SPECIAL

Scalable Policy-awarE Linked Data arChitecture for prIvacy, trAnsparency and compLiance

Deliverable No 3.4

Transparency & Compliance Release

Document version: 1.0

SPECIAL DELIVERABLE

Name, title and organisation of the scientific representative of the project's coordinator:

Ms Jessica Michel +33 4 92 38 50 89 jessica.michel@ercim.eu

GEIE ERCIM, 2004, route des Lucioles, Sophia Antipolis, 06410 Biot, France

Project website address: http://www.specialprivacy.eu/

Project	
Grant Agreement number	731601
Project acronym:	SPECIAL
Project title:	Scalable Policy-awarE Linked Data arChitecture for prIvacy, trAnsparency and compLiance
Funding Scheme:	Research & Innovation Action (RIA)
Date of latest version of DoW against which the assessment will be made:	17/10/2016
Document	
Period covered:	M18-M25
Deliverable number:	D3.4
Deliverable title	Transparency & Compliance Release
Contractual Date of Delivery:	31-01-2019
Actual Date of Delivery:	31-01-2019
Editor (s):	Sabrina Kirrane (WU), Javier Fernandez (WU), Rigo Wenning (ERCIM), Rudy Jacob (PROXIMUS), Piero Bonatti (CeRICT)
Author (s):	Wouter Dullaert, Uros Milosevic, Jonathan Langens, Arnaud S'Jongers, Nora Szepes, Vincent Goossens, Nathaniel Rudavsky-Brody, Ward Delabastita (TF), Sabrina Kirrane, Javier Fernandez (WU)
Reviewer (s):	Sabrina Kirrane, Javier Fernandez (WU), Rigo Wenning (ERCIM), Rudy Jacob (PROXIMUS), Piero Bonatti (CeRICT)
Contributor (s):	Miguel A. Martínez-Prieto, Antonio Hernández-Illera (University of Valladolid), Claudio Gutiérrez (University of Chile), Jürgen Umbrich, Magnus Knuth (AKSW/KILT, Leipzig University), Axel Polleres (WU), Simon Steyskal (WU)
Participant(s):	ERCIM, WU, CeRICT, TF, PROX
Work package no.:	3
Work package title:	Big Data Policy Engine
Work package leader:	TF
Distribution:	PU
Version/Revision:	1.0 FINAL
Total number of pages (including cover):	84

Disclaimer

This document contains description of the SPECIAL project work and findings.

The authors of this document have taken any available measure in order for its content to be accurate, consistent and lawful. However, neither the project consortium as a whole nor the individual partners that implicitly or explicitly participated in the creation and publication of this document hold any responsibility for actions that might occur as a result of using its content.

This publication has been produced with the assistance of the European Union. The content of this publication is the sole responsibility of the SPECIAL consortium and can in no way be taken to reflect the views of the European Union.

The European Union is established in accordance with the Treaty on European Union (Maastricht). There are currently 28 Member States of the Union. It is based on the European Communities and the Member States cooperation in the fields of Common Foreign and Security Policy and Justice and Home Affairs. The five main institutions of the European Union are the European Parliament, the Council of Ministers, the European Commission, the Court of Justice and the Court of Auditors (http://europa.eu/).

SPECIAL has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731601.

Contents

List of Figures

Chapter 1

Summary

The goal of this report[1] is to describe the third release of the SPECIAL platform. It builds upon the research done in WP2 by providing working implementations of many of the ideas presented in deliverables D2.5 Policy Language V2, D2.7 Transparency Framework V2 and D2.8 Transparency and Compliance Algorithms V2. It also offers an update over the previous release by reflecting on:

- *ex ante* compliance checking,
- consent backend changes,
- the personal data inventory,
- compression and encryption, and
- overall performance improvements (to be demonstrated in D3.5).

It is worth noting that, even though the work in WP2 has been finalized, not all choices are final, and some challenges will be tackled in D3.6 Final Release.

The first chapter presents the platform architecture as a whole. This will give the reader an overview of the various supported features, how the individual components interact and detailed information on some cross cutting concerns.

In subsequent chapters specific components of the architecture are discussed in more detail. SPECIAL focus is placed on documenting design decisions which might not be obvious from the source code.

At the time of publishing, the source code is available on GitHub[2], while a working version of the platform is hosted by TenForce[3].

[1] D3.4 Transparency & Compliance Release is a DE (demonstrator) type deliverable.
[2] https://github.com/specialprivacy
[3] http://projects.tenforce.com/special/demo

Chapter 2

Architecture Overview

This chapter documents the overall architecture of the SPECIAL platform as it is currently envisioned. It documents the guiding design principles, and focus on cross cutting concerns and how data flows between the various components.

Depending on the intended use case, we distinguish between two conceptually different, yet implementation-wise similar architecture setups: *ex post* and *ex ante* policy compliance checking.

Ex post compliance checking. A high level overview of the architecture is shown in Figure 2.1. As they process personal data, applications write the processing events to a processing log, which is then inspected for compliance.

Ex ante compliance checking. A similar overview of the SPECIAL-K architecture setup is given in Figure 2.2. Applications submit their requests for processing of personal data, which are then inspected for compliance. The answers are then fed back to the requesting applications via the Personal Data Gateway.

In its current state, regardless of the setup, five main components can be identified in the SPECIAL-K architecture:

1. Existing line of business applications
2. Consent management services
3. Compliance checker service
4. Transparency services
5. Personal data inventory

Each of these services will be covered in more detail in their own chapters. All the components are integrated using message passing through Apache Kafka [1].

2.1 Big Data Europe

The architecture proposed here builds on the experience from the H2020 Big Data Europe (BDE) project [2].

BDE leveraged Docker technologies to simplify installing and running big data technologies.

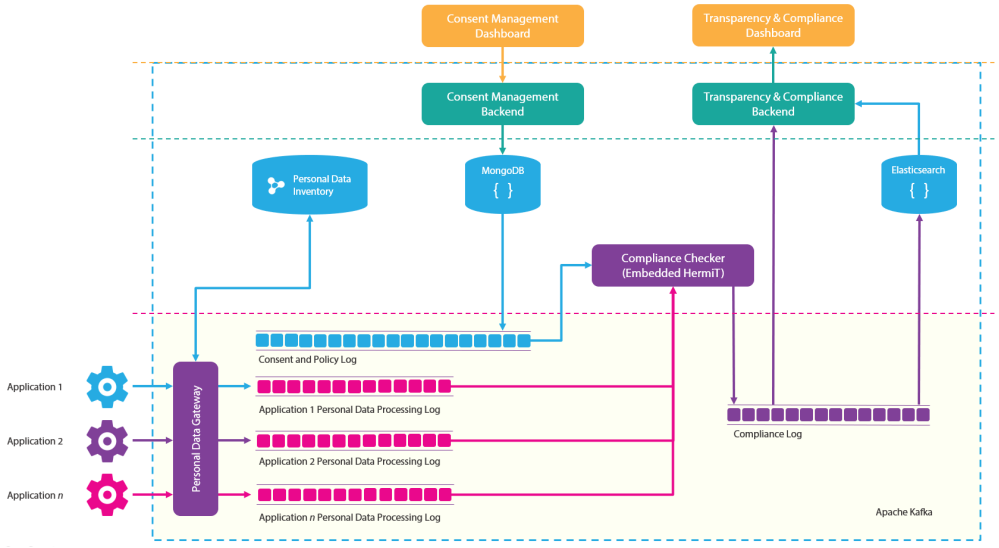


Figure 2.1: SPECIAL-K architecture setup for ex post compliance checking

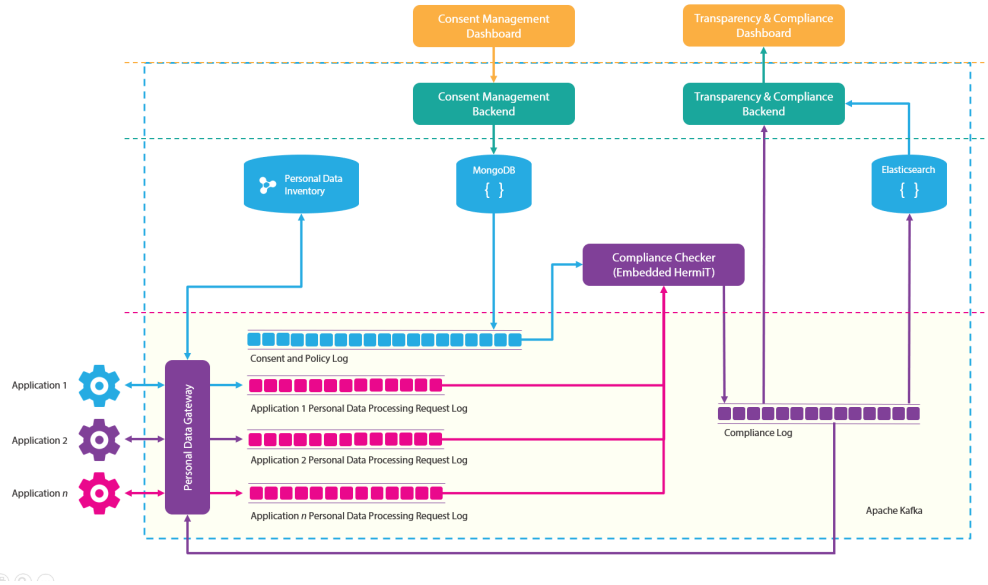


Figure 2.2: SPECIAL-K architecture setup for ex ante compliance checking

Software is packaged into Docker images for ease of distribution and composed into working systems using Docker Compose [3]. BDE leveraged Docker Swarm [4] to deploy a system onto a cluster of machines rather than a single machine.

The architecture proposed here follows all the best practices from BDE and the live prototype has been deployed using this tooling. This allows us to move the system from a single machine during development to a robust clustered deployment with ease.

2.2 Apache Kafka

Kafka [1], describes itself as a distributed streaming platform. It is easiest to think of it as a fault tolerant, append-only log. This is a very generic primitive to build robust distributed systems with.

Kafka has three main capabilities:

- Publish and subscribe to a stream of records (similar to a queue or Enterprise Service Bus (ESB))
- Store records in a fault-tolerant durable way (unlike a queue or ESB)
- Optionally process records as they occur using the Kafka Stream library

In the software system described in this deliverable, Kafka will be used as a datastore, but its data processing capabilities will not be leveraged. These will be handled using other software. This should make the approach less intrusive and make it easier for companies with existing line of business applications to adopt the platform.

Kafka has a few core abstractions:

- Kafka runs on a cluster of 1 or more servers, called *brokers*.
- Kafka stores *records* in categories called *topics*.
- Topics are subdivided into *partitions*.
- Records consist of a *key*, *offset* and *value*.

Unlike normal queueing systems, records in Kafka are persisted whether they are consumed or not. It is a kind of special purpose distributed filesystem dedicated for high-performance, low-latency commit log storage, replication, and propagation. How long records are persisted inside of Kafka is governed by a retention policy, which can be set on a topic by topic basis:

- **Time based retention**: records are kept for a certain period of time.
- **Size based retention**: records are kept in a topic until it reaches a certain size, after which the oldest records are purged until the storage quota has been met.
- **Log compaction**: Kafka ensures that at least 1 record for every key is present in the topic. Due to its importance, for the sake of clarity, log compaction is described in more detail below.

Assume there is a topic with product descriptions. Each time a product description is updated a new record is posted onto this topic with the `productId` as key and the product description as value. An example of such a topic with 6 elements is shown in Figure 2.3. In this picture the colour of the box represents the key of the record, the number below is the offset of the record.

When log compaction gets triggered, Kafka will remove all older messages for a given key, retaining only the latest one. This results in a log with "gaps" as shown in Figure 2.4. With log compaction, the size of the topic will be bounded, provided the size of the keyspace is bounded (no infinite number of products).

These various strategies give Kafka a lot of flexibility. Time based retention is great in an IOT scenario where individual records have a short half life. Size based retention is very useful in traditional queueing scenarios. Log compaction provides an elegant way to synchronise reference data between various systems.

For data consumption, Kafka combines the features of a queing system with a pub-sub system. Each consumer of data is part of a *consumer-group*. Every message on a topic will be sent to every consumer-group, implementing the broadcast behaviour of a pub-sub system. Within a consumer-group Kafka will assign the partitions of a topic to the individual consumers in the group. This allows processing of a topic to be scaled out horizontally, like what is possible with a queue and a worker pool. Because a partition can only be assigned to a single consumer within a consumer-group, the number of consumers in a consumer group can never be larger than the total number of partitions.

When compared with other storage systems, such as Hadoop, the advantage of Kafka is that it has the API of a pub-sub and queuing system. It allows us to treat data and data updates as immutable event and has well defined semantics for how to consume these, while in Hadoop's file oriented world most of the semantics need to be communicated out of band. (Are records updated in place? Are they appended to the bottom of the file?)

All these features make Kafka a very flexible data layer for our system:

- It can act as a buffer.
- It can minimize the coupling between the various components.
- Its streaming nature allows the system to do near real time data processing, while still providing support for more batch oriented workload (you can always slow down from real time, but it's hard to speed batch jobs up to realtime).
- Its easy to understand and implement semantics make it easier to build robust and scalable data processing systems.

Combined with the fact that it is mature, well supported, and proven open source software in use by some of the largest companies in the world [11, 12, 14], the authors feel confident in its selection as the data substrate for the SPECIAL system.

2.3 Authentication and Authorization

In order to authenticate users, the SPECIAL platform relies on the OpenID Connect, [7] industry standard for authentication and OAuth2 for authorization, [6].

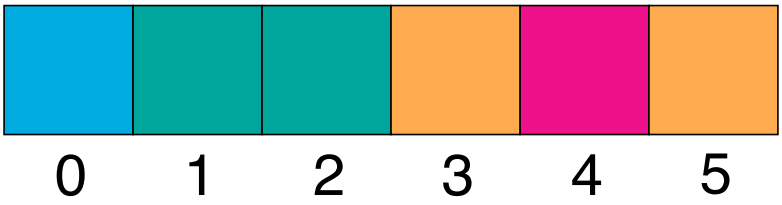


Figure 2.3: Uncompacted Log

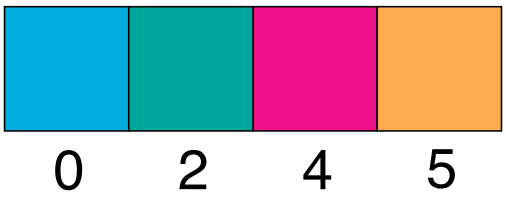


Figure 2.4: Compacted Log

2.3.1 Authentication: OpenID Connect

Authentication is the process by which the user makes a claim about his identity and proves this claim. It can losely be described as "logging in". OpenID Connect is a protocol by which both native apps and web applications[1] can delegate the authentication to a 3rd party identity provider. It builds upon OAuth2 by combining various OAuth2 message flows into an authentication flow.

Before describing the two main OpenID Connect flows in more detail, some terms are introduced:

- **Identity Provider (IDP)**: The party that offers authentication as a service. It is the service that will confirm the identity of the user (using e.g. passwords or two-factor authentication tokens). Examples of identity providers are Google, Facebook or a country's eID system.
- **Relying Party (RP)**: This party is the application which would like to establish the identity of a user. By implementing OpenID Connect it delegates this task to the IDP. The applications described in this deliverable act as RPs.
- **Claim**: This is information asserted by a user, such as name or email address.

OpenID Connect presents 3 flows:

1. **Authentication or Basic Flow**: This flow is useful for web and native applications with a trusted backend components.
2. **Implicit Flow**: This is flow is useful for web applications without a trusted backend component, such as single page web applications.

[1] Unlike the often used Security Assertion Markup Language 2.0 (SAML2) which only supports web applications.

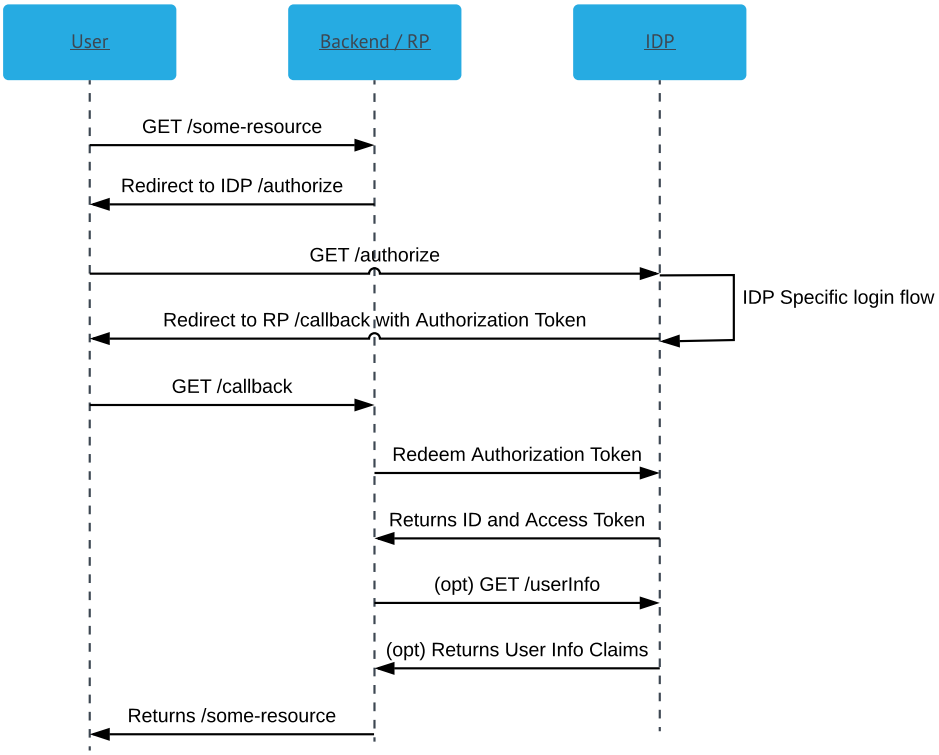


Figure 2.5: OpenID Connect Authentication Flow

3. **Hybrid Flow**: This flow is a mix of the implicit flow and authentication flow. It is hardly ever used and won't be further discussed in this deliverable.

All OpenID Connect (and OAuth2) flows assume that all communication happens over Transport Layer Security (TLS) encrypted HTTP connections (HTTPS), preventing any secrets or tokens transmitted from being leaked to attackers which evesdrop on the network connection. This moves a lot of encryption and security complexity away from developers implementing these standards in their application, into the underlying infrastructure.

The following subsections will describe the Authentication Flow (2.3.1.1) and the Implicit Flow (2.3.1.2) at a relatively high level. The goal is for the reader to get an idea how these flows work, why they are secure and that they cover the authentication needs of the platform, without losing ourselves into too many implementation details.

A more detailed overview of the tradeoffs to consider when chosing the flow an application should use can be found in [10].

2.3.1.1 Authentication Flow

The *Authentication Flow* is the most secure, and most commonly used OpenID Connect flow. A call diagram is shown in Figure 2.5

The flow describes an interaction between the following parties:

- **User-agent**. This could be a client application written by the RP, but it can also be any other application that can speak to the backend. It is an untrusted party. In Figure 2.5, this is represented as the user.
- **Backend**. This is the Relying Party (RP) which wishes to authenticate a user. It is a trusted party.
- **IDP**. The Identity Provider to which the RP wants to delegate the proving of the identity of the user.

The flow consists of the following steps:

1. The user requests a resource, or performs an action which requires him to be authenticated.
2. The RP notices that this is not an authenticated session and redirects the user to the `/authorize` of the IDP, embedding information about the RP.
3. The user follows the redirection and goes through the IDP login flow.
4. After the user has successfully followed the login flow the user is redirected to a callback at the RP. This callback embeds an *Authorization Token*.
5. The RP retrieves the *Authorization Token* from the callback and sends a request to the IDP to redeem it.
6. The IDP verifies that the *Authorization Token* is valid and issued for the RP that tries to redeem it and returns an *Access Token*, an *ID Token* and optionally a *Refresh Token*.
7. The RP can now optionally call the `/userInfo` endpoint at the IDP with the *Access Token* if it needs more user claims than those included in the *ID Token*.
8. If the token is valid and has the necessary grants, the IDP will return the claim (user information) to the RP.

At the end of this flow, the IDP will have produced 3 different tokens, each of which serves a different purpose:

- **Authorization Token**: Since the IDP does not have backchannel to talk to the RP directly, it needs to relay the results of the login flow through the untrusted user-agent. In order to prevent any confidential data from leaking, the IDP sends this single-use Authorization Token with a short time to live (typically less than an hour). This token can be exchanged only by the RP for the actual Identity and Access Tokens.
- **ID Token**: This token encodes the claims (user data such as email and name) the RP has requested from the end user. While it is theoretically possible to embed any user data the IDP has, most IDPs put limits here, requiring the RP to call the `/userInfo` endpoint for additional, more sensitive, information.

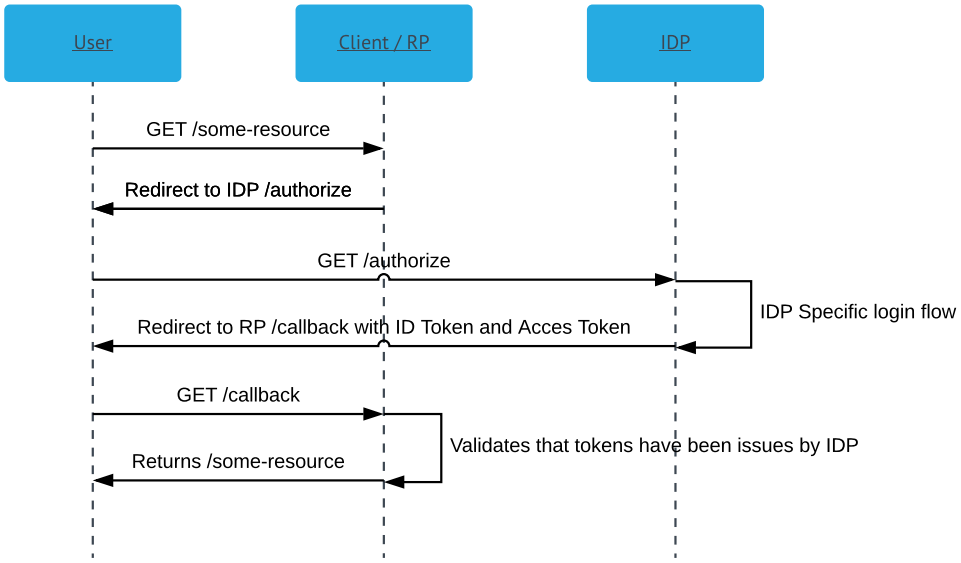


Figure 2.6: OpenID Connect Implicit Flow

- **Access Token**: The access token grants its bearer the right to call the `/userInfo` at the IDP for specific user information. The RP can use this to retrieve additional user claims not included in the ID Token, but it can also pass this token in request to downstream services which might need to verify the end users identity. The token can optionally encode additional authorizations.
- **Refresh Token**: This token can be used to refresh the Identity and Access token. This allows the time to live on these tokens to be short (limiting potential damage should they get leaked), but allows the RP to renew them without forcing the end user to go through the login flow repeatedly.

The main advantage of OpenID Connect is that the IDP can introduce new and better ways of verifying the identity claim of the user (such as 2FA or biometric methods) without any code or logic changes in the RP. The protocol does not need to change.

2.3.1.2 Implicit Flow

The implicit flow is useful for single page applications without a trusted backend. A call diagram is shown in Figure 2.6. The flow describes an interaction between the following parties:

- **User-agent**: In this flow, the user-agent is typically the browser used to interact with a client side application. It is an untrusted party.
- **Client**: This is the Relying Party (RP) trying to establish the identity of the user. In this flow the RP is assumed to run in an untrusted environment.
- **IDP**: The Identity Provider to which the RP wants to delegate the proving of the identity of the user.

The flow consists of the following steps:

1. The user requests a resource, or performs an action which requires him to be authenticated.
2. The RP notices that this is not an authenticated session and redirects the user to the `/authorize` of the IDP, embedding information about the RP.
3. The user follows the redirection and goes through the IDP login flow.
4. After finishing the login flow the IDP redirects the user to a callback at the RP. This callback embeds the *Identity Token* and an optional *Access Token*.

The flow is very similar to the Authentication flow described in Section 2.3.1.1, but in this case the RP is not running in a trusted environment. This means that the additional step of redeeming an authorization token adds no practical security: the ID and Access Token will end up in an untrusted environment anyway. Also because the RP runs in an untrusted environment, the IDP places less trust in it and will not issue a refresh token in this flow.

2.3.2 Authorization: OAuth2

OAuth2 is an authorization protocol. That means it concerns what a particular entity has access to rather than who that particular entity is. The specification describes a large amount of flows which can be implemented and that each have their own security tradeoffs. Because it can be very useful to know who an entity is when deciding what it has access to, quite a few OAuth2 flows also authenticate a user, but because the spec is focused on authorization, these aspects are often underspecified, leaving room for interpretation or custom implementations. This obviously gets in the way of interoperability and are the gaps that OpenID Connect aims to fill.

Because the OpenID Connect flows allow us to obtain authorization at the same time as authenticating a user, they currently satisfy the needs of this architecture. However in case a need for more intricate authorization flows presents itself, additional OAuth2 authorization flows can be introduced.

2.3.3 Implementation

In the demonstrator implementation Redhat Keycloak [5] has been selected as Identity provider and OpenID Connect / OAuth2 server. It is a fully featured Open Source product in use by companies big and small. Notable features of the product are:

- Federation of other identity providers through Active Directory or LDAP
- Federation of social logins such as Google or Facebook
- Broad support for authentication and authorization protocols such as SAML 2, OAuth2 and OpenID Connect

In case a company trying to adopt the system does not have an identity provider which supports OpenID Connect out of the box, Keycloak can be used to provide an OpenID Connect server without invasive changes to the existing landscape.

It is worth reiterating that while the demo system uses Keycloak, there is no strict requirement on it. The system has standardised on the OpenID Connect and OAUth2 protocols, not this particular implementation.

Chapter 3

Consent Management

This chapter describes the backend of the consent management service. Even though the demonstrator includes a frontend, the frontend / UX discussions are handled in WP4 and, more specifically, D4.1 Transparency Dashboard and Control Panel Release V1. The included frontend is just there to make it easier to present and evaluate the backend features.

The purpose of this service is to provide data subjects and data controllers a way to manage their policies. These type of services are commonly refered to as CRUD services: they need to (C)reate, (R)ead, (U)pdated and (D)elete data entities. An architecture that is commonly used to implement CRUD services, augmented with an audit log, has been chosen:

- An API Layer which allows frontends and other clients to call its services. Data validation and authorization checks happen here as well (Section 3.1).
- A database layer which persists the data in a format which is optimized for use by the API Layer (Section 3.2).
- Audit logs which record all transactions (Section 3.3).

3.1 API Design

The consent management API allows the manipulation of 3 different entities:

1. Applications
2. Users
3. Policies

Each of these entities is manipulated in a similar way. The following subsections will briefly describe the various endpoints and show some example payloads. This is not a full nor a final API specification, but should give the interested reader a decent understanding of how the API should be used. It is highly likely that specifics of the API will change as the platform evolves: this is very much a work in progress and no backward compatibility guarantees are given.

The API calls which allow for the retrieval of policy data are intended for use by UI clients

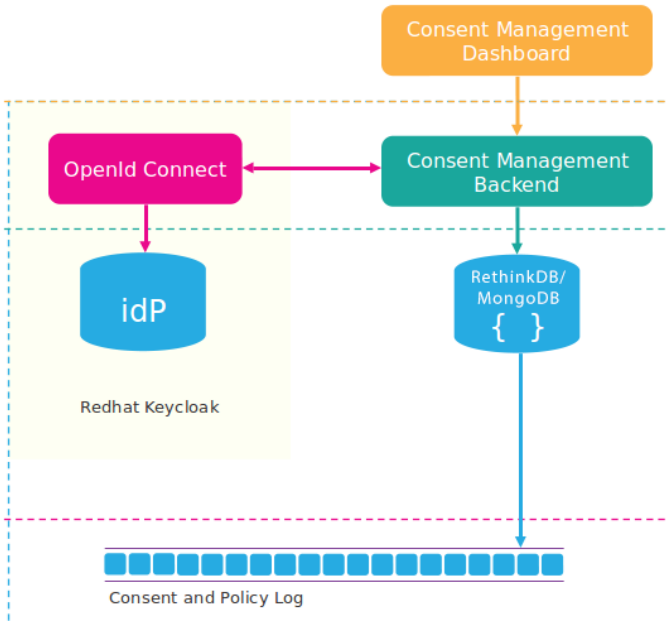


Figure 3.1: Consent Management

which wish to render an individual users policies. Services which require much more intensive use of policy data, such as the compliance checker or a potential authorization server, should preferably consume the policies from the full policy Kafka topic (see Section 3.3). This provides better decoupling, relaxes performance requirements on this service and provides consuming services with the option of reshaping the policy data to better fit their needs.

3.1.1 Applications

The `/applications` endpoints allow applications and their associated policies to be registered with the system.

- GET `/applications`
  Returns a list of all currently registered applications.

  Example Response:

```
[
  {
    "id": "d5aca7a6-ed5f-411c-b927-6f19c36b93c3",
    "name": "invoicer",
    "links": {
      "policies": "/applications/d5aca7a6-ed5f-411c-b927-6
          f19c36b93c3/policies"
    }
  },
  {
    "id": "58916c9e-3ce2-4fdb-94a4-369525582e75",
```

```
    "name": "marketing-machine",
    "links": {
      "policies": "/applications/58916c9e-3ce2-4fdb-94a4-369525582
          e75/policies"
    }
  }
]
```

- POST `/applications`
  Creates a new application. Returns the created application with its generated ID if the request was sucessful.

  Example Request:

```
{
  "name": "new-application"
}
```

  Example Response:

```
{
  "id": "ca775d49-c3a3-4e08-9e6a-9ac49612ad62",
  "name": "new-application"
  "links": {
    "policies": "/applications/ca775d49-c3a3-4e08-9e6a-9
        ac49612ad62/policies"
  }
}
```

- GET `/applications/:id`
  Returns a single registered application.

  Example Response:

```
{
  "id": "d5aca7a6-ed5f-411c-b927-6f19c36b93c3",
  "name": "invoicer",
  "links": {
    "policies": "/applications/d5aca7a6-ed5f-411c-b927-6
        f19c36b93c3/policies"
  }
}
```

- PUT `/applications/:id`
  Updates a single registered application. Returns the updated data.

  Example Request:

```
{
  "id": "d5aca7a6-ed5f-411c-b927-6f19c36b93c3",
  "name": "accounting"
}
```

Example Response:

```
{
  "id": "d5aca7a6-ed5f-411c-b927-6f19c36b93c3",
  "name": "accounting",
  "links": {
    "policies": "/applications/d5aca7a6-ed5f-411c-b927-6
       f19c36b93c3/policies"
  }
}
```

- DELETE `/applications/:id`
  Deletes a single registered application. Takes no payload and returns no data.
- GET `/applications/:id/policies`
  Returns the policies associated with a single application.

  Example Resonpse:

```
{
  "policies": [
    "d5bbb4cc-59c0-4077-9f7e-2fad74dc9998",
    "54ff9c00-1b47-4389-8390-870b2ee9a03c",
    "d308b593-a2ad-4d9f-bcc3-ff47f4acfe5c",
    "fcef1dbf-7b3d-4608-bebc-3f7ff6ae4f29",
    "be155566-7b56-4265-92fe-cb474aa0ed42",
    "8a7cf1f6-4c34-497f-8a65-4c985eb47a35"
  ]
}
```

3.1.2 Users

The `/users` endpoints allow the policies of individual data subjects to be retrieved and modified. Registration of data subjects with the system is handled by the identity provider, so the API does not provide any specific endpoints for these actions.

- GET `/users/:id`
  Return a single user information and its policies.

  Example Response:

```
{
  "id": "9b84f8a5-e37c-4baf-8bdd-92135b1bc0f9",
  "name": "Bernard Antoine",
  "links": {
    "policies": "/users/9b84f8a5-e37c-4baf-8bdd-92135b1bc0f9/
       policies"
  }
}
```

- PUT `/users/:id`
  Update a single user information and its policies

Example Request:


```
{
  "id": "9b84f8a5-e37c-4baf-8bdd-92135b1bc0f9",
  "policies": [
    "d5bbb4cc-59c0-4077-9f7e-2fad74dc9998",
    "0cb2b717-a442-4da5-818c-1c1c2e762201"
  ]
}
```


Example Response:


```
{
  "id": "9b84f8a5-e37c-4baf-8bdd-92135b1bc0f9",
  "name": "Bernard Antoine",
  "links": {
    "policies": "/users/9b84f8a5-e37c-4baf-8bdd-92135b1bc0f9/
       policies"
  }
}
```


- GET `/users/:id/policies`
  Returns the policies associated with a particular user.

Example Response:


```
{
  "policies": [
    "d5bbb4cc-59c0-4077-9f7e-2fad74dc9998",
    "0cb2b717-a442-4da5-818c-1c1c2e762201"
  ]
}
```


3.1.3 Policies

- GET `/policies`
  Returns a list of all policies currently registered in the system.

Example Response:


```
[
  {
    "id": "d5bbb4cc-59c0-4077-9f7e-2fad74dc9998",
    "dataCollection": "http://www.specialprivacy.eu/vocabs/data#
       Anonymized",
    "locationCollection": "http://www.specialprivacy.eu/vocabs/
       data#EU",
    "processCollection": "http://www.specialprivacy.eu/vocabs/
       data#Collect",
    "purposeCollection": "http://www.specialprivacy.eu/vocabs/
       data#Account",
```

```
    "recipientCollection": "http://www.specialprivacy.eu/vocabs/
       data#Delivery",
    "explanation": "I consent to the collection of my anonymized
       data in Europe for the purpose of accounting."
  },
  {
    "id": "54ff9c00-1b47-4389-8390-870b2ee9a03c",
    "dataCollection": "http://www.specialprivacy.eu/vocabs/data#
       Derived",
    "locationCollection": "http://www.specialprivacy.eu/vocabs/
       data#EULike",
    "processCollection": "http://www.specialprivacy.eu/vocabs/
       data#Copy",
    "purposeCollection": "http://www.specialprivacy.eu/vocabs/
       data#Admin",
    "recipientCollection": "http://www.specialprivacy.eu/vocabs/
       data#Same",
    "explanation": "I consent to the copying of my derived data
       in Europe-like countries for the purpose of administration
       ."
  }
]
```


- POST `/policies`
  Creates a new policy. Returns the created policies if the operation was successful.

  Example Request:


```
{
  "dataCollection": "http://www.specialprivacy.eu/vocabs/data#
     Computer",
  "locationCollection": "http://www.specialprivacy.eu/vocabs/data
     #ThirdParty",
  "processCollection": "http://www.specialprivacy.eu/vocabs/data#
     Move",
  "purposeCollection": "http://www.specialprivacy.eu/vocabs/data#
     Browsing",
  "recipientCollection": "http://www.specialprivacy.eu/vocabs/
     data#Public",
  "explanation": "I consent to the moving of my computer data on
     third-party servers for the purpose of browsing."
}
```


  Example Response:


```
{
  "id": "d308b593-a2ad-4d9f-bcc3-ff47f4acfe5c",
  "dataCollection": "http://www.specialprivacy.eu/vocabs/data#
     Computer",
  "locationCollection": "http://www.specialprivacy.eu/vocabs/data
     #ThirdParty",
  "processCollection": "http://www.specialprivacy.eu/vocabs/data#
     Move",
```

```
  "purposeCollection": "http://www.specialprivacy.eu/vocabs/data#
     Browsing",
  "recipientCollection": "http://www.specialprivacy.eu/vocabs/
     data#Public",
  "explanation": "I consent to the moving of my computer data on
     third-party servers for the purpose of browsing."
}
```


- GET `/policies/:id`
  Returns an individual policy.

  Example Response:


```
{
  "id": "d5bbb4cc-59c0-4077-9f7e-2fad74dc9998",
  "dataCollection": "http://www.specialprivacy.eu/vocabs/data#
     Anonymized",
  "locationCollection": "http://www.specialprivacy.eu/vocabs/data
     #EU",
  "processCollection": "http://www.specialprivacy.eu/vocabs/data#
     Collect",
  "purposeCollection": "http://www.specialprivacy.eu/vocabs/data#
     Account",
  "recipientCollection": "http://www.specialprivacy.eu/vocabs/
     data#Delivery",
  "explanation": "I consent to the collection of my anonymized
     data in Europe for the purpose of accounting."
}
```


- PUT `/policies/:id`
  Updates an individual policy. The updated policy is returned if the operation was successful.

  Example Request:


```
{
  "id": "d5bbb4cc-59c0-4077-9f7e-2fad74dc9998",
  "locationCollection": "http://www.specialprivacy.eu/vocabs/data
     #EULike"
}
```


  Example Response:


```
{
  "id": "d5bbb4cc-59c0-4077-9f7e-2fad74dc9998",
  "dataCollection": "http://www.specialprivacy.eu/vocabs/data#
     Anonymized",
  "locationCollection": "http://www.specialprivacy.eu/vocabs/data
     #EULike",
  "processCollection": "http://www.specialprivacy.eu/vocabs/data#
     Collect",
  "purposeCollection": "http://www.specialprivacy.eu/vocabs/data#
     Account",
```

```
  "recipientCollection": "http://www.specialprivacy.eu/vocabs/
     data#Delivery",
  "explanation": "I consent to the collection of my anonymized
     data in Europe for the purpose of accounting."
}
```

- DELETE `/policies/:id`
  Removes an individual policy. References to this policy are also removed from applications and users. Takes no payload and returns no data.

3.1.4 Authorization

The consent management service relies on the OpenID Connect Authentication Flow (see 2.3.1). This is the most secure OpenID Connect flow. The consent management service uses the data from the ID Token to bootstrap a user in the system, if it does not yet exist. This is why no POST `/users` and DELETE `/users/:id` endpoints exist. The entire lifecycle of a user is outsourced to the identity provider.

Similarly the information contained in the ID Token is used to filter data to just the data from the authenticated user.

3.2 Database Layer

For the database layer, originally, RethinkDB, [8], had been chosen. As stated in D3.2, the choice for RethinkDB was never critical for the SPECIAL platform, as most databases can be easily used for CRUD services. As the Proximus use case (as stated in D5.1 Pilot implementations and testing plans) relies on MongoDB, a conscious choice was made to switch over to MongoDB as the default consent management backend data store, while still offering RethinkDB as an alternative.

3.2.1 Document Store

Both MongoDB and RethinkDB are document stores which can persist native JSON. Because our API layer communicates using JSON, this aspect minimizes the impedance mismatch between the two parts. The document model is also very flexible and allows us to easily modify the schema of the data. Unlike Rethinkdb, MongoDB offers full support for multi-document ACID transactions, which can be seen as an advantage over its predecessor (even though document-level atomicity can be sufficient for our purposes with careful document design).

3.2.2 Streaming Queries

Rethinkdb offers first class support for streaming queries. In most databases queries will return the matching data at the point the query was issued. If the application is interested in updates on this data it will need to poll the database, by regularly issuing the query again. Rethinkdb on the other hand allows a client to subscribe to a query. When the subscription starts Rethinkdb will return the results of the query as usual, but when changes to the database happen, which impact the results of the query, Rethinkdb will push the delta between the original query result and the

current query result to the client.

This feature makes it very easy to implement the audit log. A query can be created which returns the data in exactly the right shape. The data can then be written onto Kafka for long term persistance as it arrives.

MongoDB, on the other hand, has *change streams*, allowing applications to easily subscribe to changes on the data in the system, which is necessary to populate the audit log. The *change events* emitted by these streams contain only the updated information of the record, which is not always sufficient. If this is the case, conventional database calls are used to gather the necessary information after successfully updating the consent information.

3.3 Change Feeds

The consent management service provides two change feeds:

1. Transaction Log
2. Full Policy Log

The consent management service is the only service which can write data to either of these logs. The access control mechanisms in Kafka are used to enforce this. The logs allow the reconstruction of the current state by replaying them in their entirety, but they do serve distinct purposes which are described in more detail in the following subsections.

3.3.1 Transaction Log

The transaction log is retained for audit purposes. It logs every command sent by a client. It could be described as the log of the intent of the user. It is strictly ordered and contains only the differences between two states. For example (this is not the actual format used on the log, but a more human readable version):

```
SET "3bd2731b-2361-4de6-b0e5-dd12e64827a9" [{"id": "3bd2731b-2361-4
    de6-b0e5-dd12e64827a9", "purpose": "charity"}]
```

It can be used to figure out who changed what data at a particular point in time. It also allows the state of the consent management service to be reconstructed at any particular point in time, by replaying the log in a fresh instance until that timestamp. This can be useful for what-if analysis or proving to an auditor that particular processing was lawfull at a particular time.

3.3.2 Full Policy Log

The full policy log is retained for integration purposes. It is a compacted Kafka topic (see Section 2.2): only the latest version of the policies of a data subject are retained. This makes it easy for services like the compliance checker to bootstrap their own materialised view of the policies and consume updates to those policies, without placing any load on the consent management service.

The data subject ID is stored in key of the record. The record value is a json represenation of the data subject policies. For example:

```
{
  "simplePolicies":[
    {
      "data":"http://www.specialprivacy.eu/vocabs/data#Anonymized",
      "processing":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyProcessing",
      "purpose":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyPurpose",
      "recipient":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyRecipient",
      "storage":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyDuration"
    },{
      "data":"http://www.specialprivacy.eu/vocabs/data#AnyData",
      "processing":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyProcessing",
      "purpose":"http://www.specialprivacy.eu/langs/usage-policy#
         Charity",
      "recipient":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyRecipient",
      "storage":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyDuration"
    }
  ]
}
```

Because this log is compacted, it cannot be used to reconstruct the state of the consent management system at an arbitrary point in the past.

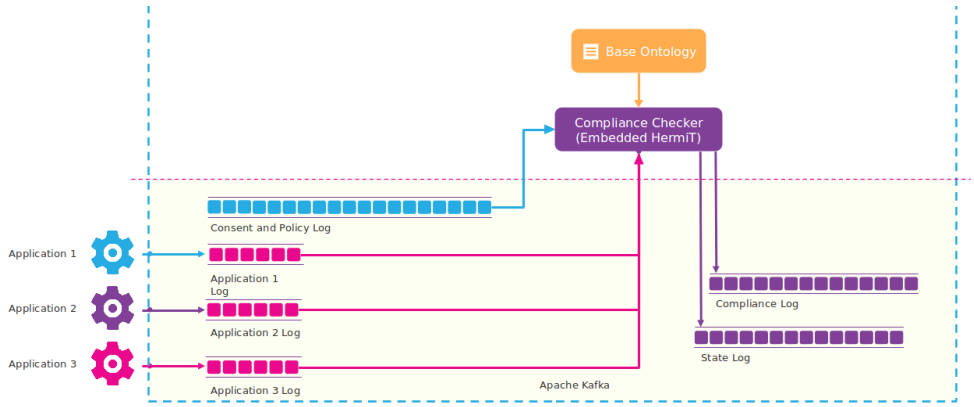


Figure 4.1: Compliance Checker

Chapter 4

Compliance Checking

This chapter describes the compliance checker service in more detail. Its purpose is to validate that application logs are compliant with a users policy. These application logs are delivered in the format described in deliverable D2.3, the policies are an implementation of the policy language described in deliverable D2.1.

Figure 4.1 shows an overview of the components that will be discussed in this section.

4.1 Data Flow

The compliance checker can be seen as a stream processor which takes in a stream of application logs and emits an augmented stream of logs. The system has the following data inputs:

- **Application Log Topic**: This is a normal Kafka topic that contains all application logs which need to be checked for compliance.
- **Policies Topic**: This is a compacted Kafka topic which holds the complete policies for all data subjects.

- **Base Ontology**: The base ontology are the vocabularies and class relationships which define the policy language as described in deliverable D2.1.

The system has the following outputs:

- **Compliance Topic**: This is a normal Kafka topic which contains the augmented application logs.
- **State Topic**: This is a compacted Kafka topic where the compliance checker can checkpoint the latest offset is has processed. This allows it easily restore its state in case it needs to restart.

4.1.1 Application Log Topic

The application log topic contains, as the name implies, the logs produced by the various line of business applications in the broader ecosystem. The compliance checker assumes that the logs are written in, or have been transformed into, the json serialization of the format described in D2.3. An example log can look as follows:

```
{
  "timestamp": 1524223395245,
  "process": "send-invoice",
  "purpose": "http://www.specialprivacy.eu/vocabs/purposes#Payment",
  "processing": "http://www.specialprivacy.eu/vocabs/processing#Move
     ",
  "recipient": "http://www.specialprivacy.eu/langs/usage-policy#
     AnyRecipient",
  "storage": "http://www.specialprivacy.eu/vocabs/locations#
     ControllerServers",
  "userID": "49d40b22-4337-4652-b463-41b1c23c6b08",
  "data": [
    "http://www.specialprivacy.eu/vocabs/data#OnlineActivity",
    "http://www.specialprivacy.eu/vocabs/data#Purchase", "http://www.
       specialprivacy.eu/vocabs/data#Financial"
  ]
}
```

In a future version of the service, a jsonld context will most likely be added to this file. Alternatively, a turtle serialization could be used to make parsing the logs easier.

Records on this topic use the data subject ID as a key, so that the data can be easily partitioned by data subject. This is helpfull when scaling up the work, see Section 4.3.

4.1.2 Policies Topic

The policies topic is a compacted Kafka topic which contains full policies for all data subjects. Its content is produced by the consent management service, described in 3. The records have the data subject ID as a key, so that the data can be easily partitioned by data subject. This is helpfull when scaling up the work, see Section 4.3.

Data subject policies are stored in a json serialization, an example of which can look as follows:

```
{
  "simplePolicies":[
    {
      "data":"http://www.specialprivacy.eu/vocabs/data#Anonymized",
      "processing":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyProcessing",
      "purpose":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyPurpose",
      "recipient":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyRecipient",
      "storage":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyDuration"
    },{
      "data":"http://www.specialprivacy.eu/vocabs/data#AnyData",
      "processing":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyProcessing",
      "purpose":"http://www.specialprivacy.eu/langs/usage-policy#
         Charity",
      "recipient":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyRecipient",
      "storage":"http://www.specialprivacy.eu/langs/usage-policy#
         AnyDuration"
    }
  ]
}
```

When a record is read from the topic, it is transformed into an OWL XML representation and saved to a temporary file, which can be indexed by the subject ID. When a new version of the policy for a particular data subject is read from Kafka, the existing temporary file is completely overwritten. In a future version of this service, this file based index will be replaced with a (potentially in-memory) key-value store.

4.1.3 Base Ontology

The base ontology is a collection of OWL statements which describe the various classes and their relationships, which are used to express policies. Without these definitions the OWL reasoner does not know how to make sense of any policies. The base ontology is saved in OWL XML format and loaded from disk at startup. These files are shipped together with the binary. In the current version of the policy checker, there is no way to load additional or different versions of the base ontology, they are effectively hard coded.

A planned improvement is to turn the base ontology into data which the compliance checker consumes from a Kafka topic. This will allow companies to more easily update the base ontology, or define additional vocabularies which define company specific attributes to use in policies.

4.2 Compliance Checking

4.2.1 Application Log Flow

When a compliance checker instance starts up, it initialises itself by first loading the base ontology into memory.

The compliance of each application log is checked by going through the following steps:

1. Clone the base ontology into a new `OWLOntology`
2. Read the data subject ID from the key of the application log
3. Lookup the data subject policy by the data subject ID
4. Load the policy into the cloned `OWLOntology`
5. Peform the subsumption check (see Section 4.2.2)
6. Add the check result to the application log data structure
7. Write the augmented application log to the Compliance Kafka topic
8. Discard the cloned `OWLOntology`

It is worth noting that application log is never persisted by the compliance checker and only retained in memory for the duration of the subsumption check. Because the `OWLOntology` used for the subsumption check only contains a small amount of information, these checks are evaluated very quickly. An additional performance improvement was accomplished by avoiding unnecessary (de)serialization steps. This will be demonstrated in D3.5 Scalability and Robustness testing report V2.

4.2.2 Subsumption

As can be seen in deliverables D2.1 and D2.3, it can be verified that an application log is compliant with a data subject policy by performing a subsumption check. The subsumption algorithm used by the compliance checker is OWL API 3.4.3[1] compliant. It creates an `OWLSubclassOfAxiom` that takes 2 `OWLClasses` and returns an `OWLAxiom` object that states that the first class is a superclass of the second class.

This is then passed on to the `isEntailed` method which returns `true` or `false`. The implementation of the reasoner is HermiT but this should be easily swapped with any other OWL API 3.4.3 compliant reasoner.

4.3 Scaling and Fault-Tolerance

Section 4.2 details how an individual application log flows through the compliance checker and is validated. This section zooms out a bit and details how the compliance checker can scale to support a load beyond what a single instance can handle. Because scaling the computation to

[1] http://owlcs.github.io/owlapi/

multiple instances turns the compliance checker into a distributed system, it is also important to look at how the processing will handle the various failures which will inevitably occur.

Since all of the data being processed by the compliance checker is stored on Kafka, it is important to understand the primitives it gives us to build a distributed stream processing system (see also Section 2.2). Topics in Kafka are divided into partitions. Partitions are the actual log structures which are persisted on disk. As a result ordering between records is only guaranteed within a partition. If the record producer does not specify a partition explicitly, Kafka decides which partition a record gets written to by using a partition function, which can take the key of the record into account. Because a partition can only be assigned to a single consumer in a consumer group, the number of partitions puts an upper limit to how far the processing of records can be scaled out.

The total number of partitions of the application log topic will decide how many instances of the compliance checker can process the data in parallel. Because the records are assigned to a partition based on the data subject ID, it is guaranteed that an individual compliance checker instance will see all logs about a particular data subject in the order they occured. This is currently not necessary for the compliance algorithm to work, but keeps the option open to take into account multiple log messages to make decisions about compliance.

Kafka automatically assigns partitions to individual consumers, and will rebalance the partition assignment in case consumers get added or removed from the consumer group. This ensures that all messages get processed and that each consumer gets a fair share of the work, even if individual consumers fail.

In case of catastrophic failure, where all consumers die, the last processed offsets per partition can be recovered from the state topic. This prevents the new compliance checker instances from redoing work which was already done previously. Provided the new instances are spawned before the existing log messages fall out of retention, this catastrophic failure will also not result in data loss.

It is also worth noting that in order to scale out the work and provide fault tolerance, no other functionality, other than a few primities provided by Kafka, is being relied upon. There are no restrictions on how the compliance checker is programmed or deployed: no special libraries or resource scheduler is required. In fact the compliance checker looks just like any other java application from an operational perspective. This is in stark contrast with data processing frameworks like Spark, which require that the processing is implemented in their own specific framework and deployed on dedicated clusters using specific resource managers.

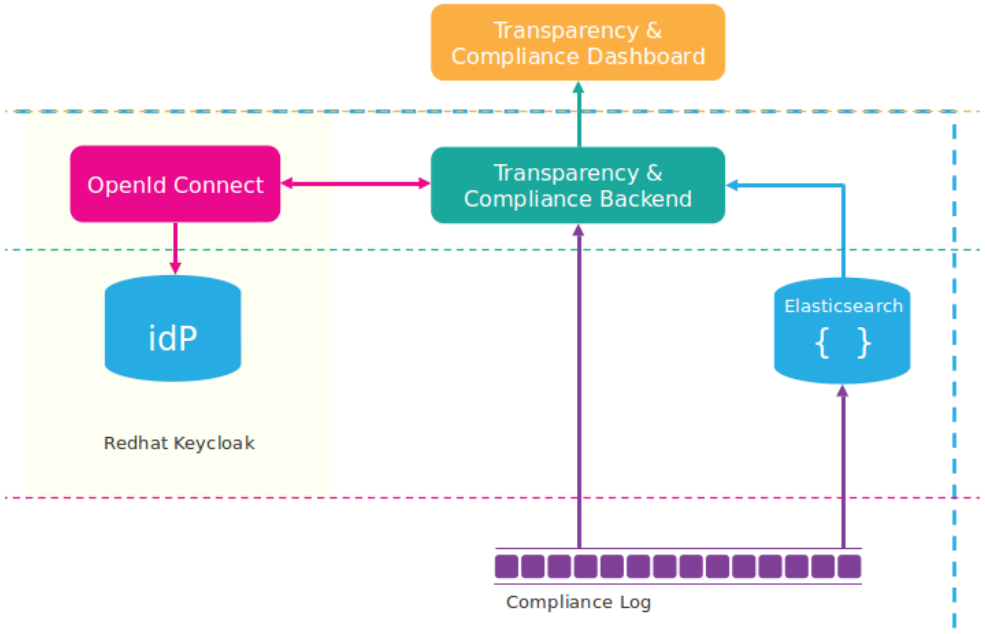


Figure 5.1: Transparency Dashboard

Chapter 5

Transparency Dashboard

This chapter covers the backend of the transparency and compliance dashboard (also referred to as "the privacy dashboard"). Frontend / UX concerns are handled in WP4 and D4.1.

At the moment the transparency backend is still fairly minimal. The focus for this deliverable has been on the consent management, compliance checking and overall integration mechanisms. Figure 5.1 shows the current architecture proposal. The actual results will be documented in deliverable D3.4.

5.1 Overview of Components

As can be seen in Figure 5.1, the proposal for the transparency service will consist of the following components

- **Compliance Log**: This is the output of the compliance checker and will serve as the reference for any visualisations
- **Elasticsearch**: Elasticsearch will contain an indexed version of the compliance log and will provide faceted browsing, easy lookups and aggregations.
- **Transparency Backend**: The transparency backend will act as the sole entrypoint for any UIs built as part of deliverable 4.1. It will provide access control and enforce authorized access to the data in elasticsearch or the compliance log.

5.2 Current State

The version of the transparency service which is currently available, consists of the following components:

- **Compliance Log**: This is the output of the compliance checker
- **Transparency Backend**: A service which exposes the compliance log as server sent events, [9], to a web client
- **Placeholder Dashboard**: A temporary dashboard which visualises the events on the compliance log in real time

These components prove that it is possible to stream a Kafka topic in real time in a web client. Any other features expected to present in the final solution, such as access control and faceted browsing, have not yet been implemented.

Chapter 6

Personal Data Inventory

As described in D2.7 Transparency Framework V2, PII[1]-oriented discovery tools and metadata repositories are insufficient and proposes an alternative approach to building data subject-centric digital enterprise inventories. Such an inventory is a requirement for compliance.

6.1 Personal Data Inventory Architecture

Figure 6.1 shows the SPECIAL Personal Data Inventory backend architecture. The microservice setup involves three distinct layers which will be described further below:

- Dispatch Layer (yellow)
- Business Layer (blue)
- Data Layer (green)

6.1.1 Dispatch Layer

The dispatch layer is simple and consists of a single microservice. The **dispatcher/identifier** needs to:

- add a cookie,
- add an identification header to a request if it is not present,
- remove blacklisted headers from outgoing requests, and
- dispatch requests to other microservices.

6.1.2 Business Layer

This middle layer implements the business logic and consists of ten distinct services.

Indexer Service. The indexer service will index the entire dataset everytime a dataset is added to the catalog in Elastic Search. The created index stores a hashed representation of the original data.

[1] Personally Identifiable Information

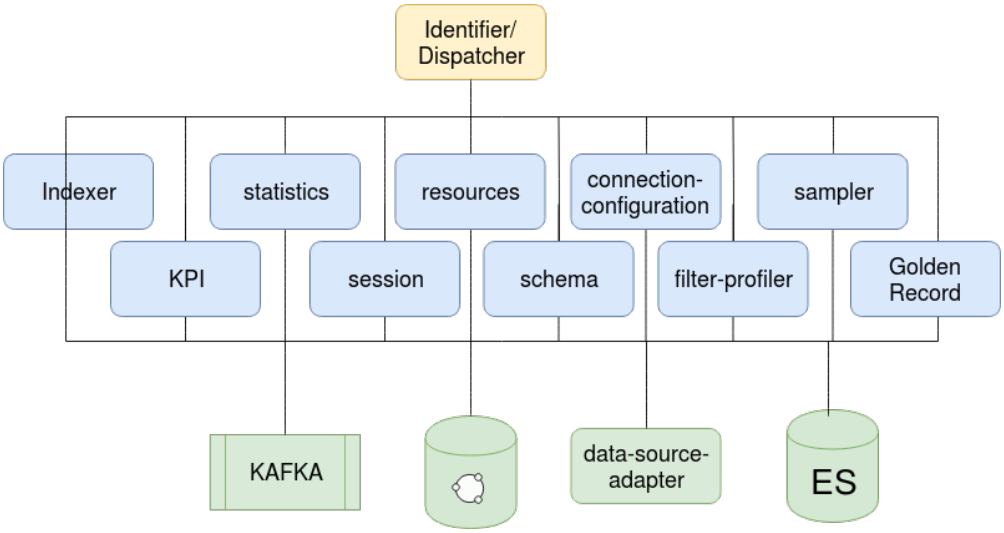


Figure 6.1: Personal Data Inventory backend architecture

Session Service. The session service facilitates user authentication. It can take into account 3rd party OAuth services.

Golden Record Service. The golden record service handles identity samples (*golden records*) discovers identity shapes. A side effect of the workings of this service is that some identities might be created in the process of discovering the general data shape.

Connection Configuration Service. The connection service offers to obtain JSON API compliant connection information and also creates connections. When it does, it also ensures that the connection object is valid prior to storing it in the database.

Schema Service. For any given connection that is stored in the database, the schema service analyzes the data source and stores its schema in the database. Furthermore, the schema service will offer JSON API compliant responses on schema objects.

Filter Profiler Service. Given a connection object with a schema object, the profile service runs a profile on that connection and adds it to the database. The service can also return profiles as resources to the frontend consumers.

Data Service. The data service provides the frontend with small (at first, "raw") samples of data for a given connection.

Resource Service. The resource service provides all plain resource objects that can be consumed by the front end.

Sampler Service. The sampler service extracts a data sample for early profiling from a cataloged data source.

Statistics Service. The statistics service returns meaningful statistics for frontend charting libraries. The only supported statistic at the time of writing is a Sankey diagram.

KPI Service. The KPI Service can retrieve KPI queries from the triple store and run them. Moreover, it also offers JSON API resources on the run KPI queries.

6.2 Data Layer

The data layer stores the inventory data, stored in a Lucene index, and metadata, including the graph (RDF) representation of the data catalog and the identities.

Connector Service. The connector offers an abstraction to outside data sources for which we want to offer specific services (e.g. get schema, get sample, or get connection info).

Triple Store. The RDF store is OpenLink Virtuoso here. The triple store is the single source of truth.

Kafka. Events are published in Kafka topics. These topics can be subscribed to by any interested microservice.

Elasticsearch (ES). We use Elasticsearch to index all of our data assets. Only hashes of data and id's are stored here.

6.3 Personal Data Inventory Gateway

In the SPECIAL-K architecture, any request for personal data would go through a personal data gateway, which would in turn consult the inventory for relevant metadata, including ownership, making policy compliance checking per data subject possible.

Interfacing with Line-of-Business applications will be the focus of year three research, the results of which will be presented in D3.6 Final release.

Chapter 7

RDF Compression

This chapter briefly motivates and reviews the most important works on RDF compression (Section 7.1). Then, we present our current efforts on managing compressed RDF datasets (Section 7.2) and RDF archives, i.e. versions of evolving data (Section 7.3). Our initial insights set the basis for a new generation of compressed-based Big Semantic Data stores that can be integrated in our SPECIAL platform to manage large and evolving semantic data at Web scale.

7.1 Compressing RDF Data

The steady adoption of Linked Data, together with the support of key open projects (such as DBpedia, Wikidata or Bio2RDF), have promoted RDF as a de-facto standard to represent facts about arbitrary knowledge in the Web, organized around the emerging notion of knowledge graphs. This impressive growth in the use of RDF has irremediably led to increasingly large RDF datasets and consequently to scalability challenges in *Big Semantic Data* management.

RDF is an extremely simple model where a graph is a set of *triples*, a ternary structure (subject, predicate, object), which does not impose any physical storage solution. RDF data management is traditionally based on human-readable serializations, which add unnecessary processing overheads in the context of a large-scale and machine-understandable Web. For instance, the latest DBpedia (2016-10) consists of more than 13 billion triples. Even though transmission speeds and storage capacities grow, such graphs can quickly become cumbersome to share, index and consume. This scenario calls for efficient and functional representation formats for RDF as an essential tool for RDF preservation, sharing, and management.

RDF compression can be defined as the problem of encoding an RDF dataset using less bits than that required by text-based traditional serialization formats like `RDF/XML`, `NTriples`, or `Turtle`, among others [**?**]. These savings immediately lead to more efficient storage (i.e. archival) and less transmission costs (i.e. less bits over the wire). Although this problem can be easily solved through universal compression (e.g. gzip or bzip2), optimized RDF-specific compressors take advantage of the particular features of RDF datasets (such as semantic redundancies) in order to save more bits or to provide retrieval operations on the compressed information.

In the following, we first review the state of the art on specific RDF compressors (Section 7.1.1). Then, we list and discuss the most important applications in the area (Section 7.1.2).

7.1.1 Classification of RDF compressors

RDF compressors can be classified into *physical* and *logical*: the former exploits symbolic/syntactic redundancy, while the latter focuses on semantic-based redundancy. Finally, *hybrid* compressors perform at physical and logical levels. An ortogonal view considers the functionality of the compression scheme, where RDF self-indexes allows for efficient RDF retrieval on compressed space.

RDF-specific physical compressors usually perform *dictionary compression*. That is, they translate the original RDF graph into a new representation which includes an string dictionary and an ID-graph encoding:

- The **dictionary** organizes the RDF vocabulary, which comprises all different terms used in the dataset.
- The **ID-graph** replaces the original terms by their corresponding IDs in the dictionary.

Physical compressors propose different approaches to organize and compress RDF dictionaries, and to encode the corresponding ID-graph representations. *Dictionary compression* has not received much particular attention, in spite that representing the RDF vocabulary usually takes more space than the ID-graph encoding [**?**]. Martínez-Prieto et al. [**?**] proposes different approaches to compress RDF dictionaries.D_{comp} [**?**] is a dictionary technique that splits the dictionary by role (subject, object and predicate) and vocabulary (URIs, blank nodes, and literals). The resulting multi-dictionary organization allows for choosing the best compression technique for each collection of RDF terms, reporting competitive compression ratios and enabling fine-grained retrieval operations to be performed.

Once removed symbol repetitions, *ID-graph compression* looks for syntactic redundancy on the resulting ID-graph. These techniques model the graph in terms of adjacency lists or matrices, and look for regularities or patterns, which are succinctly encoded. HDT [**?**] proposes *BitmapTriples*, one of the pioneer approaches for (RDF) ID-graph compression. In essence, it transforms the graph into a forest of three-level trees: each tree is rooted by a subject ID, having its adjacency list of predicates in the second level and, for each of them, the adjacency list of related objects in the third (leaf) level. The whole forest is then compressed using two ID sequences (for predicates and objects), and two bitsequences which encode the number of branches and leaves of each tree. This simple encoding reports interesting compression ratios ($10 - 25\%$ of the original space), while supporting efficient triple decoding. Furthermore, Bitmap Triples allows subject-based queries to be resolved by traversing subject trees from the root. HDT consolidates a binary serialization format by joining FrontCoding and BitmapTriples to compress dictionaries and ID-graphs respectively.

OFR [**?**] proposes another compression scheme for ID-graphs. It first performs dictionary compression (terms are organized into a multi-dictionary using differential encoding), and the resulting ID-graph is sorted by objects and subjects. In this case, *run-length* and *delta* compression [**?**] are applied to exploit multiple object occurrences, and the non-decreasing order of the consecutive subjects, respectively. OFR compressed files are then re-compressed using universal techniques like `gzip` or `7zip`. The resulting OFR effectiveness improves HDT, but its inner data organization discourages any chance of efficient retrieval.

Logical compressors look for (redundant) triples that can be inferred from others. These triples are removed from the original graph, and only the resulting *canonical subgraph* is finally

serialized. Different approaches have been followed to obtain these canonical subgraphs. The initial approaches [**?** **?**] are based on the notion of *lean subgraph*. The lean subgraph is a subset of the original graph that has the property of being the smallest subgraph that is instance of the original graph. The number of removed triples by a lean subgraph strongly depends on the graph features, but a reasonable lower limit is two removed triples per blank node [**?**]. Nevertheless, some triples of a lean graph can still be derived from others, hence some semantic redundancy can still be present [**?**].

The rule-based (RB) compressor [**?**] uses mining techniques to detect objects that are commonly related to a particular predicate (intra-property patterns) and to group frequent predicate-object pairs (inter-property patterns). These patterns are then used as generative rules to remove triples that can be inferred from such patterns. RB is not so effective by itself, and only inter-property patterns enable significant amount of triples to be removed. [**?**] state that frequent patterns are not so expressive to capture semantic redundancy, and suggest that effectiveness can be improved using more expressive rules. In this case, Horn rules are mined from the dataset, and all triples matching their head part are removed. The resulting canonical subgraph is then compressed using RB. This Horn-rule based compressor outperforms RB effectiveness, but it introduces latencies in compression and decompression processes.

Hybrid compressors compact the RDF graph by first using a logical approach to remove redundant triples, and then performing physical compression at serialization level. Although these techniques could combine the best of logical and physical compression, their application has received relatively little attention until now.

HDT++ [**?**] revisits HDT to introduce some methods to detect syntactic and semantic redundancy. HDT++ brings out the inherent structure of RDF by detecting and grouping the different set of predicates (*predicate families*) used to describe subjects. The original RDF graph is encoded as a set of subgraphs, one per predicate family. The `rdf:type` values are attached to each predicate family, hence removing these triples from the subgraphs. Finally, HDT++ uses local IDs for the terms in each subgraph, thus reducing the number of required bits. As a result, HDT++ reduces the original HDT ID-graph space requirements up to 2 times for more structured datasets, and reports significant improvements even for highly semi-structured datasets.

The *graph-pattern based* (GPB) compressor [**?**] shares some common features with HDT++, also grouping subjects by predicate families, called *entity description patterns* (EDPs). Each EDP is encoded as a pair which includes the corresponding pattern and all instances matching it. This policy consolidates the simplest GPB encoding scheme (LV0), but patterns are then merged to obtain better patterns (LV1), and the merging process can be recursively performed (LV2). GPB results are not compared with other physical compressors, but they excel at logical level, where GPB-LV2 is able to remove more triples than RB.

In turn, **RDF Self-indexes** address efficient RDF retrieval on compressed space. These approaches do not just compress the ID-graph, but also provide indexing capabilities over it. HDT-FoQ [**?**] enhances HDT to also support predicate and object based queries, adding inverted indexes for predicate and object adjacency lists that, all together, provide excellent performance for resolving SPARQL triple patterns. k^2-triples [**?**] provides an alternative organization of the ID-graph, encoding a (binary) adjacency matrix of (subject, object) pairs per predicate. These matrices are very sparse and can be easily compressed using k^2-trees [**?**]. The k^2-triples approach improves HDT-FoQ compression ratios, and reports competitive numbers for all triple patterns binding the predicate, but results in a poor performance in those queries with unbounded

predicates. This is mitigated by adding two additional indexes to store the predicates related to each subject and object, but the pattern that only binds the predicate remains slow. RDFCSA [**?**] is the most recent RDF self-index, which encodes the ID-graph as a Compressed Suffix Array (CSA). RDFCSA also ensures efficient lookup performance, competing with k^2-triples at the cost of using more space.

7.1.2 Applications of RDF compressors

RDF compression has been widely adopted by the Semantic Web community as a standard technique to reduce storage and transmission costs when downloading RDF datasets. Most of the publishers in the Linked Open Data (LOD) cloud, make use of universal compression, given its simplicity, usability and widespread adoption. This is particularly true for projects publishing massive amounts of RDF data, such as DBpedia or Bio2RDF.

Nonetheless, RDF-specific compressors, and in particular RDF self-indexes, are receiving increased attention. Projects like LOD Laundromat [**?**] or Triple Pattern Fragments (TPF) [**?**] describe two interesting use cases exploiting compressed RDF. LOD Laundromat is an initiative to crawl and clean (removing syntax errors) RDF data from the LOD cloud. As a result, it exposes more than 650K cleansed datasets which are delivered in HDT format and can be queried using TPF interfaces. TPF focuses on alleviating the burden of endpoints by serving simple SPARQL triple patterns, paginating the results. This simplification allows servers to scale, while clients can always execute more complex SPARQL queries on top of TPFs by taking care of integrating and filtering the results. Given the simplicity of the required infrastructure at the server, TPF interfaces can make use of RDF self-indexes to serve low-cost operations, being HDT the most used backend in practice. The recently published *LOD-a-lot* dataset [**?**] combines the benefits from both projects to provide a practical example of efficient management of compressed Big Semantic Data. *LOD-a-lot* integrates all data from LOD Laundromat into a cross domain mashup of more than 28 billion triples and several terabytes of space (in NTriples). This dataset is then exposed as HDT and the corresponding TPF interface. The queryable self-indexed HDT of such large portion of the LOD cloud takes 524 GB, and can serve fast triple pattern resolution with an affordable memory footprint (in practice, 15.7 GB). These numbers are a strong evidence of how RDF compression contributes to make Big Semantic Data management feasible in most Linked Data servers (for online consumption) and clients (for downloading and offline consumption).

RDF compression and self-indexes have also been actively used in other Semantic Web areas such as i) SPARQL querying and recommender systems [**?** **?**], leveraging the retrieval operations supported by self-indexes to support more complex queries, ii) reasoning [**?**], optimizing the RDF dictionary and triples encoding to serve inference capabilities, iii) versioned RDF or RDF archives [**?**], where RDF compression is used to preserve (and query) the history of an RDF dataset, and iv) constrained and mobile devices [**?**] in order to maximize the exploitation of their storage/processing capabilities.

Finally, although it is not the focus of this review, RDF compression has also been highlighted by RDF stream processing systems [**?** **?**] (cf. [**?**] for a more complete survey).

In the following, we present our approach to enhance the aforementioned HDT technique to consider RDF datasets (quads). Then, we focus on the problem of versioned RDF or RDF archives.

7.2 HDTQ: Managing RDF Datasets in Compressed Space

As we motivated in the previous section, HDT [**?**] is a compact, self-indexed serialization of RDF that keeps big datasets compressed for RDF preservation and sharing and –at the same time– provides basic query functionality without prior decompression. HDT has been widely adopted by the community, (i) used as the main backend of Triple Pattern Fragments (TPF) [**?**] interface, which alleviates the traditional burden of LOD servers by moving part of the query processing onto clients, (ii) used as a storage backend for large-scale graph data [**?**], or (iii) as the store behind LOD Laundromat [**?**], serving a crawl of a very big subset of the LOD Cloud, to name but a few.

One of the main drawbacks of HDT so far is its inability to manage RDF datasets with multiple RDF graphs. HDT considers that all triples belong to the same graph, the *default graph*. However, triples in an RDF dataset can belong to different (named) graphs, hence the extension to the so-called RDF quadruples (subject, predicate, object, graph), or *quads*. The graph (also called context) is used to capture information such as trust, provenance, temporal information and other annotations [**?**]. Since RDF 1.1 [**?**] there exist standard RDF syntaxes (such as N-Quads or Trig) for representing RDF named graphs. SPARQL, with its *GRAPH* keyword, allows for querying and managing RDF named graphs, which most common triple stores have implemented. Interestingly, while RDF compression has been an active research topic for a while now, there is neither a compact RDF serialization nor a self-indexed RDF store for quads, to the best of our knowledge.

In the following, we present HDTQ[1], our current efforts on extending HDT to cope with quads and keep its compact and queryable features[2]. First, we briefly review RDF concepts (Section 7.2.1) and the most important components of HDT (Section 7.2.2). Then, we present the required extensions to add and query graph information in HDT (Section 7.2.3). Finally, we discuss the approach and our initial results (Section 7.2.7).

7.2.1 RDF preliminaries

An *RDF graph* G is a finite set of triples (subject, predicate, object) from $(I \cup B) \times I \times (I \cup B \cup L)$, where I, B, L denote IRIs, blank nodes and RDF literals, respectively. RDF graphs can be grouped and managed together, conforming an *RDF dataset*, that is, a collection of RDF graphs [**?**]. Borrowing terminology from [**?**], an *RDF dataset* is a set $DS = \{G, (g_1, G_1), \ldots, (g_n, G_n)\}$ consisting of a (non-named) default graph G and *named graphs* s.t. $g_i \in I$ are graph names. Figure 7.1 represents a dataset DS consisting of two named graphs (*aka* subgraphs), `graphWU` and `graphTU`, coming from different sources (e.g. from two universities). Note that terms[3] (i.e. subjects, predicates and objects) and triples can belong to different named graphs. For instance, the triple $(Vienna, locatedIn, Europe)$ is shared among the two subgraphs.

An *RDF quad* can be seen as an extension of a triple with the graph name (*aka* context). Formally, an RDF quad q from an RDF dataset DS, is a quadruple (subject, predicate, object, g_i) from $(I \cup B) \times I \times (I \cup B \cup L) \times I$. Note that the graph name g_i can be used in other triples or quads to provide further meta-knowledge, e.g. the subgraph provenance. We also

[1] The HDTQ approach and the supporting images presented herein have been adapted from [**?**].
[2] Further details on the current HDTQ prototype and its performance can be found in [**?**]
[3] All terms are IRIs whose prefix, `http://example.org/`, has been omitted for simplicity.

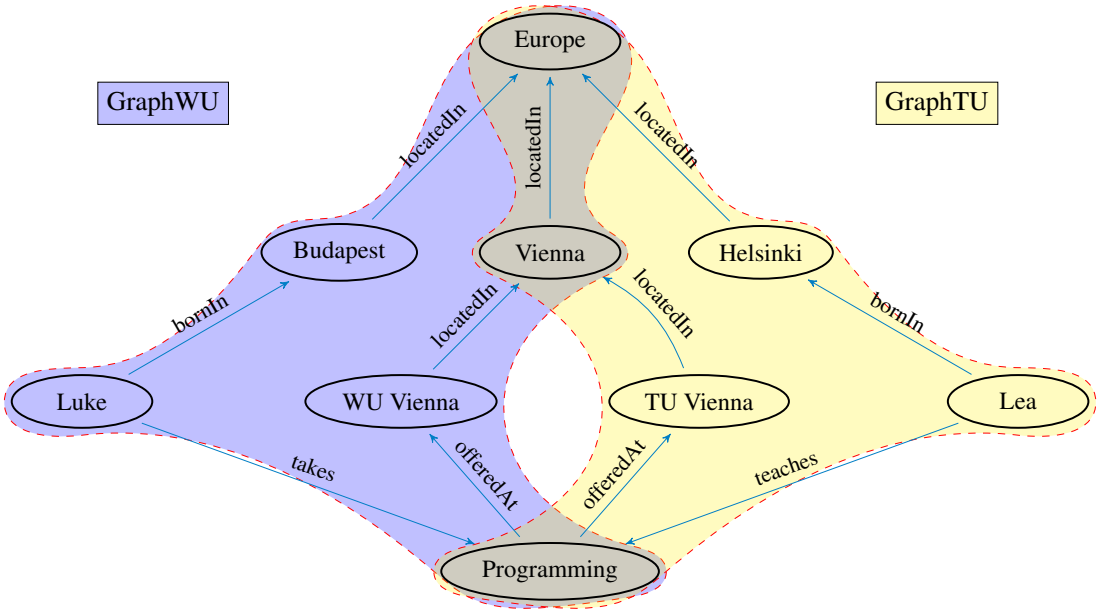


Figure 7.1: An RDF dataset DS consisting of two graphs, `GraphWU` and `GraphTU`.

note that quads and datasets (with named graphs) are in principle interchangeable in terms of expressiveness, i.e. one can be represented by the other.

RDF graphs and datasets are traditionally queried using the well-known SPARQL [**?**] query language. SPARQL is based on graph pattern matching, where the core component is the concept of a triple pattern, i.e. a triple where each subject, predicate and object are RDF terms or SPARQL variables. Formally, assuming a set V of variables, disjoint from the aforementioned I, B and L, a triple pattern tp is a tuple from $(I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$. In turn, SPARQL defines ways of specifying and querying by graph names (or the default graph), using the GRAPH keyword. To capture this, following the same convention as the triple pattern, we define a quad pattern qp as an extension of a triple pattern where also the graph name can be provided or may be a variable to be matched. That is, a quad pattern qp is a pair $tp \times (I \cup V)$ where the last component denotes the graph of the pattern (an IRI or variable).

7.2.2 HDT preliminaries

HDT [**?**] is a compressed serialization format for single RDF graphs, which also allows for triple pattern retrieval over the compressed data. HDT encodes an RDF graph G into three components: the *Header* holds metadata (provenance, signatures, etc.) and relevant information for parsing; the *Dictionary* provides a catalog of all RDF terms in G and maps each of them to a unique identifier; and the *Triple* component encodes the structure of the graph after the ID replacement. Figure 7.2 shows the HDT dictionary and triples for all RDF triples in Figure 7.1, i.e. disregarding the name graphs.

7.2.2.1 HDT Dictionary

The HDT dictionary of a graph G, denoted as D_G, organizes all terms in four sections, as shown in Figure 7.2 (a): SO includes terms occurring both as subject and object, mapped to the ID-

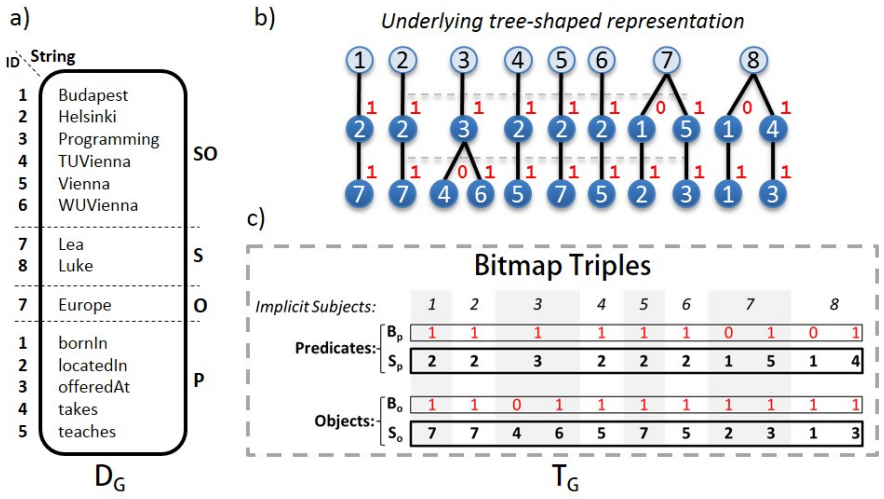


Figure 7.2: HDT Dictionary and Triples for a graph G (merging all triples of Fig. 7.1).

range `[1,|SO|]`. Sections S and O comprise terms that only appear as subjects or objects, respectively. In order to optimize the range of IDs, they are both mapped from `|SO|+1`, ranging up to `|SO|+|S|` and `|SO|+|O|`, respectively. Finally, section P stores all predicates, mapped to `[1,|P|]`. Note that (i) no ambiguity is possible once we know the role played by the term, and (ii) the HDT dictionary provides fast lookup conversions between IDs and terms.

7.2.2.2 HDT Triples

The Triples component of a graph G, denoted as T_G, encodes the *structure* of the RDF graph after ID replacement. Logically speaking, T organizes all triples into a forest of trees, one per different subject, as shown in Figure 7.2 (b): subjects are the roots of the trees, where the middle level comprises the ordered list of predicates associated with each subject, and the leaves list the objects related to each (subject, predicate) pair. This *underlying representation* is practically encoded with the so-called *BitmapTriples* approach [**?**], shown in Figure 7.2 (c). It comprises *two sequences*: `Sp` and `So`, concatenating all predicate IDs in the middle level and all object IDs in the leaves, respectively; and *two bitsequences*: `Bp` and `Bo`, which are aligned with `Sp` and `So` respectively, using a 1-bit to mark the end of each list. Bitsequences are then indexed to locate the 1-bits efficiently. These enhanced bitsequences are usually called *bitmaps*. HDT uses the Bitmap375 [**?**] technique that takes 37.5% extra space on top of the original bitsequence size.

7.2.2.3 Triple Pattern resolution with HDT

As shown, *BitmapTriples* is organized by subject, conforming a SPO index that can be used to efficiently resolve subject-bounded triple pattern queries [**?**] (i.e. triples where the subject is provided and the predicate and object may be a variable) as well as listing all triples. HDT-Focused on Querying (HDT-FoQ) [**?**] extends HDT with two additional indexes (PSO and OPS) to speed up the resolution of all triple patterns.

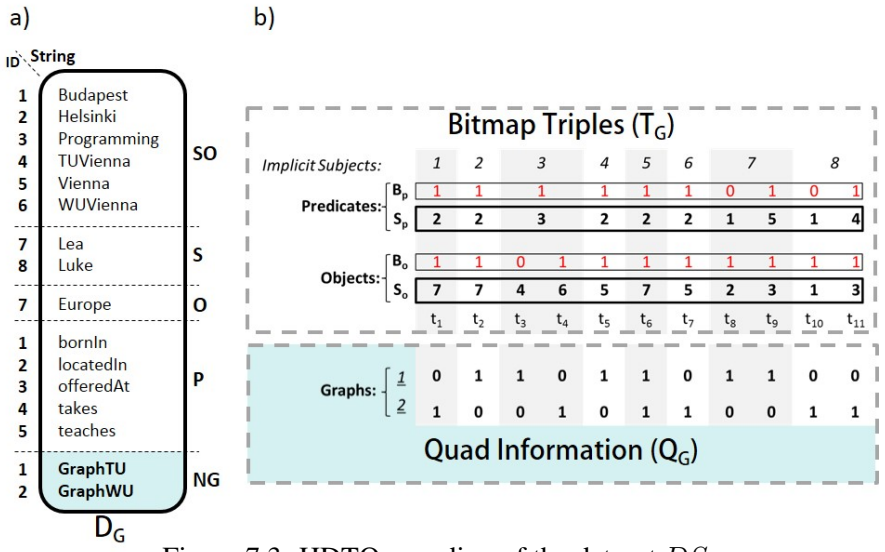


Figure 7.3: HDTQ encoding of the dataset DS.

7.2.3 HDTQ: Adding Graph Information to HDT

This section introduces HDTQ, an extension of HDT that involves managing RDF quads. We consider hereinafter that the original source is an RDF dataset as defined in Section 7.2.1, potentially consisting of several named graphs. For simplicity, we assume that graphs have no blank nodes in common, otherwise a re-labeling step would be possible as pre-processing.

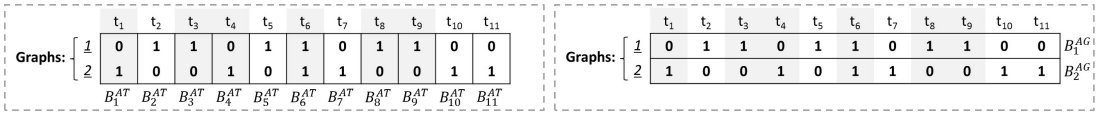
7.2.4 Extending the HDT Components

HDT was originally designed as a flexible format that can be easily extended, e.g. to include different dictionary and triples components or to support domain-specific applications. In the following, we detail HDTQ and the main design decisions to extend HDT to cope with quads. Figure 7.3 shows the final HDTQ encoding for the dataset DS in Figure 7.1. We omit the header information, as the HDTQ extension only adds implementation-specific metadata to parse the components.

7.2.4.1 Dictionary

In HDTQ, the previous four-section dictionary is extended by a fifth section to store all different graph names. The IDs of the graphs are then used to annotate the presence of the triples in each graph, further explained below. Figure 7.3 (a) shows the new HDTQ dictionary encoding for the dataset DS. Compared to the dictionary shown in Figure 7.2, i.e. the HDT conversion of all triples disregarding the named graphs, two comments are in order:

- The terms of all graphs are merged together in the traditional four dictionary sections, SO, S, O, P, as explained in Section 7.2.2. This decision can potentially increase the range of IDs w.r.t an individual mapping per graph, but it keeps the philosophy of storing terms once, when possible.
- The graph names are organized in an independent graph section, NG (named graphs), mapped from 1 to n_g, being n_g the number of graphs. Note that these terms might also play a different role in the dataset, and can then appear duplicated in SO, S, O or P.



(a) Annotated Triples (b) Annotated Graphs

Figure 7.4: Annotated Triples and Annotated Graphs variants for the RDF dataset DS.

However, no ambiguity is possible with the IDs once we know the role of the term we are searching for. In turn, the storage overhead of the potential duplication is limited as we assume that the number of graphs is much less than the number of unique subjects and objects. An optimization for extreme corner cases is devoted to future work.

7.2.4.2 Triples

HDTQ respects the original *BitmapTriples* encoding and extends it with an additional *Quad Information* (Q) component, shown in Figure 7.3 (b). Q represents a boolean matrix that includes (for every *triple - graph* combination) the information on whether a specific triple appears in a specific graph. Formally, having a triple-ID t_j (where $j \in \{1..m\}$, being m the total number of triples in the dataset DS), and a graph-ID k (where $k \in \{1..n_g\}$), the new Q component defines a boolean function $graph(t_j, k) = \{0, 1\}$, where 1 denotes that t_j appears in the graph k, or 0 otherwise.

7.2.5 Quad Indexes: Graph and Triples Annotators

HDTQ proposes two approaches to realize the Q matrix, namely Annotated Triples (HDT-AT) and Annotated Graphs (HDT-AG). They both rely on bitmaps, traditionally used in HDT (see Section 7.2.2).

7.2.5.1 Annotated Triples

Using the Annotated Triples approach, a bitmap is assigned to each triple, marking the graphs in which the corresponding triple is present. A dataset containing m triples in n different graphs has $\{B_1^{AT}, \cdots, B_m^{AT}\}$ bitmaps each of size n. Thus, if $B_j^{AT}[i] = 1$, it means that the triple t_j is present in the i^{th} graph, being $B_j^{AT}[i] = 0$ otherwise. This can be seen in Figure 7.4 (a), where 11 bitmaps (one per triple) are created, each of them of two positions, corresponding to the two graphs. In this example, the bitmap for the first triple holds $\{0, 1\}$, meaning that the first triple, (1,2,7), only appears in the second graph, which is `graphWU`.

Intuitively, Annotated Triples favors quad patterns having the graph component as a variable, like `SPO?`, as only a single bitmap needs to be browsed. On the other hand, if the graph is given, like in the pattern `???G`, all of the bitmaps need to be browsed.

7.2.5.2 Annotated Graphs

This approach is orthogonal to Annotated Triples: a bitmap is assigned to each graph, marking the triples present in the corresponding graph. Thus, a dataset containing m triples in n different graphs has $\{B_1^{AG}, \cdots, B_n^{AG}\}$ bitmaps each of size m. Thus, if $B_j^{AG}[i] = 1$, it means that the triple t_i is present in the j^{th} graph, being $B_j^{AG}[i] = 0$ otherwise. This can be seen in Figure 7.4 (b), including 2 bitmaps, each of size 11. For instance, the bitmap for the first graph, `graphTU`,

holds $\{0,1,1,0,1,1,0,1,1,0,0\}$ meaning that it consists of the triples $\{t_2, t_3, t_5, t_6, t_8, t_9\}$, which can be found in the respective positions in BitmapTriples.

Compared to Annotated Triples, Annotated Graphs favors quad patterns in which the graph is given, like `???G`, as only a single bitmap (the bitmap of the given graph G) needs to be browsed. On the other hand it penalizes patterns with graph variables, as all bitmaps need to be browsed to answer the query.

Finally note that, both in HDT-AT and HDT-AG, depending on the data distribution, the bitmaps can be long and sparse. However, in practice, HDT-AT and HDT-AG can be implemented with compressed bitmaps [**?**] to minimize the size of the bitsequences.

7.2.6 Search Operations

The resolution of quad patterns in HDTQ builds on top of two operations inherently provided by the *BitmapTriples* component (BT):

- **BT.getNextSolution(quad, startPosition)**. Given a $quad$ pattern, BT removes the last graph term and resolves the triple pattern, outputting a pair $(triple, posTriple)$ corresponding to the next triple solution and its position in BT. The search starts at the $startPosition$ provided, in BT. For instance, in our example in Figure 7.3, with a pattern $quad$ = `7???`, an operation `BT.getNextSolution(quad,8)` will jump the first 8 triples in BT,$\{t_1, \cdots, t_8\}$, hence the only solution is the pair $((7,5,3),9)$ or, in other words, t_9.

- **BT.getSolutionPositions(quad)**. This operation finds the set of triple positions where solution candidates appear. In subject-bounded queries, these positions are actually a consecutive range $\{t_x, .., t_y\}$ of BT. Otherwise, in queries such as `?P?G`, `??OG` and `?POG`, the positions are spread across BT. For instance, t_2 and t_5 are solutions for $quad$ =`?2?1`, but t_3 and t_4 do not match the pattern.

Note that we assume that the HDT-FoQ [**?**] indexes (PSO and OPS) are created, hence BT can provide these operations for all patterns. In the following, we detail the resolution depending on whether the graph term is given or it remains unbounded.

7.2.6.1 Quad Pattern Queries with Unbounded Graph

Algorithm 1 shows the resolution of quad patterns in which the graph term is not given, i.e. `????`, `S???`, `?P??`, `??O?`, `SP??`, `S?O?`, `?PO?` and `SPO?`. It is mainly based on iterating through the solutions of the traditional HDT and, for each triple solution, returning all the graphs associated to it. Thus, the algorithm starts by getting the first solution in BT (Line 2), using the aforementioned operation *getNextSolution*. While the end of BT is not reached (Line 3), we get the next graph associated with the current triple (Line 4), or $null$ if it does not appear in any further graph. This is provided by the operation *nextGraph* of Q, explained below. If there is a graph associated with the triple (Line 5), both are appended to the results (Line 6). Otherwise, we look for the next triple solution (Line 8).

The auxiliary *nextGraph* operation of Q returns the next graph in which a given triple appears, or $null$ if the end is reached. Algorithm 2 shows this operation for HDT-AT. First, the bitmap corresponding to the given triple is retrieved from Q (Line 1). Then, within this bitmap,

Algorithm 1: SEARCHQUADS - *quad patterns with unbounded graphs*

```
   Input: BitmapTriples BT, Quad Information Q, quad pattern q
   Output: The quads matching the given pattern
1  result ← (); graph ← 0
2  (triple, posTriple) ← BT.getNextSolution(q, 0)
3  while posTriple ≠ null do
4  |   graph ← Q.nextGraph(posTriple, graph + 1)
5  |   if graph ≠ null then
6  |   |   result.append(triple, graph)
7  |   else
8  |   |   (triple, posTriple) ← BT.getNextSolution(q, posTriple)
9  |   |   graph ← 0
10 |   end
11 end
12 return result
```

Algorithm 2: NEXTGRAPH - AT

```
   Input: Quad Information Q, int posTriple, int graph
   Output: The position of the next graph
1  bitmap ← Q[posTriple]
2  return bitmap.getNext1(graph)
```

Algorithm 3: NEXTGRAPH - AG

```
   Input: Quad Information Q, int posTriple, int graph
   Output: The position of the next graph
1  do
2  |   bitmap ← Q[graph]
3  |   if bitmap[posTriple] = 1 then
4  |   |   return graph
5  |   else
6  |   |   graph ← graph + 1
7  |   end
8  while graph ≤ Q.size()
9  return null
```

Algorithm 4: SEARCHQUADSG - *quad patterns with bounded graphs*

```
   Input: BitmapTriples BT, Quad Information Q, quad pattern q
   Output: The quads matching the given pattern
1  graph ← getGraph(q); result ← ()
2  sol[] ← BT.getSolutionPositions(q)
3  while !sol.isEmpty() do
4  |   posTripleCandidateBT ← sol.pop()
5  |   posTripleCandidateQT ← Q.nextTriple(posTripleCandidateBT, graph)
6  |   if posTripleCandidateBT = posTripleCandidateQT then
7  |   |   (triple, posTriple) ← BT.getNextSolution(q, posTripleCandidateBT − 1)
8  |   |   result.append(triple, graph)
9  |   else
10 |   |   sol.removeLessThan(posTripleCandidateQT)
11 |   end
12 end
13 return result
```

the location of the next 1 starting with the provided graph ID is retrieved (or null if the end is reached) and returned (Line 2). This latter is natively provided by the bitmap indexes.

Algorithm 3 shows the same process for HDT-AG. In this case, a bitmap is associated with each graph. Thus, we iterate on graphs and access one bitmap after the other (Line 1-7). The process ends when a 1-bit is found (Line 3), returning the graph (Line 4), or the maximum number of graphs is reached (Line 7), returning null (Line 8).

7.2.6.2 Quad Pattern Queries with Bounded Graph

Algorithm 4 resolves all quad patterns where the graph is provided. To do so, the graph ID is first retrieved from the quad pattern (Line 1). The aforementioned *getSolutionPositions* operation of BT finds the triple positions in which the solutions can appear (Line 2). Then, we iterate on this set of candidate positions until it is empty (Line 3). For each *posTripleCandidateBT* extracted from the set (Line 4), we check if this position is associated with the given graph (Line 5), using the operation *nextTriple* of the Q structure. This operation, omitted for the sake of concision as it is analogous to *nextGraph* (see Algorithms 2 and 3), starts from *posTripleCandidateBT* and returns the next triple position (*posTripleCandidateQT*) that is associated to the given graph. Thus, if this position is exactly the current candidate position (Line 6), the actual triple is obtained for that position (Line 7), and appended to the final resultset (Line 8). Otherwise, the candidate position was not a valid solution (it was not related to the graph), and we can remove, from the set of candidate solutions, all positions lesser than *posTripleCandidateQT* (Line 10), given that none of them are associated to the given graph.

7.2.7 HDTQ Discussion

In this section we have presented HDTQ, an extension of HDT, a compact and queryable serialization of RDF, to support RDF datasets including named graphs (quads). HDTQ considers a new dictionary to uniquely store all different named graphs, and a new Quad Information component to annotate the presence of the triples in each graph of the RDF dataset. Two realizations of this component are proposed, HDT-AG and HDT-AT, and space/performance tradeoffs are evaluated against different datasets and state-of-the-art stores.

Our initial results [**?**] show that HDTQ keeps the same HDT features, positioned itself as a highly compact serialization for RDF quads that remains competitive in quad pattern resolution. Our ongoing work focuses on inspecting an hybrid AT-AG strategy for the quad information and supporting full SPARQL 1.1. on top of HDTQ. To do so, we plan to use HDTQ as a compressed backend store within existing Big Semantic Data frameworks, supporting the scalable needs of our SPECIAL platform.

7.3 Strategies to Evaluate the Performance of RDF Archives

There is an emerging demand on efficiently archiving and (temporal) querying different versions of evolving semantic data. As novel archiving systems are starting to address this challenge, foundations/standards for benchmarking RDF archives are needed to evaluate its storage space efficiency and the performance of different retrieval operations. To this end, in this section[4] we first review the most common strategies to manage versioned RDF data, i.e. RDF archives (7.3.1). We then provide theoretical foundations on the design of data and queries to evaluate emerging RDF archiving systems (Section 7.3.2). Then, we instantiate these foundations along a concrete set of queries on the basis of a real-world evolving datasets (Section 7.3.3). These concepts are crystallized in BEAR, a benchmark for the evaluation of RDF archives. Further details on the current BEAR prototype and its evaluation results on current RDF store systems can be found in [**?**]. Section 7.3.4 discusses the approach and our initial results.

[4]The evaluation approach and the supporting images presented herein have been adapted from [**?**].

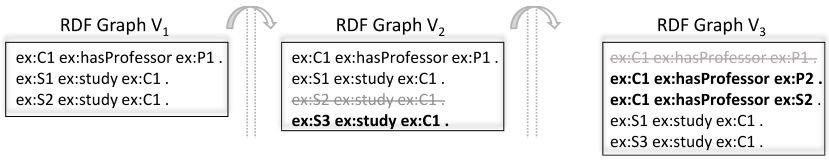


Figure 7.5: Example of RDF graph versions.

Type / Focus	Materialisation	Structured Queries	
		Single time	**Cross time**
Version	Version Materialisation *-get snapshot at time t_i*	Single-version structured queries *-lectures given by certain teacher at time t_i*	Cross-version structured queries *-subjects who have played the role of student and teacher of the same course*
Delta	Delta Materialisation *-get delta at time t_i*	Single-delta structured queries *-students leaving a course between two consecutive snapshots, i.e. between t_{i-1}, t_i*	Cross-delta structured queries *-largest variation of students in the history of the archive*

Table 7.1: Classification and examples of retrieval needs.

All in all, these foundations and results are meant to serve as a baseline of future developments of the SPECIAL platform, guiding the efficient management and querying of evolving RDF data.

7.3.1 Preliminaries on RDF Archives

We briefly summarise current archiving techniques for dynamic Linked Open Data. The use case is depicted in Figure 7.5, showing an evolving RDF graph with three versions V_1, V_2 and V_3 : the initial version V_1 models two students *ex:S1* and *ex:S2* of a course *ex:C1*, whose professor is *ex:P1*. In V_2, the *ex:S2* student disappeared in favour of a new student, *ex:S3*. Finally, the former professor *ex:P1* leaves the course to a new professor *ex:P2*, and the former student *ex:S2* reappears also as a professor.

7.3.1.1 Retrieval Functionality

Given the relative novelty of archiving and querying evolving semantic Web data, retrieval needs are neither fully described nor broadly implemented in practical implementations (described below). Table 7.1 shows a first classification [**?** **?**] that distinguishes six different types of retrieval needs, mainly regarding the query type (materialisation or structured queries) and the main focus (version/delta) of the query.

Version materialisation is a basic demand in which a full version is retrieved. In fact, this is the most common feature provided by revision control systems and other large scale archives, such as current Web archiving that mostly dereferences URLs across a given time point.[5]

Single-version structured queries are queries which are performed on a specific version. One could expect to exploit current state-of-the-art query resolution in RDF management systems, with the additional difficulty of maintaining and switching between all versions.

[5] See the Internet Archive effort, http://archive.org/web/.

Cross-version structured queries, also called time-traversal queries, must be satisfied across different versions, hence they introduce novel complexities for query optimization.

Delta materialisation retrieves the differences (deltas) between two or more given versions. This functionality is largely related to RDF authoring and other operations from revision control systems (merge, conflict resolution, etc.).

Single-delta structured queries and **cross-delta structured queries** are the counterparts of the aforementioned version-focused queries, but they must be satisfied on change instances of the dataset.

7.3.1.2 Archiving Policies and Retrieval Process

Main efforts addressing the challenge of RDF archiving fall in one of the following three storage strategies [**?**]: *independent copies (IC)*, *change-based (CB)* and *timestamp-based (TB)* approaches.

Independent Copies (IC) [? ?] is a basic policy that manages each version as a different, isolated dataset. It is, however, expected that IC faces scalability problems as static information is duplicated across the versions. Besides simple retrieval operations such as version materialisation, other operations require non-negligible processing efforts. A potential retrieval mediator should be placed on top of the versions, with the challenging tasks of (i) computing deltas at query time to satisfy delta-focused queries, (ii) loading/accessing the appropriate version/s and solve the structured queries, and (iii) performing both previous tasks for the case of structured queries dealing with deltas.

Change-based approach (CB) [? ? ?] partially addresses the previous scalability issue by computing and storing the differences (deltas) between versions. For the sake of simplicity, we focus here on low-level deltas (added or deleted triples).

A query mediator for this policy manages a materialised version and the subsequent deltas. Thus, CB requires additional computational costs for delta propagation which affects version-focused retrieving operations. Although an alternative policy could always keep a materialisation of the current version and store *reverse deltas* with respect to this latter [**?**], such deltas still need to be propagated to access previous versions.

Timestamp-based approach (TB) [? ? ?] can be seen as a particular case of time modelling in RDF, where each triple is annotated with its temporal validity. Likewise, in RDF archiving, each triple locally holds the timestamp of the version. In order to save space avoiding repetitions, compression techniques can be used to minimize the space overheads, e.g. using self-indexes, such as in v-RDFCSA [**?**], or delta compression in B+Trees [**?**].

Hybrid-based approaches (HB) [? ? ?] combine previous policies to inspect other space/performance tradeoffs. On the one hand, Dong-Hyuk et al. [**?**] and the TailR [**?**] archiving system adopt a hybrid IC/CB approach (referred to as HB$^{IC/CB}$ hereinafter), which can be complemented with a theoretical cost model [**?**] to decide when a fresh materialised version (IC) should be computed. These costs highly depend on the difficulties of constructing and reconstructing versions and deltas, which may depend on multiple and variable factors. On the other hand, R43ples [**?**] and other practical approaches [**? ? ?**] follow a TB/CB approach (referred to as HB$^{TB/CB}$ hereinafter) in which triples can be time-annotated only when they are added or deleted (if present). In these practical approaches, versions/deltas are often managed under named/virtual graphs, so that the retrieval mediator can rely on existing solutions

providing named/virtual graphs. Except for delta materialisation, all retrieval demands can be satisfied with some extra efforts given that (i) version materialisation requires to rebuild the delta similarly to CB, and (ii) structured queries may need to skip irrelevant triples [**?**].

Finally, [**?**] builds a partial order index keeping a hierarchical track of changes. This proposal, though, is a limited variation of delta computation and it is only tested with datasets having some thousand triples.

7.3.2 Evaluation of RDF Archives: Challenges and Guidelines

Previous considerations on RDF archiving policies and retrieval functionality set the basis of future directions on evaluating the efficiency of RDF archives. The design of a benchmark for RDF archives should meet three requirements:

- The benchmark should be **archiving-policy agnostic** both in the dataset design/generation and the selection of queries to do a fair comparison of different archiving policies.
- Early benchmarks should mainly focus on simpler queries against an increasing number of snapshots and introduce complex querying once the policies and systems are better understood.
- While new retrieval features must be incorporated to benchmark archives, one should consider lessons learnt in previous recommendations on benchmarking RDF data management systems [**?**].

Although many benchmarks are defined for RDF stores [**?** **?**] (see the Linked Data Benchmark Council project [**?**] for a general overview) and related areas such as relational databases (e.g. the well-known TPC[6] and recent TPC-H and TPC-C extensions to add temporal aspects to queries [**?**]) and graph databases [**?**], to the best of our knowledge, none of them are designed to address these particular considerations in RDF archiving. The preliminary EvoGen [**?**] data generator is one of the first attempts in this regards, based on extending the Lehigh University Benchmark (LUBM) [**?**] with evolution patterns. However, the work is focused on the creation of such synthetic evolving RDF data, and the functionality is restricted to the LUBM scenario. Nonetheless, most of the well-established benchmarks share important and general principles. We briefly recall here the four most important criteria when designing a domain-specific benchmark [**?**], which are also considered in our approach: Relevancy (to measure the performance when performing typical operations of the problem domain, i.e. archiving retrieval features), portability (easy to implement on different systems and architectures, i.e. RDF archiving policies), scalability (apply to small and large computer configurations, which should be extended in our case also to data size and number of versions), and simplicity (to evaluate a set of easy-to-understand and extensible retrieval features).

We next formalize the most important features to characterize data and queries to evaluate RDF archives. These will be instantiated in the next section to provide a concrete experimental testbed.

[6] http://www.tpc.org/.

7.3.2.1 Dataset Configuration

We first provide semantics for RDF archives and adapt the notion of *temporal RDF graphs* by Gutierrez et al. [**?**]. We make a syntatic-sugar modification to put the focus on version labels instead of temporal labels. Note, that time labels are a more general concept that could lead to time-specific operators (intersect, overlaps, etc.), which is complementary –and not mandatory– to RDF archives. Let $\mathcal{N}$ be a finite set of version labels in which a total order is defined.

Definition 1 (RDF Archive) *A version-annotated triple is an RDF triple* (s, p, o) *with a label* $i \in \mathcal{N}$ *representing the version in which this triple holds, denoted by the notation* $(s, p, o) : [i]$. *An RDF archive graph* $\mathcal{A}$ *is a set of version-annotated triples.*

Definition 2 (RDF Version) *An RDF version of an RDF archive* $\mathcal{A}$ *at snapshot* i *is the RDF graph* $\mathcal{A}(i) = \{(s, p, o)|(s, p, o) : [i] \in \mathcal{A}\}$. *We use the notation* V_i *to refer to the RDF version* $\mathcal{A}(i)$.

As basis for comparing different archiving policies, we introduce four main features to describe the dataset configuration, namely *data dynamicity*, *data static core*, *total version-oblivious triples* and *RDF vocabulary*.

Data dynamicity. This feature measures the number of changes between versions, considering these differences at the level of triples (low-level deltas [**?**]). Thus, it is mainly described by the *change ratio* and the *data growth* between versions. We note that there are various definitions of change and growth metrics conceivable, and we consider our framework extensible in this respect with other, additional metrics. At the moment, we consider the following definitions of *change ratio*, *insertion ratio*, *deletion ratio* and *data growth*:

Definition 3 (change ratio) *Given two versions* V_i *and* V_j, *with* $i < j$, *let* $\Delta^+_{i,j}$ *and* $\Delta^-_{i,j}$ *two sets respectively denoting the triples added and deleted between these versions, i.e.* $\Delta^+_{i,j} = V_j \setminus V_i$ *and* $\Delta^-_{i,j} = V_i \setminus V_j$. *The change ratio between two versions denoted by* $\delta_{i,j}$, *is defined by*

$$\delta_{i,j} = \frac{|\Delta^+_{i,j} \cup \Delta^-_{i,j}|}{|V_i \cup V_j|}.$$

That is, the change ratio between two versions should express the ratio of *all triples* in $V_i \cup V_j$ that have changed, i.e., that have been either inserted or deleted. In contrast, the insertion and deletion ratios provide further details on the proportion of inserted and add triple wrt. *the original version*:

Definition 4 (insertion ratio, deletion ratio) *The insertion* $\delta^+_{i,j} = \frac{|\Delta^+_{i,j}|}{|V_i|}$ *and deletion* $\delta^-_{i,j} = \frac{|\Delta^-_{i,j}|}{|V_i|}$ *denote the ratio of "new" or "removed" triples with respect to the original version.*

Finally, the data growth rate compares the number of triples between two versions:

Definition 5 (data growth) *Given two versions* V_i *and* V_j, *having* $|V_i|$ *and* $|V_j|$ *different triples respectively, the data growth of* V_j *with respect to* V_i, *denoted by,* $growth(V_i, V_j)$, *is defined by*

$$growth(V_i, V_j) = \frac{|V_j|}{|V_i|}$$

In archiving evaluations, one should provide details on three related aspects, $\delta_{i,j}$, $\delta^{+}_{i,j}$ and $\delta^{-}_{i,j}$, as well as the complementary version data growth, for all pairs of consecutive versions. Additionally, one important aspect of measurement could be the rate of changed triples *accumulated* overall across *non-consecutive* versions. That is, as opposed to the (absolute) metrics defined so far, which compare between the original and the final version only, here we want to also be able to take all intermediate changes into account. To this end, we can also define an *accumulated change rate* $\delta^{*}_{i,j}$ between two (not necessarily consecutive) versions as follows:

Definition 6 *The accumulated change ratio $\delta^{*}_{i,j}$ between two versions V_i, V_j with $j = i + h$, with $h > 0$, is defined as*

$$\delta^{*}_{i,j} = \frac{\Sigma^{j}_{k=i}\delta_{k,k+1}}{h}$$

The rationale here is that $\delta^{*}_{i,j}$ should be 1 iff all triples changed *in each version* (even if eventually the changes are reverted and $V_i = V_j$), 0 if $V_i = V_k$ for each $i \leq k \leq j$, and non-0 otherwise, i.e. measuring the accumulation of changes over time.

Note that most archiving policies are affected by the frequency and also the type of changes, that is both absolute change metrics and accumulated change rates play a role. For instance, IC policy duplicates the static information between two consecutive versions V_i and V_j, whereas the size of V_j increases with the added information ($\delta^{+}_{i,j}$) and decreases with the number of deletions ($\delta^{-}_{i,j}$), given that the latter are not represented. In contrast, CB and TB approaches store all changes, hence they are affected by the general dynamicity ($\delta_{i,j}$).

Data static core. It measures the triples that are available in all versions:

Definition 7 (Static core) *For an RDF archive $\mathcal{A}$, the static core $\mathcal{C}_{\mathcal{A}} = \{(s,p,o)|\forall i \in \mathcal{N}, (s,p,o) : [i] \in \mathcal{A}\}$.*

This feature is particularly important for those archiving policies that, whether implicitly or explicitly, represent such static core. In a change-based approach, the static core is not represented explicitly, but it inherently conforms the triples that are not duplicated in the versions, which is an advantage against other policies such as IC. It is worth mentioning that the static core can be easily computed taking the first version and applying all the subsequent deletions.

Total version-oblivious triples. This computes the total number of different triples in an RDF archive independently of the timestamp. Formally speaking:

Definition 8 (Version-oblivious triples) *For an RDF archive $\mathcal{A}$, the version-oblivious triples $\mathcal{O}_{\mathcal{A}} = \{(s,p,o)|\exists i \in \mathcal{N}, (s,p,o) : [i] \in \mathcal{A}\}$.*

This feature serves two main purposes. First, it points to the diverse set of triples managed by the archive. Note that an archive could be composed of few triples that are frequently added or deleted. This could be the case of data denoting the presence or absence of certain information, e.g. a particular case of RDF streaming. Then, the total version-oblivious triples are in fact the set of triples annotated by temporal RDF [**?**] and other representations based on annotation (e.g. AnQL [**?**]), where different annotations for the same triple are merged in an annotation set (often resulting in an interval or a set of intervals).

RDF vocabulary. In general, we cover under this feature the main aspects regarding the different subjects (S_A), predicates (P_A), and objects (O_A) in the RDF archive $\mathcal{A}$. Namely, we put the focus on the RDF *vocabulary per version and delta* and the *vocabulary set dynamicity*, defined as follows:

Definition 9 (RDF vocabulary per version) *For an RDF archive $\mathcal{A}$, the vocabulary per version is the set of subjects (S_{V_i}), predicates (P_{V_i}) and objects (O_{V_i}) for each version V_i in the RDF archive $\mathcal{A}$.*

Definition 10 (RDF vocabulary per delta) *For an RDF archive $\mathcal{A}$, the vocabulary per delta is the set of subjects ($S_{\Delta^+_{i,j}}$ and $S_{\Delta^-_{i,j}}$), predicates ($P_{\Delta^+_{i,j}}$ and $P_{\Delta^-_{i,j}}$) and objects ($O_{\Delta^+_{i,j}}$ and $O_{\Delta^-_{i,j}}$) for all consecutive (i.e., $j = i + 1$) V_i and V_j in $\mathcal{A}$.*

Definition 11 (RDF vocabulary set dynamicity) *The dynamicity of a vocabulary set K, being K one of $\{S, P, O\}$, over two versions V_i and V_j, with $i < j$, denoted by $vdyn(K, V_i, V_j)$ is defined by*

$$vdyn(K, V_i, V_j) = \frac{|(K_{V_i} \setminus K_{V_j}) \cup (K_{V_j} \setminus K_{V_i})|}{|K_{V_i} \cup K_{V_j}|}.$$

The vocabulary set dynamicity for insertions and deletions is defined by $vdyn^+(K, V_i, V_j) = \frac{|K_{V_j} \setminus K_{V_i}|}{|K_{V_i} \cup K_{V_j}|}$ and $vdyn^-(K, V_i, V_j) = \frac{|K_{V_i} \setminus K_{V_j}|}{|K_{V_i} \cup K_{V_j}|}$ respectively.

The evolution (cardinality and dynamicity) of the vocabulary is specially relevant in RDF archiving, since traditional RDF management systems use dictionaries (mappings between terms and integer IDs) to efficiently manage RDF graphs. Finally, whereas additional graph-based features (e.g. in-out-degree, clustering coefficient, presence of cliques, etc.) are interesting and complementary to our work, our proposed properties are feasible (efficient to compute and analyse) and grounded in state-of-the-art of archiving policies.

7.3.2.2 Design of Benchmark Queries

There is neither a standard language to query RDF archives, nor an agreed way for the more general problem of querying temporal graphs. Nonetheless, most of the proposals (such as T-SPARQL [**?**], stSPARQL [**?**], SPARQL-ST [**?**] and the most recent SPARQ-LTL [**?**]) are based on SPARQL modifications.

In this scenario, previous experiences on benchmarking SPARQL resolution in RDF stores show that benchmark queries should report on the query type, result size, graph pattern shape, and query atom selectivity [**?**]. Conversely, for RDF archiving, one should put the focus on data dynamicity, without forgetting the strong impact played by query selectivity in most RDF triple stores and query planning strategies [**?**].

Let us briefly recall and adapt definitions of query cardinality and selectivity [**?** **?**] to RDF archives. Given a SPARQL query Q, where we restrict to SPARQL Basic Graph Patterns (*BGP*s[7]) hereafter, the evaluation of Q over a general RDF graph $\mathcal{G}$ results in a bag of solution mappings $[[Q]]_G$, where Ω denotes its underlying set. The function $card_{[[Q]]_G}$ maps each mapping $\mu \in \Omega$ to its cardinality in $[[Q]]_G$. Then, for comparison purposes, we introduce three main features, namely *archive-driven result cardinality and selectivity*, *version-driven result cardinality and selectivity*, and *version-driven result dynamicity*, defined as follows.

Definition 12 (Archive-driven result cardinality) *The archive-driven result cardinality of Q over the RDF archive $\mathcal{A}$, is defined by*

[7] Sets of triple patterns, potentially including a FILTER condition, in which all triple patterns must match.

$$CARD(Q, \mathcal{A}) = \sum_{\mu \in \Omega} card_{[[Q]]_A}(\mu).$$

In turn, the archive-driven query selectivity accounts how selective is the query, and it is defined by $SEL(Q, \mathcal{A}) = |\Omega|/|\mathcal{A}|$.

Definition 13 (Version-driven result cardinality) *The version-driven result cardinality of* Q *over a version* V_i*, is defined by*

$$CARD(Q, V_i) = \sum_{\mu \in \Omega_i} card_{[[Q]]_{V_i}}(\mu),$$

where Ω_i *denotes the underlying set of the bag* $[[Q]]_{V_i}$*. Then, the version-driven query selectivity is defined by* $SEL(Q, V_i) = |\Omega_i|/|V_i|$.

Definition 14 (Version-driven result dynamicity) *The version-driven result dynamicity of the query* Q *over two versions* V_i *and* V_j*, with* $i < j$*, denoted by* $dyn(Q, V_i, V_j)$ *is defined by*

$$dyn(Q, V_i, V_j) = \frac{|(\Omega_i \setminus \Omega_j) \cup (\Omega_j \setminus \Omega_i)|}{|\Omega_i \cup \Omega_j|}.$$

Likewise, we define the version-driven result insertion $dyn^+(Q, V_i, V_j) = \frac{|\Omega_j \setminus \Omega_i|}{|\Omega_i \cup \Omega_j|}$ *and deletion* $dyn^-(Q, V_i, V_j) = \frac{|\Omega_i \setminus \Omega_j|}{|\Omega_i \cup \Omega_j|}$ *dynamicity.*

The archive-driven result cardinality is reported as a feature directly inherited from traditional SPARQL querying, as it disregards the versions and evaluates the query over the set of triples present in the RDF archive. Although this feature could be only of peripheral interest, the knowledge of this feature can help in the interpretation of version-agnostic retrieval purposes (e.g. ASK queries).

As stated, result cardinality and query selectivity are main influencing factors for the query performance, and should be considered in the benchmark design and also known for the result analysis. In RDF archiving, both processes require particular care, given that the results of a query can highly vary in different versions. Knowing the version-driven result cardinality and selectivity helps to interpret the behaviour and performance of a query across the archive. For instance, selecting only queries with the same cardinality and selectivity across all version should guarantee that the index performance is always the same and as such, potential retrieval time differences can be attributed to the archiving policy. Finally, the version-driven result dynamicity does not just focus on the number of results, but how these are distributed in the archive *timeline*.

In the following, we introduce five foundational query atoms to cover the broad spectrum of emerging retrieval demands in RDF archiving. Rather than providing a complete catalog, our main aim is to reflect basic retrieval features on RDF archives, which can be combined to serve more complex queries. We elaborate these atoms on the basis of related literature, with special attention to the needs of the well-established *Memento Framework* [**?**], which can provide access to prior states of RDF resources using datetime negotiation in HTTP.

Version materialisation, $Mat(Q, V_i)$**:** it provides the SPARQL query resolution of the query Q at the given version V_i. Formally, $Mat(Q, V_i) = [[Q]]_{V_i}$.

Within the Memento Framework, this operation is needed to provide mementos (URI-M) that encapsulate a prior state of the original resource (URI-R).

versions	$\|V_0\|$	$\|V_{57}\|$	$\overline{growth}$	$\overline{\delta}$	$\overline{\delta^-}$	$\overline{\delta^+}$	$\mathcal{C}_\mathcal{A}$	$\mathcal{O}_\mathcal{A}$
58	30m	66m	101%	31%	32%	27%	3.5m	376m

Table 7.2: BEAR-A Dataset configuration

Delta materialisation, $Diff(Q, V_i, V_j)$: it provides the different results of the query Q between the given V_i and V_j versions. Formally, let us consider that the output is a pair of mapping sets, corresponding to the results that are present in V_i but not in V_j, that is $(\Omega_i \setminus \Omega_j)$, and viceversa, i.e. $(\Omega_j \setminus \Omega_i)$.

A particular case of delta materialisation is to retrieve all the differences between V_i and V_j, which corresponds to the aforementioned $\Delta^+_{i,j}$ and $\Delta^-_{i,j}$.

Version Query, $Ver(Q)$: it provides the results of the query Q *annotated* with the version label in which each of them holds. In other words, it facilitates the $[[Q]]_{V_i}$ solution for those V_i that contribute with results.

Cross-version join, $Join(Q_1, V_i, Q_2, V_j)$: it serves the join between the results of Q_1 in V_i, and Q_2 in V_j. Intuitively, it is similar to $Mat(Q_1, V_i) \bowtie Mat(Q_2, V_j)$.

Change materialisation, $Change(Q)$: it provides those consecutive versions in which the given query Q produces different results. Formally, $Change(Q)$ reports the labels i, j (referring to the versions V_i and V_j) $\Leftrightarrow Diff(Q, V_i, V_j) \neq \emptyset, j = i + 1$.

Within the Memento Framework, change materialisation is needed to provide timemap information to compile the list of all mementos (URI-T) for the original resource, i.e. the basis of datetime negotiation handled by the timegate (URI-G).

These query features can be instantiated in domain-specific query languages (e.g. DIACHRON QL [**?**]) and existing temporal extensions of SPARQL (e.g. T-SPARQL [**?**], stSPARQL [**?**], SPARQL-ST [**?**], and SPARQ-LTL [**?**]). An instantiation of this queries in AnQL is provided in [**?**].

7.3.3 BEAR: A Test Suite for RDF Archiving

This section presents BEAR, a prototypical (and extensible) test suite to demonstrate the new capabilities in benchmarking the efficiency of RDF archives using our foundations, and to highlight current challenges and potential improvements in RDF archiving. BEAR comprises three main datasets, namely BEAR-A, BEAR-B, and BEAR-C, each having different characteristics.

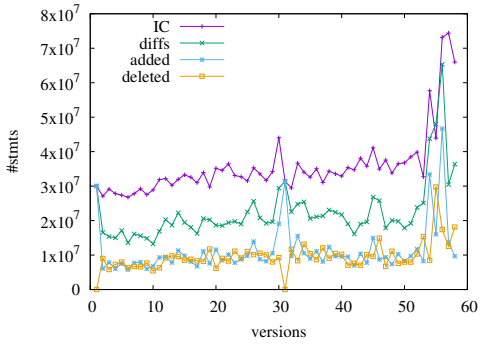
The complete test suite (data corpus, queries, archiving system source codes, evaluation and additional results) is available at the BEAR repository[8].

7.3.3.1 BEAR-A: Dynamic Linked Data

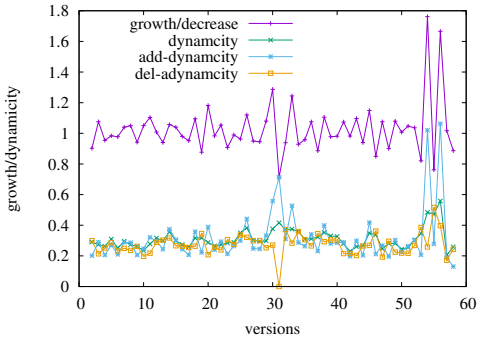
The first benchmark we consider provides a realistic scenario on queries about the evolution of Linked Data in practice.

Description. We build our RDF archive on the data hosted by the Dynamic Linked Data Observatory[9], monitoring more than 650 different domains across time and serving weekly crawls

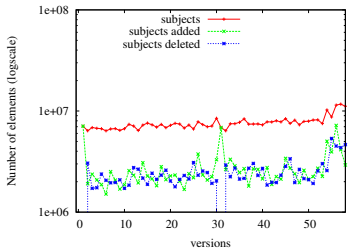
[8] https://aic.ai.wu.ac.at/qadlod/bear.
[9] http://swse.deri.org/dyldo/.



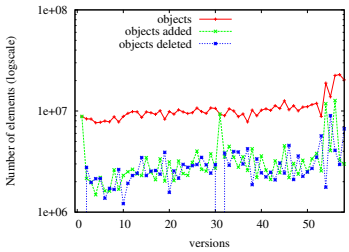
(a) Number of statements



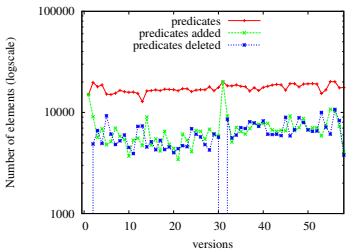
(b) Relative growth and dynamicity



(c) Subject vocabulary



(d) Object vocabulary



(e) Predicate vocabulary

Figure 7.6: Dataset description.

of these domains. BEAR data are composed of the first 58 weekly snapshots, i.e. 58 versions, from this corpus. Each original week consists of triples annotated with their RDF document provenance, in N-Quads format. We focus on archiving of a single RDF graph, so that we remove the context information and manage the resultant set of triples, disregarding duplicates. The extension to multiple graph archiving can be seen as future work. In addition, we replaced Blank Nodes with Skolem IRIs[10] (with a prefix *http://example.org/bnode/*) in order to simplify the computation of diffs.

We report the data configuration features (cf. Section 7.3.2) that are relevant for our purposes. Table 7.2 lists basic statistics of our dataset, further detailed in Figure 7.6, which shows the figures per version and the vocabulary evolution. Data growth behaviour (dynamicity) can be identified at a glance: although the number of statement in the last version ($|V_{57}|$) is more than double the initial size ($|V_0|$), the mean version data growth ($\overline{growth}$) between versions is almost marginal (101%).

A closer look to Figure 7.6 (a) allows to identify that the latest versions are highly contribut-

[10]https://www.w3.org/TR/rdf11-concepts/#section-skolemization

ing to this increase. Similarly, the version change ratios[11] in Table 7.2 ($\overline{\delta}$, $\overline{\delta^-}$ and $\overline{\delta^+}$) point to the concrete adds and delete operations. Thus, one can see that a mean of 31% of the data change between two versions and that each new version deletes a mean of 27% of the previous triples, and adds 32%. Nonetheless, Figure 7.6 (b) points to particular corner cases (in spite of a common stability), such as V_{31} in which no deletes are present, as well as it highlights the noticeable dynamicity in the last versions.

Conversely, the number of version-oblivious triples ($\mathcal{O}_\mathcal{A}$), $376m$, points to a relatively low number of different triples in all the history if we compare this against the number of versions and the size of each version. This fact is in line with the $\overline{\delta}$ dynamicity values, stating that a mean of 31% of the data change between two versions. The same reasoning applies for the remarkably small static core ($\mathcal{C}_\mathcal{A}$), $3.5m$.

Finally, Figures 7.6 (c-e) show the RDF vocabulary (different subjects, predicates and objects) per version and per delta (adds and deletes). As can be seen, the number of different subjects and predicates remains stable except for the noticeable increase in the latests versions, as already identified in the number of statements per versions. However, the number of added and deleted subjects and objects fluctuates greatly and remain high (one order of magnitude of the total number of elements, except for the aforementioned V_{31} in which no deletes are present). In turn, the number or predicates are proportionally smaller, but it presents a similar behaviour.

Test Queries. BEAR-A provides triple pattern queries Q to test each of the five atomic operations defined in our foundations (Section 7.3.2). Note that, although such queries do not cover the full spectrum of SPARQL queries, triple patterns (i) constitute the basis for more complex queries, (ii) are the main operation served by lightweight clients such as the Linked Data Fragments [**?**] proposal, and (iii) they are the required operation to retrieve prior states of a resource in the Memento Framework. For simplicity, we present here atomic lookup queries Q in the form (S??), (?P?), and (??O), which are then extended to the rest of triple patterns (SP?), (S?O), (?PO), and (SPO)[12]. For instance, Listing 7.1 shows an example of a materialization of a basic predicate lookup query in version 3.

Listing 7.1: Materialization of a (?P?) triple pattern in version 3.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT * WHERE {
?s dc:language ?p : 3 }
}
```

As for the generation of queries, we randomly select such triple patterns from the 58 versions of the Dynamic Linked Data Observatory. In order to provide comparable results, we consider entirely dynamic queries, meaning that the results always differ between consecutive versions. In other words, for each of our selected queries Q, and all the versions V_i and V_j ($i < j$), we assure that $dyn(Q, V_i, V_j) > 0$. To do so, we first extract subjects, predicates and objects that appear in all $\Delta_{i,j}$.

Then, we follow the foundations and try to minimise the influence of the result cardinality on the query performance. For this purpose, we sample queries which return, for all versions, result sets of similar size, that is, $CARD(Q, V_i) \approx CARD(Q, V_j)$ for all queries and versions. We

[11] Note that $\overline{\delta} = \delta^*_{1,n}$, so we use them interchangeably.

[12] The triple pattern (???) retrieves all the information, so no sampling technique is required.

QUERY SET	lookup position	$\overline{CARD}$	$\overline{dyn}$	#queries
Q_L^S-ϵ=0.2	subject	6.7	0.46	50
Q_L^P-ϵ=0.6	predicate	178.66	0.09	6
Q_L^O-ϵ=0.1	object	2.18	0.92	50
Q_H^S-ϵ=0.1	subject	55.22	0.78	50
Q_H^P-ϵ=0.6	predicate	845.3	0.12	10
Q_H^O-ϵ=0.6	object	55.62	0.64	50

Table 7.3: Overview of BEAR-A lookup queries

granularity	versions	$\lvert V_0 \rvert$	$\lvert V_{last} \rvert$	$\overline{growth}$	$\overline{\delta}$	$\overline{\delta^-}$	$\overline{\delta^+}$	$\mathcal{C}_\mathcal{A}$	$\mathcal{O}_\mathcal{A}$
instant	21,046	33,502	43,907	100.001%	0.011%	0.007%	0.004%	32,094	234,588
hour	1,299	33,502	43,907	100.090%	0.304%	0.197%	0.107%	32,303	178,618
day	89	33,502	43,907	100.744%	1.778%	1.252%	0.526%	32,448	83,134

Table 7.4: BEAR-B Dataset configuration

introduce here the notation of a ϵ-stable query, that is, a query for which the min and max result cardinality over all versions do not vary by more than a factor of $1 \pm \epsilon$ from the mean cardinality, i.e., $\max_{\forall i \in \mathcal{N}} CARD(Q, V_i) \leq (1+\epsilon) \cdot \frac{\sum_{\forall i \in \mathcal{N}} CARD(Q,V_i)}{|\mathcal{N}|}$ and $\min_{\forall i \in \mathcal{N}} CARD(Q, V_i) \geq (1-\epsilon) \cdot \frac{\sum_{\forall i \in \mathcal{N}} CARD(Q,V_i)}{|\mathcal{N}|}$.

Thus, the previous selected dynamic queries are effectively run over each version in order to collect the result cardinality. Next, we split subject, objects and predicate queries producing low (Q_L^S, Q_L^P, Q_L^O) and high (Q_H^S, Q_H^P, Q_H^O) cardinalities. Finally, we filter these sets to sample at most 50 subject, predicate and object queries which can be considered ϵ-stable for a given ϵ. Table 7.3 shows the selected query sets with their epsilon value, mean cardinality and mean dynamicity. Although, in general, one could expect to have queries with a low ϵ (i.e. cardinalities are equivalent between versions), we test higher ϵ values in objects and predicates in order to have queries with higher cardinalities. Even with this relaxed restriction, the number of predicate queries that fulfil the requirements is just 6 and 10 for low and high cardinalities respectively.

7.3.3.2 BEAR-B: DBpedia Live

Our next benchmark, rather than looking at arbitrary Linked Data, is focused on the evolution of DBpedia, which directly reflect Wikipedia edits, where we can expect quite different change/evolution characteristics.

Dataset Description. The BEAR-B dataset has been compiled from DBpedia Live changesets[13] over the course of three months (August to October 2015). DBpedia Live [**?**] records all updates to Wikipedia articles and hence re-extracts and instantly updates the respective DBpedia Live resource descriptions. The BEAR-B contains the resource descriptions of the 100 most volatile resources along with their updates. The most volatile resource (`dbr:Deaths_in_-`

[13] http://live.dbpedia.org/changesets/

2015) changes 1,305 times, the least volatile resource contained in the dataset (Once_Upon_-a_Time_(season_5)) changes 263 times.

As dataset updates in DBpedia Live occur instantly, for every single update the dataset shifts to a new version. In practice, one would possibly aggregate such updates in order to have less dataset modifications. Therefore, we also aggregated these updates on an hourly and daily level. Hence, we get three time granularities from the changesets for the very same dataset: *instant* (21,046 versions), *hour* (1,299 versions), and *day* (89 versions).

Detailed characteristics of the dataset granularities are listed in Table 7.4. The dataset grows almost continuously from 33,502 triples to 43,907 triples. Since the time granularities differ in the number of intermediate versions, they show different change characteristics: a longer update cycle also results in more extensive updates between versions, the average version change ratio increases from very small portions of 0.011% for instant updates to 1.8% at the daily level. It can also be seen that the aggregation of updates leads to omission of changes: whereas the instant updates handle 234,588 version-oblivious triples, the daily aggregates only have 83,134 (hourly: 178,618), i. e. a reasonable number of triples exists only for a short period of time before they get deleted again. Likewise, from the different sizes of the static core, we see that triples which have been deleted at some point are re-inserted after a short period of time (in the case of DBpedia Live this may happen when changes made to a Wikipedia article are reverted shortly after).

Test Queries. BEAR-B allows one to use the same sampling methodology as BEAR-A to retrieve dynamic queries. Nonetheless, we exploit the real-world usage of DBpedia to provide realistic queries. Thus, we extract the 200 most frequent triple patterns from the DBpedia query set of *Linked SPARQL Queries dataset* (LSQ) [**?**] and filter those that produce results in our BEAR-B corpus. We then obtain a batch of 62 lookup queries, mixing (?P?) and (?PO) queries. The full batch has a $\overline{CARD}$=80 in BEAR-B-day and BEAR-B-hour, and $\overline{CARD}$=54 in BEAR-B-instant. Finally, we build 20 join cases using the selected triple patterns, such as the join in Listing 7.2. Further statistics on each query are available at the BEAR repository.

Listing 7.2: Example of a join query in BEAR-B

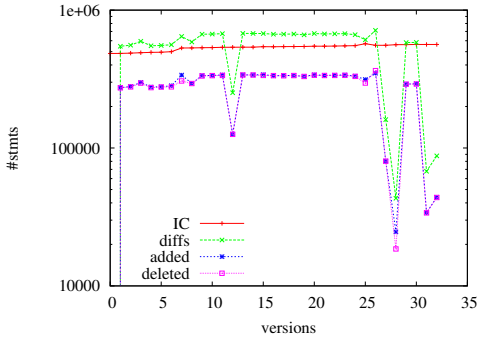
```
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
{
  ?film dbo:director ?director .
  ?director dbp:name ?name .
}
```

7.3.3.3 BEAR-C: Open Data portals

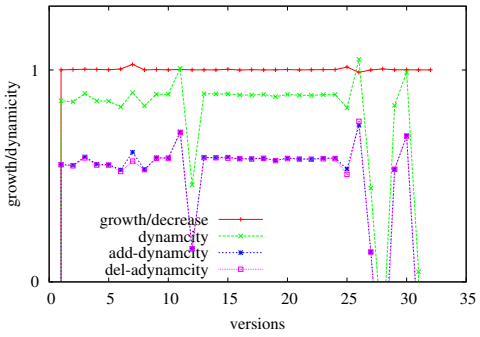
The third dataset is taken from the Open Data Portal Watch project, a framework that monitors over 260 Open Data portals in a weekly basis and performs a quality assessment. The framework harvests the dataset descriptions in the portals and converts them to their DCAT representation. We refer to [**?**] for more details.

granularity	versions	$\|V_0\|$	$\|V_{last}\|$	$\overline{growth}$	$\overline{\delta}$	$\overline{\delta^-}$	$\overline{\delta^+}$	$\mathcal{C}_\mathcal{A}$	$\mathcal{O}_\mathcal{A}$
portal	32	485,179	563,738	100.478%	67.617%	33.671%	33.946%	178,484	9,403,540

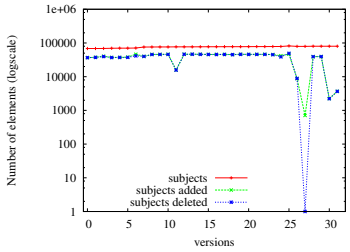
Table 7.5: BEAR-C Dataset configuration



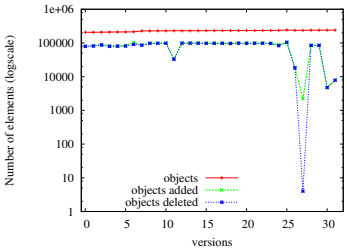
(a) Number of statements



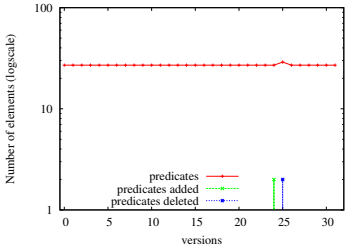
(b) Relative growth and dynamicity



(c) Subject vocabulary



(d) Object vocabulary



(e) Predicate vocabulary

Figure 7.7: Dataset description.

Dataset Description. For BEAR-C, we decided to take the datasets descriptions of the European Open Data portal[14] for 32 weeks, or 32 snapshots respectively. Table 7.5 and Figure 7.7 show the main characteristics of the dataset. Each snapshot consists of roughly 500m triples with a very limited growth as most of the updates are modifications on the metadata, i.e. adds and deletes report similar figures as shown in Figure 7.7 (a-b). Note also that this dynamicity is also reflected in the subject and object vocabulary (Figures 7.7 (c-d)), whereas the metadata is always described with the same predicate vocabulary (Figure 7.7 (e)), in spite of a minor modification in version 24 and 25. Note that, as in BEAR-A, we also replaced Blank Nodes with Skolem IRIs.

[14] http://data.europa.eu/euodp/en/data/

Test Queries. Selected triple patterns in BEAR-A cover queries whose dynamicity is well-defined, hence it allows for a fine-grained evaluation of different archiving strategies (and particular systems). In turn, BEAR-B adopts a realistic approach and gather real-word queries from DBpedia. Thus, we provide complex queries for BEAR-C that, although they cannot be resolved in current archiving strategies in a straightforward and optimized way, they could help to foster the development and benchmarking of novel strategies and query resolution optimizations in archiving scenarios.

With the help of Open Data experts, we created 10 queries that retrieve different information from datasets and files (referred to as *distributions*, where each dataset refers to one or more distributions) in the European Open Data portal. For instance, `Q1` in Listing 7.3 retrieves all the datasets and their file URLs. See the BEAR repository for the full list of queries[15].

Listing 7.3: BEAR-C Q1: Retrieve portals and their files.

```
PREFIX dcat: <http://www.w3.org/ns/dcat#>
{
  ?dataset rdf:type dcat:Dataset .
  ?dataset dcat:distribution ?distribution .
  ?distribution dcat:accessURL ?URL .
}
```

7.3.4 Discussion

RDF archiving is still in an early stage of research. Novel solutions have to face the additional challenge of comparing the performance against other archiving policies or storage schemes, as there is not a standard way of defining neither a specific data corpus for RDF archiving nor relevant retrieval functionalities.

In this section we have presented foundations to guide future evaluation of RDF archives, which can guide the implementation of future versions of the SPECIAL platform. First, we formalized dynamic notions of archives, allowing to effectively describe the data corpus. Then, we described the main retrieval facilities involved in RDF archiving, and have provided guidelines on the selection of relevant and comparable queries. We provide a concrete instantiation of archiving queries and instantiate our foundations in a prototypical benchmark suit, BEAR, composed of three real-world and well-described data corpus and query testbeds. Our prototypical evaluation [**?**] considers different state-of-the-art archiving policies, using independent copies (IC), change-based (CB), timestamp (TB) and hybrid (HB) approaches, and stores (Jena TDB, HDT, v-RDFCSA, TailR, R43ples).

Our initial results clearly confirm challenges (in terms of scalability) and strengths of current archiving approaches, and highlight the influence of the number of versions and the dynamicity of the dataset in order to select the right strategy (as well as an input for hybrid approaches in order to decide when and how to materialize a version), guiding future developments.In particular, in terms of space, CB, TB and hybrid policies (such as TB/CB in R43ples and IC/CB in TailR) achieve better results than IC in less dynamic datasets, but they are penalized in highly dynamic datasets due to index overheads. In this case, the TB policy reports overall good space

[15]Note that queries are provided as group graph pattern, such that they can be integrated in the AnQL notation. BEAR-C queries intentionally included UNION and OPTIONAL to extend the application beyond Basic Graph Patterns.

figures but it can be penalized at increasing number of versions. Regarding query resolution performance, the evaluated archiving policies excel at different operations but, in general, the IC, TB and CB/TB policies show a very constant behaviour, while CB and IC/CB policies degrade if more deltas have to be queried. Results also show that specific functional RDF compression techniques such as HDT and RDFCSA emerge as promising solutions for RDF archiving in terms of space requirements and query performance. These valuable insights can be then integrated in future versions of the SPECIAL platform.

Chapter 8

Encryption

This chapter briefly motivates and reviews the most important works on RDF encryption. Following on from this we present two different proposals for encrypting RDF data, once based on functional encryption and the based on symmetric encryption. The adaption of the existing SPECIAL platform to cater for encrypted RDF data will form part of the final SPECIAL release.

8.1 Encrypting RDF Data

Encryption techniques for RDF have received very little attention to date, with work primarily focusing on the partial encryption of RDF data, the querying of encrypted data and the signing of RDF graphs.

?] demonstrate how public-key encryption can be used to partially encryption RDF fragments (i.e. subjects, objects, or predicates). The ciphertext and the corresponding metadata (algorithm, key, hash etc...) are represented using a literal that they refer to as an encryption container. When only the object is encrypted, the object part of the triple is replaced with a blanknode (i.e. an anonymous resource) and a new statement is created with the blanknode as the subject, the encryption container as the object and a new `renc:encNLabel` predicate (cf. *Figure* 8.1). The treatment of encrypted subjects is analogous. The encrpytion of predicates is a little more difficult, as reification (a technique used to make statements about resources) is needed to associate the new blanknode with the relevant subject, object and encryption container.

Rather than simply storing the encrypted data and metadata in a literal, **?**] discuss now the metadata can be represented using multiple triples using their crypto ontology. The encrypted element or subgraph is replaced with a new unique identifier and new statements are added for the encrypted data and the corresponding metadata (cf. *Figure* 8.2).

?] in turn focus on querying encrypted data. In the proposed framework each triple is encrypted eight times according to the eight different triple pattern binding possibilities. The proposed approach allows for graph pattern queries to be executed over the ciphertext, at the cost of storing multiple ciphers for each statement. An alternative approach by **?**] demonstrates how functional encryption can be used to generate query keys based on triple patterns, whereby each key can decrypt all triples that match the corresponding triple pattern. While, other work by **?**] investigates enabling the signing of graph data at different levels of granularity.

The approach and the supporting images presented in this section have been adapted from **?**].

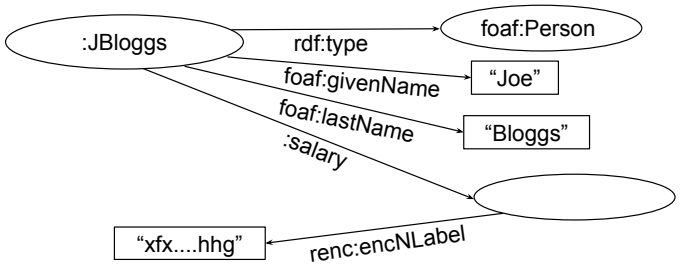


Figure 8.1: Partially Encrypted RDF graph

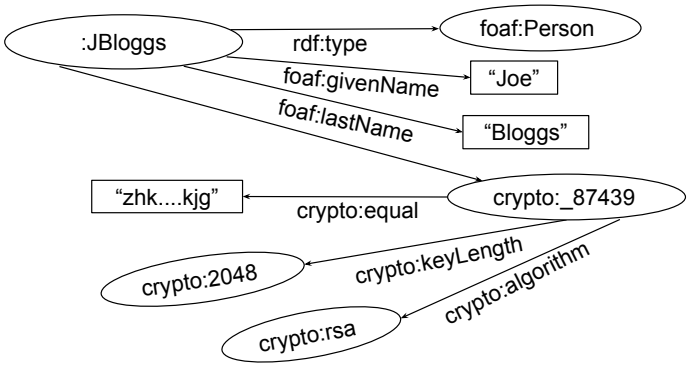


Figure 8.2: Partially Encrypted RDF graph and Metadata

8.2 Fine-grained Encryption for RDF

In this section, we discuss how functional encryption can be used together with RDF patterns to encryption RDF data in a very flexible manner. The approach and the supporting imaged presented herein have been adapted from **?**]. Common public-key encryption schemes usually follow an all-or-nothing approach (i.e., given a particular decryption key, a ciphertext can either be decrypted or not) which in turn requires users to manage a large amount of keys, especially if there is a need for more granular data encryption [**?**]. Recent advances in public-key cryptography, however, have led to a new family of encryption schemes called *Functional Encryption (FE)* which addresses aforementioned issue by making encrypted data self-enforce its access restrictions, hence, allowing for fine-grained access over encrypted information. In a functional encryption scheme, each decryption key is associated with a boolean function and each ciphertext is associated with an element of some attribute space Σ; a decryption key corresponding to a boolean function f is able to decrypt a particular ciphertext associated with $I \in \Sigma$ iff $f(I) = 1$. A functional encryption scheme is defined as a tuple of four distinct algorithms (**Setup**, **Enc**, **KeyGen**, **Dec**) such that:

Setup is used for generating a master public and master secret key pair.

Enc encrypts a plaintext message m given the master public key and an element $I \in \Sigma$. It returns a ciphertext c.

KeyGen takes as input the master secret key and generates a decryption key (i.e., secret key) SK_f for a given boolean function f.

Dec takes as input a secret key SK_f and a ciphertext c. It extracts I from c and computes $f(I)$.

In this section, we propose a flexible and dynamic mechanism for securely storing and efficiently querying RDF datasets. By employing an encryption strategy based on Functional Encryption (FE) in which controlled data access does not require a trusted mediator, but is instead enforced by the cryptographic approach itself, we allow for fine-grained access control over encrypted RDF data while at the same time reducing the administrative overhead associated with access control management.

8.2.1 A Functional Encryption Scheme for RDF

While there exist various different approaches for realising functional encryption schemes, we build upon the work of Katz et al. [**?**] in which functions correspond to the computation of inner-products over $\mathbb{Z}_N$ (for some large integer N). In their construction, they use $\Sigma = \mathbb{Z}_N^n$ as set of possible ciphertext attributes of length n and $\mathcal{F} = \{f_{\vec{x}} | \vec{x} \in \mathbb{Z}_N^n\}$ as the class of decryption key functions. Each ciphertext is associated with a (secret) attribute vector $\vec{y} \in \Sigma$ and each decryption key corresponds to a vector $\vec{x}$ that is incorporated into its respective boolean function $f_{\vec{x}} \in \mathcal{F}$ where $f_{\vec{x}}(\vec{y}) = 1$ iff $\sum_{i=1}^{n} y_i x_i = 0$.

In the following, we discuss how this encryption scheme can be utilised (i.e., its algorithms adopted[1]) to provide fine-grained access over encrypted RDF triples. Thus, allow for querying encrypted RDF using triple patterns such that a particular decryption key can decrypt all triples that satisfy a particular triple pattern (i.e., one key can open multiple locks). For example, a decryption key generated from a triple pattern `(?,p,?)` should be able to decrypt all triples with `p` in the predicate position.

8.2.1.1 Encrypting RDF Triples (Enc)

To be able to efficiently encrypt large RDF datasets, we adopt a strategy commonly used in public-key infrastructures for securely and efficiently encrypting large amounts of data called *Key Encapsulation* [**?**]. Key encapsulation allows for secure but slow asymmetric encryption to be combined with simple but fast symmetric encryption by using asymmetric encryption algorithms for deriving a symmetric encryption key (usually in terms of a seed) which is subsequently used by encryption algorithms such as AES [**?**] for the actual encryption of the data. We illustrate this process in Figure 8.3.

Thus, to encrypt an RDF triple $t = (s, p, o)$, we first compute its respective triple vector (i.e., attribute vector) $\vec{y_t}$ and functionally encrypt (i.e., compute **Enc** as defined in [**?**]) a randomly generated seed m_t using $\vec{y_t}$ as the associated attribute vector. Triple vector $\vec{y_t}$ where $\vec{y_t} = (y_s, y'_s, y_p, y'_p, y_o, y'_o)$ for triple t is constructed as follows, where σ denotes a mapping function that maps a triple's subject, predicate, and object value to elements in $\mathbb{Z}_N$:

$$y_l := -r \cdot \sigma(l),\ y'_l := r, \text{ with } l \in \{s, p, o\} \text{ and random } r \in \mathbb{Z}_N$$

Table 8.1 illustrates the construction of a triple vector $\vec{y_t}$ based on RDF triple t.

We use AES to encrypt the actual plaintext triple t with an encryption key derivable from our previously generated seed m_t and return both, the resulting AES ciphertext of t denoted by $\hat{t}$ and the ciphertext of the seed denoted by $\hat{m}_t$ as final ciphertext triple $c_t = \langle \hat{t}, \hat{m}_t \rangle$.

[1] The **Setup** algorithm remains unchanged.

TRIPLE t	TRIPLE VECTOR $\vec{y_t}$
$t_1 = (\mathtt{s}_1, \mathtt{p}_1, \mathtt{o}_1)$	$\vec{y}_{t_1} = (-r_1 \cdot \sigma(s_1), r_1, -r_2 \cdot \sigma(p_1), r_2, -r_3 \cdot \sigma(o_1), r_3)$
$t_2 = (\mathtt{s}_2, \mathtt{p}_2, \mathtt{o}_2)$	$\vec{y}_{t_2} = (-r_4 \cdot \sigma(s_2), r_4, -r_5 \cdot \sigma(p_2), r_5, -r_6 \cdot \sigma(o_2), r_6)$
...	...
$t_n = (\mathtt{s}_n, \mathtt{p}_n, \mathtt{o}_n)$	$\vec{y}_{t_n} = (-r_{3n-2} \cdot \sigma(s_n), r_{3n-2}, -r_{3n-1} \cdot \sigma(p_n), r_{3n-1}, -r_{3n} \cdot \sigma(o_n), r_{3n})$

Table 8.1: Computing the triple vector $\vec{y_t}$ of an RDF triple t.

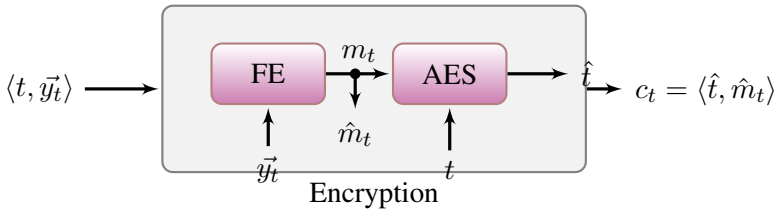


Figure 8.3: Process of encrypting an RDF triple t.

8.2.1.2 Generating Decryption Keys (KeyGen)

As outlined above, decryption keys must be able to decrypt all triples that satisfy their inherent triple pattern (i.e., one query key can open multiple locks). In order to compute a decryption key based on a triple pattern $tp = (s, p, o)$ with s, p, and o either bound or unbound, we define its corresponding vector $\vec{x}$ as $\vec{x}_{tp} = (x_s, x'_s, x_p, x'_p, x_o, x'_o)$ with:

$$\text{if } l \text{ is bound: } x_l := 1, x'_l := \sigma(l), \text{ with } l \in \{s, p, o\}$$
$$\text{if } l \text{ is not bound: } x_l := 0, x'_l := 0, \text{ with } l \in \{s, p, o\}$$

Again, σ denotes a mapping function that maps a triple pattern's subject, predicate, and object value to elements in $\mathbb{Z}_N$. Table 8.2 illustrates the construction of a query vector $\vec{x}_{tp}$ that corresponds to a triple pattern tp.

TRIPLE PATTERN tp	QUERY VECTOR $\vec{x}_{tp}$
$tp_1 = (?, ?, ?)$	$\vec{x}_{tp_1} = (0, 0, 0, 0, 0, 0)$
$tp_2 = (\mathtt{s}_2, ?, ?)$	$\vec{x}_{tp_2} = (1, \sigma(s_2), 0, 0, 0, 0)$
$tp_3 = (\mathtt{s}_3, \mathtt{p}_3, ?)$	$\vec{x}_{tp_3} = (1, \sigma(s_3), 1, \sigma(p_3), 0, 0)$
...	...
$tp_n = (\mathtt{s}_n, \mathtt{p}_n, \mathtt{o}_n)$	$\vec{x}_{tp_n} = (1, \sigma(s_n), 1, \sigma(p_n), 1, \sigma(o_n))$

Table 8.2: Computing the query vector $\vec{x}_{tp}$ that corresponds to a triple pattern tp

8.2.1.3 Decryption of RDF Triples (Dec)

To verify whether an encrypted triple can be decrypted with a given decryption key, we compute the inner-product of their corresponding triple vector $\vec{y_t}$ and query vector $\vec{x}_{tp}$, with $t = (s_t, p_t, o_t)$ and $tp = (s_{tp}, p_{tp}, o_{tp})$:

$$\vec{y_t} \cdot \vec{x}_{tp} = y_{s_t} x_{s_{tp}} + y'_{s_t} x'_{s_{tp}} + y_{p_t} x_{p_{tp}} + y'_{p_t} x'_{p_{tp}} + y_{o_t} x_{o_{tp}} + y'_{o_t} x'_{o_{tp}}$$

Only when $\vec{y}_t \cdot \vec{x}_{tp} = 0$ is it possible to decrypt the encrypted seed $\hat{m}_t$, hence the corresponding symmetric AES key can be correctly derived and the plaintext triple t be returned. Otherwise (i.e., $\vec{y}_t \cdot \vec{x}_{tp} \neq 0$), an arbitrary seed $m' \neq m_t$ is generated hence encrypted triple c_t cannot be decrypted [**?**].

8.2.2 Optimising Query Execution over Encrypted RDF

The *secure data store* holds all the encrypted triples, i.e. $\{c_{t_1}, c_{t_2}, \cdots, c_{t_n}\}$, being n the total number of triples in the dataset. Besides assuring the confidentiality of the data, the data store is responsible for enabling the querying of encrypted data.

In the most basic scenario, since triples are stored in their encrypted form, a user's query would be resolved by iterating over all triples in the dataset, checking whether any of them can be decrypted with a given decryption key. Obviously, this results in an inefficient process at large scale. As a first improvement one can distribute the set of encrypted triples among different peers such that decryption could run in parallel. In spite of inherent performance improvements, such a solution is still dominated by the available number of peers and the – potentially large – number of encrypted triples each peer would have to process. Current efficient solutions for querying encrypted data are based on (a) using indexes to speed up the decryption process by reducing the set of potential solutions; or (b) making use of specific encryption schemes that support the execution of operations directly over encrypted data [**?**]. Our solution herein follows the first approach, whereas the use of alternative and directly encryption mechanisms (such as homomorphic encryption [**?**]) is complementary and left to future work.

In our implementation of such a secure data store, we first encrypt all triples and store them in a key-value structure, referred to as an `EncTriples Index`, where the keys are unique integer IDs and the values hold the encrypted triples (see Figure 8.4 and Figure 8.5 (right)). Note that this structure can be implemented with any traditional *Map* structure, as it only requires fast access to the encrypted value associated with a given ID. In the following, we describe two alternative approaches, i.e., one using *three individual indexes* and one based on *Vertical Partitioning (VP)* for finding the range of IDs in the `EncTriples Index` which can satisfy a triple pattern query. In order to maintain simplicity and general applicability of the proposed store, both alternatives consider key-value backends, which are increasingly used to manage RDF data [**?**], especially in distributed scenarios. It is also worth mentioning that we focus on basic triple pattern queries as (i) they are the cornerstone that can be used to build more complex SPARQL queries, and (ii) they constitute all the functionality to support the Triple Pattern Fragments [**?**] interface.

8.2.2.1 3-Index Approach.

Following well-known indexing strategies, such as from CumulusRDF [**?**], we use three key-value B-Trees in order to cover all triple pattern combinations: `SPO`, `POS` and `OSP Indexes`. Figure 8.4 illustrates this organisation. As can be seen, each index consists of a *Map* whose keys are the securely hashed (cf. PBKDF2 [**?**]) subject, predicate, and object of each triple, and values point to IDs storing the respective ciphertext triples in the `EncTriples Index`.

Algorithm 5 shows the resolution of a `(s,p,o)` triple pattern query using the `3-Index` approach. First, we compute the secure hashes `h(s)`, `h(p)` and `h(o)` from the corresponding `s`, `p` and `o` provided by the user (Line 1). Our $hash(s, p, o)$ function does not hash unbounded terms in the triple pattern but treats them as a wildcard '`?`' term (hence all terms will be retrieved

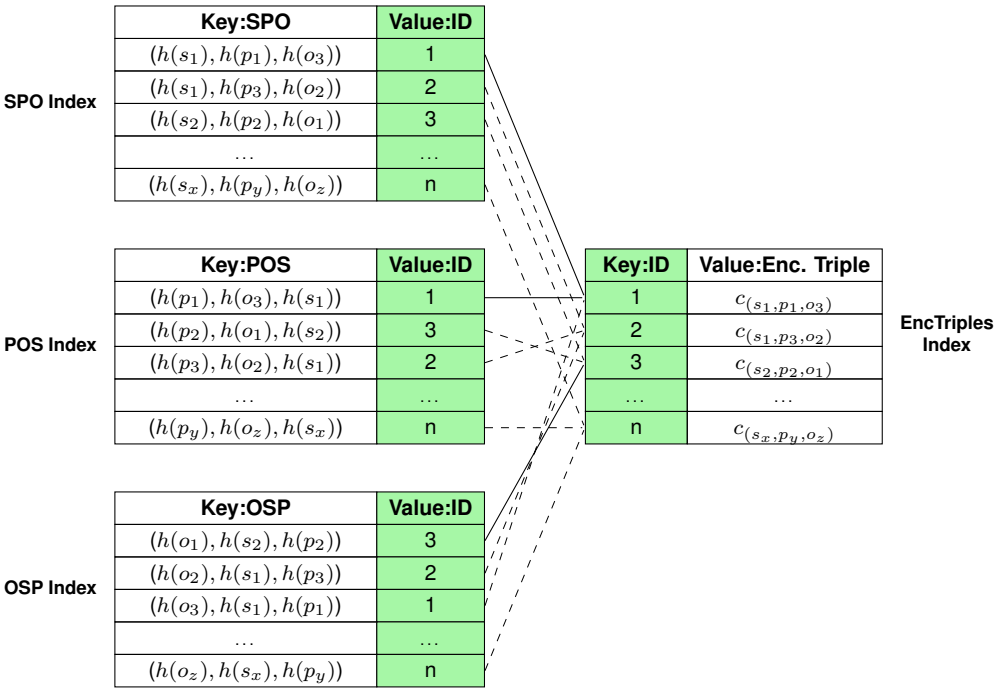


Figure 8.4: `3-Index` approach for indexing and retrieval of encrypted triples.

in the subsequent range queries). Then, we select the best index to evaluate the query (Line 2). In our case, the `SPO Index` serves `(s,?,?)` and `(s,p,?)` triple patterns, the `POS Index` satisfies `(?,p,?)` and `(?,p,o)`, and the `OSP Index` index serves `(s,?,o)` and `(?,?,o)`. Both `(s,p,o)` and `(?,?,?)` can be solved by any of them. Then, we make use of the selected index to get the range of values where the given `h(s), h(p), h(o)` (or 'anything' if the wildcard '`?`' is present in a term) is stored (Line 3). Note that this search can be implemented by utilising B-Trees [**?** **?**] for indexing the keys. For each of the candidate ID values in the range (Line 4), we retrieve the encrypted triple for such ID by searching for this ID in the `EncTriples Index` (Line 5). Finally, we proceed with the decryption of the encrypted triple using the `key` provided by the user (Line 6). If the status of such decryption is *valid* (Line 7) then the decryption was successful and we output the decrypted triples (Line 8) that satisfy the query.

Thus, the combination of the three `SPO`, `POS` and `OSP Indexes` reduces the search space of the query requests by applying simple range scans over hashed triples. This efficient retrieval has been traditionally served through tree-based map structures guaranteeing $log(n)$ costs for searches and updates on the data, hence we rely on B-Tree stores for our practical materialisation of the indexes. In contrast, supporting all triple pattern combinations in `3-Index` comes at the expense of additional space overheads, given that each `(h(s),h(p),h(o))` of a triple is stored three times (in each `SPO`, `POS` and `OSP Indexes`). Note, however, that this is a typical scenario for RDF stores and in our case the triples are encrypted and stored just once (in `EncTriples Index`).

Algorithm 5: `3-Index_Search(s,p,o,key)`

1 $(h(s), h(p), h(o)) \leftarrow hash(s, p, o)$; $index \leftarrow selectBestIndex(s, p, o)$; ▷
$index = \{SPO|POS|OSP\}$ $IDs[\,] \leftarrow index.getRangeValues(h(s), h(p), h(o))$;
for *each* $(id \in IDs)$ **do**
2 $encryptedTriple \leftarrow EncTriples.get(id)$;
$< decryptedTriple, status > \leftarrow Decrypt(encryptedTriple, key)$; **if**
$(status = valid)$ **then**
3 **output**$(decryptedTriple)$;
4 **end**
5 **end**

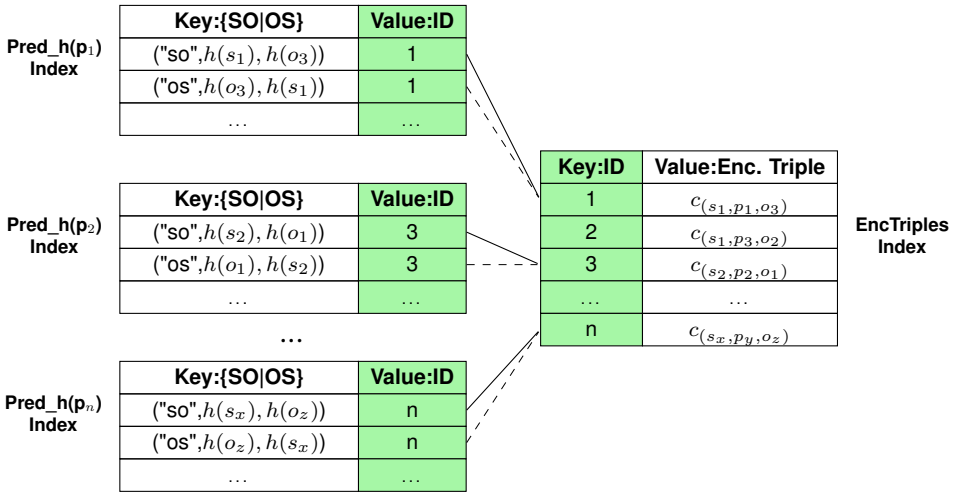


Figure 8.5: `Vertical Partitioning (VP)` approach for indexing and retrieval of encrypted triples.

8.2.2.2 Vertical Partitioning Approach.

Vertical partitioning [**?**] is a well-known RDF indexing technique motivated by the fact that usually only a few predicates are used to describe a dataset [**?**]. Thus, this technique stores one "table" per predicate, indexing `(S,O)` pairs that are related via the predicate. In our case, we propose to use one key-value B-Tree for each `h(p)`, storing `(h(s),h(o))` pairs as keys, and the corresponding ID as the value. Similar to the previous case, the only requirement is to allow for fast range queries on their map index keys. However, in the case of an `SO` index, traditional key-value schemes are not efficient for queries where the first component (the subject) is unbound. Thus, to improve efficiency for triple patterns with unbounded subject (i.e. `(?,`p_y`,`o_z`)` and `(?,?,`o_z`)`), while remaining in a general key-value scheme, we duplicate the pairs and introduce the inverse `(h(o),h(s))` pairs. The final organisation is shown in Figure 8.5 (left), where the predicate maps are referred to as `Pred_h(`p_1`)`, `Pred_h(`p_2`)`,..., `Pred_h(`p_n`) Indexes`. As depicted, we add `"so"` and `"os"` keywords to the stored composite keys in order to distinguish the order of the key.

Algorithm 6 shows the resolution of a `(s,p,o)` triple pattern query with the `VP` organisation. In this case, after performing the variable initialisation (Line 1) and the aforementioned

Algorithm 6: `VerticalPartitioning_Search(s,p,o,key)`

```
1  IDs[ ] ← (); (h(s), h(p), h(o)) ← hash(s, p, o);
   Indexes[] ← selectPredIndex(h(p)); ▷
   Indexes ⊆ {Pred_h(p_1), ··· , Pred_h(p_n)Index} for each (index ∈ Indexes) do
2      if (s =?) then
3          IDs[ ] ← index.getRangeValues("os", h(o), ?);
4      else
5          IDs[ ] ← index.getRangeValues("so", h(s), h(o));
6      end
7      for each (id ∈ IDs) do
8          encryptedTriple ← EncTriples.get(id);
           < decryptedTriple, status >← Decrypt(encryptedTriple, key); if
           (status = valid) then
9              output(decryptedTriple);
10         end
11     end
12 end
```

secure hash of the terms (Line 2), we inspect the predicate term `h(p)` and select the corresponding predicate index (Line 3), i.e., `Pred_h(p)`. Nonetheless, if the predicate is unbounded, all predicate indexes are selected as we have to iterate through all tables, which penalises the performance of such queries. For each predicate index, we then inspect the subject term (Lines 5-9). If the subject is unbounded (Line 5), we will perform a `("os",h(o),?)` range query over the corresponding predicate index (Line 6), otherwise we execute a `("so",h(s),h(o))` range query. Note that in both cases the object could also be unbounded. The algorithm iterates over the candidates IDs (Lines 10-end) in a similar way to the previous cases, i.e., retrieving the encrypted triple from `EncTriples Index` (Line 11) and performing the decryption (Lines 12-14).

Overall, `VP` needs less space than the previous `3-Index` approach, since the predicates are represented implicitly and the subjects and objects are represented only twice. In contrast, it penalises the queries with unbound predicate as it has to iterate through all tables. Nevertheless, studies on SPARQL query logs show that these queries are infrequent in real applications [**?**].

8.2.2.3 Protecting the Structure of Encrypted Data.

The proposed hash-based indexes are a cornerstone for boosting query resolution performance by reducing the encrypted candidate triples that may satisfy the user queries. The use of secure hashes [**?**] assures that the terms cannot be revealed but, in contrast, the indexes themselves reproduce the structure of the underlying graph (i.e., the in/out degree of nodes). However, the structure should also be protected as hash-based indexes can represent a security risk if the data server is compromised. State-of-the-art solutions (cf., [**?**]) propose the inclusion of spurious information, that the query processor must filter out in order to obtain the final query result.

In our particular case, this technique can be adopted by adding dummy triple hashes into the indexes with a corresponding ciphertext (in `EncTriples Index`) that cannot be decrypted by any key, hence will not influence the query results. Such an approach ensures that both the

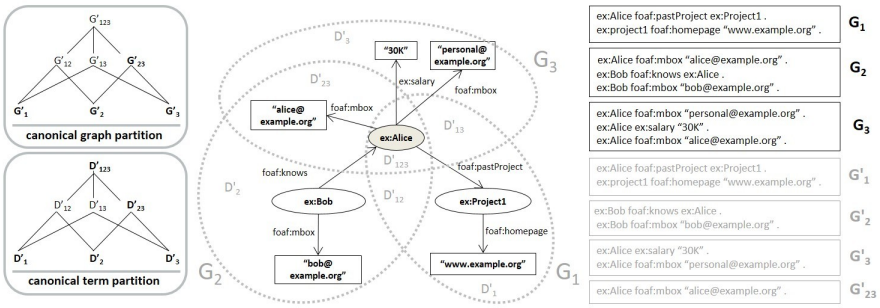


Figure 8.6: An access-restricted RDF dataset such that G comprises three separate access-restricted subgraphs G_1, G_2, G_3; the graph's canonical partition is comprised of four non-empty subgraphs $G'_1, G'_2, G'_3, G'_{23}$, whereas the *terms* in these graphs can be partitioned into five non-empty subsets corresponding to the dictionaries $D'_1, D'_2, D'_3, D'_{23}, D'_{123}$.

triple hashes and their corresponding ciphertexts are not distinguishable from real data.

8.3 HDT$_{crypt}$: Extending HDT for Encryption

The publication and interchange of RDF datasets online has experienced significant growth in recent years, promoted by different but complementary efforts, such as Linked Open Data, the Web of Things and RDF stream processing systems. However, the current Linked Data infrastructure does not cater for the storage and exchange of sensitive or private data. On the one hand, data publishers need means to limit access to confidential data (e.g. health, financial, personal, or other sensitive data). On the other hand, the infrastructure needs to compress RDF graphs in a manner that minimises the amount of data that is both stored and transferred over the wire. In this section, we discuss how HDT – a compressed serialization format for RDF – can be extended to cater for supporting encryption. We propose a number of different graph partitioning strategies and discuss the benefits and tradeoffs of each approach. The approach and the supporting images presented herein have been adapted from **?**].

We introduce HDT_{crypt}, an extension of HDT that involves encryption of RDF graphs. We first define the notion of access-restricted RDF datasets and the implications for HDT (Section 8.3.1). Then, we show an extension of the HDT header component to cope with access-restricted RDF datasets (Section 8.3.2), which leads to the final HDT_{crypt} encoding. Finally, as HDT_{crypt} can manage several HDT Dictionary components, we describe the required operations to integrate different Dictionary components within an HDT collection (Section 8.3.3). These operations will be the basis to represent the shared components between access-restricted datasets efficiently, addressed in Section 8.3.4.

8.3.1 Representing access-restricted RDF datasets

We consider hereinafter that users wishing to publish *access-restricted RDF datasets* divide their complete graph of RDF triples G into (named) graphs, that are accessible to other users, i.e. we assume that access rights are already materialised per user group in the form of a set (cover) of separate, possibly overlapping, RDF graphs, each of which are accessible to different sets of users.

Borrowing terminology from [**?**], an *access restricted RDF dataset* (or just "dataset" in the following) is a set $DS = \{G, (g_1, G_1), \ldots, (g_n, G_n)\}$ consisting of a (non-named) default graph G and named graphs s.t. $g_i \in I$ are graph names, where in our setting we require that $\{G_1, \ldots, G_n\}$ is a cover[2] of G. We further call DS a *partition* of G if $G_i \cap G_j = \emptyset$ for any $i \neq j$; $1 \leq i, j \leq n$. Note that from any dataset DS, a *canonical partition* DS' can be trivially constructed (but may be exponential in size) consisting of all non-empty (at most $2^n - 1$) subsets G'_S of triples $t \in G$ corresponding to an index set $S \in 2^{1,\ldots,i}$ such that $G'_S = \{t \mid t \in \bigcap_{i \in S} G_i \wedge \neg \exists S' : (S' \supset S \wedge t \in \bigcap_{j \in S'} G_j)\}$.

Figure 8.6 shows an example of such a dataset composed of three access-restricted subgraphs (or just "subgraphs" in the following) G_1, G_2, G_3 for the previous full graph G (Figure **??**a). Intuitively, this corresponds to a scenario with three access rights: users who can access general information about projects in an organisation (graph G_1); users who have access to public email accounts and relations between members in the organisation (graph G_2); and finally, users who can view personal information of members, such as the salary and personal email accounts (graph G_3). As can be seen, the triple (ex:Alice foaf:mbox "alice@example.org") is repeated in subgraphs G_2 and G_3, showing a redundancy which can produce significant overheads in realistic scenarios with large-scale datasets and highly overlapping graphs. Canonical partitioning groups these triples into disjoint sets so that no repetitions are present. In our example in Figure 8.6, the set $G'_{\{2,3\}}$, which can simply be written as G'_{23}, holds this single triple, (ex:Alice foaf:mbox "alice@example.org"), hence this triple is not present in G'_2 and G'_3. In this simple scenario, G'_1 is equivalent to G_1 as it does not share triples with other graphs.

Thus, we consider hereinafter an *HDT collection* corresponding to a dataset DS denoted by $HDT(DS) = (H, \mathcal{D}, \mathcal{T})$ as a single H, plus sets $\mathcal{D} = \{D_1, \ldots, D_n\}$, $\mathcal{T} = \{T_1, \ldots, T_m\}$ of dictionary and triple components, respectively, such that the union of triple components encodes a cover of G, i.e. the overall graph of all triples in the dataset DS. We do not assume that there is a one-to-one correspondence between individual triple components in $\mathcal{T}$ and graphs in DS; different options of mapping subgraphs to HDT components will be discussed in Section 8.3.4 below. The relation between the dictionaries and the triple components (in other words, which dictionaries are used to codify which triple components) is also flexible and must be specified through metadata properties. In our case, we assume $H = \{R, M\}$ to contain a relation $R \subseteq \mathcal{D} \times \mathcal{T}$, which we call the *dictionary-triples map* with the implicit meaning that dictionary components encode terms used in the corresponding triple components, and M is comprised of additional header metadata (as mentioned above, the header contains a variety of further (meta-)information in standard HDT [**?**], which we skip for the considerations herein). It is worth noting that we do not prescribe that either D or T do not overlap. However, it is clear that one should find an unambiguous correspondence to decode the terms under $ids(T)$.

Thus, we define the following admissibility condition for R. An HDT collection is called *admissible* if:

- $\forall D_i, D_j \in \mathcal{D} : (D_i, T), (D_j, T) \in R \wedge i \neq j \implies terms(D_i) \cap terms(D_j) = \emptyset$
- $\forall T \in \mathcal{T} : i \in ids(T) \implies \exists (D, T) \in R \wedge i \in ids(D)$

For any admissible HDT collection HDT we define the *T-restricted collection* HDT^T as the collection obtained from removing: (i) all triple components $T' \neq T$ from HDT; (ii) the corresponding D' such that (D', T') is in R and (D', T) is not in R; and (iii) the relations

[2] In the set-theoretic sense.

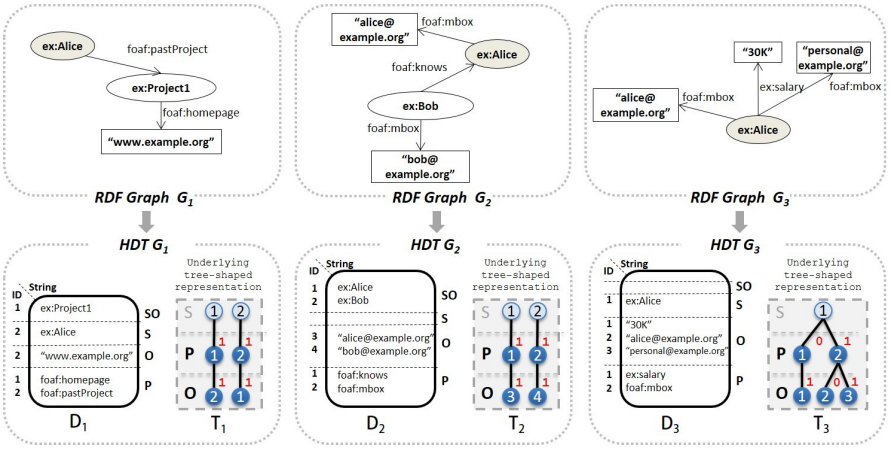


Figure 8.7: HDT$_{crypt-A}$, create and encrypt one HDT per partition.

(D', T') from R. Thus allowing an HDT collection to be filtered by erasing all dictionary and triple components that are not required for T.

8.3.2 HDT$_{crypt}$ encoding

We now introduce the final encoding of the HDT_{crypt} extension. HDT_{crypt} uses AES (Advanced Encryption Standard) [**?**] to encrypt the D and triple components of an HDT collection and extends the header H with a keymap $kmap : \mathcal{D}_{crypt} \cup \mathcal{T}_{crypt} \mapsto I$ that maps *encrypted components* to identifiers (IRIs), which denote AES keys that can be used to decrypt these components.

Thus, $HDT_{crypt} = (H, \mathcal{D}_{crypt}, \mathcal{T}_{crypt})$ where $H = \{R, kmap, M\}$, $R \subseteq \mathcal{D}_{crypt} \times \mathcal{T}_{crypt}$, and the components in $\mathcal{D}_{crypt}$ and $\mathcal{T}_{crypt}$ are encrypted with keys identified in $kmap$.

The operations to $encrypt$ and $decrypt$ the dictionary and triples are described as follows. First, the operation $encrypt$ takes one or more dictionary and triples and encrypts the components with a given key. Formally, we write $encrypt(c, key_{crypt}) = c_{crypt}$, where $c \in \mathcal{D} \cup \mathcal{T}$ to denote the component $c_{crypt} \in \mathcal{D}_{crypt} \cup \mathcal{T}_{crypt}$ obtained by encrypting c with the key key_{crypt}. While, we add an identifier of the components to the header metadata. In other words, $id(c_{crypt}) \mapsto IRI(key_{crypt})$ is added to the kmap, where id denotes the ID of the component in $\mathcal{D}_{crypt}$ and $\mathcal{T}_{crypt}$ and IRI a unique identifier for the symmetric key.

For the decryption, it is assumed that an authorized user u has partial knowledge about these keys, i.e. they have access to a partial function $key_u : I_u \mapsto K$ that maps a finite set of "user-owned" key IDs $I_u \subseteq I$ to the set of AES (symmetric) keys K. The decryption simply takes the given compressed component(s) and performs the decryption with the given symmetric key. Formally, we write $decrypt(c_{crypt}, key_{crypt}) = c$, where $c_{crypt} \in \mathcal{D}_{crypt} \cup \mathcal{T}_{crypt}$ to denote the component $c \in \mathcal{D} \cup \mathcal{T}$ obtained from decrypting c_{crypt} with the key $key_{crypt} = key(kmap(c_{crypt}))$. Further we write $decrypt(HDT_{crypt}, I_u)$ to denote the non-encrypted HDT collection consisting of all decrypted dictionary and triple components of HDT_{crypt} which can be decrypted with the keys in $\{key_u(i) \mid i \in I_u\}$. In other words, the T-restriction of HDT_{crypt} is defined analogously to the above-said.

8.3.3 Integration operations

Finally, we define two different ways of integrating dictionaries $D_1, \ldots, D_k \in \mathcal{D}$ within an HDT collection: *D-union* and *D-merge*. In the former, we replace dictionaries with a new dictionary that includes the union of all terms. In the latter, we establish one of the dictionaries as the dictionary baseline and rename the IDs of the other dictionaries.

8.3.3.1 D-union

The *D-union* is only defined for $D_1, \ldots, D_k \subseteq \mathcal{D}$ if the following condition holds on R: $\forall (D_i, T) \in R : (\neg \exists D_j \notin D_1, \ldots, D_k$ such that $(D_j, T) \in R)$. In other words, we can perform a D-union if all T-components depending on dictionaries in the set $D_1, \ldots, D_k$ only depend on these dictionaries. Then, we can define a trivial D-union of HDT wrt. $D_1, \ldots, D_k$, written $HDT_{D_1 \cup \ldots \cup D_k}$, as follows:

- replace $\{D_1, \ldots, D_k\}$ dictionaries with a single dictionary $D_{1\ldots k} = D_1 \cup \ldots \cup D_k$, such that $\forall x \in terms(D_1) \cup \ldots \cup terms(D_k)$
  - $x \in terms(D_{1\ldots k})$
  - $id(x, D_{1\ldots k})$ is obtained by sequentially numbering the terms in $terms(D_1) \cup \ldots \cup terms(D_k)$ upon an (arbitrary) total order, e.g., lexicographically ordering the terms (as it is done in HDT dictionaries by default).
- replace all $(D_i, T) \in R$, $i \in \{1, \ldots, k\}$, with new $(D_{1\ldots k}, T')$ relations, where T' is obtained from T by replacing the original IDs from D_i with their corresponding new IDs in $D_{1\ldots k}$.

8.3.3.2 D-merge

In the more general case where the condition for D-unions does not hold on $D_1, \ldots, D_k \subseteq \mathcal{D}$, we can define another operation, *D-merge*, written $HDT_{D_1 \triangleright \ldots \triangleright D_k}$. We start with the binary case, where only two dictionaries D_1 and D_2 are involved; $HDT_{D_1 \triangleright D_2}$ is obtain as follows:

- replace D_1 and D_2 with a single $D_{12} = D_1 \triangleright D_2$,[3] such that
  - $\forall x \in terms(D_1) : id(x, D_{12}) = id(x, D_1)$
  - $\forall x \in terms(D_2) \setminus terms(D_1) : id(x, D_{12}) = id(x, D_2) + max(ids(D_1))$
- replace all $(D_1, T_1) \in R$ with (D_{12}, T_1)
- replace all $(D_2, T_2) \in R$ with (D_{12}, T_2'), where T_2' is obtained from T_2 by analogous ID changes.

D-merge can then be trivially generalized to a sequence of dictionaries assuming left-associativity of $\triangleright$ operator. That is, $HDT_{D_1 \triangleright D_2 \triangleright \ldots \triangleright D_k} = HDT_{((D_1 \triangleright D_2) \triangleright \ldots) \triangleright D_k}$.

For convenience, we extend the notation of $T(G, D)$ from Section 7.2.2.2 to D-unions and D-merges: let $(D_1, \ldots, D_k)$ be a sequence of dictionaries and G an RDF graph such that $terms(G) = \bigcup_{D_i \in (D_1, \ldots, D_k)} terms(D_i)$. Then we will write $T(G, (D_1 \cup \ldots \cup D_k))$ and $T(G, (D_1 \triangleright \ldots \triangleright D_k))$ for the triples part generated from G according to the combined dictionary $((D_1 \cup D_2) \cup \ldots) \cup D_k$ and $((D_1 \triangleright D_2) \triangleright \ldots) \triangleright D_k$ respectively. Finally, we note that for any admissible HDT collection, both D-union and D-merge preserve admissibility.

[3] We use the directed operator $\triangleright$ instead of $\cup$ here, since this operation is not commutative.

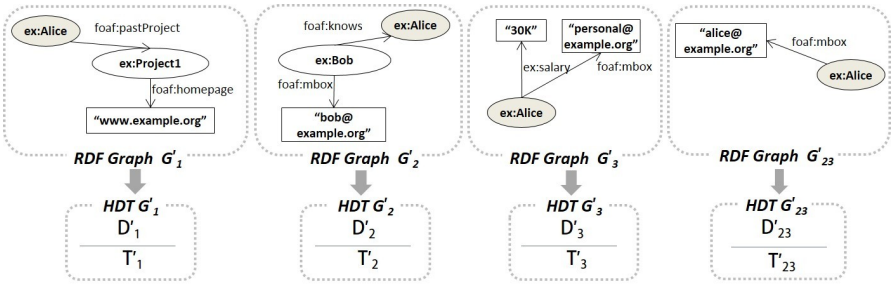


Figure 8.8: HDT$_{crypt-B}$, extracting non-overlapping triples.

8.3.4 Efficient Partitioning HDT$_{crypt}$

After having introduced the general idea of HDT$_{crypt}$ and the two different ways of integrating dictionaries within an HDT collection, we now discuss four alternatives strategies that can be used for distributing a dataset DS across dictionary and triple components in an HDT$_{crypt}$ collection. These alternatives, referred to as HDT$_{crypt-A}$, HDT$_{crypt-B}$, HDT$_{crypt-C}$ and HDT$_{crypt-D}$, provide different space/performance tradeoffs that will be evaluated in Section **??**. We note that HDT behaves differently than the normal RDF merge regarding blank nodes in different "partitions" as, by default, HDT does not rename the blank nodes to avoid shared labels [**?**]: the original blank nodes are skolemized to constants (unique per RDF graph) and preserved across partitions, so that we do not need to consider blank node (re-)naming separately.

8.3.4.1 HDT$_{crypt-A}$: A Dictionary and Triples per Named Graph in DS

The baseline approach is straightforward, we construct separate HDT components $D_i = D(G_i)$ and $T_i = T(G_i, D_i)$ per graph G_i in the dataset, see Figure 8.7, thereafter each of these components is encrypted with a respective, separate key, identified by a unique IRI $id_i \in I$, i.e., $kmap(D_i) = kmap(T_i) = id_i$ and $R = \{(D_i, T_i) \mid G_i \in DS\}$. For re-obtaining graph G_i a user must only have access to the key corresponding to id_i, and can thereby decrypt D_i and T_i and extract the restricted collection HDT^{T_i}, which corresponds to G_i. Obviously, this approach encodes a lot of overlaps in both dictionary and triples parts: that is, for our running example from Figure 8.7, the IRI for ex:alice is encoded in each individual D component and the overlapping triples in graphs G_2 and G_3 appear in both T_2 and T_3 respectively (cf., Figure 8.7).

8.3.4.2 HDT$_{crypt-B}$: Extracting non-overlapping Triples in DS'

In order to avoid the overlaps in the triple components, a more efficient approach could be to split the graphs in the dataset DS according to their canonical partition DS' and again construct separate (D,T)-pairs for each subset $G'_S \in DS'$, see Figure 8.8. That is, we create $D'_S = D(G'_S)$ and $T'_S = T(G'_S, D'_S)$ per graph $G'_S \in DS'$, where $S \in 2^{1,\ldots,i}$ denotes the index set corresponding to a (non-empty) subset of DS'. R in turn contains pairs (D'_S, T'_S) and $kmap$ entries for keys identified by I'_S per G'_S used for the encryption/decryption of the relevant D'_S and T'_S. The difference for decryption now is that any user who is allowed access to G_i must have all keys corresponding to any I'_S such that $i \in S$ in order to re-obtain the original graph G_i.

First, the user will decrypt all the components for which they have keys, i.e. obtaining a non-encrypted collection HDT' consisting of components $\mathcal{D}' = \{D'_1, \ldots, D'_k\}, \mathcal{T}' = \{T'_1, \ldots, T'_k\}$

consisting of the components corresponding to a partition of G_i. Then, for decompressing the original graph G_i, we create separate T'_S-restricted HDTs, which are decompressed separately, with G_S being the union of the resulting subgraphs.

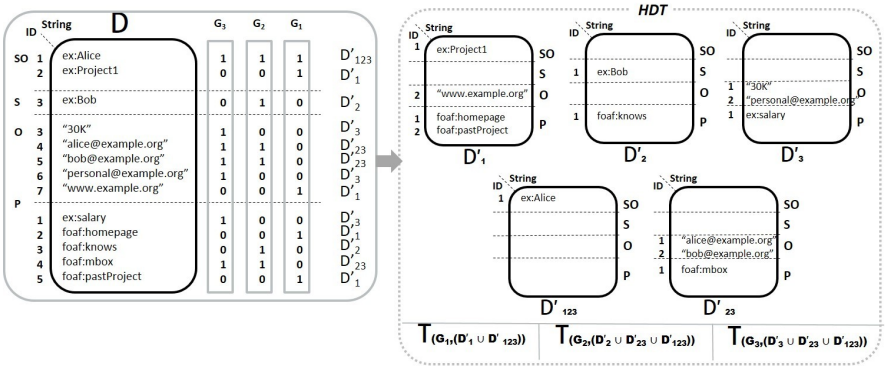


Figure 8.9: HDT$_{crypt-C}$, extracting non-overlapping dictionaries.

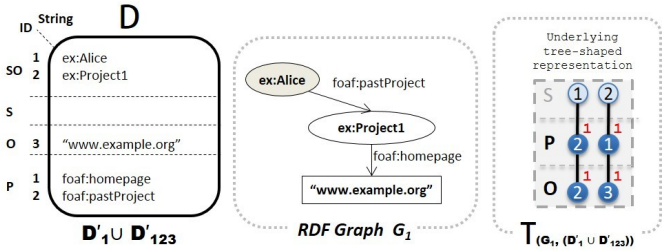


Figure 8.10: Union of dictionaries (in HDT$_{crypt-C}$) to codify the non-overlapping dictionaries of a partition.

8.3.4.3 HDT$_{crypt-C}$: Extracting non-overlapping Dictionaries in DS'

Note that in the previous approach, we have duplicates in the dictionary components. An alternative strategy would be to create a *canonical partition* of *terms* instead of triples, and create separate dictionaries $D'_S \in \mathcal{D}'$ for each non-empty term-subset,[4] respectively. Figure 8.9 shows the canonical partition of terms in our running example: as can be seen, the original dictionary is split into five non-empty terms-subsets corresponding to the dictionaries D'_{123} (terms shared in all three graphs), D'_{23} (terms shared in graphs G_2 and G_3 that are not in D'_{123}) and D'_1, D'_2, D'_3 (terms in either G_1, G_2 or G_3 resp. and are not shared between graphs). This partition can be computed efficiently, thanks to the HDT dictionary D of the full graph G, which we assume to be available[5]. To do so, we keep[6] an auxiliary bitsequence per graph G_i (see Figure 8.9, top left), each of size $terms(D)$. Then, we iterate through triples in each graph G_i and, for

[4] Again, here $S \in 2^{1,\ldots,n}$ represents an index set.

[5] All HDT$_{crypt}$ strategies are evaluated from an existing full graph G. Our evaluation in Section **??** also reports the time to create the HDT representation of the full graph G

[6] This auxiliary structure is maintained just at compression time and it is not shipped with the encrypted information.

each term, we search its ID in D, marking such position with a 1-bit in the bitsequence of G_i. Finally, the dictionaries of the subsets can be created by inspecting the combinations of 1-bits in the bitsequences: terms in $D'_{xy\cdots z}$ will be those with a 1-bit in the bitsequences of graphs $xy\cdots z$ and 0-bits in other graphs. For instance, in Figure 8.9, D'_{123} is constituted only by ex:alice, because it is the only term with three 1-bits in the bitsequences of G_1, G_2 and G_3. In contrast, ex:Project1 will be part of D'_1 as it has a 1-bit only in the bitsequence of G_1.

The number of triple components in this approach are as in HDT$_{crypt-A}$, one per graph G_i. However, they are constructed slightly differently as, in this case, we have a canonical partition of terms and one user will just receive the dictionaries corresponding to subsets that correspond to terms in the graph G_i that they have been granted access to. In other words, the IDs used in each T_i should unambiguously correspond to terms, but these terms may be distributed across several dictionaries.[7] Thus, we encode triples with a D-union (see Section 8.3.3) of the D'_S such that $i \in S$. That is, for each G_i we construct $T_i = T(G_i, (\bigcup_{i\in S} D'_S))$, and add the respective pairs (D'_S, T_i) in R.

Figure 8.10 illustrates this merge of dictionaries for the graph G_1 and the respective construction of $T(G_1, (D'_1 \cup D'_{123}))$. The decompression process after decryption is the exact opposite. For decompressing the graph G_i, the decrypted dictionaries $\bigcup_{i\in S} D'_S$ are used to create a D-union D_i which can be used to decompress the triples T_i in one go. Finally, as a performance improvement at compression time, note that, although the canonical partition of terms has to be built to be shipped in the compressed output, we can actually skip the creation of the D-union dictionaries to encode the IDs in the triples. To do so, we make use of the bitsequences to get the final IDs that are used in the triples: One should note that the ID of a term in a D-union of a graph G_i is the number of previous 1-bits in the bitsequence of G_i (for each *SO*, S, O, and P section). For instance, in our example in Figure 8.10, ex:Project1 is encoded with the ID=2. Instead of creating D_1, we can see that in the bitsequence of G_1 (see Figure 8.9, top right) we have two 1-bits in the predicate section up to the position where ex:Project1 is stored in the original dictionary, hence its ID=2.

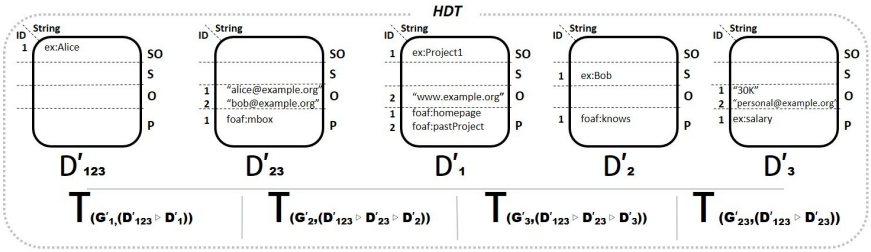


Figure 8.11: HDT$_{crypt-D}$, extracting non-overlapping dictionaries and triples.

8.3.4.4 HDT$_{crypt-D}$: Extracting non-overlapping Dictionaries and Triples in DS'

In HDT$_{crypt-D}$, we combine the methods of both HDT$_{crypt-B}$ and HDT$_{crypt-C}$. That is, we first create a *canonical partition of terms* as in HDT$_{crypt-C}$, and a *canonical partition of triples* DS' as in HDT$_{crypt-B}$. Then, we codify the IDs in the subsets of DS' with the IDs from the dictionaries. Note, however, that in this case there is – potentially – an n:m between the resulting dictionary and triple components. In other words, triples in T'_S can include terms that

[7] Given the partition definition, it is nonetheless true that a term appears in one and only one term-subset.

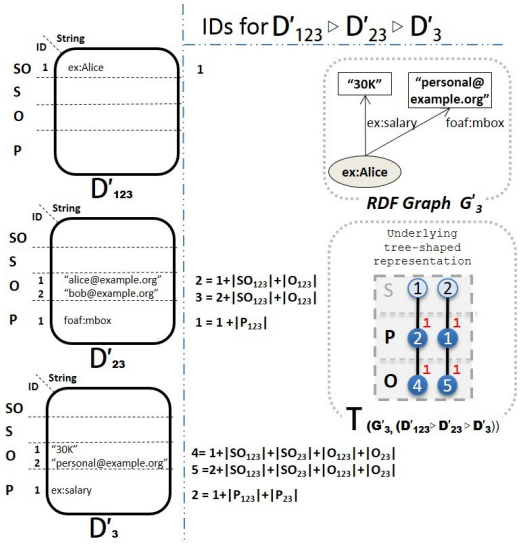


Figure 8.12: Merge of dictionaries (in HDT$_{crypt-D}$) to codify the non-overlapping dictionaries and triples of a partition.

are not only in D'_S as they may be distributed across several term-subsets. For instance, in our running example, T'_1 in HDT$_{crypt-B}$ includes ex:Alice (see Figure 8.8) which is stored in D'_{123} in HDT$_{crypt-C}$ (see Figure 8.9). One alternative could be to create a D-union of each graph G'_S and codify triples in T'_S with the corresponding IDs. However, it is trivial to see that this would lead to an exponential number of D-union dictionaries, one per T'_S component. In addition, we would need to physically recreate all these dictionaries at compression time, and also at decompression time in order to decompress each single graph G'_S. Thus, we perform a D-merge approach (see the definition in Section 8.3.3), which fits perfectly with n:m-relations. This is illustrated in Figure 8.11. As can be seen, triples in each G'_S of the canonical partition are encoded with an appropriate D-merge of term-subsets. A practical example is shown in Figure 8.12, representing the encoding of graph G'_3. As defined in D-merge, IDs are assigned in order, that is for a merge $D'_1 \triangleright \ldots \triangleright D'_k$, the IDs in D'_k are shifted $max(ids(D'_1)) + \ldots + max(ids(D'_{k-1}))$. For instance, in our example, the predicate ex:salary will be encoded in G'_3 with the ID=2, because its local ID in D'_3 is 1, and it has to be shifted $max(ids(D'_{123})) + max(ids(D'_{23})) = 1$, hence its final ID= $1+max(ids(D'_{123}))+max(ids(D'_{23})) = 2$. Note that here we restrict the dictionaries D' per section (SO, S, O and P). Given the special numbering of IDs in HDT, where S and O IDs follow from SO as explained in Section 7.2.2.1. This is illustrated in our example, e.g. the object "30K" with local ID=1 in D'_3 is mapped in the D-merge dictionary with 4, as it sums up all the previous objects and subjects IDs in D'_{123} and D'_{23}.

It is worth mentioning that no ambiguity is present in the order of the D-merge as it is implicitly given by the partition DS' as per the canonical term partition. Thus, the decompression follows the opposite process: for each graph T'_S in the partition of the graph G_i, the user processes each ID and, depending of the value, they get the associated term in an appropriate term subset. For instance, if the user is accessing the predicate ID=2 in our example, one can easily

see that $2 > |P_{123}| + |P_{23}|$, so dictionary D'_3 has to be used[8]. The local ID to look at is then $2 - |P_{123}| - |P_{23}| = 1$, hence the predicate ID=1 in D'_3 is inspected and then foaf:pastProject is retrieved. Finally, note that not all terms in a D-merge are necessarily used when encoding a particular T'_S. For instance, in our example in Figure 8.12, the object "bob@example.org" with ID=2 in D'_{23} (and ID=3 in the D-merge) is not used in T'_3. However, this ID is "blocked": it cannot be used by a different object in T'_3 as this ID is taken into account when encoding the present objects ("30K" and "personal@example.org"), once we sum the $max(ids(D'_{23}))$ as explained above. The same consequence applies to subjects, so that subject IDs are not necessarily correlative in T'_S. This constitutes a problem for the HDT Bitmap Triples encoding (presented in Section 7.2.2.2), given that it represents subjects implicitly assuming that they are correlative. Thus, HDT$_{crypt-D}$ has to explicitly state the ID of each subject, which constitutes a space overhead and a drawback of this approach, despite the fact that duplicate terms and triples are avoided. Technically, instead of a forest of trees, triples are codified as tuples of three IDs, using an existing HDT triples representation called *Plain Triples* [**?**].

[8]We abuse notation to denote the cardinality of a set, e.g. $|P_{123}|$, as the maximum id represented in such dictionary set.

Chapter 9

Discussion

This deliverable gives an insight into the current state of the SPECIAL-K architecture, highlighting the changes with respect to compliance checking, consent management, personal data inventory, compression, encryption, and performance improvements. As explained earlier, not all choices are final, and some problems will be tackled in D3.6 Final Release, including the challenge of interfacing with Line of Business applications.

Optimizations with respect to data handling in in the compliance checker are expected to lead to significant improvements in terms of performance. This will be demonstrated in D3.5 Scalability and Robustness Testing Report V2.

Bibliography

[1] Apache kafka. URL https://kafka.apache.org/.

[2] Big data europe. URL https://www.big-data-europe.eu/.

[3] Docker compose, . URL https://docs.docker.com/compose/.

[4] Docker swarm mode, . URL https://docs.docker.com/engine/swarm/.

[5] keycloak. URL https://www.keycloak.org/.

[6] Rfc 6749: The oauth 2.0 authorization framework. URL https://tools.ietf.org/html/rfc6749.

[7] Openid connect core 1.0. URL https://openid.net/specs/openid-connect-core-1_0.html.

[8] Rethinkdb. URL https://rethinkdb.com.

[9] Server sent events. URL https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events.

[10] R. Broeckerlmann. When to use which (oauth2) grants and (oidc) flows, 2017. URL https://medium.com/@robert.broeckelmann/when-to-use-which-oauth2-grants-and-oidc-flows-ec6a5c00d864.

[11] L. Engineering. Running kafka at scale, 2015. URL https://engineering.linkedin.com/kafka/running-kafka-scale.

[12] N. Engineering. Kafka inside keynote pipeline, 2016. URL https://medium.com/netflix-techblog/kafka-inside-keystone-pipeline-dd5aeabaf6bb.

[13] M. Glukhovsky. Rethinkdb joins the linux foundation, 2017. URL https://rethinkdb.com/blog/rethinkdb-joins-linux-foundation/.

[14] B. Svingen. Publishing with apache kafka at the new york times, 2017. URL https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/.