo raggruppare le Construct si è in seguito cercato di diminuirne il numero risparmiando il tempo di generazione, inoltro ed esecuzione di tutte le Construct che potessero risultare superflue. Prendendo tutte le tipologie di update che vengono analizzate per ottenere le relative Construct, si può notare che nel caso di una UpdateDataInsert e una UpdateDataDelete, non essendoci la clausola *"where"*, le triple o quadruple trattate sono già tutte esplicite e quindi, in questi casi, non sono necessarie le Construct. Pertanto alla recezione di una richiesta di update di tipo UpdateDataInsert oppure UpdateDataDelete, vengono direttamente estrapolate le quadruple per poi passarle alla fase successiva, in cui vengono eseguite le Ask.

6.1.2 Ottimizzazione delle ASK

Per le richieste delle Ask le possibili ottimizzazioni sono le medesime che per le richieste di tipologia Construct, con l'aggravante che data una update il numero di Ask necessarie è mediamente molto più elevato del numero di Construct, dato che per ogni quadrupla interessata dalla richiesta di update serve una richiesta di tipo Ask. Il problema principale rimane l'overhead della richiesta HTTP, come visto per le richieste di Construct, che viene moltiplicato per il numero di Ask necessarie.
SPARQL1.1 come nel caso della Construct non permette la concatenazione di più query di tipo Ask, si è dunque cercato di ricondursi ad un unica query di tipo Select , che potesse sostituire un insieme di richieste di tipo Ask, ottenendo lo stesso risultato. La richiesta di Select sostitutiva dovrà dunque essere equivalente all'insieme di richieste Ask, ponendosi come obbiettivo la convalida delle quadruple ottenute dalle fasi precedenti, per conoscere quali siano già presenti sulla base di conoscenza e quali no. Non tutte le alternative trovate consistono in una sola richiesta query per ogni update, alcune necessitano di più richieste Select distinte.
Le query di tipo Select candidate come sostitute sono:

- AsksAsSelectExplicitGraph, necessita di due Select per ogni grafo che contiene sia triple da aggiungere che da rimuovere e una sola Select per ogni grafo che possiede solo triple da aggiungere o solo triple da rimuovere. Utilizza la clausola "VALUES" su soggetto, predicato ed oggetto. Restituisce la lista di triple già esistenti nella base di dati.

```
SELECT ?s ?p ?o {
  GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0> {
      ?s ?p ?o.
  }
  VALUES (?s ?p ?o) {
      (<S0><P0><O0>)
      (<S1><P1><O1>)
            ...
      (<Sn><Pn><On>)
  }
}
```

Listing 10: Esempio di AsksAsSelectExplicitGraph.

- AsksAsSelectGraphAsVar, necessita al più di due Select, una per le triple da aggiungere a prescindere dal grafo di appartenenza e una per le triple da rimuovere sempre per tutti i grafi coinvolti. Utilizza la clausola "VALUES" per generare le quadruple, anche il grafo è posto come variabile controllata dalla clausola.

```
SELECT ?g ?s ?p ?o {
      VALUES (?g ?s ?p ?o) {
          (<G0><S0><P0><O0>)
          (<G1><S1><P1><O1>)
            ...
          (<Gn><Sn><Pn><On>)
      }
      GRAPH ?g {
        ?s ?p ?o.
      }
}
```

Listing 11: Esempio di AsksAsSelectGraphAsVar.

- AsksAsSelectFilter, molto simile alla AsksAsSelectExplicitGraph, necessità di due Select per ogni grafo che contiene sia triple da aggiungere che da rimuovere e una sola Select per ogni grafo che possiede solo triple da aggiungere o solo triple da rimuovere. Viene utilizzata la clausola "FILTER" in alternativa alla clausola "VALUES" usata in "AsksAsSelectExplicitGraph".

```
SELECT  ?s ?p ?o WHERE{
  GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0>{
    ?s ?p  ?o.
  }
  FILTER(
    ?s = <S0> &&  ?p = <P0> && ?o = <O1> ||
    ?s = <S1> &&  ?p = <P1> && ?o = <O1> ||
              ...
    ?s = <Sn> &&  ?p = <Pn> && ?o = <On> ||
  )
}
```

Listing 12: Esempio di AsksAsSelectFilter.

- AsksAsSelectExistsList, necessita di una sola Select, dato che una sola richiesta è in grado di coprire tutti i grafi sia per le triple da aggiungere sia per quelle da rimuovere. Viene utilizzata la clausola "VALUES" per generare ennuple composte dalla quadrupla e da un numero utilizzato come identificativo della quadrupla stessa. L'identificatore servirà a ricondurre la quadrupla alla sua origine che è la lista di quadruple appartenenti al grafo che a sua volta appartiene alla lista delle aggiunte o alla lista delle rimosse. Grazie alla concatenazione delle clausole "BIND" ed "EXISTS" siamo in grado di ottenere come risultato della Select, un booleano, associato al numero identificativo della quadrupla, che indica se la quadrupla esiste o non è presente sulla base di dati.

```
SELECT ?x ?i {
    VALUES (?g ?s ?p ?o ?i) {
      (<G0><S0><P0><O0>0)
      (<G1><S1><P1><O1>1)
              ...
      (<Gn><Sn><Pn><On>n)
    }
    BIND(EXISTS{GRAPH ?g {?s ?p ?o}} AS ?x)
}
```

Listing 13: Esempio di AsksAsSelectExistsList.

Ricordiamo che utilizzando la vera richiesta Ask fornita da SPARQL1.1, servirebbe una richiesta Ask per ogni quadrupla e che per l'ottimizzazione di questa fase dell'algoritmo, si punta a diminuire il numero di richieste all'endpoint. Le sopra elencate alternative alla Ask forniscono tutte tempi di esecuzione inferiori rispetto all'esecuzione di molteplici richieste di Ask.
Sono state scartate tutte quelle richieste di select che dipendevano dal numero di grafi coinvolti, come la richiesta di "AsksAsSelectExplicitGraph" e la richiesta di "AsksAsSelectFilter". Questo perchè al crescere del numero di grafi cresceva il tempo di esecuzione, a causa del crescere del numero di richieste HTTP verso l'endpoint. Le richieste rimanenti, sono state confrontate ed è emerso che la richiesta migliore risulta essere "AsksAsSelectExistsList", necessitando di una sola richiesta in ogni casistica, a differenza dell'utilizzo della rimanente "AsksAsSelectGraphAsVar" che necessita di al più due richieste verso

l'endpoint. In figura 11 sono presenti i grafici dei test effettuati su Virtuoso e su Blazegraph, utilizzando la Ask canonica e i grafici delle due migliori select alternative. I grafici e i test relativi seguono la struttura e le modalità riportate al paragrafo 6.2.3, in cui vengono spiegati nel dettaglio. In breve sull'asse delle ordinate vi è il tempo espresso in millisecondi, sull'asse delle ascisse vi è il numero di triple utilizzate nel test. Ogni grafico è il risultato di 7 campioni, le richieste sono state fatte con $1, 2, 4, 8, 16, 32$ e 64 triple, provando 6 update distinte, sulla base di dati caricata sugli endpoint formata da più di 100.000 triple.

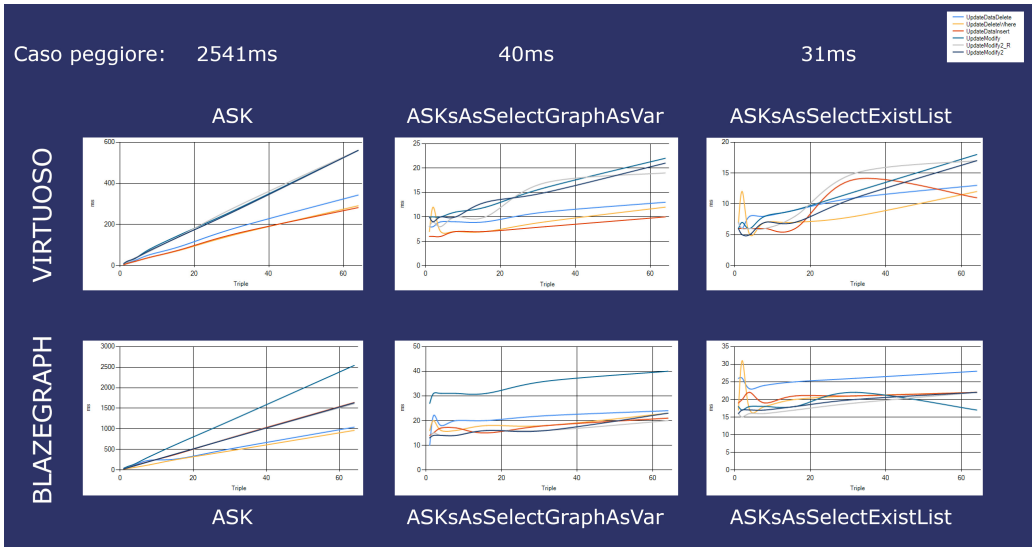

Figura 11: Grafici delle due alternative più significative alla richiesta di Ask.

6.1.3 Implementazione dell'algoritmo

Rappresentiamo il flusso dell'algoritmo AR in figura 12, indicando con rombi i blocchi di scelta, con i rettangoli blu le procedure e con i rettangoli bianchi le iterazioni. Lo schema si riferisce all'algoritmo finale, che prevede le ottimizzazioni sulle Construct e sulle Ask. Le fasi riportate sono quelle descritte all'inizio del capitolo, con qualche dettaglio aggiuntivo. In particolare la prima fase si comporta in modo diverso a seconda della tipologia rilevata dell'update in ingresso. Nel caso la update sia di tipo UpdateDataInsert o UpdateDataDelete vengono direttamente estratte le quadruple esplicite, raggruppandole per il loro grafo di appartenenza. In questo caso non sarà necessaria la costruzione di alcuna richiesta di Construct. Notare che nel caso di una UpdateDataInsert le relative strutture dati che contengono la lista di grafi e annesse triple che rappresenta la clausola *"delete"* saranno vuote, in ugual modo per una UpdateDataDelete le strutture dati che rappresentano la clausola di *"insert"* saranno vuote. Se nessun grafo è esplicitato nella update, le strutture dati che possiedono i vari grafi coinvolti conterranno un solo elemento, il DefaultGraph, se è presente la clausola *"with"* il DefaultGraph verrà sostituito dal grafo indicato dalla clausola. Nel caso di una update di tipologia UpdateDeleteWhere oppure UpdateModify vengono estratte le informazioni riguardanti i grafi coinvolti, la clausola *"where"*, le triple esplicite e non esplicite. In presenza di una di queste due ultime tipologie, verranno anche generate una o più richieste di Construct. Per eseguire il primo passaggio della prima fase, il riconoscimento della tipologia della update, è stata utilizzata la libreria Jena [1], utile anche come supporto per l'estrazione delle quadruple dalle richieste SPARQL1.1.
Possiamo notare dallo schema dell'algoritmo, che la seconda fase non sia obbligatoria per tutte le tipologie di update, nello specifico per le tipologie UpdateDataInsert e UpdateDataDelete, essendo prive di Construct.
L'obbiettivo delle prime due fasi è quello di risolvere tutte le quadruple che contengono delle variabili, in modo da ottenere solamente quadruple esplicite. La terza fase si occuperà poi di filtrare le quadruple esplicite formando così le added e removed.
Ricordiamo che la terza fase era stata pensata per svolgere una richiesta Ask per ogni quadrupla, ma che questo causava una perdita prestazionale notevole, obbligandoci a ricondurci ad una query SPARQL1.1 sostitutiva. In questo capitolo sono state esposte varie tipologie di query sostitutive. Non tutte le query necessitano di una sola richiesta, per alcune tipologie è necessaria la creazione e l'invocazione di più richieste verso l'endpoint. Nel diagramma, alla terza fase, si mostra solo la casistica in cui venga utilizzata la query "AsksAsSelectExistsList" che svolge il compito delle Ask con una sola invocazione, nel paragrafo 6.1.2 è stato dimostrato come questa sia la più performante. Vedremo in seguito che l'algoritmo è stato attrezzato con la possibilità di cambiare alcune parti della sua logica applicativa in base all'endpoint che si sta utilizzando, nel caso specifico della terza fase l'algoritmo può selezionare la query sostitutiva più adatta. Nel caso venga utilizzata una tipologia di query sostitutiva che richieda più iterazioni, concettualmente verrà rieseguita più volte la terza fase. Facendo un esempio concreto, se si utilizzasse la query "AsksAsSelectGraphAsVar", la terza fase verrebbe eseguita al più due volte, una per le quadruple da aggiungere e una per le quadruple da rimuovere. Come ultima fase è mostrata la generazione della UpdateInsertDelete sostituta della update originale.

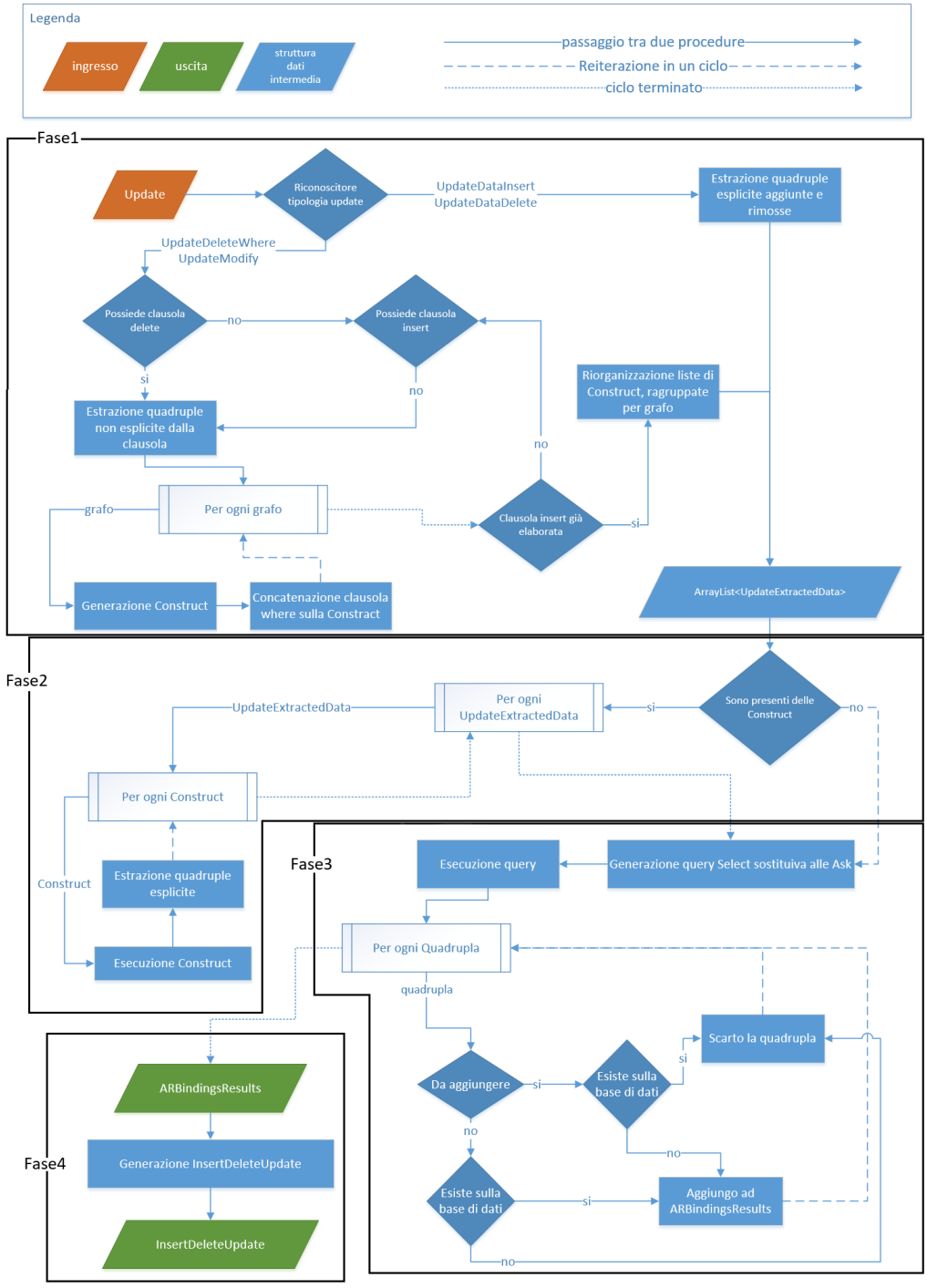

Figura 12: Diagramma di flusso dell'algoritmo AR.

Le strutture dati principali sono "UpdateExtractedData", con lo scopo di agevolare la manipolazione dei dati durante l'esecuzione dell'algoritmo, ed "ARBindingsResults" che racchiude il risultato finale da cui verrà anche generata la update InputDeleteUpdate durante la quarta fase. Ogni "UpdateExtractedData" si riferisce ad un grafo coinvolto nella update, contenendo le eventuali due Construct (sotto forma di stringa), una per le triple da aggiungere e una per quelle da rimuovere. Conterrà anche un booleano che indica l'assenza di entrambe le Construct, nel caso di UpdateDataInsert e UpdateDataDelete, utilizzato all'inizio della seconda fase, per capire se è possibile evitarla passando direttamente alla successiva. Un singolo "UpdateExtractedData" racchiude anche una coppia di "BindingsResults", uno per le aggiunte e uno per le rimosse. Questa coppia di "BindingsResults" racchiudono dunque tutte le triple che fanno riferimento al grafo corrente, candidate a diventare added e removed e che verranno verificate nella terza fase. I "BindingsResults", contenenti una lista di bindings, verranno popolati durante la prima fase, nel caso di update di tipologia UpdateDataInsert e UpdateDataDelete, oppure durante la terza fase nei rimanenti casi. Il risultato finale "ARBindingsResults" conterrà esclusivamente le quadruple da aggiungere e rimuovere filtrate nella terza fase, anche chiamata fase di ASKs. Le quadruple finali non sono più contenute in liste di triple separate per grafo di appartenenza, ma in due liste di quadruple, una per le aggiunte e una per le rimosse. Le due liste di "ARBindingsResults" sono rappresentate tramite due "BindingsResults". La differenza sostanziale tra le due strutture dati è dunque l'organizzazione delle triple e dei loro grafi. La struttura dati utilizzata durante il processamento consiste in una lista di "UpdateExtractedData" che rappresenta una lista di grafi, in cui ad ogni grafo vengono collegate al più due liste di triple, una per le aggiunte e una per le rimosse. La struttura finale è invece riorganizzata dal punto di vista delle aggiunte e rimosse, contenendo due liste di quadruple, raggruppate non per il grafo di appartenenza ma per l'operazione di aggiunta o rimozione a cui mira la update.

6.1.4 Casi particolari legati ai vari endpoint

Esistono aspetti che SPARQL1.1 non formalizza nel suo standard, lasciandoli così aperti a possibili interpretazioni differenti. Non tutti gli endpoint si comportano allo stesso modo su alcuni fronti, come la rappresentazione o il nome del DefaultGraph, la scelta di come realizzare l'ActiveGraph, l'ordine con cui una query o una update viene interpretata ed eseguita. Queste diverse implementazioni, oltre a sfociare in possibili problemi di incompatibilità, potrebbero in alcuni casi essere oggetto di studio per la ricerca di ottimizzazioni mirate e specifiche sugli endpoint.
Per far fronte a queste differenze, essendo il SEPA compatibile con tutti gli endpoint conformi a SPARQL1.1 Protocol [7], si è creata una sezione di specializzazione per poter agire al meglio a seconda dell'endpoint, in molti casi anche obbligatoriamente per evitare malfunzionamenti.
Prendendo Virtuoso e Blazegraph come endpoint di riferimento, questi ad esempio implementano in modo differente la rappresentazione dei booleani. Virtuoso, alla richiesta di una Ask o della risoluzione di una espressione booleana, restituisce degli interi "0" oppure "1", mentre Blazegraph restituisce i booleani sotto forma di "true" oppure "false". I due endpoint hanno anche vincoli diversi per la Construct, nello specifico Virtuoso restituisce i valori di soggetto, predicato ed oggetto specificando i nomi delle tre variabili come ["s","p","o"], mentre Blazegraph utilizza ["subject","predicate","object"].
Nei vari test effettuati, abbiamo notato che l'ordine all'interno della clausola where, cambia drasticamente i tempi di esecuzione di alcune richieste. Prendendo in particolare

la richiesta Select AsksAsSelectGraphAsVar in sostituzione alle Ask, da prima abbiamo notato una forte differenza del tempo necessario alla sua esecuzione tra Virtuoso e Blazegraph. Facendo lo stesso test di sforzo, all'ultimo campionamento, il più impegnativo, Virtuoso impiegava decine di secondi e il Blazegraph decine di millisecondi. Sono riportati di seguito i due grafici (figura 13 ed figura 14) che rappresentano il risultato di questo test, i grafici in questione seguono la struttura che verrà spiegata in dettagli al paragrafo 6.2.3.

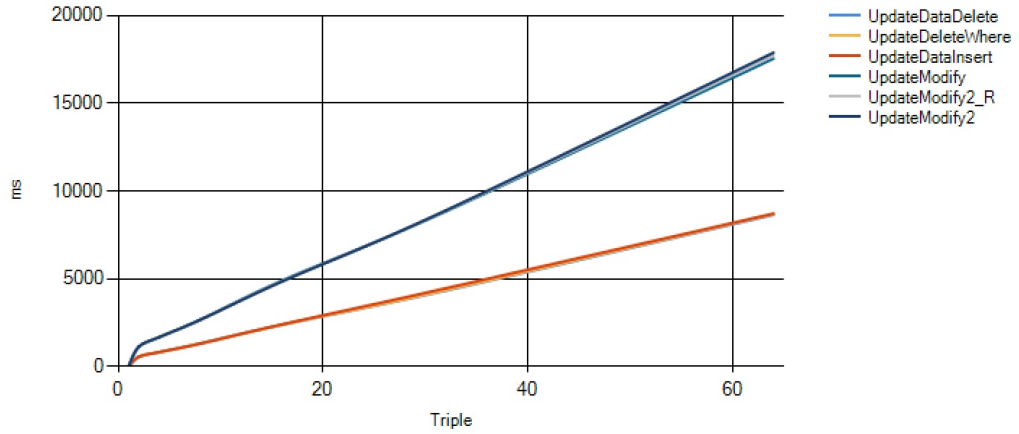

Figura 13: Grafico AsksAsSelectGraphAsVar lenta su Virtuoso.

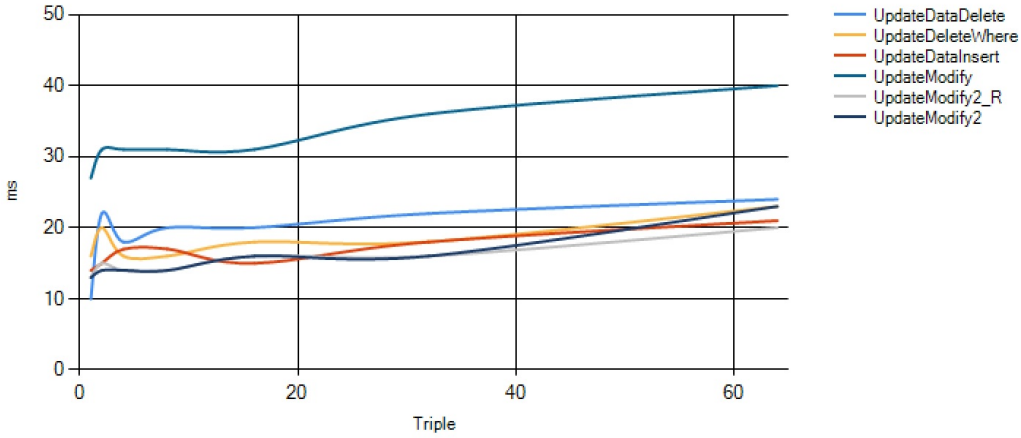

Figura 14: Grafico AsksAsSelectGraphAsVar con Blazegraph.

Si intuiva che il problema dovesse essere legato al tipo di Select, dato che in generale Virtuoso risulta più veloce di Blazegraph e che una differenza tale da cambiare ordine di grandezza, da secondo a millisecondi, era inaccettabile. Dopo aver ricercato la possibile causa, questa è risultata essere l'ordine delle due clausole "VALUES" ed "GRAPH" all'interno della query AsksAsSelectGraphAsVar. Su Blazegraph non produce alcuna differenza invertirne l'ordine, mentre Virtuoso risponde in pochissimo tempo se la clausola

"VALUES" viene definita prima delle clausola "GRAPH", come riportato nel listing 11. I risultati del medesimo test con questa variante di AsksAsSelectGraphAsVar risultano perfino migliori su Virtuoso rispetto che su Balzegraph, come mostra il grafico di figura 15.

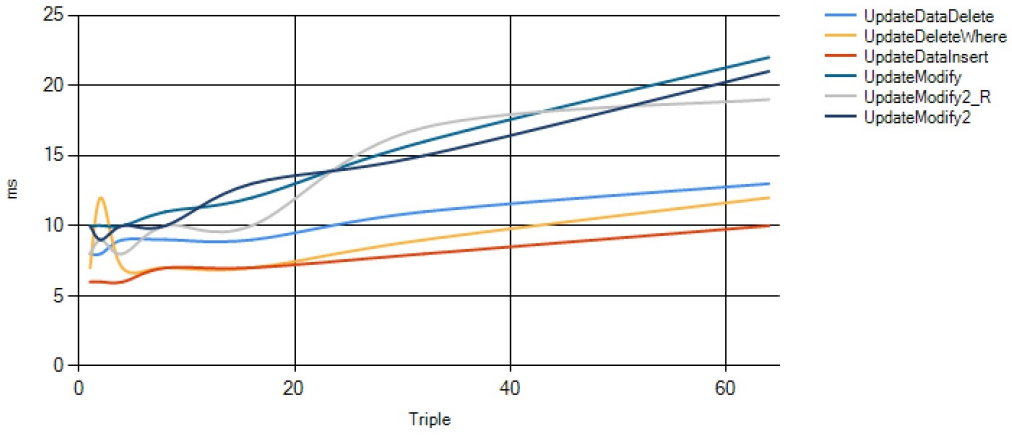

Figura 15: Grafico AsksAsSelectGraphAsVar con Virtuoso.

Dopo altri test, si è notato anche un errore nell'endpoint Virtuoso che ha reso insicuro l'utilizzo della "AsksAsSelectExistsList". Nell'implementazione della sezione di specializzazione viene quindi data la possibilità di selezionare l'alternativa alle Ask più opportuna. Nel caso il SEPA facesse riferimento a Virtuoso come endpoint, la sezione di specializzazione forzerà l'utilizzo della richiesta "AsksAsSelectGraphAsVar" altrimenti continuerà ad essere impiegata la richiesta "AsksAsSelectExistsList". In questo modo si fornisce l'ottimizzazione migliore e si evitano problematiche di compatibilità.
L'errore presente solo su Virtuoso, sembrerebbe riguardare la risoluzione del tipo dell'oggetto delle triple. Ci sono delle casistiche in cui una richiesta di "AsksAsSelectExistsList" che tratta triple con oggetti di tipologia mista non risponde come dovrebbe. Poniamoci nel caso specifico in cui la stessa richiesta trattasse una tripla con in oggetto un URI, che denominiamo A e un'altra con in oggetto un literale che denominiamo B. Stiamo coinvolgendo un solo grafo e questo possiede solo la tripla A. Ci si aspetterebbe che la richiesta "AsksAsSelectExistsList" fornisca come risposta l'affermazione dell'esistenza della tripla A e la negazione dell'esistenza della tripla B. Su Virtuoso questa casista fornisce la negazione dell'esistenza di entrambe le triple, negazione non corretta. Lo stesso esperimento è stato svolto con Blazegraph che invece risponde correttamente, come ci si aspetterebbe. La query in questione è stata analizzata e risolta seguendo la grammatica SPARQL1.1 ed è risultata corretta. Il caso è stato segnalato alla repository di Github ed è stata confermata la presenza dell'errore.

6.2 Validazione e analisi dell'algoritmo AR

Dopo aver ricercato e progettato l'algoritmo AR, sono stati progettati dei test specifici per poter misurare i tempi necessari al calcolo delle quadruple aggiunte e rimosse, per poter testare più alternative e selezionare le più performanti. In questa fase della tesi c'è stata una forte alternanza tra test e sviluppo, per poter implementare l'algoritmo AR consolidandone l'efficacia, la correttezza e la completezza. In questa fase sono state implementate e testate più opzioni di sottoprocedure dell'algoritmo, per poi poterne selezionare le migliori.
Per testare al meglio l'algoritmo AR è stato progettato e implementato un apposito tester, LumbTest, con relativo programma per la raccolta e l'analisi dei risultati dei test.

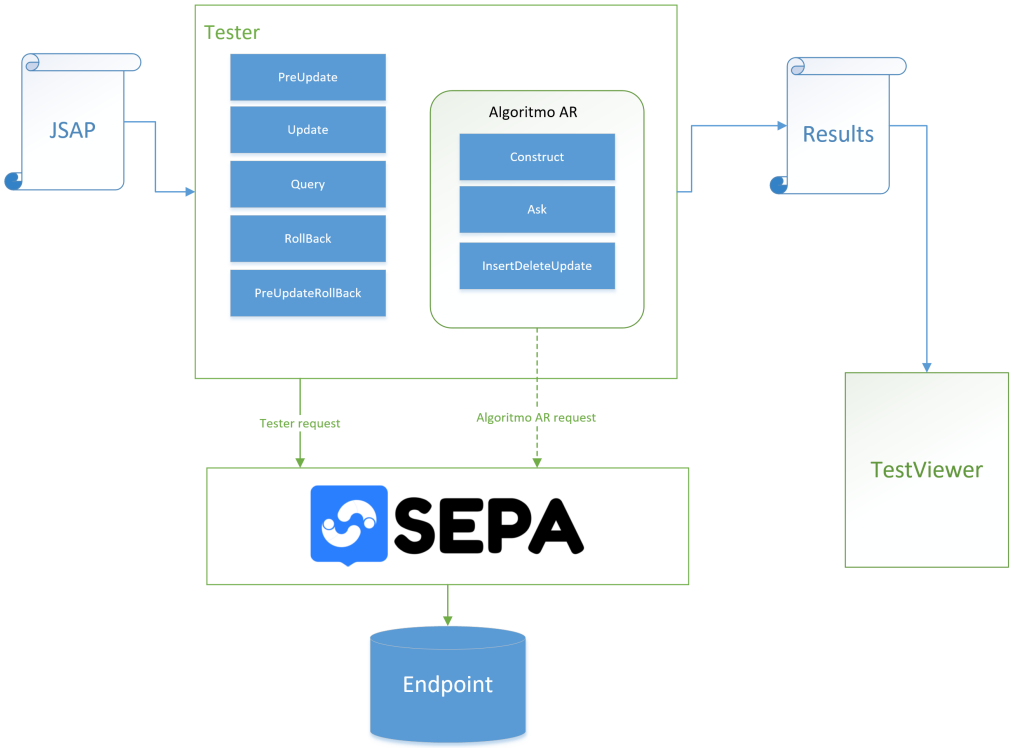

Figura 16: Schema del tester costruito appositamente per algoritmo AR.

Per lavorare al meglio sul nuovo algoritmo e poterlo disaccoppiare il più possibile dal resto del SEPA, si è deciso di operare sul nuovo algoritmo dall'esterno. Lasciando invariato il SEPA, sarà il tester a implementare l'algoritmo AR. Tutto questo è possibile poiché l'algoritmo AR si pone a monte della gestione delle sottoscrizioni, svolgendo un compito globale, il suo operato serve in ugual modo a tutte le sottoscrizioni e può essere visto come un servizio isolato.
Ogni test si basa su una update sulla quale verrà eseguito il nuovo algoritmo, tutte le richieste di update e di query necessarie verranno inoltrate al SEPA. L'interazione tra le parti, in questi test può essere vista come se l'algoritmo AR fosse un cliente del SEPA. Spostando la logica del nuovo algoritmo a livello di utilizzatore del SEPA, possiamo

ipotizzare un'ulteriore overhead dovuto all'inoltro delle richieste dal tester al SEPA, richieste che scaturiscono dell'algoritmo stesso e che quindi invece di essere generate dal SEPA e inviate all'endpoint, vengono generate dal tester, inoltrate al SEPA che a sua volta le invia all'endpoint. Con questo inoltro in più, vi è dunque la presenza di un tempo aggiuntivo nei risultati che riguardano le fasi che generano richieste di query o update direttamente dall'algoritmo (richieste indicate con linee tratteggiate in figura 16). Come semplificazione per la raccolta delle metriche temporali, questo overhead viene ignorato, in quanto stimato come fattore aggiuntivo lineare peggiorativo dei tempi, il che significa un'ulteriore diminuzione dei tempi quando l'algoritmo verrà processato direttamente nel SEPA. Si stima che il valore di questo overhead sia pari all'overhead del protocollo (HTTP) utilizzato per la comunicazione tra cliente e SEPA moltiplicato per il numero di richieste generate dall'algoritmo. Come mostrato in figura 16, il tester viene configurato per l'esecuzione di uno o più test tramite un file JSAP, dove vengono indicate le update, le query e tutti i parametri necessari all'esecuzione di ogni test. Di seguito una parte di un file JSAP esteso, utilizzato per il tester:

```
extended: {
    eps: "BLAZEGRAPH",
    tests: {
      UpdateDataInsert : {
        PreUpdateLink: "INSERT_DATA",
        PreUpdateRollbackLink : "DELETE_WHERE",
        UpdateLink : "INSERT_DATA",
        RollbackLink : "DELETE_WHERE",
        QueryLink: "QUERY",
        Pot : 4,
        Reiteration: 3,
        AskTestEnable: true,
        PreparationPercentage:0
      }
    }
},   ...
updates: {
    INSERT_DATA: {
        sparql:"INSERT DATA  {
            GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0> {
              ?tripleBaseInsert0
            }
            GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph1> {
              ?tripleBaseInsert1
            }
          }",
      forcedBindings: {
        tripleBaseInsert0: {
          type: "literal",
          value: "<http://www.unibo.it/Student0__X__> ub:memberOf <http://
    www.unibo.it>"
        },
        tripleBaseInsert1: {
          type: "literal",
          value: "<http://www.unibo.it/Student1__X__> ub:memberOf <http://
    www.unibo.it>"
        }
      }
    },  ...
}
```

Listing 14: Esempio JSAP esteso.

Il file JSAP in questione, è un'estensione del file JSAP utilizzato dai clienti del SEPA. In particolare è stata utilizzata la sezione "extended" del JSON nativo per aggiungere tutti i descrittori necessari ai test, come l'endpoint di riferimento "eps" e l'elenco dei test "tests".
Alla fine dell'esecuzione di tutti i test specificati nel JSAP, sarà disponibile un file JSON contenente tutte le metriche e i risultati dei test, quest'ultimo può essere consultato graficamente dal TestViewer, che fornisce vari strumenti per l'analisi dei tempi e di alcune correlazioni importanti al fine degli obiettivi perseguiti.
Il tester, LumbTest, si baserà sulle update spiegate al paragrafo 6.2.4 e caricherà una base di dati basata sul progetto LUMB. Verranno generati N grafi contenenti ciascuno circa 100.000 triple, dove N varia a seconda dei grafi necessari all'esecuzione del test. Sarà anche presente la rispettiva ontologia LUMB caricata sull'endpoint.

6.2.1 Obiettivi

L'obiettivo principale di questi test è quello di confrontare le tempistiche necessarie alla gestione delle update in una situazione di evoluzione costante dell'algoritmo interessato. I test saranno eseguiti al variare di alcune scelte implementative, al crescere del numero di triple e del numero di grafi. L'obiettivo dei test è inoltre quello di validare i vincoli di completezza e correttezza del nuovo algoritmo.
I test saranno suddivisi in fasi, alcune delle quali sono specifiche dell'algoritmo AR, come la fase di esecuzione delle Construct e la fase di esecuzione delle Ask. Parte dell'obiettivo è quella di raccogliere metriche sull'esecuzione di ciascuna di queste fasi, per poter ottenere un quadro generale delle performance e poter ricavare i tempi di overhead dell'algoritmo AR, a confronto principalmente con i tempi della richiesta di update che scaturisce l'esecuzione stessa dell'algoritmo.
Per controllare il vincolo di completezza verrà eseguita una serie di test per ogni tipologia di update. Per controllare il vincolo di correttezza, verranno convalidati i risultati del nuovo algoritmo, tramite comparazioni sulle triple risultanti dalle Ask e confrontando gli effetti, sulla base di dati, della update originale con gli effetti della InsertDeleteUpdate che viene generata dall'algoritmo AR per sostituire l'update originale.

6.2.2 Meta-Test

In questo tester, quando si parla di test, ci sono diversi livelli di astrazione. Il primo livello è il Meta-Test, che consiste in un modello di un test che possa essere eseguito una o più volte al cambiare del carico di triple. Con un Meta-Test possiamo definire una base per un test che può cambiare a run-time seguendo le regole dichiarative fornite dal file JSAP di configurazione. Ogni figlio del percorso "exstended.tests" del file JSAP rappresenta un Meta-Test.
Definiamo Single-Test ogni test che viene generato a partire da un Meta-Test, passando così ad un secondo livello di astrazione.
I Single-Test appartenenti allo stesso Meta-Test si differenziano tra loro per il numero di triple generate per il test, che aumenta seguendo la potenza di 2. Nel file JSAP, "Pot" rappresenta l'esponente massimo della potenza di 2, il quale risultato è il numero massimo di triple utilizzato dal Single-Test più impegnativo, nonché, per difetto di 1 il numero di Single-Test del Meta-Test corrente.
In questo paragrafo considereremo la porzione del file di esempio al listing 14, avendo

"Pot:4" per il Meta-Test denominato "UpdateDataInsert", verranno generati 5 Single-Test, di cui il primo con 1 sola tripla, a seguire gli altri con $2, 4, 8$ e 16 triple. Un Single-Test può essere visto come un raccoglitore delle esecuzioni del medesimo test con le stesse impostazioni.
Per poter ottenere delle buone metriche, bisogna cercare di eliminare i rumori causati dall'ambiente di esecuzione, come processi di sistema, ritardi o ottimizzazioni causati dalla JVM o congestioni di rete inusuali. Anche gli endpoint possono aggiungere rumore con possibili ottimizzazioni interne non trasparenti, che potrebbero influire sui tempi di esecuzione delle query e delle update. Una metodica molto semplice consiste nella riesecuzione del medesimo test più volte per poi poter calcolare medie e mediane, ottenendo così tempistiche più affidabili. Viene fornita per questo la possibilità di rieseguire in successione lo stesso test, una singola esecuzione prende il nome di Single-Execution, un Single-Test raccoglie n Single-Execution, dove n viene impostato nel file JSAP tramite "Reiteration", nel file di esempio ogni Single-Test conterrà 3 Single-Execution.
Una Single-Execution di un test è divisa in fasi (Phases), ogni fase è una parte del test da cui si è interessati ad ottenere delle metriche, la più importante è il tempo di esecuzione. Una fase però è in grado di raccogliere potenzialmente qualsiasi informazione e non tutte le fasi sono obbligate a raccogliere le stesse tipologie di informazioni. Ad esempio alcune di loro vengono utilizzate per recuperare indicatori di errore riguardanti il codice racchiuso da quella fase, altre invece contengono i valori di altri indicatori di correttezza. Le fasi si possono creare o spostare a proprio piacimento all'interno del codice del test, non tutte le fasi sono necessarie per l'esecuzione di un test e una fase può racchiudere altre fasi.
In alcune sono necessarie delle richieste SPARQL1.1, ad esempio la update contenente le triple generate a seconda dell'attributo "Pot". La base di queste richieste è definita nelle sezioni "update" e "query" del file JSAP. Quest'ultime verranno modificate seguendo anche le indicazioni dei "forceBindings" sempre presenti nel JSAP. Nel file di esempio è presente una sola base per una richiesta SPARQL1.1 di update, denominata "INSERT_DATA", in questo caso i "forceBindings" vengono utilizzati per segnalare la tripla di partenza e il suo posto nella update SPARQL1.1. Il tester si occuperà di generare la richiesta SPARQL1.1 adatta per ogni Single-Test, utilizzando il nativo sistema di "forceBinginds" per la sostituzione dell'intera tripla e applicando un'ulteriore sostituzione di "__X__" con un numero progressivo. Quest'ultimo serve a differenziare le triple di un Single-Test, ad esempio avendo "Pot:4" e prendendo l'ultimo Single-Test, ci ritroveremo con 16 triple sostituite sia per "?tripleBaseInsert0" che per "?tripleBaseInsert1", alle quali verrà a loro volta sostituito "__X__" con un numero che varia da 0 a 15. I "forceBinginds" sono dunque usati in modo alternativo, dato che sono stati pensati per sostituzioni di singoli termini di una tripla, ma in questo tester marcando la risorsa come "type:literal", vengono usati per la sostituzione dell'intera tripla.
Le fasi dei test di un Meta-Test possono recuperare le richieste, di cui hanno bisogno, tramite gli attributi descrittivi del Meta-Test contenente il loro nome, ad esempio: "PreUpdateLink", "UpdateLink", "QueryLink", ecc.
Principalmente un test è composto dalle seguenti fasi, riportate seguendo l'ordine di esecuzione:

- "Preparation", la preparazione della base di conoscenza dell'endpoint, fase in cui verrà eseguita una update per popolare correttamente la base di conoscenza prima del test.

- L'update in questione viene esplicitata tramite l'attributo del JSAP "PreUpdateLink".
- Questa fase è opzionale ed è possibile impostare il numero di triple coinvolte in percentuale al numero di triple trattate nel test, tramite l'attributo "PreparationPercentage". Per esempio nel caso di un Single-Test con 16 triple, se "PreparationPercentage:50", l'update di preparazione avrà 8 triple.

- "Added removed extraction and InsertDeleteUpdate generation", fase che racchiude l'intero cuore dell'algoritmo AR, in cui vengono estratte le informazioni dalla richiesta di update SPARQL1.1 sottoposta al test, vengono generate ed eseguite le relative richieste di Construct e di Ask per ottenere le quadruple aggiunte e le quadruple rimosse dalla update, concludendo con la generazione della update sostitutiva InsertDeleteUpdate.
  - L'update originaria che viene analizzata in questa fase, può essere recuperata tramite l'attributo del JSAP "UpdateLink".
  - Questa fase contiene altre due fasi, per poter raccogliere oltre al tempo di esecuzione globale anche i tempi necessari all'esecuzione delle Construct e delle Ask.
    - "Construct", fase di esecuzione delle richieste di Construct (non tutte le tipologie di update, richiedono l'esecuzione delle Construct).
    - "ASKs" esecuzione delle richieste di Ask oppure della query sostitutiva.

- "Pre-Query", l'esecuzione di una query, indicata nel JSAP da "QueryLink", per ottenere lo stato della conoscenza prima dell'esecuzione della InsertDeleteUpdate.
  - La richiesta "QueryLink" deve coinvolgere possibilmente tutte le triple modificate dalla richiesta di update. Questa richiesta ha lo scopo di controllare gli effetti della update originaria e della sua derivata InsertDeleteUpdate sulla base di conoscenza.

- "Execution InsertDeleteUpdate", l'esecuzione della InsertDeleteUpdate.

- "Execution Query after insert-delete", l'esecuzione della stessa query della fase di "Pre-Query", per poter ottenere gli effetti della InsertDeleteUpdate sulla base di conoscenza dopo l'esecuzione della InsertDeleteUpdate, confrontando il risultato di questa fase con il risultato della fase di "Pre-Query".

- "Execution RollBack Update", esecuzione di una update che sia in grado di annullare gli effetti della update originaria, riportando così la base di conoscenza come prima dell'esecuzione della InsertDeleteUpdate e dopo la fase di "Preparation".
  - La update è accessibile tramite "RollbackLink" dal file JSAP.
  - Se l'algoritmo AR funziona correttamente le due update InsertDeleteUpdate e update originaria avranno effetti identici sulla base di conoscenza, dunque "RollbackLink" dovrebbe essere in grado di annullare gli effetti sia della prima che della seconda update.

- "Execution normal update", l'esecuzione della update originaria.

- "Execution Query after normal update", l'esecuzione della stessa query delle fasi "Pre-Query" e "Execution InsertDeleteUpdate".
    - L'attributo "AskTestEnable" presente nel file JSAP, indica se va effettuato il confronto dei risultati raccolti dalle due fasi "Execution Query after normal update" e "Execution Query after insert-delete".
- "Re-Execution RollBack", l'esecuzione della stessa update della fase "Execution RollBack Update" per annullare gli effetti della update originaria.
- "RollBack preparation", l'esecuzione di una update con lo scopo di annullare gli effetti dell'eventuale update della fase "Preparation", portando così la base di conoscenza a prima dell'esecuzione dell'intero test.
    - L'update in questione viene esplicitata tramite l'attributo del JSAP "PreUpdateRollbackLink".
    - Questa fase è opzionale.

Oltre alle fasi sopra elencate, durante i test ne sono state utilizzate altre come le fasi che delimitavano l'esecuzione di tipi diversi di query in sostituzione alla classica Ask, in modo da poter ottenere e comparare sul medesimo test i tempi di esecuzione di soluzioni diverse.
Ogni Single-Execution oltre alla lista delle proprie metriche delle fasi, possiede anche altri indicatori:

- "InsertDellChackResult", indicatore booleano che rappresenta con stato positivo se la update originale e la InsertDeleteUpdate hanno avuto lo stesso effetto sulla base di conoscenza, quindi indica se le due update sono equivalenti.
- "AskCheckDone", indicatore booleano che rappresenta con stato positivo il successo di tutte le validazioni sui risultati delle triple da aggiungere e rimuovere secondo l'algoritmo AR.
- "AskDeleteCheckResult", indicatore booleano che rappresenta con stato positivo il successo della convalidazione per le triple che secondo l'algoritmo AR sono effettivamente da rimuovere.
- "AskInsertCheckResult", indicatore booleano che rappresenta con stato positivo il successo della convalidazione per le triple che secondo l'algoritmo AR sono effettivamente da aggiungere.
- "TotalTripleTestCount", il numero totale delle triple del Single-Execution.
- "PreInseredTripleCount", il numero di triple utilizzate dalla update della fase "Preparation".
- "PreQueryTripleCount", il numero di triple ottenute dalla fase "Pre-Query" eseguendo la relativa query.
- "AskInsertTripleCount", il numero di triple che secondo l'algoritmo AR sono effettivamente da inserire.
- "AskDeleteTripleCount", il numero di triple che secondo l'algoritmo AR sono effettivamente da rimuovere.

- "QueryTripleCountAfterInsertDelete", il numero di triple ottenute dalla fase "Execution Query after insert-delete" eseguendo la relativa query.
- "QueryTripleCountAfterNormalUpdate", il numero di triple ottenute dalla fase "Execution Query normal update" eseguendo la relativa query.
- "TripleExampleBeforeTest", una stringa contenente un esempio di tripla ottenuta dalla fase "Pre-Query".
- "TripleExampleAfterInsertDelete", una stringa contenente un esempio di tripla ottenuta dalla fase "Execution Query after insert-delete".
- "TripleExampleAfterNormalUpdate", una stringa contenente un esempio di tripla ottenuta dalla fase "Execution Query after normal update".
- "TripleExampleAskInsert", una stringa contenente un esempio di tripla ottenuta dalla lista delle triple che secondo l'algoritmo AR sono effettivamente da aggiungere.
- "TripleExampleAskDelete", una stringa contenente un esempio di tripla ottenuta dalla lista delle triple che secondo l'algoritmo AR sono effettivamente da rimuovere.

Tutti questi indicatori e metriche elencati, saranno disponibili nel file JSON risultante dall'esecuzione di tutti i Meta-Test.

6.2.3 TestViewer

Per poter consultare, analizzare e studiare i risultati del tester abbiamo progettato e implementato un applicativo in C#, capace di visualizzare grafici e correlazioni tra i dati. TestViewer oltre a dare la possibilità di caricare il file JSON contenete i risultati dei test, permette di caricare il relativo file JSAP, che è stato utilizzato dal tester per generare ed eseguire i test. Così facendo si avranno a disposizione tutte le informazioni sui test oltre che sui risultati. Graficamente il TestViewer è suddiviso in sei sezioni di cui la prima, "General", offre una visione generale dei risultati caricati.

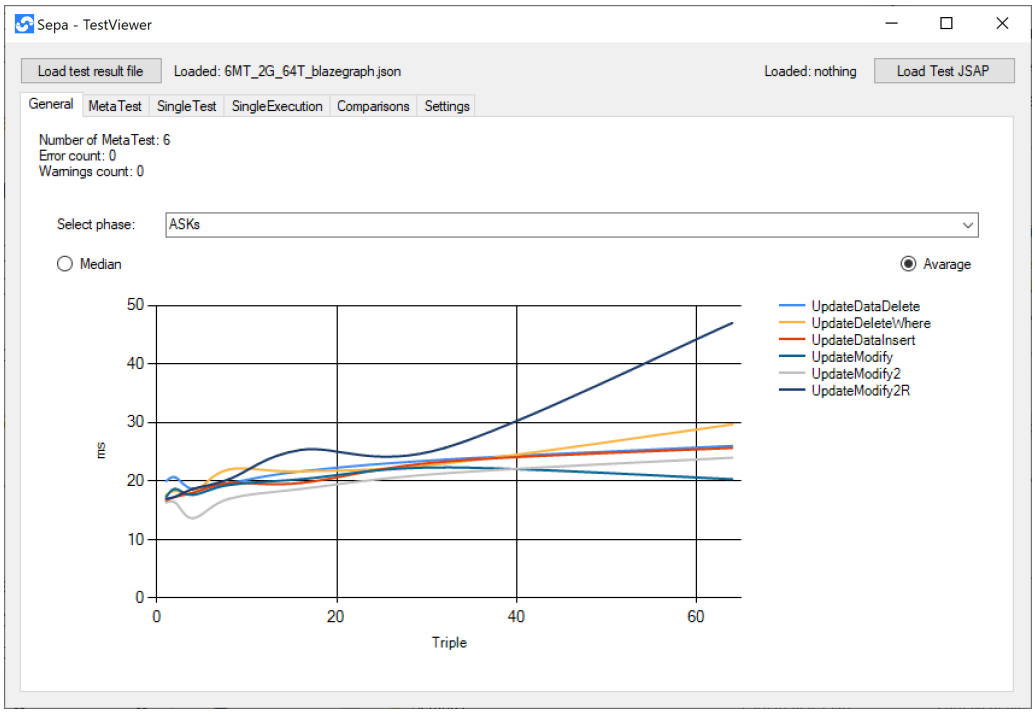

Figura 17: TestViewer General, prima sezione.

Come mostrato in figura 17, nella sezione corrente in alto a sinistra troviamo i tre contatori principali, che visualizzano il numero di Meta-Test con i relativi totali per il numero di errori e di validazioni fallite. Per errore si intende il fallimento a run-time del codice dell'algoritmo all'interno di una fase, il contatore di errori riporta il totale delle fasi che si sono imbattute in un errore di esecuzione. Per validazione fallita invece si indica il fallimento di una validazione da parte del tester rispetto ad una Single-Execution.
In primo piano viene data la possibilità di disegnare i grafici degli andamenti temporali di una fase specifica o di funzioni personalizzate che metteranno in relazione più fasi tra loro. Verranno disegnati i grafici di tutti i Meta-Test contenuti nel file caricato, differenziati per colore e indicati nella legenda riportata sulla destra del grafico. Nella figura in esempio vengono riportati i grafici delle tempistiche della fase di "ASKs", con il tempo sull'asse delle ordinate e il numero di triple del relativo Single-Test sull'asse delle ascisse. La scelta di visualizzare il grafico a partire dalla media o dalla mediana, si riferisce al trattamento dei dati di ogni Single-Test. A seconda della selezione della modalità di media o mediana,

il valore temporale della fase selezionata di ogni Single-Test sarà rispettivamente la media o la mediana dei valori di tutte le sue Single-Execution.
Dal grafico in esempio, possiamo dire che ogni campione sull'asse delle ordinate sia la media dei valori della fase "ASKs" di tutti i Single-Execution per ogni Single-Test, riferiti al proprio Meta-Test. Dato che il numero di triple segue la serie aritmetica delle potenze di due a partire da 2^0 fino ad arrivare a 2^{Pot}, i campioni reali sull'asse delle ascisse in questo caso sono: $2, 4, 8, 16, 32$ e 64.
La sezione successiva, permette una analisi più approfondita di un singolo Meta-Test.

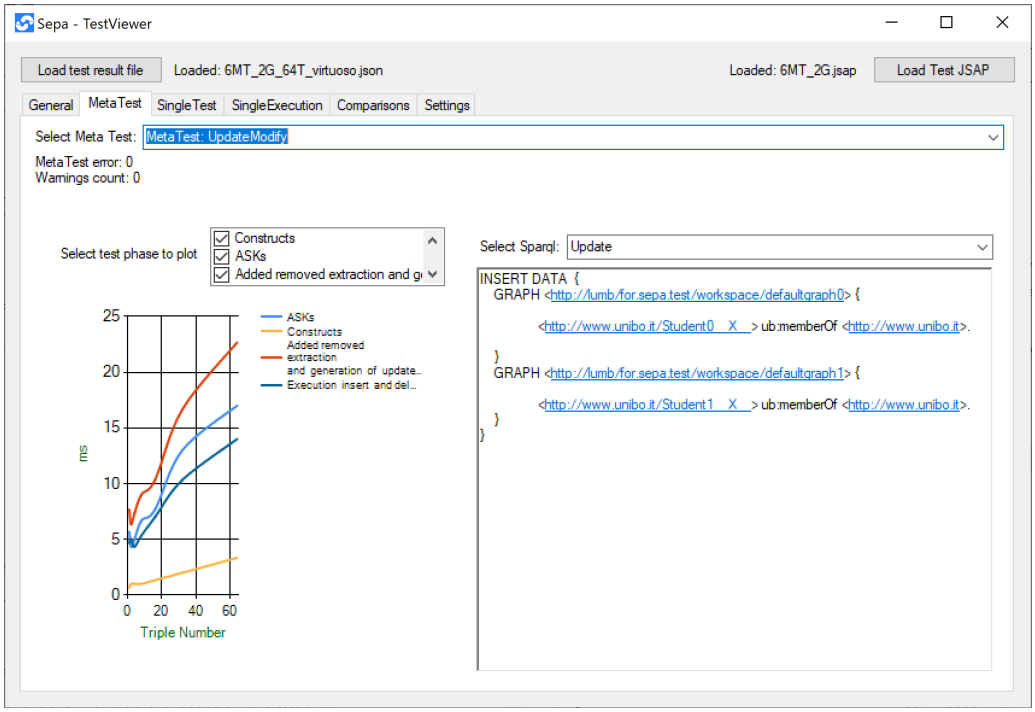

Figura 18: TestViewer Meta-Test, seconda sezione.

Come mostrato in figura 18, anche qui troviamo i contatori degli errori e delle validazioni fallite, ma riguardanti solo il Meta-Test selezionato. Il generatore di grafici di questa sezione permette di disegnare una o più fasi contemporaneamente, per studiarle singolarmente o in relazione tra loro. Nell'esempio sono selezionate quattro fasi, di cui ricordiamo che la fase di "Added removed extraction and InsertDeleteUpdate generation" racchiude anche le fasi di "ASKs" e di "Constructs". Possiamo infatti notare dal grafico che la prima sia approssimativamente la somma delle altre due.
Il grafico possiede lo stesso asse delle ordinate e lo stesso asse delle ascisse della sezione precedente, tempo di esecuzione del campione per le ordinate e numero di triple per le ascisse.
Di fianco al grafico, sulla destra, possiamo visualizzare lo SPARQL, preso dal file JSAP precedentemente caricato, di tutte le query e update relative al Meta-Test corrente.

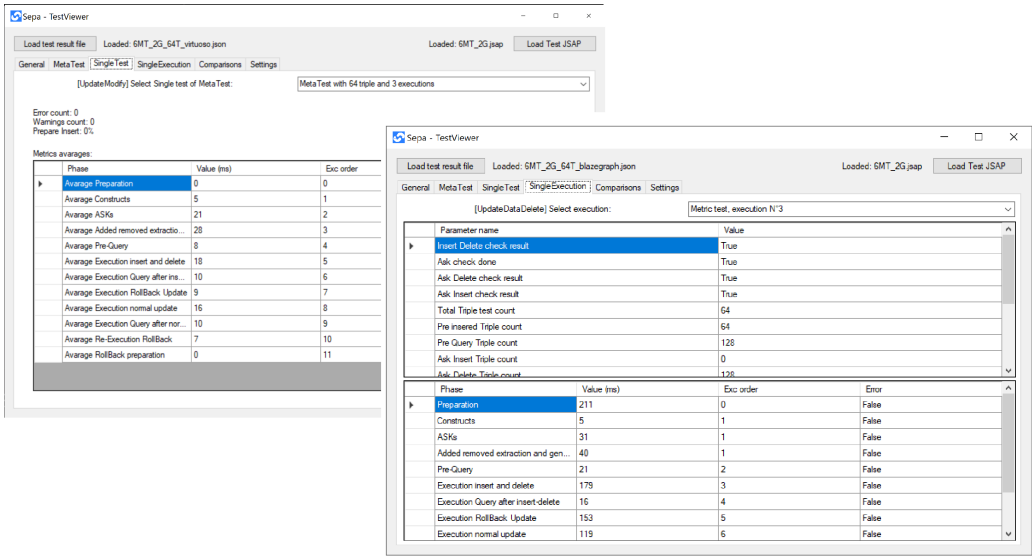

Figura 19: TestViewer Single-Test e Single-Execution, terza e quarta sezione.

La terza e la quarta sezione (figura 19) mettono a disposizione nel dettaglio i dati rispettivamente di tutti i Single-Test e tutte le Single-Execution, permettendone la selezione delle singole istanze.
Molto importanti sono gli indicatori della quarta sezione, che ci permettono di capire se l'algoritmo sta funzionando tramite i parametri di validazione, risultanti dall'apposito controllore del tester, nonché gli attributi che individuano quale fase ha incontrato errori a run-time.
La quinta sezione è la più importante, permette di effettuare comparazioni tra gli andamenti temporali delle varie fasi tenendo in considerazione tutti i Meta-Test.
Nello specifico, in figura 20 vengono mostrati gli overhead introdotti dall'algoritmo AR sui valori temporali totali. Nello specifico sono disegnati i grafici di tre andamenti temporali:

- "S1" rappresenta la somma del tempo di esecuzione delle fasi di "Construct", "ASKs" e della richiesta della nuova update, risultate dall'algoritmo AR, InsertDeleteUpdate (al posto della update originaria).
- "S2" rappresenta la somma del tempo di esecuzione delle fasi di "Construct", "ASKs" e della richiesta della update originaria.
- "S3" rappresenta il tempo di esecuzione della sola richiesta di update originaria.

In questo esempio, seguendo la figura 20, possiamo notare che l'update originaria sembra impiegare meno della InsertDeleteUpdate risultante dall'algoritmo AR, mentre ci si aspetterebbe il contrario. La richiesta InsertDeleteUpdate contiene sole triple esplicite, senza clausola *"where"*, a differenza della richiesta di update originaria che ha anche una clausola *"where"*, che quindi dovrebbe incrementare il tempo necessario allo svolgimento totale della richiesta da parte dell'endpoint.

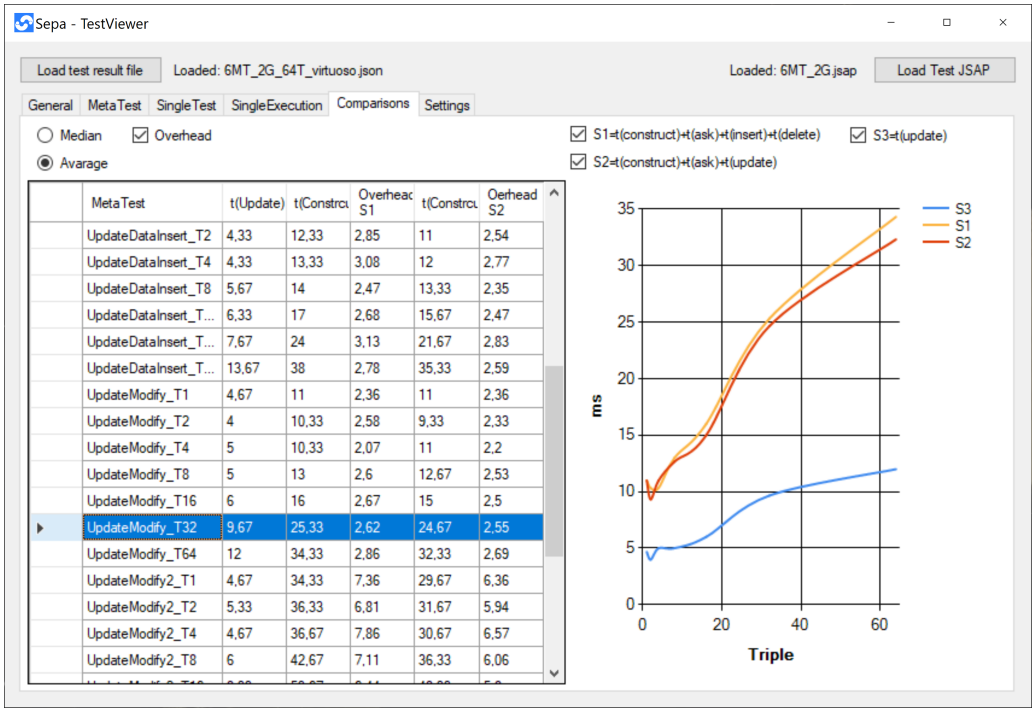

Figura 20: Single-Execution, quinta sezione.

Questa incoerenza era dovuta all'esecuzione della InsertDeleteUpdate in due richieste SPARQL1.1 distinte, nello specifico una InsertData e una DeleteData, dunque due richieste HTTP verso l'endpoint, con un corrispondente aumento dei tempi necessari rispetto alla sola richiesta originaria che necessitava del tempo di una sola richiesta HTTP.
In seguito le due richieste InsertData e DeleteData sono state inglobate in una sola richiesta di update, riducendo i tempi necessari, rendendoli inferiori alla richiesta di update originaria.
Questa penultima sezione è stata lo strumento chiave per individuare le problematiche, come quella appena descritta, ma soprattutto per stimare e poter confermare le tempistiche del nuovo algoritmo.
Infine l'ultima sezione dell'applicativo racchiude le impostazioni e le preferenze del programma.

6.2.4 SPARQL1.1 update utilizzate nei test

Per testare le tempistiche e soprattutto l'affidabilità dell'algoritmo AR tramite il tester descritto, sono state scelte delle update semplici, coprendo tutte le tipologie di update. In primo luogo si è puntato ad ottimizzare il SEPA sulla gestione di update ricorrenti e comuni, che rientrano nelle casistiche delle update selezionate per questi test o che al più avranno clausole *"where"* più complesse rispetto a quelle che abbiamo testato. La scelta semplificata di queste update è stata poi supportata dall'ipotesi di poter, in futuro, scegliere quali richieste far gestire al nuovo algoritmo e annesso filtraggio delle SPU, così da poter ottimizzare il SEPA per le richieste più comuni, che sono anche le più semplici, e gestire direttamente con l'algoritmo nativo le casistiche più complesse, che risultano anche le meno frequenti negli ambiti in cui il SEPA mira a lavorare. Tutto questo non toglie che il nuovo algoritmo possa funzionare su tutte le casistiche, anche le più complesse, semplicemente è stato scelto per i test un insieme di richieste che potesse essere il più generico possibile, ipotizzando il corretto funzionamento di tutte le update più complesse a partire da quelle testate. Nel caso in futuro si trovasse una update che non rispettasse questa ipotesi sarà sempre possibile sviluppare nuovi algoritmi di ottimizzazione specializzati nelle casistiche che potrebbero non essere gestite al meglio dal nuovo algoritmo.
Al listing 15 sono presenti degli esempi con una sola tripla di tutte le update utilizzate nel tester. Sono stati dunque testati tutti i principali costrutti quali "insert data" tramite la update UpdateDataInsert, "delete data" tramite la update UpdateDataDelete, "delete where" tramite la update UpdateDeleteWhere e infine "delete insert where" tramite le ultime tre update: UpdateModify, UpdateModify2 e UpdateModify2R.
I nomi dati alle update seguono i nomi che utilizza la libreria Jena per differenziare le tipologie di update.
Le ultime tre update di test, appartengono alla stessa tipologia di UpdateModify, con una piccola differenza che può però portare sia a problemi nella creazione della query di Construct sia a un incremento notevole dei tempi richiesti per la gestione della update. La prima delle tre, la UpdateModify ha semplicemente la clausola *"where"* vuota e quindi triple esplicite a differenza di UpdateModify2 che per la clausola di "delete", non ha triple esplicite e ha una rispettiva clausola *"where"*. La differenza tra UpdateModify2 e UpdateModify2R è l'ordine all'interno del modello della clausola *"where"*, non vuota in entrambi i casi. Quest'ultima differenza d'ordine nella clausola where, su alcuni endpoint, provoca enormi differenze sul tempo di esecuzione della update come esposto al paragrafo 6.1.4.
Ci sono anche tipologie di update che non sono state considerate, dato che i loro effetti sulla base di conoscenza sono focalizzati su interi grafi e non su singole su triple o quadruple, quindi non adatte con l'approccio logico del nuovo algoritmo. L'algoritmo AR lavora molto bene con triple o quadruple, ma abbiamo ipotizzato che sarebbe controproducente fargli gestire update di operazioni che manipolano grafi per intero. Le Update in questione, sempre seguendo la nomenclatura della libreria Jena, sono: la UpdateCreate, la UpdateLoad, la UpdateClear, la UpdateDrop, la UpdateCopy, la UpdateAdd e la UpdateMove.

```
/*--------------------UpdateDataInsert*/
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
INSERT DATA  {
    GRAPH <http://lumb/for.sepa.test/workspace/defaultgraphX> {
      <http://www.unibo.it/StudentY> ub:memberOf <http://www.unibo.it>
    }}
/*--------------------UpdateDataDelete*/
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
DELETE DATA {
  GRAPH  <http://lumb/for.sepa.test/workspace/defaultgraphX> {
    <http://www.unibo.it/StudentY> ub:memberOf <http://www.unibo.it>
  }}
/*--------------------UpdateDeleteWhere*/
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
DELETE WHERE {
  GRAPH <http://lumb/for.sepa.test/workspace/defaultgraphX> {
    ?s ub:memberOf <http://www.unibo.it>
  }}
/*--------------------UpdateModify*/
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
DELETE {
  GRAPH <http://lumb/for.sepa.test/workspace/defaultgraphX>  {
    <http://www.unibo.it/StudentY> ub:memberOf <http://www.unibo.it>
  }
} INSERT {
  GRAPH <http://lumb/for.sepa.test/workspace/defaultgraphX> {
    <http://www.unibo.it/GraduatedStudentY> ub:memberOf <http://www.unibo.
    it>
  }
} WHERE {}
/*--------------------UpdateModify2*/
DELETE {
    GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0>  {
      ?s ?p ?o
    }
} INSERT {
    GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0> {
      <http://www.unibo.it/GraduatedStudentY> ub:memberOf <http://www.unibo
    .it>
    }
} WHERE {
    ?s ?p ?o .
    ?s ub:memberOf <http://www.unibo.it>
}
/*--------------------UpdateModify2R*/
DELETE {
    GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0>  {
      ?s ?p ?o
    }
} INSERT {
    GRAPH <http://lumb/for.sepa.test/workspace/defaultgraph0> {
      <http://www.unibo.it/GraduatedStudentY> ub:memberOf <http://www.unibo
    .it>
    }
} WHERE {
    ?s ub:memberOf <http://www.unibo.it> .
    ?s ?p ?o
}
```

Listing 15: Update SPARQL1.1 usate per i Meta-Test (con una sola tripla).

6.2.5 Risultati

Dopo aver implementato e collaudato l'algoritmo AR, si è passati alla fase di test specifica per l'ottimizzazione. L'ostacolo più grande risultava essere l'esecuzione della fase di "ASKs", che originariamente prevedeva la richiesta di n query di tipologia ask, dove n è il numero di tutte le triple che potenzialmente dovrebbero essere aggiunte o rimosse. Le problematiche e le soluzioni trovate sono spiegate in dettaglio al capitolo 6.1.2.
In secondo luogo sono stati svolti dei test di stress, sempre basati sulle update precedentemente descritte. Si era interessati al rapporto tra tempo di esecuzione della richiesta di update originaria e tempo necessario all'esecuzione dell'algoritmo AR e all'esecuzione della richiesta InsertDeleteUpdate generata in sostituzione della update originaria.
Questo test è stato pensato su singolo grafo RDF e con un numero considerevole di triple. L'impostazione di "Pot" dei JSAP di tutti i Meta-Test era pari a 8, con dunque nove Single-Test con rispettivamente $1, 2, 4, 8, 16, 32, 64, 128$ e 512 triple. Nella figura 21 sono presenti i grafici disegnati tramite la prima sezione del TestViewer a partire dai risultati del test, in cui sulla prima colonna ci sono i sei Meta-Test eseguiti con Virtuoso come endpoint e sulla seconda colonna gli stessi Meta-Test eseguiti con Blazegraph come endpoint. Ricordiamo che il tester fa riferimento sempre al SEPA che a sua volta si interfaccia con l'endpoint.
Ogni grafico traccia due andamenti temporali, in blu il tempo di esecuzione della update originaria del Meta-Test al variare delle triple e in arancione la somma dei tempi di esecuzione delle fasi di Construct, ASKs e della richiesta di InsertDeleteUpdate. Questo consente di ottenere l'ammontare di tempo aggiunto dall'algoritmo AR rispetto alla singola update originaria.
Considerando che generalmente l'algoritmo nativo, che gestisce le richieste di update per poi propagare le informazioni necessarie ai suoi sottoscritti, può impiegare tempi dell'ordine dei secondi o addirittura di ordini superiori, fatto noto alla luce dei risultati dei benchmark ottenuti con il ChatTest, gli overhead dell'algoritmo AR, che risultano al più dell'ordine di centinaia di millisecondi, sono accettabili.
Abbiamo appena definito accettabili le tempistiche dell'algoritmo AR sulla base del confronto tra i risultati di due tester e scenari differenti, questo perchè in questa fase di test e sviluppo stiamo considerando solo l'esecuzione dell'algoritmo AR senza i suoi benefici, ottenibili filtrando le SPU, è dunque impossibile effettuare i benchmark del ChatTest esclusivamente sull'algoritmo AR. In questa fase non siamo interessati a quantificare con esattezza la qualità dell'ottimizzazione finale, stiamo svolgendo in primo luogo uno studio di fattibilità, per escludere il caso in cui l'algoritmo AR sia del tutto controproducente.
Definiamo i risultati ottenuti come ottimali, dato che ipotizziamo a partire da questi dati, che con l'aggiunta di poche decine di millisecondi da parte dell'algoritmo AR, verranno risparmiati secondi nella gestione delle sottoscrizioni. Questo perché l'algoritmo nativo impiega una grande quantità di tempo nell'ottenere le informazioni di triple aggiunte e rimosse da dover propagare ai suoi sottoscritti. Di norma solo una piccola parte di loro sarà interessata ai cambiamenti dovuti ad una update, attualmente sono molte le SPU che vengono attivate inutilmente. Il tempo necessario all'esecuzione dell'algoritmo AR dovrebbe essere ampiamente ricompensato dal grande numero di SPU che non verranno attivate, facendo risparmiare tempo al SEPA.

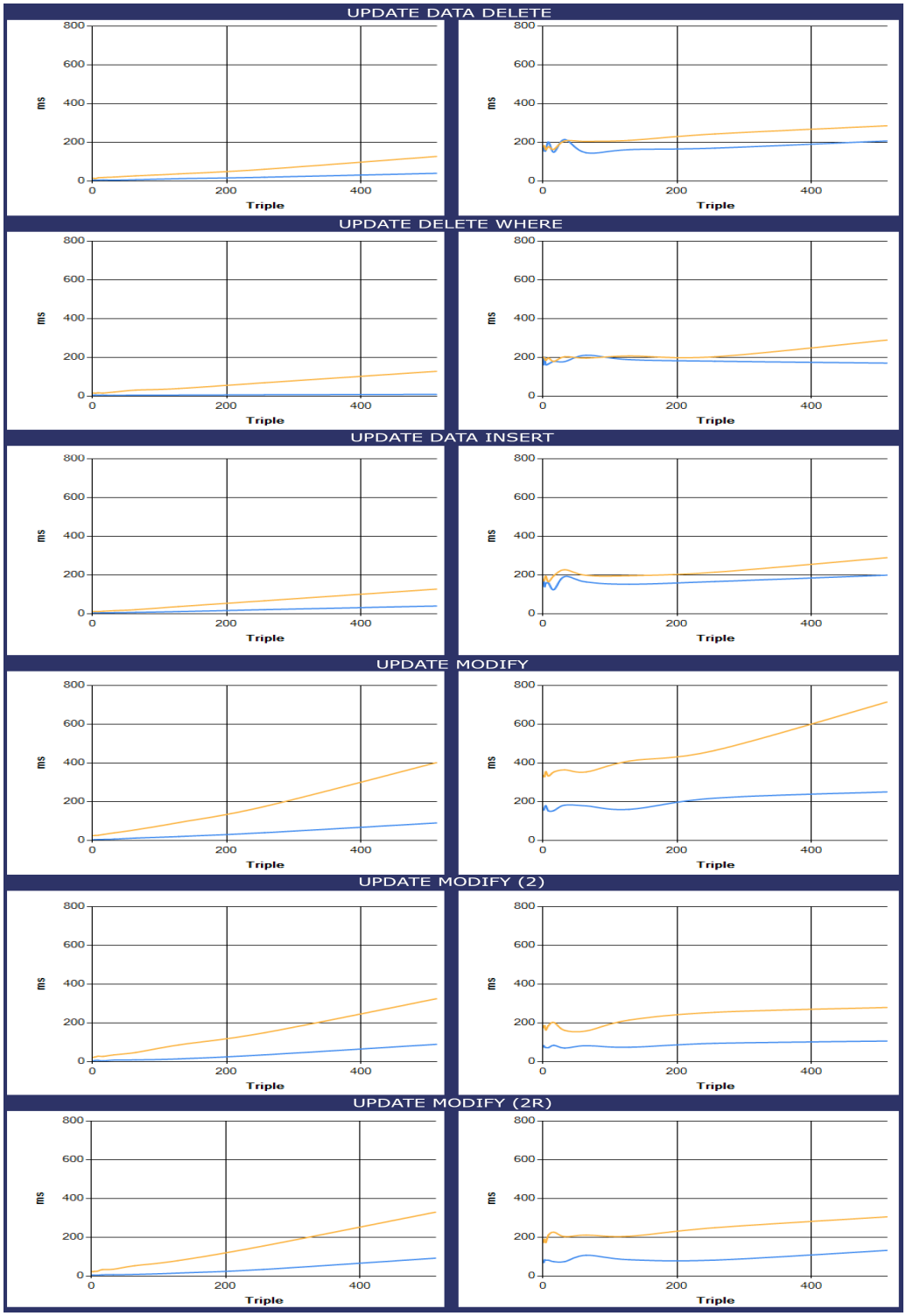

Figura 21: Risultati dei test su singolo grafo. In arancione il tempo dell'algoritmo AR con InsertDeleteUpdate e in blu il tempo della update originaria.

Qualche considerazione sugli endpoint alla luce dei risultati ottenuti: generalmente risulta che Virtuoso sia più efficiente rispetto a Blazegraph e possiamo notare una maggiore fluidità sempre da parte di Virtuoso. I grafici costruiti sui dati raccolti con l'utilizzo di Blazegraph come endpoint, spesso contengono più anomalie e andamenti temporali meno stabili.

Dopo aver testato l'operato dell'algoritmo su singolo grafo, sono state modificate le update degli stessi sei Meta-Test, in modo tale da lasciare la logica delle richieste invariata, ma passando dall'utilizzo di un singolo grafo all'utilizzo di più grafi.
Le update che coinvolgono più grafi sono una delle fonti di rallentamenti per il SEPA, in particolar modo per l'algoritmo nativo, quindi si è cercato di raccogliere stime temporali al riguardo anche per l'algoritmo AR.
In figura 23 sono presenti i grafici risultanti dai test effettuati su più grafi. A differenza dei grafici precedenti (su singolo grafo) in ognuno di questi grafici per ogni Meta-Test è disegnata solo la somma temporale delle fasi di Construct, ASKs e della richiesta di InsertDeleteUpdate.
Una nota importante sui test su più grafi che seguiranno, riguarda il legame tra il numero di grafi e l'attributo "Pot" dei Meta-Test. In questi test "Pot" è stato impostato a 6, ottenendo così sei campioni di cui l'ultimo dovrebbe avere $2^{Pot} = 2^6 = 64$ triple. Il tester però genererà 2^{Pot} triple per ogni grafo, aggiungendo così un distacco tra i dati dei test effettuati con un numero diverso di grafi, pur considerando lo stesso campionamento. Ai fini della valutazione dell'algoritmo AR questo distacco non è importante e si può comunque compensare, comparando campionamenti diversi, basandosi sul numero totale delle triple del campione. Per esempio prendendo il campione a $Pot = 6$ per i test su due grafi, le triple utilizzate dal Single-Test sarebbero 64 per grafo, con un totale di 128 triple. Per poter fare un confronto con i risultati dei test su singolo grafo, considerando lo stesso numero totale di triple utilizzate, sarà sufficiente far riferimento al Single-Test a $Pot = 7$ su singolo grafo, che possiede un totale di 128 triple, su singolo grafo.
Per quanto riguarda l'analisi mirata all'algoritmo AR, in presenza di due grafi, il tempo necessario non si distacca molto dai test effettuati su singolo grafo, in quanto prendendo il campione a $Pot = 7$ per i test effettuati su singolo grafo e a $Pot = 6$ per quelli su doppio grafo, il tempo totale richiesto per eseguire l'estrazione delle quadruple e l'aggiornamento dell'endpoint, risultate dalla sommatoria dei tempi per eseguire la fase di Construct, di ASKs e la InsertDeleteUpdate, varia di pochi millisecondi. Estrapoliamo l'intervallo di questa sommatoria di tempi di esecuzione, per ogni endpoint considerato, riportandolo in forma tabellare:

Endpoint	Numero grafi	Pot	Triple per grafo	Triple totali	Intervallo tempi di esecuzione AR
Virtuoso	1	7	128	128	[37, 92]ms
Virtuoso	2	6	64	128	[35-114]ms
Blazegraph	1	7	128	128	[196, 409]ms
Blazegraph	2	6	64	128	[148, 450]ms

L'aumento di decine di millisecondi su centinaia dei millisecondi totali, dovuto al passaggio da richieste su singolo grafo a richieste su due grafi distinti è considerato come un aumento irrisorio comparato alla nota difficoltà generale nel trattare molteplici grafi.
I dati raccolti e i grafici estrapolati dai successivi test su quattro grafi confermano il fatto che generalmente Blazegraph sia più resiliente al cambiamento delle tempistiche al variare del numero di grafi rispetto a Virtuoso, ma che sia anche di base più lento. Questi intervalli confermano che l'algoritmo continua a lavorare con ottime tempistiche anche in presenza di più grafi, trascurando le divergenze causate dagli endpoint, ad esempio il fatto che con Virtuoso i tempi siano generalmente minori.
Con i precedenti test, con un solo grafo, con due e con quattro grafi, abbiamo confermato la validità dell'algoritmo AR. Un altro punto d'interesse è stato quello di ottenere un limite superiore all'esecuzione dell'algoritmo in tempi utili. I grafici dei test con otto grafi, sono stati effettuati appositamente per evidenziare i limiti dell'algoritmo, infatti l'ultimo campione di questi test risulta lavorare con otto grafi ognuno dei quali con 64 triple, per un totale di 512 triple.
Questi ultimi test hanno evidenziato che l'algoritmo è sensibile al numero di grafi specialmente per le update di tipo UpdateModify con clausola *"where"* (nel caso specifico sia per il Meta-Test "UpdateModify2" sia per "UpdateModify2R") e UpdateDeleteWhere. Queste update hanno in comune la presenza della clausola where, che come ci si poteva aspettare tende ad essere dispendiosa, soprattutto in presenza di più grafi. Questo aspetto, si poteva già notare anche negli altri test, in cui queste update risultano sempre un pò più lente rispetto alle altre.
Analizzando le singole fasi di questi Meta-Test è evidente, dai dati, che la fase che incide su questi innalzamenti delle tempistiche sia la fase "ASKs", che ricordiamo eseguire una query in sostituzione all'esecuzione delle richieste di ask vere e proprie. Prendendo in considerazione solo l'ultimo Single-Test del Meta-Test sulla update UpdateDeleteWhere (raffigurata in giallo a figura 23), per il test su otto grafi con Virtuoso come endpoit, i valori dell'ultimo campione (con 64 triple per grafo) risultano:

$$t(Construct)+t(ASKs)+t(InsertDeleteUpdate) = (114+1852+52)ms = 2018ms \quad (5)$$

L'esecuzione della richiesta della query che sostituisce le ask (fase di ASKs) occupa la maggioranza del tempo totale. Mostriamo in figura 22 gli andamenti dei tempi del Meta-Test UpdateDeleteWhere su otto grafi.

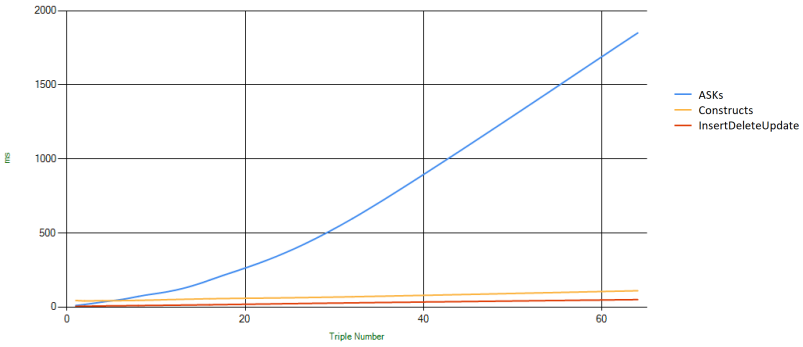

Figura 22: Risultati del test sulla UpdateDeleteWhere con 8 grafi.

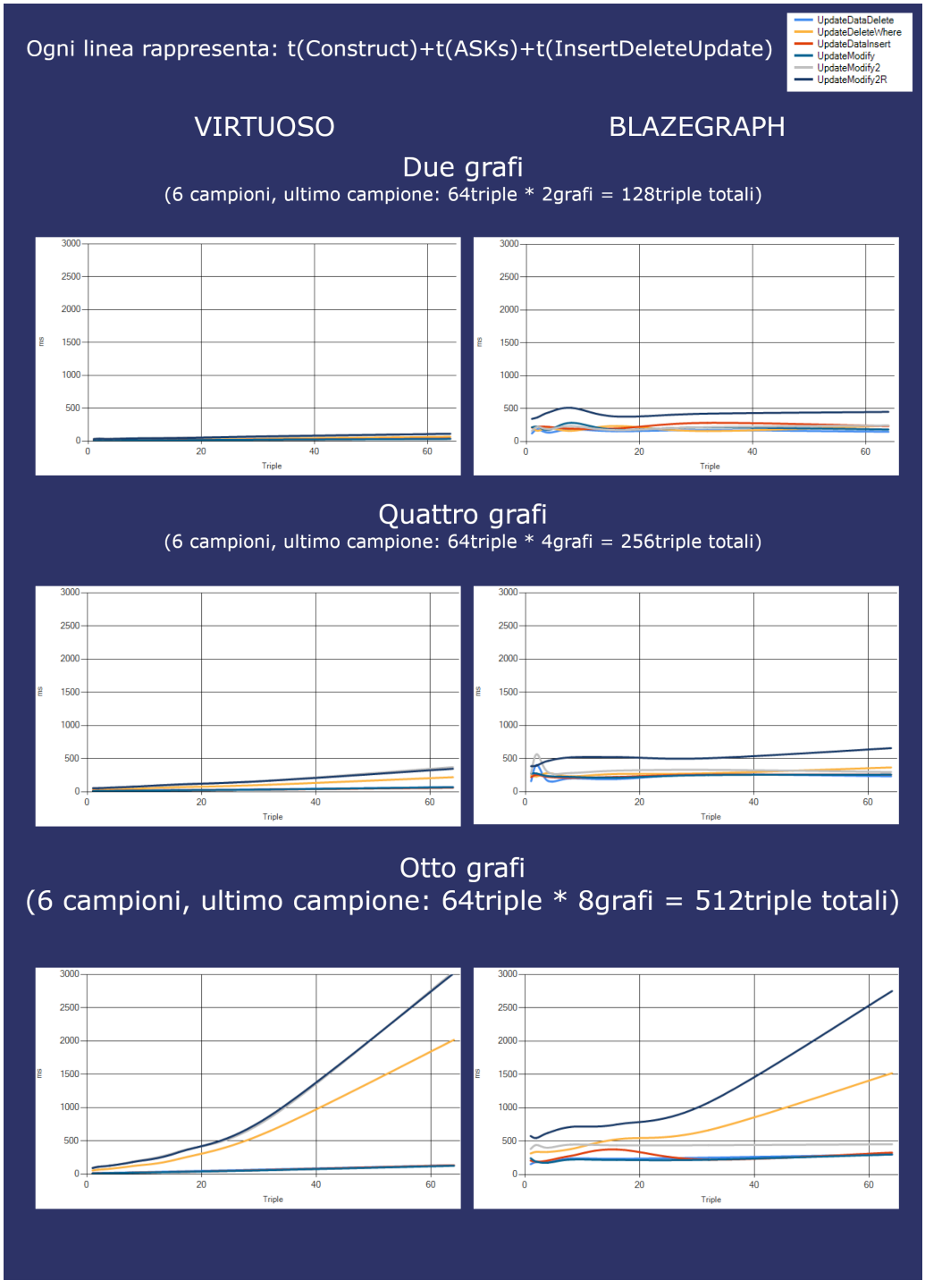

Figura 23: Risultati dei test su molteplici grafi.

Alla luce di tutto ciò, il fattore scatenante secondo teoria è la presenza della clausola where, ma non è la sua risoluzione a rallentare il processo, dato che la fase di Construct che risolve l'eventuale clausola *"where"* risulta impiegare poco tempo rispetto alla fase di "ASKs", che secondo le metriche invece è la responsabile dell'appesantimento dell'intero processo.
L'ipotesi è che sia l'elevato numero di quadruple uscenti dalla fase di Construct ad essere la causa del rallentamento sulle update con clausola *"where"* in questi ultimi test. Per confermare questa ipotesi è bastato osservare il numero di quadruple controllate dalla fase di "ASKs". Prendendo il caso specifico del Meta-Test della update UpdateModify2 risultano ben 4608 quadruple come possibili candidate dopo la fase di Construct, un numero molto alto dato che le quadruple che poi verranno effettivamente utilizzate saranno meno di 1/4 di quelle controllate. Ogni grafo in questo caso aggiunge 64 triple e ne rimuove altrettante, quindi il totale di triple realmente interessate è $8 * 64 * 2 = 1024$, di cui 512 sono le triple esplicite generate per la clausola di insert della UpdateModify2.
Oltre all'utilità generale dei dati forniti dall'algoritmo AR, uno degli scopi che si cerca di raggiungere è l'ottimizzazione della gestione delle sottoscrizioni. Principalmente l'obiettivo è quello di non attivare le SPU che non sono interessate all'aggiornamento della base di dati dopo una update e che quindi eseguirebbero a vuoto la loro query di sottoscrizione. Tramite i dati forniti dall'algoritmo AR si può realizzare un filtro più selettivo, per l'attivazione delle SPU, da accodare al filtro basato sui grafi. Il filtro in questione prende il nome di Look Up Triples Table (LUTT) [18].
Possiamo fornire una stima del tempo che si risparmierà con l'ulteriore filtro delle SPU. Facendo riferimento alla formula (3) all'inizio del capitolo, l'algoritmo AR aggiunge un overhead all'esecuzione della update $t(u)$ e permette di filtrare le SPU diminuendo il numero di sottoscrizioni da gestire N. Semplificando, poniamo il tempo di gestione di una sottoscrizione alla sola esecuzione della relativa query $t(Q)$. Definiamo Sf come la media del numero minimo di SPU che non devono essere attivate, quindi filtrate dal filtro basato sui risultati dell'algoritmo AR, per poter compensare il tempo medio di $t(AR)$. Considerando tutti i test effettuati sull'algoritmo AR che come configurazione avevano 16 triple, lavorando su 1 e 2 grafi, riportiamo di seguito i dati relativi indicando con $t(AR)$ il tempo di esecuzione dell'algoritmo AR e con $t(Q)$ la media dei tempi di esecuzione delle query effettuate nel relativo test.
Risulta che il tempo necessario per l'esecuzione dell'algoritmo AR venga recuperato con mediamente $3 - 4$ SPU non attivate. Pertanto l'ottimizzazione conferirà un ottimo guadagno di tempo nel gestire le sottoscrizioni, dato che il tempo risparmiato nel filtrare le SPU non necessarie è superiore a quello impiegato dall'algoritmo AR dopo la soglia delle 3 - 4 Sf.

Ednpoint	MetaTest	Numero grafi	Triple	t(AR) [ms]	t(Q) [ms]	t(AR)/t(Q)
Virtuoso	UpdateDataDelete	1	16	14	5	2,80
Virtuoso	UpdateDeleteWhere	1	16	13	6	2,17
Virtuoso	UpdateDataInsert	1	16	9	3	3,00
Virtuoso	UpdateModify	1	16	22	4	5,50
Virtuoso	UpdateModify2	1	16	18	4	4,50
Virtuoso	UpdateModify2_R	1	16	23	4	5,75
Blazegraph	UpdateDataDelete	1	16	25	15	1,67
Blazegraph	UpdateDeleteWhere	1	16	37	15	2,47
Blazegraph	UpdateDataInsert	1	16	22	14	1,57
Blazegraph	UpdateModify	1	16	57	15	3,80
Blazegraph	UpdateModify2	1	16	51	14	3,64
Blazegraph	UpdateModify2_R	1	16	39	14	2,79
Virtuoso	UpdateDataDelete	2	16	15	5	3,00
Virtuoso	UpdateDeleteWhere	2	16	26	5	5,20
Virtuoso	UpdateDataInsert	2	16	11	4	2,75
Virtuoso	UpdateModify	2	16	10	5	2,00
Virtuoso	UpdateModify2	2	16	41	5	8,20
Virtuoso	UpdateModify2_R	2	16	37	5	7,40
Blazegraph	UpdateDataDelete	2	16	28	16	1,75
Blazegraph	UpdateDeleteWhere	2	16	63	15	4,20
Blazegraph	UpdateDataInsert	2	16	24	15	1,60
Blazegraph	UpdateModify	2	16	23	15	1,53
Blazegraph	UpdateModify2	2	16	65	16	4,06
Blazegraph	UpdateModify2_R	2	16	89	16	5,56
			Media	31,75	9,79	Sf = 3 - 4

7 Conclusione

Con le attività svolte durante il periodo di tesi sono state convalidate le performance del SEPA e le sue potenzialità ancora parzialmente inespresse. Il meccanismo fornito per le sottoscrizioni basate su query SPARQL1.1 si è dimostrato uno strumento ad ampio spettro, questo suo aspetto, offrendo la più totale generalità d'utilizzo si è pertanto rivelato poco performante in alcuni scenari. Le ottimizzazioni studiate e implementate si pongono il problema di snellire l'algoritmo attuale che gestisce le sottoscrizioni, in modo che il SEPA sia generalmente più reattivo.
Il primo tester, il benchmark ChatTest implementato a partire dal progetto SepaChat, è stato messo all'opera per raccogliere dati, riguardo all'utilizzo di più grafi nella logica delle richieste SPARQL1.1 e alle tempistiche del SEPA con e senza il filtro basato sui grafi per le sottoscrizioni. Con la conclusione che l'utilizzo del filtro porta a una grande diminuzione dei tempi necessari per la gestione delle richieste in presenza di più grafi coinvolti.
Al seguito di questa prima fase si è passati allo sviluppo dell'algoritmo AR, che è stato trattato in contemporanea allo sviluppo del tester basato sull'ontologia LUMB. Per poter analizzare al meglio l'algoritmo AR si è scelto di eseguirlo esternamente al SEPA, pertanto l'algoritmo è stato implementato direttamente all'interno del tester.
Si è realizzato l'applicativo TestViewer finalizzato a studiate nel dettaglio tutte le metriche dei test inerenti l'esecuzione delle fasi dell'algoritmo AR, che si è dimostrato efficace nel recuperare le quadruple aggiunte e rimosse con lassi di tempo accettabili. L'algoritmo AR è stato infine implementato all'interno del SEPA e correlato dei rispettivi test Junit per la sua convalida.
In conclusione, i dati raccolti e le analisi effettuate hanno permesso di validare l'algoritmo e stabilire quali parti del processo di estrazione delle quadruple aggiunte e rimosse siano più onerose, in relazione al tipo di update. La soglia di Subscription Processing Unit (SPU) filtrate, dopo la quale il tempo necessario per l'esecuzione dell'algoritmo AR viene compensato, risulta ottimale dato che il tempo non impiegato a gestire le SPU filtrate diventa risparmiato a partire dalla quarta SPU filtrata.
Gli sviluppi futuri dovrebbero consistere nel testare l'algoritmo AR con ulteriori update, in particolare utilizzando clausole come *"from"* e *"from named"*, inoltre si dovrebbero implementare le LUTT per poter filtrare le SPU confrontando le quadruple aggiunte e rimosse fornite dall'algoritmo AR con le LUTT stesse.

Riferimenti bibliografici

[1] *Apache Jena*. URL: https://jena.apache.org/.

[2] Brad Bebee. *Blazegraph*. 2020. URL: https://github.com/blazegraph/database/wiki.

[3] *CoAP*. URL: https://coap.technology/.

[4] Richard Cyganiak, David Wood e Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014*. 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

[5] *Dashboard*. URL: https://github.com/arces-wot/SEPA-Dashboard.

[6] *Docker*. URL: https://docs.docker.com/engine/docker-overview/.

[7] Lee Feigenbaum et al. *SPARQL 1.1 Protocol, W3C Recommendation 21 March 2013*. 2013. URL: http://www.w3.org/TR/sparql11-protocol/.

[8] Paula Gearon, Alexandre Passant e Axel Polleres. *SPARQL 1.1 Update, W3C Recommendation 21 March 2013*. 2013. URL: http://www.w3.org/TR/sparql11-update/.

[9] Steve Harris e Andy Seaborne. *SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013*. 2013. URL: http://www.w3.org/TR/sparql11-query/.

[10] Jeff Heflin. *LUMB*. URL: http://swat.cse.lehigh.edu/projects/lubm/.

[11] Gregory Todd Williams; Rensselaer Polytechnic Institute. *SPARQL 1.1 Service Description*. 2013. URL: https://www.w3.org/TR/sparql11-service-description/.

[12] Jakob Jenkov. *JHM*. 2015. URL: http://tutorials.jenkov.com/java-performance/jmh.html.

[13] *JSAP*. URL: http://mml.arces.unibo.it/TR/jsap.html.

[14] *Linked Data Notifications*. URL: https://www.w3.org/TR/ldn/.

[15] *LUMB default generation*. URL: http://swat.cse.lehigh.edu/projects/lubm/profile.htm.

[16] *LUMB owl*. URL: http://swat.cse.lehigh.edu/onto/univ-bench.owl.

[17] *MQTT*. URL: https://mqtt.org/.

[18] Luca Roffia et al. "A Semantic Publish-Subscribe Architecture for the Internet of Things". In: *IEEE Internet of Things Journal* (dic. 2016). DOI: 10.1109/JIOT.2016.2587380.

[19] Luca Roffia et al. "Dynamic Linked Data: A SPARQL Event Processing Architecture". In: *Future Internet* 10.4 (2018), p. 36.

[20] *SEPA*. URL: http://mml.arces.unibo.it/TR/sepa.html.

[21] *Sepa chat*. URL: https://github.com/arces-wot/SEPA-Chat.

[22] OpenLink Software. *Virtuoso Universal Server*. 2019. URL: https://virtuoso.openlinksw.com/.

[23] *SPARQL 1.1 Secure Event*. URL: http://mml.arces.unibo.it/TR/sparql11-se-protocol.html.

[24] *SPARQL 1.1 Subscribe Language*. URL: http://mml.arces.unibo.it/TR/sparql11-subscribe.html.