



VRIJE  
UNIVERSITEIT  
BRUSSEL



# **BELONTO**

Open Information Systems

**Name:** Robin De Haes  
**Student Number:** 0547560  
**Email:** robin.de.haes@vub.be

**Academic year 2021-2022**

## Abstract

*In this document we describe and justify the design and implementation of BELONTO, an open information system for booking tickets with Brussels Airlines. BELONTO was created for academic purposes in the context of the course Open Information Systems given by Prof. Bas Ketsman at Vrije Universiteit Brussel. However, the system should also support practical applications related to the booking and management of flights. We demonstrate this with some non-trivial use cases that seem to indicate correctness and usefulness of the system. Nevertheless, further extensions would still be needed for most real-world usage.*

## 1 Introduction

With the growing data availability and consensus on its usefulness, interoperability of systems and a semantic understanding of the information provided by such systems has become more important. In this context, ontologies are often proposed that formally and explicitly specify a shared conceptualization and externalize the semantics outside the information system (Gruber, 1995; Studer et al., 1998).

For the course Open Information Systems (OIS), given by Prof. Bas Ketsman at Vrije Universiteit Brussel (VUB), we designed and implemented BELONTO. BELONTO<sup>1</sup> is an ontology that supports usage for a ticket booking system for Brussels Airlines. To demonstrate the applicability of BELONTO we provided an actual implementation of a booking system in addition to designing the ontology it is based on. SPARQL queries are used to further illustrate how such a system could be used.

First we will discuss the design of our booking system in subsection 1.1, focusing on noteworthy details in our Entity Relationship Diagram (ERD). Secondly, in section 2 we delve deeper into decisions made during the design of BELONTO. Section 3 explains how we then used the database we created, based on our ERD, in combination with an R2RML mapping to generate an RDF/turtle-based version of our proposed ticket booking system. Subsequently, we demonstrate the usage of our system in section 4 by providing SPARQL queries that can solve some required use cases. Finally, we will conclude this document with a general discussion of the design process and possible future work.

### 1.1 Design of the Booking System

#### 1.1.1 ERD model

Since our system is supposed to represent one aspect of a larger system, we decided to mainly focus on the minimal requirements present in the specification and develop these as good as possible. This does not mean that our system does not support other functionalities than those mentioned in the specification, but the additional functionalities that have been included will be closely related to the ones from the minimal requirements. More elaborate functionalities that would generally be considered to be part of a different subsystem or module were considered to be out-of-scope. In the paragraphs below, we will briefly explain the most important design choices that were made. For more detailed information, we refer the reader to our ERD annotated with comments about entities and relations. This ERD is included in Appendix A.

#### Bookings & Flights

The Booking entity is directly or indirectly related to all other entities in our model which allows a user to get access to all necessary information once the booking number is known. This central position of Booking is sensible as our system revolves around booking creation and management.

A distinction should also be made between a booking of flights and a flight. We consider a flight to start once it departs from an airport and to stop once it arrives at another airport. When layovers occur, connecting flights are seen as multiple separate flights within one booking. This approach prevents having to use more weak entities in our design. Alternatively, we could have designed a Flight entity consisting of one or more FlightSegments that would be either weak entities or have an artificial key. These FlightSegments would often also only exist within

---

<sup>1</sup>BELONTO is an amalgamation of the code BEL used for Brussels Airlines by the International Civil Aviation Organization and the word ‘ontology’

the context of a Booking. That is why in our model a FlightSegment is in effect modeled as a standalone Flight but a Booking can consist of a Flight sequence that is ordered via Flight departure times. Such a sequence can be built during the booking process by potential passengers. The final price of the booking will depend on many factors, such as seats that are left at the time of booking, whether the flight is during a busy period and whether it is part of a round-trip. Therefore, total price of the sequence of flights is stored on the Booking entity as a complicated formula is generally used to compute the final price for the flights and it should not be seen as a simple sum of individual flight prices.

Outward flights and return flights are further subsumed by two separate bookings that are connected via a ‘return relation’. These two types of flights are generally booked at the same time by a passenger but they can have different flight options and management which is one of the reasons why we separate them into two bookings. In a user interface, however, they could easily be visually merged together under the booking number of the original outward flight if required. Furthermore, this approach with two separate bookings also supports use by booking intermediaries where the intermediary books two one-way tickets with different airlines but presents them to the passenger as a single round-trip booking. Although this two-bookings approach is less relevant for Brussels Airlines specifically, it is interesting from an ontology perspective. Therefore, our approach seems justified by reduced modeling complexity and broader potential use.

Finally, it is important to note that all flight timings are mentioned using Central European Time. If required we could keep track of time zones as well for each address and compute the local time. For the purposes of this project, however, this seemed superfluous.

### **Entities or Attributes**

Some entities and relations in our ERD could theoretically be modeled as composite or multi-valued attributes, since many of these entities have only one relation to another entity in our case. Some examples are the Address entity which is only used in the context of the Airport with a one-to-one relation and booking options such as PaymentOption and Insurance. However, we chose to model these as entities because they could be seen as larger meaningful physical or abstract concepts that would also be modeled as entities in other domains. Therefore, modeling them as entities allows for easier extensibility later on. Although there currently is no reuse for the Address entity and its fields could simply be included as attributes in the Airport entity, there is a high probability that other systems might use addresses as well. In this case it will be easier to connect them via an ontology if we already model an address as an entity instead of as a sum of simple attributes on a non-address entity.

In general we preferred to use simple attributes and primary key attributes. Composite attributes were only used when the composition is very small and straightforward, e.g., a name consisting of a first name and last name. We also didn’t explicitly include any derived attributes in our model to reduce complexity and improve readability. Moreover, we tried to avoid redundancy and potential inconsistencies by including all information only once. For a passenger we don’t separately keep birth date and age for example, since age can be derived from birth date. We show how to perform some more important derivations in our queries file.

### **Airline**

We designed a booking system that is specifically made for Brussels Airlines. Therefore, we chose to not explicitly include an Airline entity in our ERD model and instead the assumption should be made that entities are implicitly related to Brussels Airlines. This choice has primarily been made to reduce complexity and improve readability of the model. If we included both a Booking entity and an Airline entity, we would have had two central entities with relations to a high number of other entities. Flights are provided by a specific airline, luggage options and loyalty programs can differ per airline, a booking is with a specific airline, etc. Explicitly modeling

an Airline entity would thus bring no relevant information, since the system would only have Brussels Airlines as such an entity, while this inclusion would lead to many crossing relation lines in the ERD and thus a less clear model.

One potential downside to not including an Airline entity is that it might be harder for external systems to interpret the information from this system which makes it less ‘open’. However, in regular circumstances it seems safe to assume that other systems would know at least the source of their external data and can thus imply that the LuggageReservation from our system is specific to Brussels Airlines.

### Extensions

As mentioned before most entities, relations and attributes that are not part of the specified minimal requirements will at least be closely related to them. Since we actually don’t have any prior experience with flying, most of these extensions are based on existing options from the current booking interface from the Brussels Airlines’ website (Brussels Airlines, 2022). Below we briefly list some extensions that have been included, but more details can be found as comments in the ERD itself which has been provided in Appendix A.

- Bookings should be possible for multiple passengers at once, while only having one main contact person or booker. This means contact information, such as a unique email address, is not required for every passenger which led to the use of an artificial key to identify passengers. Relations indicating who actually made the booking and which passengers are included in a booking have also been added.
- Some passengers can be minors and at least one adult accompanying them has to be included in the booking, which necessitates including a date of birth field.
- Reservation of a specific seat is often possible during a booking, which led to the additional entities Seat and SeatReservation.
- A minimum connection time (MCT) has been added to airports to ensure scheduled connecting flights are actually feasible.
- An airline can support multiple loyalty programs, but each booking can only earn points for the card of one LoyaltyProgram or use an AwardTicket from one LoyaltyProgram.

It should be noted that our ticket booking system mostly focuses on the creation and management of bookings before the actual flight, but not on the process of flight execution. This is important as it means that we considered the check-in process, assignment of gates, etc. to be part of a different subsystem and have not included these functionalities in our model.

#### 1.1.2 Target Users

The booking system mostly focuses on three types of users with similar needs. The first type of users are potential passengers that want to search for flights that fulfill their needs and then book them. The second type of users are handling agents, working for the airline, that want to verify and perhaps modify bookings. Furthermore, the system also supports some operations of a third type of users namely an internal ‘planning agent’. This agent needs access to the number of passengers a flight currently has and the fuel cost of a flight. This information can for example be used to set the price of last-minute tickets or to (re-)assign planes based on the current popularity of a flight.

Most functionalities needed by the aforementioned users, however, are quite trivial. Seeing a simple overview of all bookings of one passenger or making small changes is useful, but have

short and straightforward queries. Therefore, we chose to not describe these kind of features that consist of rather simple selects, inserts or updates. Instead we focused on flight searches for the first user type, more complex booking verifications for the second user type and cost computations for the third user. We also show how flight-related overviews can be generated that might be useful for every type of user. On a final note, it should be mentioned that our information system can also be used in the context of data analytics by including a query to get insight in historic data for business strategy purposes. However, as the specification of the system did not seem to focus on such data analytics we mostly focused on demonstrating other use cases. A short discussion and demonstration of potential queries in the form of SPARQL queries can be found in section 4, while the SQL version of these queries have been added in the SQL file *de-haes-robin-queries.sql* that also contains more explanations and the expected output of each query.

### 1.1.3 Data Sources

Non-flight specific data, e.g. passenger names or account numbers, were mostly automatically generated by Mockaroo (2022). To have more realistic use cases, however, flight-related data such as airports and airplanes have been randomly selected from the open source tool OpenFlights (Patokallio, 2022). Furthermore, we also consulted the website of Brussels Airlines (2022) as inspiration since OpenFlights does not provide information about certain flight options. Finally, we also slightly modified some information from the aforementioned sources or even invented some data ourselves to ensure we would be able to clearly demonstrate certain functionalities of our system. In summary, the following data sources have been used:

- <https://www.mockaroo.com>
- <https://openflights.org/data.html>
- <https://www.brusselsairlines.com>

## 2 Ontology

Although we already have a working usable database system, as demonstrated by the SQL queries presented in *de-haes-robin-queries.sql*, our ticket booking system is seen as one aspect of a larger system with different aspects being developed separately. To allow better integration of these different subsystems, we developed the ontology BELONTO to turn our database system into a richer information system. However, BELONTO will still be focused on the ticket booking functionalities provided by our subsystem. Furthermore, as the name BELONTO suggests, the ontology prioritizes data integration for Brussels Airlines’ subsystems. It should allow the conversion of independently developed database subsystems into a joint federated information system, but the entities’ semantics are expected to be interpreted in the context of Brussels Airlines. Therefore, analogously to our ERD design in section 1.1, classes such as Airline are not explicitly included as there would only be one fixed individual ‘Brussels Airlines’ that does not provide much additional contextual information. In other words, BELONTO is a more specialized ontology and does not aim to be as general as possible. A visual representation of BELONTO, generated with WebVOWL (Lohmann et al., 2016), has been made available in Appendix B.

## 2.1 OWL Profile

The OWL Profile we chose for our ontology is OWL 2 QL. Although this profile is less expressive than alternatives, it seems suited for our purposes. This profile choice was based on the information provided by the World Wide Web Consortium (W3C;Calvanese et al., 2012) and the original architects of OWL 2 QL (Fikes et al., 2004).

OWL 2 QL is aimed at applications that involve querying knowledge bases with very large volumes of instance data, in which query answering is the main reasoning task. Sound, complete and relatively fast query answering is possible in OWL 2 QL. It has a LOGSPACE complexity with respect to the dataset size and it is even possible to implement the query answering by rewriting queries into SQL queries. Although the latter is currently unneeded, it could be an interesting advantage as the internal subsystems are currently expected to be SQL databases.

Our view on the integrated information system for Brussels Airlines, using BELONTO, matches the recommended application of OWL 2 QL quite well. The information system we envision would be a federated database, composed of data from multiple SQL database systems within Brussels Airlines, allowing fast queries over the combined data. Therefore, BELONTO's goal is mostly providing a semantic vocabulary to allow good data integration and it is not complex reasoning. Furthermore, it is also to be expected that the number of passengers, bookings and additional options stored in our system would lead to very large volumes of instance data which is another indication OWL 2 QL is well-suited in this context.

Alternative profiles will have more expressive power, as the expressiveness of OWL 2 QL is necessarily limited to achieve its query answering properties. However, our information system currently does not make extensive use of rule-based reasoning engines which makes efficient query answering the most important aspect. Therefore, OWL 2 QL was preferred over OWL 2 RL and OWL 2 EL. OWL 2 Full was not used as it is too expressive in general making both query answering and reasoning very complex or even undecidable. OWL 1 has not been taken into consideration as OWL 2 is the most recent recommendation of W3C adding new features based on real-world applications and user experiences which made it a better choice overall for the creation of a long-term and well-supported system for Brussels Airlines.

For more information about the restrictions put in place by OWL 2 QL, we refer to the documentation provided by W3C at [https://www.w3.org/TR/owl2-profiles/#Feature\\_Overview\\_2](https://www.w3.org/TR/owl2-profiles/#Feature_Overview_2). Generally, common OWL features that are not supported by OWL 2 QL have to do with cardinality restrictions (functional, inverse-functional, max and min cardinalities), transitive properties and key specifications. Supported features such as domain-range properties, class disjointness, inverse properties, (a)symmetry and (ir)reflexivity have been used in BELONTO where relevant. However, it should be noted that we only included these kind of properties where they were deemed appropriate or needed and sometimes left certain entities and properties less restricted to allow broader use if more subsystems would be added.

### 2.1.1 Ontology Alignment

Although our ontology is more specialized towards Brussels Airlines, we still decided to investigate the possibility of alignment with existing ontologies in case we would decide to make BELONTO more generally applicable for external use. In this context we searched for three types of ontologies, namely airline reservation ontologies, travel- or vacation-related ontologies and general ontologies providing broadly usable classes and properties.

When it comes to airline reservation ontologies, we found multiple ontologies to potentially align with but eventually chose not to use any of them for the following reasons. First, some ontologies are either meant to be seen as demonstrative examples, small academic proof-of-concepts or simply failed to gather much support and further development (Firat et al., 2003; Noy

and Musen, 1999; Vukmirovic et al., 2006). Secondly, there are ontologies that seem adequately detailed and practical when it comes to the domain of aviation, e.g., ICARUS (Stefanidis et al., 2020), but they are very recent and currently lack general adoption which makes them less interesting to align with. Thirdly, ontologies such as the Open Travel Alliance (OTA) are broad and already have real-world use but they seem to mostly provide an XML- or JSON-based communication language related to travel reservations without focusing on actual query answering or reasoning. Therefore, the goal of OTA seems different from ours. Finally, the Air Traffic Management ontology (ATMONT; M. Keller, 2019) of the US National Aeronautics and Space Administration (NASA) seems to be closest to the type of ontology we would be interested in. It is based on real-world air traffic data, is well-documented and knows some reuse in other ontologies (Stefanidis et al., 2020; Vukmirovic et al., 2006). It also provides some classes and properties that are similar to ours, such as Airport, Flight and Aircraft. However, we still decided to not use it because its design uses the OWL 2 RL profile which would make these classes incompatible with our chosen profile. Furthermore, we would still need to extend most of these classes with custom properties and the actual reuse would be quite limited. Therefore, ATMONT was not aligned with.

When it comes to travel-related ontologies, we mostly found some interesting ones about accommodation reservations (Chaves et al., 2012; Hepp, 2013). These could be useful as an extension of BELONT to make it more generally applicable in a broader domain, but such an extension is out-of-scope for now.

Lastly, we looked at some well-known general RDF vocabularies and linked data projects. Two interesting ones in this context are Friend-of-a-friend (FOAF) to describe people and relations between people and DBPedia which is an extensive ontology and knowledge base based on Wikipedia. Although DBPedia provides concepts related to almost any domain, we still decided to not use its concepts for our specialized domain of a ticket booking system for Brussels Airlines. A connection to DBPedia was deemed to be mostly useful for larger reasoning systems that can make use of DBPedia’s extensive knowledge base and external links. This is not closely related to our current goals. FOAF does provide a useful vocabulary that allows us to adequately describe passengers, which are of course people. However, it uses functional properties which is not supported in OWL 2 QL. Since we would also only use foaf:Person and its related properties, we decided to model a passenger and his attributes ourselves instead.

In summary, we decided not to align our ontology with any external ontologies for three reasons. First, our ontology is meant to be used in the more specialized domain of Brussels Airlines which sometimes makes it easier to use custom developed classes and properties to ensure correct semantics. Secondly, most useful ontologies use the profile of OWL 2 RL which is incompatible with our chosen OWL profile of OWL 2 QL. Thirdly, many external ontologies have limited long-term support and maintenance.

### 2.1.2 General design

Since we already made the ERD design of our database system quite general and our ontology is currently only based on this system, the classes, object properties and data properties of our ontology are quite similar to respectively the entities, relationships and attributes of our ERD. Following the ontology naming convention prescribed as a best practice by Bergman (2018), however, we consistently used camelcasing for classes, object properties and data properties in our ontology with classes starting with a capital letter and object properties and data properties starting with a lowercase verb.

The main difference between our ontology and the ERD design is that we are unable to add cardinality restrictions or specify properties to be functional or inverse functional. Furthermore,

the HasKey construct is also not supported in OWL 2 QL which means we could not formally specify which properties could serve as a primary key for individuals of a certain class. Nevertheless, the added semantic explanations and comments generally mention which properties can serve as a unique identifier and the primary key properties will also be used to form the IRI of an individual as explained in section 3.

The relationships of our ERD are also present as an object property in our ontology. However, the name and direction can vary. While the ERD already contains meaningful names for the relationships, we did not consider the direction that would be used most. Therefore, in our ontology we sometimes created the inverse of that relationship if it seemed more appropriate. An example is the usedFor relationship from AwardTicket to Booking. In our system it is to be expected that users generally start from the Booking and from there follow a relation to the AwardTicket. Therefore, we created a usedAwardFrom object property that goes from Booking to AwardTicket. Furthermore, sometimes both directions are likely to be used in which case we created two object properties that are each other's inverse. Going from Passenger to Booking and vice versa is equally likely for example to check who has which booking. Therefore, both hasFlightBooking and isBookedFor were included and specified to be each other's inverse. We only created inverse object properties where useful, since it can more easily lead to inconsistencies with a mistake being made in one of the directions during mapping processes. Intermediary tables from our SQL database are not explicitly present in our ontology, since many-to-many relationships can be expressed directly with an object property. One special case is AwardedPoints, which is similar to an intermediate entity, but has been kept following the W3C specification for modeling N-ary relations that have an additional attribute (Hayes and Welty, 2006). Finally, we specified whether object properties are (a)symmetric or (ir)reflexive when deemed appropriate. However, we only added these restrictions if using the object property in another way would be rather meaningless. If meaningful use of an object property without this restriction could be possible, we did not add the restriction even if this restriction would always hold in our system. We chose to do this to keep our system more extensible, although it should be noted that almost all object properties have still been specified to be asymmetric and irreflexive with the exception of hasReturnBooking for which we could imagine an extension being possible that allows this property to be used in another way.

Classes and data properties are generally the same as the entities and attributes, but some names have been modified to give them a clearer, more explicit meaning. Booking for example is named FlightBooking in our ontology and recurring attributes such as points on FrequentFlyerCard and points on AwardedPoints have been renamed to hasAccumulatedFFPoints and costsFFPoints respectively. The latter was not only done to give a more explicit readable meaning, as the meaning of the attribute is slightly different for both, but also because OWL 2 QL does not allow ObjectUnionOf constructs which complicates the reuse of the same data property for multiple classes without the creation of a superclass. We decided to create only one superclass, which is Reservation that has subclasses FlightBooking, LuggageReservation and SeatReservation. Here we chose to use a superclass for their common attributes, such as reservation ID and creation time, because they don't only have shared attributes but are also semantically related as they all are some type of reservation. Therefore, creating a hierarchy seemed appropriate here. We also added axioms saying classes are disjoint with one another, since an Airport and a Passenger can clearly never be the same individual in a meaningful way. For data properties such as price, weight and others that actually represent a quantity and a unit we decided to not create a hierarchy of QuantityUnit classes as it would increase query complexity. Instead we mention for every data property in which unit the value has to be interpreted. Although artificially made primary keys might not be very meaningful outside of our system, they could be useful if the ontology will mostly be used in the context of Brussels Airlines. Subsystems



might actually reuse the same kind of artificial keys. Therefore, we still included them as a data property instead of only using them in the IRI during the mapping process of section 3.

When it comes to data types, we limited ourselves to existing ones. Although we sometimes could have put more restrictions on certain properties, the tradeoff between the increase in complexity and the decrease in potentially incorrect values did not seem worth it. An IATA code for example currently has `xsd:string` as the data type for its range, while we know for certain that it is always alphanumeric and only three characters long. Price classes also do not consist of fixed enumerations, but are simply stated to have a range of `xsd:string`. Besides the fact that the existing data types seemed to already serve the purpose of our system adequately, this choice was also partially made because OWL 2 QL’s support of new data types is low.

On a final note, many comments added to classes and properties to further explain their semantics in our ontology are based on entries from the Oxford English Dictionary (Simpson and Weiner, 1989).

### 3 Mapping

R2RML (W3C, 2022) was used to specify a relational mapping from our database to RDF/turtle triples that can be used in the richer information system based on BELONTO. Our database system contains two types of tables. The first type of table is used to represent an actual entity from our ERD, e.g., the ‘flight’ table. The second type of table is used to represent a many-to-many relationship between two entities from our ERD, e.g., the ‘flight\_meal\_service’ table representing relations between ‘flight’ and ‘meal\_service’ entities. Instead of using JOIN statements to gather all needed information from both types of tables for completely mapping an entity, we handled both types of tables with separate triples maps as we found that to be a more readable organized approach.

First we defined a triples map for each relation of the first type. As naming convention we decided to use the plural form of the name of the ontology class, e.g., FlightBooking has a triples map called `#FlightBookings`. Since the general design of our ontology and our ERD is quite analogous, most of the times we could directly use all fields from the relational table without having to write a more specialized SQL query to process the data first. There were only three exceptions to this rule. First, we sometimes had to split up some database fields as was the case for the field ‘street’ which is divided into ‘street name’ and ‘street number’ for our ontology. Secondly, timestamps in PostgreSQL have a different format than the `xsd:dateTime` format used in our ontology. Therefore, a conversion from ‘YYYY-MM-DD hh:mm:ss’ to ‘YYYY-MM-DDThh:mm:ss’ was included. Thirdly, OWL 2 QL supports `xsd:dateTime` but not `xsd:date`. Therefore, date fields were converted to `xsd:dateTime` with the time part consisting of all zeroes. The Subject Map for each triple specified the ontology class an entity belongs to and followed a fixed naming convention for the IRI consisting of `http://www.semanticweb.org/belonto/{CLASS NAME}#{PRIMARY KEYS}`, with PRIMARY KEYS being all primary keys for that entity separated by an underscore (.). This can lead to long IRIs, but that should not be a problem in practice. All data properties can be obtained directly from a column in the table through a Predicate Object Map. Language tags have been added where relevant, although only an English version is currently provided. Relations are mapped through the use of object maps with join conditions as this provides a more flexible and maintainable mapping than directly using the template for the related individual. The latter could lead to inconsistency issues when changes are made to a template, which should be less of an issue with the approach we used.

For every relational table of the second type, we also defined a triples map. As naming convention we used a description of the relation and the entities it connects, i.e.,

#Class1RelationshipClass2Mapping for Relationship going from Class1 to Class2. A Subject Map with the same IRI for individuals of Class1 is used to ensure this relation is added to the existing triples generated by the previous mappings. However, these Subject Maps do not specify the ontology class as these have already been added by the aforementioned previous mappings. Furthermore, these tables do not encode data properties so the triples maps only contain Predicate Object Maps with object maps and join conditions.

On a final note, we would like to mention that inclusion of other databases of Brussels Airlines might lead to modifications of the described mapping. More complicated SQL queries and IRI templates could be needed to be able to combine the data coming from every system. However, currently the mappings could be done in a rather straightforward manner.

## 4 Queries

As mentioned in section 1.1.2, we chose to demonstrate our system using more non-trivial use cases instead of simple selects, inserts or updates. In this section, we will describe five such use cases and their corresponding SPARQL query. We chose queries that include every entity and relation at least once to also demonstrate coverage of our model as much as possible. More information about the queries and their expected results can be found in the file *de-haes-robin-queries.sql* and Appendix C. The same results were obtained by the SPARQL queries below, verified by executing them with the lightweight cross-platform SPARQL endpoint Oxigraph<sup>2</sup>. Although Oxigraph supports some SPARQL extensions, we limited ourselves to standard SPARQL features to ensure broad support over different SPARQL engines.

SPARQL queries allow to give back the IRI to a whole individual, but we chose to only give back the individual's fields of interest instead to more easily compare and verify the results with those of the SQL queries. Furthermore, for reasons of readability and clarity of the SPARQL queries we also chose to not use complicated property paths. Instead we used intermediate variables with a meaningful name to traverse longer paths, even though those intermediate variables have no further use. To reduce complexity and length of our queries, we also did not specify the class to which each variable should belong since this could often already be implied by the used data and object properties. Finally, in our queries we sometimes had to perform data type conversions to ensure correct execution. OWL 2 QL supports owl:real but does not support xsd:float, while SPARQL does not support arithmetic with owl:real. Therefore, we converted owl:real values inside our queries. Although Oxigraph does actually support the datatype xsd:duration, this data type and certain date and time arithmetic is not supported in standard SPARQL. Therefore, we chose to convert our xsd:dateTime values to integer timestamps to be able to execute arithmetic operations on them and convert them to a custom string format representing an interval afterwards. This makes our queries longer and a bit more complicated, but ensures adherence to the SPARQL standard and thus broader support by SPARQL engines. Specific remarks for each SPARQL query have also been added as comments in the queries presented below. Moreover, these queries have also been provided in the file *de-haes-robin-sparql.txt* for readability purposes.

### 4.1 Flight Searches

Potential passengers might want to perform advanced flight searches to find flights that fulfill all their needs. As a use case we present a query in which a passenger is looking to find a round-trip in October between Brussels and New York for a stay of at least 5 days. It should have a

---

<sup>2</sup><https://github.com/oxigraph/oxigraph>

maximum total duration of 16 hours (for the outward and return flights) and should have a first class window seat available with a vegan meal option.

This query should adequately demonstrate the type of advanced searches that are possible with our system to book a flight and covers many entities. This query also involves the computation of the number of seats of each type on a plane, which can be seen as a derived attribute of Airplane.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX belonto: <http://www.semanticweb.org/belonto#>

SELECT
    ?outFlightDescription
    ?outFlightDesignator
    ?outDepartureDateTime
    ?outArrivalDateTime
    ?outFlightDuration

    ?returnFlightDescription
    ?returnFlightDesignator
    ?returnDepartureDateTime
    ?returnArrivalDateTime
    ?returnFlightDuration
WHERE {
    # Both the outward and return flight have common constraints, but using
    # CONSTRUCT to first create a "view" that is then nested inside a SELECT
    # query is currently not supported. Therefore, a lot of similar statements
    # are made for both the outward and return flight.
    # Later, both flights are connected with each other by additional
    # constraints.

    # relevant flight information that we want to show
    ?outFlight a belonto:Flight .
    ?outFlight belonto:hasFlightDesignator ?outFlightDesignator .
    ?outFlight belonto:hasDepartureDateTime ?outDepartureDateTime .
    ?outFlight belonto:hasArrivalDateTime ?outArrivalDateTime .

    ?outFlight belonto:departsFrom ?outDepartureAirport .
    ?outFlight belonto:arrivesAt ?outArrivalAirport .
    ?outDepartureAirport belonto:hasIATACode ?outDepartureAirportIATA .
    ?outArrivalAirport belonto:hasIATACode ?outArrivalAirportIATA .
    # formatted textual representation of the flight
    BIND(CONCAT(?outDepartureAirportIATA, " -> ", ?outArrivalAirportIATA) AS
    ↪ ?outFlightDescription ) .

    ?returnFlight a belonto:Flight .
    ?returnFlight belonto:hasFlightDesignator ?returnFlightDesignator .
    ?returnFlight belonto:hasDepartureDateTime ?returnDepartureDateTime .
```

```

?returnFlight belonto:hasArrivalDateTime ?returnArrivalDateTime .

?returnFlight belonto:departsFrom ?returnDepartureAirport .
?returnFlight belonto:arrivesAt ?returnArrivalAirport .
?returnDepartureAirport belonto:hasIATACode ?returnDepartureAirportIATA .
?returnArrivalAirport belonto:hasIATACode ?returnArrivalAirportIATA .
BIND(CONCAT(STR( ?returnDepartureAirportIATA ), " -> ", STR(
  ↳ ?returnArrivalAirportIATA )) AS ?returnFlightDescription ) .

# xsd:duration is not supported in standard SPARQL, so for interval
# comparisons we translate dateTime to a timestamp (starting from the year 0)
BIND(((((YEAR(?outDepartureDateTime) * 365 +
  ↳ MONTH(?outDepartureDateTime)) * 12) + DAY(?outDepartureDateTime)) * 24)
  ↳ + HOURS(?outDepartureDateTime)) * 60) + MINUTES(?outDepartureDateTime))
  ↳ * 60 + SECONDS(?outDepartureDateTime) AS ?outDepartureTS)
BIND(((((YEAR(?outArrivalDateTime) * 365 + MONTH(?outArrivalDateTime)) *
  ↳ 12) + DAY(?outArrivalDateTime)) * 24) + HOURS(?outArrivalDateTime)) *
  ↳ 60) + MINUTES(?outArrivalDateTime)) * 60 + SECONDS(?outArrivalDateTime)
  ↳ AS ?outArrivalTS)

BIND(((((YEAR(?returnDepartureDateTime) * 365 +
  ↳ MONTH(?returnDepartureDateTime)) * 12) +
  ↳ DAY(?returnDepartureDateTime)) * 24) +
  ↳ HOURS(?returnDepartureDateTime)) * 60) +
  ↳ MINUTES(?returnDepartureDateTime)) * 60 +
  ↳ SECONDS(?returnDepartureDateTime) AS ?returnDepartureTS)
BIND(((((YEAR(?returnArrivalDateTime) * 365 +
  ↳ MONTH(?returnArrivalTime)) * 12) + DAY(?returnArrivalDateTime)) *
  ↳ 24) + HOURS(?returnArrivalDateTime)) * 60) +
  ↳ MINUTES(?returnArrivalDateTime)) * 60 + SECONDS(?returnArrivalDateTime)
  ↳ AS ?returnArrivalTS)

# xsd:duration is not supported in standard SPARQL and neither is computing
# date differences, so instead we compute the difference between the
# timestamps and convert it to a meaningful string representation.
BIND((?outArrivalTS - ?outDepartureTS) AS ?outDurationTS)
BIND(FLOOR(?outDurationTS/(24*3600)) AS ?outDays)
BIND(FLOOR((?outDurationTS - (?outDays * 24)) / 3600) AS ?outHours)
BIND(FLOOR((?outDurationTS - (((?outDays * 24) + ?outHours) * 3600)) / 60)
  ↳ AS ?outMins)
BIND((?outDurationTS - (((?outDays * 24) + ?outHours) * 60) + ?outMins) *
  ↳ 60) AS ?outSecs)
BIND(CONCAT(STR(?outDays), "D", STR(?outHours), "H", STR(?outMins), "M",
  ↳ STR(?outSecs), "S") AS ?outFlightDuration)

BIND((?returnArrivalTS - ?returnDepartureTS) AS ?returnDurationTS)
BIND(FLOOR(?returnDurationTS/(24*3600)) AS ?returnDays)
BIND(FLOOR((?returnDurationTS - (?returnDays * 24)) / 3600) AS ?returnHours)

```

```

BIND(FLOOR((?returnDurationTS - (((?returnDays * 24) + ?returnHours) *
→ 3600)) / 60) AS ?returnMins)
BIND((?returnDurationTS - (((?returnDays * 24) + ?returnHours) * 60) +
→ ?returnMins * 60)) AS ?returnSecs)
BIND(CONCAT(STR(?returnDays), "D", STR(?returnHours), "H", STR(?returnMins),
→ "M", STR(?returnSecs), "S") AS ?returnFlightDuration)

# the flights should offer a vegan meal
?outFlight belonto:offersMealService ?outMealService .
?outMealService belonto:offersMealType "vegan"@en .

?returnFlight belonto:offersMealService ?returnMealService .
?returnMealService belonto:offersMealType "vegan"@en .

# we compute the number of first class seats the planes of each flight have
# (there should be at least one such seat to be a valid flight)
{
  SELECT ?outFlight (COUNT(*) AS ?outAvailableSeats)
  WHERE {
    ?outFlight belonto:isFlownWithPlane ?outPlane .
    ?outPlane belonto:containsSeat ?outSeat .
    ?outSeat belonto:hasSeatClass "first"@en .
  } GROUP BY ?outFlight
}

{
  SELECT ?returnFlight (COUNT(*) AS ?returnAvailableSeats)
  WHERE {
    ?returnFlight belonto:isFlownWithPlane ?returnPlane .
    ?returnPlane belonto:containsSeat ?returnSeat .
    ?returnSeat belonto:hasSeatClass "first"@en .
  } GROUP BY ?returnFlight
}

# we compute the number of first class seats that are already booked on each
# flight because these are not available anymore
{
  SELECT ?outFlight (SUM(IF(BOUND(?outBooking), 1, 0)) AS ?outBookedSeats)
  WHERE {
    ?outFlight a belonto:Flight .

    # OPTIONAL since it is possible no seats have been booked so far
    OPTIONAL {
      ?outFlight belonto:isBookedWith ?outBooking .
      ?outBooking belonto:hasBookingClass "first"@en .
      ?outBooking belonto:isBookedFor ?outPassenger .
    }
  } GROUP BY ?outFlight
}

```

```

{
  SELECT ?returnFlight (SUM(IF(BOUND(?outBooking),1,0)) AS
    ↪ ?returnBookedSeats)
  WHERE {
    ?returnFlight a belonto:Flight .

    OPTIONAL {
      ?returnFlight belonto:isBookedWith ?returnBooking .
      ?returnBooking belonto:hasBookingClass "first"@en .
      ?returnBooking belonto:isBookedFor ?returnPassenger.
    }
  } GROUP BY ?returnFlight
}

# we compute the number of window seats that are still available on each
# flight, i.e., for which a seat reservation does not exist yet
{
  SELECT ?outFlight (COUNT(*) AS ?outRemainingWindowSeats)
  WHERE {
    ?outFlight belonto:isFlownWithPlane ?outPlane .
    ?outPlane belonto:containsSeat ?outSeat .
    ?outSeat belonto:hasSeatClass "first"@en .
    ?outSeat belonto:hasSeatType "window"@en .

    # check if the seat is already reserved
    FILTER NOT EXISTS {
      ?outFlight belonto:isBookedWith ?outBooking .
      ?outBooking belonto:containsSeatReservation ?outSeatReservation .
      ?outSeatReservation belonto:reserverSeat ?outSeat .
    }
  } GROUP BY ?outFlight
}

{
  SELECT ?returnFlight (COUNT(*) AS ?returnRemainingWindowSeats)
  WHERE {
    ?returnFlight belonto:isFlownWithPlane ?returnPlane .
    ?returnPlane belonto:containsSeat ?returnSeat .
    ?returnSeat belonto:hasSeatClass "first"@en .
    ?returnSeat belonto:hasSeatType "window"@en .

    FILTER NOT EXISTS {
      ?returnFlight belonto:isBookedWith ?returnBooking .
      ?returnBooking belonto:containsSeatReservation ?returnSeatReservation .
      ?returnSeatReservation belonto:reserverSeat ?returnSeat .
    }
  } GROUP BY ?returnFlight
}

```

```

# we are looking for round-trip flights between BRU and JFK
FILTER(?outDepartureAirportIATA = "BRU" && ?outArrivalAirportIATA = "JFK")
FILTER(?returnDepartureAirport = ?outArrivalAirport && ?returnArrivalAirport
↪ = ?outDepartureAirport)

# we want to travel in October 2022 (i.e., the 10th month of 2022)
FILTER(YEAR(?outDepartureDateTime) = YEAR(?outArrivalDateTime) &&
↪ YEAR(?outDepartureDateTime) = 2022
&& MONTH(?outDepartureDateTime) = MONTH(?outArrivalDateTime) &&
↪ MONTH(?outDepartureDateTime) = 10)

FILTER(YEAR(?returnDepartureDateTime) = YEAR(?returnArrivalDateTime) &&
↪ YEAR(?returnDepartureDateTime) = 2022
&& MONTH(?returnDepartureDateTime) =
↪ MONTH(?returnArrivalDateTime) &&
↪ MONTH(?returnDepartureDateTime) = 10)

# the maximum duration of each flight should be below 16 hours
FILTER((?outArrivalTS - ?outDepartureTS) < (16 * 3600))
FILTER((?returnArrivalTS - ?returnDepartureTS) < (16 * 3600))

# the passenger wants to stay at the destination for at least 5 days
FILTER(?returnDepartureTS > (?outArrivalTS + (5 * 24 * 3600)))

# there should be at least 1 first class seat still available on each flight
FILTER((?outAvailableSeats - ?outBookedSeats) > 0)
FILTER((?returnAvailableSeats - ?returnBookedSeats) > 0)

# not all window seats should be reserved
FILTER(?outRemainingWindowSeats > 0)
FILTER(?returnRemainingWindowSeats > 0)
}
ORDER BY ?outDepartureDateTime ?returnDepartureDateTime

```

## 4.2 Overbooking Mitigation

Airlines generally allow overbookings up to a certain percentage, because it is possible passengers do not show up on the day of the flight. However, if too many seats of a certain price class have been booked a handling agent might need to upgrade a passenger. To decide which passenger should be upgraded, the number of frequent flyer points could be used. As a use case we present a query in which we are looking for a passenger to upgrade. This passenger should have an economy seat on an overbooked flight and should have the most frequent flyer points for a specific loyalty program that actually offers upgrades like this. However, he cannot be a minor or the only adult accompanying minor(s) since multiple people should be upgraded in that case.

This query should adequately demonstrate how a handling agent could use our system to help him upgrade, change or otherwise manage flight bookings.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

```

```

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX belonto: <http://www.semanticweb.org/belonto#>

SELECT
  ?passengerID
  ?email
  ("Miles & More"@en AS ?loyaltyProgramName)
  ?points
  ?bookingDateTime
WHERE {
  ?passenger a belonto:Passenger .
  ?passenger belonto:hasPassengerID ?passengerID .
  ?passenger belonto:hasFlightBooking ?booking .
  ?booking belonto:hasReservationID ?bookingNumber .

  # we are checking a specific flight
  ?flight belonto:hasFlightDesignator "SN1510" .
  ?flight belonto:hasDepartureDateTime ?departureDT .
  ?booking belonto:containsFlight ?flight .

  # only the booker (i.e. main contact is required to provide an email address)
  # so if the selected passenger has no direct contact information, return the
  # main contact's information
  ?booking belonto:isBookedBy ?booker .
  ?booker belonto:hasEmail ?bookerEmail .
  OPTIONAL
  {
    ?passenger belonto:hasEmail ?passengerEmail .
  }
  BIND(IF(BOUND(?passengerEmail), ?passengerEmail, ?bookerEmail) AS ?email) .

  # economy is overbooked, so we are looking for passengers with such a booking
  ?booking belonto:hasBookingClass "economy"@en .

  # we will check later whether the loyalty program they have a frequent flyer
  # card from supports upgrades to first class (because business class is
  # already fully booked as well)
  ?passenger belonto:hasFrequentFlyerCard ?ffc .
  ?ffc belonto:allowsParticipationInLoyaltyProgram ?loyaltyProgram .
  ?loyaltyProgram belonto:hasLoyaltyProgramName ?loyaltyProgramName .
  ?loyaltyProgram belonto:offersAwardTicket ?awardTicket .
  ?awardTicket belonto:awardsUpgradeType "first class"@en.

  # the passenger with most points and earliest booking time gets priority
  # (i.e., these values will be used for ordering)
  ?ffc belonto:hasAccumulatedFFPoints ?points .
  ?booking belonto:isReservedOn ?bookingDateTime .

```



```

# we compute the age of the passenger to know whether he's a minor or an
# adult (with a minor being under 12 and an adult being at least 18)
?passenger belonto:hasBirthDate ?birthDate .
# Age can be computed by computing the difference between the years and then
# subtracting one year if the passenger's birthday is in a later month or on
# a later day in the same month in the departure year
BIND((YEAR(?departureDT) - YEAR(?birthDate)) - (IF((MONTH(?departureDT) <
→ MONTH (?birthDate)) || ((MONTH(?departureDT) = MONTH(?birthDate)) &&
→ (DAY(?departureDT) < DAY(?birthDate))), 1, 0)) AS ?age )

# we won't give an upgrade to an adult if it's the only adult accompanying a
# minor
{
  SELECT ?booking ?departureDT
  WHERE {
    ?ePassenger a belonto:Passenger .
    ?ePassenger belonto:hasFlightBooking ?booking .
    ?ePassenger belonto:hasBirthDate ?eBirthDate .

    BIND("2022-08-20T11:30:00"^^xsd:dateTime AS ?departureDT)
    BIND((YEAR(?departureDT) - YEAR(?eBirthDate)) - (IF((MONTH(?departureDT)
→ < MONTH (?eBirthDate)) || ((MONTH(?departureDT) =
→ MONTH(?eBirthDate)) && (DAY(?departureDT) < DAY(?eBirthDate))), 1,
→ 0)) AS ?eAge )
  }
  GROUP BY ?booking ?departureDT
  # to be eligible you should not be accompanying a minor or there should be
  # more than 1 adult accompanying the minor(s)
  HAVING (SUM(if(?eAge >= 18, 1, 0)) > 1 || SUM(if(?eAge < 12, 1, 0)) = 0)
}

# we won't give an upgrade to minors (because they have to be accompanied by
# their adult)
FILTER(?age >= 12)
}

# the passengers with more points and earlier booking time get priority
ORDER BY DESC(?points) ?bookingDateTime

```

### 4.3 Booking Price & Awarded Points

A passenger should be able to see how much a booking costs in total, including all subparts such as flight fares, luggage registrations, insurance and seat reservations. As an example use case we show how all the necessary information can be gathered and presented in a decomposed way starting from just a specific booking number. Additionally, we will also show the method of payment and the amount of frequent flyer points that have been awarded for this booking.

This query should adequately demonstrate how both passengers and handling agents can get an organized view of relevant booking information with our system.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

```

```

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX belonto: <http://www.semanticweb.org/belonto#>

# We will perform 2 subqueries that compute the price per passenger for
# different parts of the bill for the outward and return flight, with parts of
# the bill being insurance, luggage, flight cost, etc.
# The results of these 2 subqueries will then be combined to compute the total
# cost of the booking.
SELECT
  ?bookingNumber
  ?bookerID
  # the total cost of the booking can be computed by summing up all the
  # separate parts for all the passengers in the booking
  (SUM(?insuranceCost)
   + SUM(?outFlightCost)
   + SUM(?outLuggageCost)
   + SUM(?outSeatCost)
   + SUM(?returnFlightCost)
   + SUM(?returnLuggageCost)
   + SUM(?returnSeatCost) AS ?totalPrice)
  ?paymentType
  ?loyaltyProgramName
  ?points
WHERE {
  ?booking a belonto:FlightBooking .
  ?booking belonto:hasReservationID ?bookingNumber .
  # we keep track of the actual booker, since it is the passenger that will be
  # awarded the frequent flyer miles (in case of a group booking)
  ?booking belonto:isBookedBy ?booker .
  ?booker belonto:hasPassengerID ?bookerID .
  # we will show how everything was paid for
  ?booking belonto:isPaidWith ?paymentOption .
  ?paymentOption belonto:hasPaymentType ?paymentType .

  # subquery for costs related to the outward flight of the booking, grouped
  # per passenger
  {
    SELECT
      ?passenger
      ?booking
      ?insuranceCost
      # the total luggage cost for a passenger is the sum of the costs for
      # each type of luggage
      (SUM(?outBagsCost) AS ?outLuggageCost)
      ?outFlightCost
      ?outSeatCost
    WHERE {
      ?passenger a belonto:Passenger .

```

```

?passenger belonto:hasFlightBooking ?booking .

# the price for the actual flight itself
# (i.e. transportation of the passenger) is stored on the booking;
# owl:real is not a supported datatype in standard SPARQL, so we convert
# it to xsd:float to perform arithmetic later
?booking belonto:hasFlightsPrice ?outFlightsPrice .
BIND(xsd:float(xsd:string(?outFlightsPrice)) AS ?outFlightCost)

# OPTIONAL is used since all the following parts are not required to be
# present in a booking

# the group insurance is only charged to the booker (and only once for
# the whole round-trip);
# owl:real is not a supported datatype in standard SPARQL, so we convert
# it to xsd:float to perform arithmetic later and we use a conditional
# BIND statement instead of COALESCE since it allows for more
# straightforward GROUP BY statements (grouping would otherwise have to
# be on ?insurancePrice instead of ?insuranceCost)
OPTIONAL
{
  ?booking belonto:isBookedBy ?passenger .
  ?booking belonto:isInsuredBy ?insurance .
  ?insurance belonto:hasInsurancePremium ?insurancePrice .
}
BIND(IF(BOUND(?insurancePrice), xsd:float(xsd:string(?insurancePrice)),
↪ "0.0"^^xsd:float) AS ?insuranceCost)

# the price for bringing luggage can be computed by summing up (the
# number of bags we have of each type) times (the cost for that type of
# bag) ;
# we already compute the products used in the sum here while summing up
# happens in the SELECT statement
OPTIONAL
{
  ?passenger belonto:hasLuggageReservation ?luggageReservation .
  ?booking belonto:containsLuggageReservation ?luggageReservation .
  ?luggageReservation belonto:reservesLuggageService ?luggageService .
  ?luggageReservation belonto:reservesNumberOfBags ?outNumberOfBags .
  ?luggageService belonto:hasPricePerBag ?outPricePerBag .
}
BIND(IF(BOUND(?outPricePerBag), xsd:float(xsd:string(?outPricePerBag)) *
↪ ?outNumberOfBags, "0.0"^^xsd:float) AS ?outBagsCost)

# the price for reserving a seat is stored via a seat reservation
OPTIONAL
{
  ?passenger belonto:getsSeatVia ?outSeatReservation .
  ?booking belonto:containsSeatReservation ?outSeatReservation .

```

```

        ?outSeatReservation belonto:hasSeatPrice ?outSeatPrice .
    }
    BIND(IF(BOUND(?outSeatPrice), xsd:float(xsd:string(?outSeatPrice)),
        ↪ "0.0"^^xsd:float) AS ?outSeatCost)
}
# except for luggage cost, all prices are unique and can therefore be
# safely included in the GROUP BY statement
GROUP BY ?passenger ?booking ?insuranceCost ?outFlightCost ?outSeatCost
}

# subquery for costs related to the return flight of the booking, grouped
# per passenger;
# same computations should occur (except insurance which is only paid once),
# but everything is optional as the return flight itself is optional as well
{
    SELECT
        ?passenger
        ?booking
        ?returnFlightCost
        (SUM(?returnBagsCost) AS ?returnLuggageCost)
        ?returnSeatCost
    WHERE {
        ?passenger a belonto:Passenger .
        ?passenger belonto:hasFlightBooking ?booking .

        # belonto:hasReturnBooking is included in every OPTIONAL statement
        # because the next OPTIONAL statements would otherwise not be related to
        # the original booking when there is no return booking as the first
        # OPTIONAL statement would not create a binding for ?returnBooking
        OPTIONAL
        {
            ?booking belonto:hasReturnBooking ?returnBooking .
            ?returnBooking belonto:hasFlightsPrice ?returnFlightsPrice .
        }
        BIND(IF(BOUND(?returnFlightsPrice),
            ↪ xsd:float(xsd:string(?returnFlightsPrice)), "0.0"^^xsd:float) AS
            ↪ ?returnFlightCost)

        OPTIONAL {
            ?booking belonto:hasReturnBooking ?returnBooking .
            ?passenger belonto:hasLuggageReservation ?returnLuggageReservation .
            ?returnBooking belonto:containsLuggageReservation
                ↪ ?returnLuggageReservation .
            ?returnLuggageReservation belonto:reservesLuggageService
                ↪ ?returnLuggageService .
            ?returnLuggageReservation belonto:reservesNumberOfBags
                ↪ ?returnNumberOfBags .
            ?returnLuggageService belonto:hasPricePerBag ?returnPricePerBag .
        }
    }
}

```

```

    BIND(IF(BOUND(?returnPricePerBag),
      ↪ xsd:float(xsd:string(?returnPricePerBag)) * ?returnNumberOfBags,
      ↪ "0.0"^^xsd:float) AS ?returnBagsCost)

    OPTIONAL {
      ?booking belonto:hasReturnBooking ?returnBooking .
      ?passenger belonto:getsSeatVia ?returnSeatReservation .
      ?returnBooking belonto:containsSeatReservation ?returnSeatReservation .
      ?returnSeatReservation belonto:hasSeatPrice ?returnSeatPrice .
    }
    BIND(IF(BOUND(?returnSeatPrice),
      ↪ xsd:float(xsd:string(?returnSeatPrice)), "0.0"^^xsd:float) AS
      ↪ ?returnSeatCost)
  }
  GROUP BY ?passenger ?booking ?returnFlightCost ?returnSeatCost
}

# OPTIONAL is also used for retrieving the awarded points, since having an
# eligible frequent flyer card is not required
OPTIONAL {
  ?booker belonto:hasFrequentFlyerCard ?frequentFlyerCard .
  ?booking belonto:earnsFFPoints ?awardedPoints .
    ?awardedPoints belonto:consistsOfFFPoints ?points .
  ?awardedPoints belonto:isAddedToFFCard ?frequentFlyerCard .
  ?frequentFlyerCard belonto:allowsParticipationInLoyaltyProgram
    ↪ ?loyaltyProgram .
  ?loyaltyProgram belonto:hasLoyaltyProgramName ?loyaltyProgramName .
}

# all computations are made for a specific booking
# (for clarity we limited the computation to one specific order, but a
# correct and complete overview per booking would be given otherwise) ;
# replace the filter below by
# FILTER NOT EXISTS { ?outBooking belonto:hasReturnBooking ?booking . }
# to get an overview for all bookings
FILTER(?bookingNumber = "B00000000017")
}

# all these fields are unique and can therefore be safely included in the
# GROUP BY statement
GROUP BY ?bookingNumber ?bookerID ?paymentType ?loyaltyProgramName ?points

```

## 4.4 Cost Computations

Multiple planes might be available for a certain flight, but not all of them might be equally cost-effective. To be able to efficiently compare planes for a flight an airline agent would benefit from being able to find eligible planes and estimate their fuel cost. Although in reality more factors than fuel cost are used to decide which plane is chosen for a flight, we will present an example use case that filters planes on eligibility and then orders them only on fuel cost in order to make a decision. Eligibility of a plane for a specific flight is dependent on its speed, seating

capacity and the maximum distance it can cover.

This query should adequately demonstrate how an airline agent could use our system to decide which planes to assign to certain flights.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX belonto: <http://www.semanticweb.org/belonto#>

SELECT ?tailNumber ?aircraftType ?fuelCost
WHERE {
  # this subquery counts the number of booked seats per price class
  # (i.e., the minimum number of seats we need)
  {
    SELECT
      ?flight
      (SUM(if(?bookingClass = "economy"@en, 1, 0)) AS ?minEconomy)
      (SUM(if(?bookingClass = "business"@en, 1, 0)) AS ?minBusiness)
      (SUM(if(?bookingClass = "first"@en, 1, 0)) AS ?minFirst)
    WHERE {
      ?booking a belonto:FlightBooking .
      ?booking belonto:hasBookingClass ?bookingClass .
      # one booking can be for multiple passengers
      ?booking belonto:isBookedFor ?passenger .

      # we are counting the seats for a specific (known) flight
      ?booking belonto:containsFlight ?flight .
      ?flight belonto:hasFlightDesignator "SN4508" .
      ?flight belonto:hasDepartureDateTime "2022-10-01T13:15:00"^^xsd:dateTime
      ↪ .
    }
    GROUP BY ?flight
  }

  # this subquery counts the number of seats per price class that are
  # available on an airplane
  {
    SELECT
      ?airplane
      (SUM(if(?seatClass = "economy"@en, 1, 0)) AS ?planeEconomy)
      (SUM(if(?seatClass = "business"@en, 1, 0)) AS ?planeBusiness)
      (SUM(if(?seatClass = "first"@en, 1, 0)) AS ?planeFirst)
    WHERE {
      ?airplane a belonto:Airplane .
      ?airplane belonto:containsSeat ?seat .
      ?seat belonto:hasSeatClass ?seatClass .
    }
    GROUP BY ?airplane
  }
}
```

```

# we want to display tail number and type in the results
?airplane belonto:hasTailNumber ?tailNumber .
?airplane belonto:hasAircraftType ?aircraftType .

# we will use flight and airplane information to put some constraints on the
# eligibility of airplanes
?flight belonto:hasFlightDistance ?flightDistance .
?flight belonto:hasDepartureDateTime ?flightDeparture .
?flight belonto:hasArrivalDateTime ?flightArrival .
?airplane belonto:hasFuelCapacity ?fuelCapacity .
?airplane belonto:hasFuelEconomy ?fuelEconomy .
?airplane belonto:hasCruisingSpeed ?cruisingSpeed .

# owl:real is not a supported datatype in standard SPARQL,
# so we convert it to xsd:float to perform arithmetic later
BIND(xsd:float(xsd:string(?flightDistance)) AS ?flightDistanceF)
BIND(xsd:float(xsd:string(?fuelCapacity)) AS ?fuelCapacityF)
BIND(xsd:float(xsd:string(?fuelEconomy)) AS ?fuelEconomyF)
BIND(xsd:float(xsd:string(?cruisingSpeed)) AS ?cruisingSpeedF)

# xsd:duration is not supported in standard SPARQL, so for comparisons with
# the interval between flight departure and arrival we translate dateTime to
# a timestamp (starting from the year 0)
BIND((((((((YEAR(?flightDeparture) * 365 + MONTH(?flightDeparture)) * 12) +
  ↳ DAY(?flightDeparture)) * 24) + HOURS(?flightDeparture)) * 60) +
  ↳ MINUTES(?flightDeparture)) * 60 + SECONDS(?flightDeparture) AS
  ↳ ?flightDepartureTS)
BIND((((((((YEAR(?flightArrival) * 365 + MONTH(?flightArrival)) * 12) +
  ↳ DAY(?flightArrival)) * 24) + HOURS(?flightArrival)) * 60) +
  ↳ MINUTES(?flightArrival)) * 60 + SECONDS(?flightArrival) AS
  ↳ ?flightArrivalTS)

# we use a different (arbitrarily chosen) price for a liter of kerosene and
# gasoline so we can use that price to compute the fuel cost of the plane
?airplane belonto:hasFuelType ?fuelType .
BIND(IF(?fuelType = "kerosene"@en, (?flightDistanceF / ?fuelEconomyF) * 0.4,
  ↳ (?flightDistanceF / ?fuelEconomyF) * 1.4) AS ?fuelCost) .

# the airplane should have enough economy, business and first class seats
# available to cover the existing bookings
FILTER((?planeEconomy > ?minEconomy) && (?planeBusiness > ?minBusiness) &&
  ↳ (?planeFirst > ?minFirst))
# the airplane should have enough fuel to cover the required distance
FILTER(?fuelCapacityF > (?flightDistanceF / ?fuelEconomyF))
# the airplane should be able to travel fast enough so it won't arrive late
FILTER(((?flightDistanceF / ?cruisingSpeedF) * 3600) < (?flightArrivalTS -
  ↳ ?flightDepartureTS))
}

```

```

# we are currently mainly using fuel cost to make our final decision,
# so we order on fuel cost
ORDER BY ?fuelCost

```

## 4.5 Flight Overview

Getting an overview of possible flights that are currently offered in combination with some properties can be beneficial for both airline agents and potential passengers. As a use case we provide an overview of all flights that leave from Brussels, with flights being either direct or with layovers. This overview is useful for airline agents to get a good view on the current offer or it can serve as inspiration for passengers that want to travel but don't have a specific destination in mind yet. Additionally, the type of flight will also be included to clearly indicate if a flight is continental or intercontinental.

This query should adequately demonstrate how useful knowledge can be extracted from our system to inspire decisions or get a better view of what is currently available which might influence future decisions. For in-depth data analysis this overview could be extended with additional information about popularity of flights for example. This query also involves the computation of the flight type, i.e., continental or intercontinental, which can be seen as a derived attribute of Flight.

On a final note, it is important to mention that indirect flights have been limited to only one layover. Although SPARQL queries more easily support cycles of flexible length, this is not so straightforward for SQL queries.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX belonto: <http://www.semanticweb.org/belonto#>

# Results will be obtained as the union of 2 queries.
# The first query searches for all direct flights
# and the second query searches for flights with one layover.
SELECT DISTINCT ?flightDescription ?flightType ?hasLayovers
WHERE {
  # direct flights
  {
    ?flight a belonto:Flight .

    # get the origin and destination airport of the direct flight
    # (and their IATA codes)
    ?flight belonto:departsFrom ?originAirport .
    ?originAirport belonto:hasIATACode ?originCode .
    ?flight belonto:arrivesAt ?destinationAirport .
    ?destinationAirport belonto:hasIATACode ?destinationCode .

    # formatted textual representation of the flight
    BIND(CONCAT(STR( ?originCode ), " -> ", STR( ?destinationCode )) AS
    ↪ ?flightDescription ) .

    # this query will find direct flights

```



```

    BIND("false"^^xsd:boolean AS ?hasLayovers) .
}

UNION

# flights with layovers
{
    # flight1 is the flight to the layover destination
    ?flight1 a belonto:Flight .
    # flight2 is the flight from the layover stop to the final destination
    ?flight2 a belonto:Flight .

    # get the origin, intermediate and destination airport
    # of the direct flight (and their IATA codes)
    ?flight1 belonto:departsFrom ?originAirport .
    ?originAirport belonto:hasIATACode ?originCode .
    ?flight1 belonto:arrivesAt ?intermediateAirport .
    ?flight2 belonto:departsFrom ?intermediateAirport .
    ?intermediateAirport belonto:hasIATACode ?intermediateCode .
    ?flight2 belonto:arrivesAt ?destinationAirport .
    ?destinationAirport belonto:hasIATACode ?destinationCode .

    # get the arrival time, departure time and the layover airport's
    # minimum connection time to ensure both flights can be connected safely
    ?flight1 belonto:hasArrivalDateTime ?intermediateArrivalDateTime .
    ?flight2 belonto:hasDepartureDateTime ?intermediateDepartureDateTime .
    ?intermediateAirport belonto:hasMCT ?mct .

    # xsd:duration is not supported in standard SPARQL, so to compare with MCT
    # later on we translate dateTime to a timestamp (starting from the year 0)
    # to later compare the difference between departure and arrival with MCT
    BIND(((((((YEAR(?intermediateArrivalDateTime) * 365 +
        ↪ MONTH(?intermediateArrivalDateTime)) * 12) +
        ↪ DAY(?intermediateArrivalDateTime)) * 24) +
        ↪ HOURS(?intermediateArrivalDateTime)) * 60) +
        ↪ MINUTES(?intermediateArrivalDateTime)) * 60 +
        ↪ SECONDS(?intermediateArrivalDateTime) AS ?intermediateArrivalTS)
    BIND(((((((YEAR(?intermediateDepartureDateTime) * 365 +
        ↪ MONTH(?intermediateDepartureDateTime)) * 12) +
        ↪ DAY(?intermediateDepartureDateTime)) * 24) +
        ↪ HOURS(?intermediateDepartureDateTime)) * 60) +
        ↪ MINUTES(?intermediateDepartureDateTime)) * 60 +
        ↪ SECONDS(?intermediateDepartureDateTime) AS ?intermediateDepartureTS)

    # formatted textual representation of the flight
    BIND(CONCAT(STR( ?originCode ), " -> ", STR(?intermediateCode), " -> ",
        ↪ STR( ?destinationCode )) AS ?flightDescription ) .

    # this query will find flights with layovers

```

```

BIND("true"^^xsd:boolean AS ?hasLayovers) .

# round-trip flights are excluded as that stop is not considered a layover
FILTER(?originCode != ?destinationCode) .

# the layover flight should depart after the initial flight arrives
FILTER(?intermediateDepartureDateTime > ?intermediateArrivalDateTime) .

# for two flights to be considered as a possible connection in 1 larger
# encompassing flight, a limit of maximally 20 hours between them has been
# specified (which was an arbitrary choice by us)
FILTER((?intermediateDepartureTS - ?intermediateArrivalTS) < (20 * 3600)) .

# the layover airport's minimum connection time should allow both flights
# to be connected safely
FILTER((?mct * 60) < (?intermediateDepartureTS - ?intermediateArrivalTS)) .
}

# a flight is continental if it has source and target airport in the same
# continent (independent of whether the intermediate airport is in another
# continent for simplicity sake), otherwise it is intercontinental
?originAirport belonto:hasAddress ?originAddress .
?destinationAirport belonto:hasAddress ?destinationAddress .
?originAddress belonto:isLocatedInContinent ?originContinent .
?destinationAddress belonto:isLocatedInContinent ?destinationContinent .
BIND(IF(?originContinent = ?destinationContinent, "continental"^^xsd:string,
↪ "intercontinental"^^xsd:string) AS ?flightType) .

# only consider flights leaving from BRU
FILTER (?originCode = "BRU")

} ORDER BY ?flightType ?flightDescription

```

## 5 Discussion

Going from data to a rich information system involves technologies, tools and design decisions on multiple levels that all have their own benefits and drawbacks. The development of the ontology BELONTO and the related ticket booking system for Brussels Airlines clearly shows the impact a decision on one level can have on the rest of the process. We will briefly discuss the made decisions and their consequences.

First we had to make an initial design for a PostgreSQL database system, based on a textual specification of some minimal requirements. In our case this step was probably the most time-consuming one for three reasons. The first reason is that we are not experienced airplane travelers which means we had to do some prior research to adequately understand the domain so we would be able to build a system that is actually useful, that satisfies the requirements and is also general enough to allow extensions if needed. Secondly, we should acknowledge the fact that our entity relationship diagram and database system was created knowing that an ontology and information system would be based on it in a next phase. This led to a more thoughtful design as early

decisions would heavily influence the subsequent phases. Thirdly, our design differs a bit from a traditional database design. Since integration and openness of the system was paramount, we focused on using as many natural keys and meaningful entities as possible from the start to more easily decide if an individual is the same in different systems. If we only focused on developing a database system, we might have used more artificial identifiers and other connections for ease of use in our particular context. This might lead to less time being spent on the initial design, but could also lead to more issues during generalization or integration.

After creating a PostgreSQL database, SQL queries were written to demonstrate and verify its functionalities. In practice, this step was quite intertwined with the design of the entity relationship diagram. Insights in the domain obtained during the initial design led to the idea for a useful SQL query which in turn led to changes being made to the initial design if some entities or attributes were still missing. In practice, generating and gathering enough data to get useful results and verifying these results took more time than writing the queries themselves. We had a lot of entities and relations, so care had to be taken to ensure our data is consistent and our results are correct.

Following the verification of our database system, we developed the ontology BELONTO to augment it and facilitate potential integration with other subsystems. As previously stated, our initial design already tried to be as meaningful and informative as possible which made the creation of an ontology based on our entity relationship diagram straightforward. Furthermore, we did not actually have multiple database systems to integrate which further facilitates ontology design as conflicting database design choices are absent. Therefore, most work actually went into investigating the pros and cons of different OWL profiles and researching existing ontologies. Although OWL 2 QL seemed most appropriate for our specific use cases, it complicates alignment with other ontologies as many ontologies are written using a more expressive OWL profile. The choice for using OWL 2 QL thus might need to be reconsidered if alignment with other ontologies would be required. Moreover, if more complex reasoning would be needed OWL 2 QL might not suffice either.

After the development of BELONTO, we used R2RML to map our existing database system to a triple store based on our ontology. Just like designing the ontology, creating this mapping was straightforward for two main reasons. Our entity relationship diagram was already close to our ontology design and we only had one database system which means real conflicts or the need for merging was generally absent. Therefore, most work during the mapping design had to do with formatting differences between our database system and the available ontology formats. These differences, however, were not so difficult to overcome. We would also like to note that the common OWL profiles only seem to support `xsd:dateTime` and not `xsd:date` or `xsd:time`. This was not a problem for our design, but we can see how this might be problematic for generating mappings in some other cases.

Finally, we translated our SQL queries to SPARQL queries to ensure our BELONTO information system can still provide the same functionalities as our original database system. Although we succeeded in this effort, a couple of remarks should be made in this context. First, we would like to mention that in a lot of cases SPARQL allowed to easily go from one individual to another using a simple property path, while in SQL multiple tables and JOIN conditions were involved. However, in general the SPARQL standard currently seems quite limited in what it supports. Our SQL queries used WITH-statements to create and then reuse ‘views’ in a SELECT-statement. This was not possible with the SPARQL standard which led to repetition of nested SELECT statements and therefore longer queries. Furthermore, date and time arithmetic is also not well-supported in standard SPARQL. Comparisons of date and time are possible, but everything related to durations and intervals needed a lot of additional logic. It was also surprising that OWL 2 QL supports `owl:real` instead of `xsd:float` or `xsd:double`, while

SPARQL cannot perform computations with owl:real. This is easily solved by performing a data type conversion in SPARQL, but this discrepancy seemed counterintuitive as OWL 2 QL focuses on allowing efficient querying behavior and SPARQL is often used in this context. However, we should also note that there are many SPARQL engines available with extensions for date and time arithmetic, for computations with owl:real and even with support of WITH statements. Nevertheless, these extensions are currently not part of the SPARQL standard.

In conclusion, the process from data gathering to having an information system supporting SPARQL querying knows some difficulties but intelligent design during the initial phases makes this process manageable. We were able to create a system that demonstrates practical applications for the booking and management of flights for Brussels Airlines. Nevertheless, BELONTO and the related system should be seen as an academic project. Actual integration with existing systems of Brussels Airlines would lead to more conflicts that have to be solved and a generalization or extension of BELONTO would definitely be needed. Furthermore, we are not domain experts and used dummy data which might cause parts of our design to inadequately represent some real-world use cases.

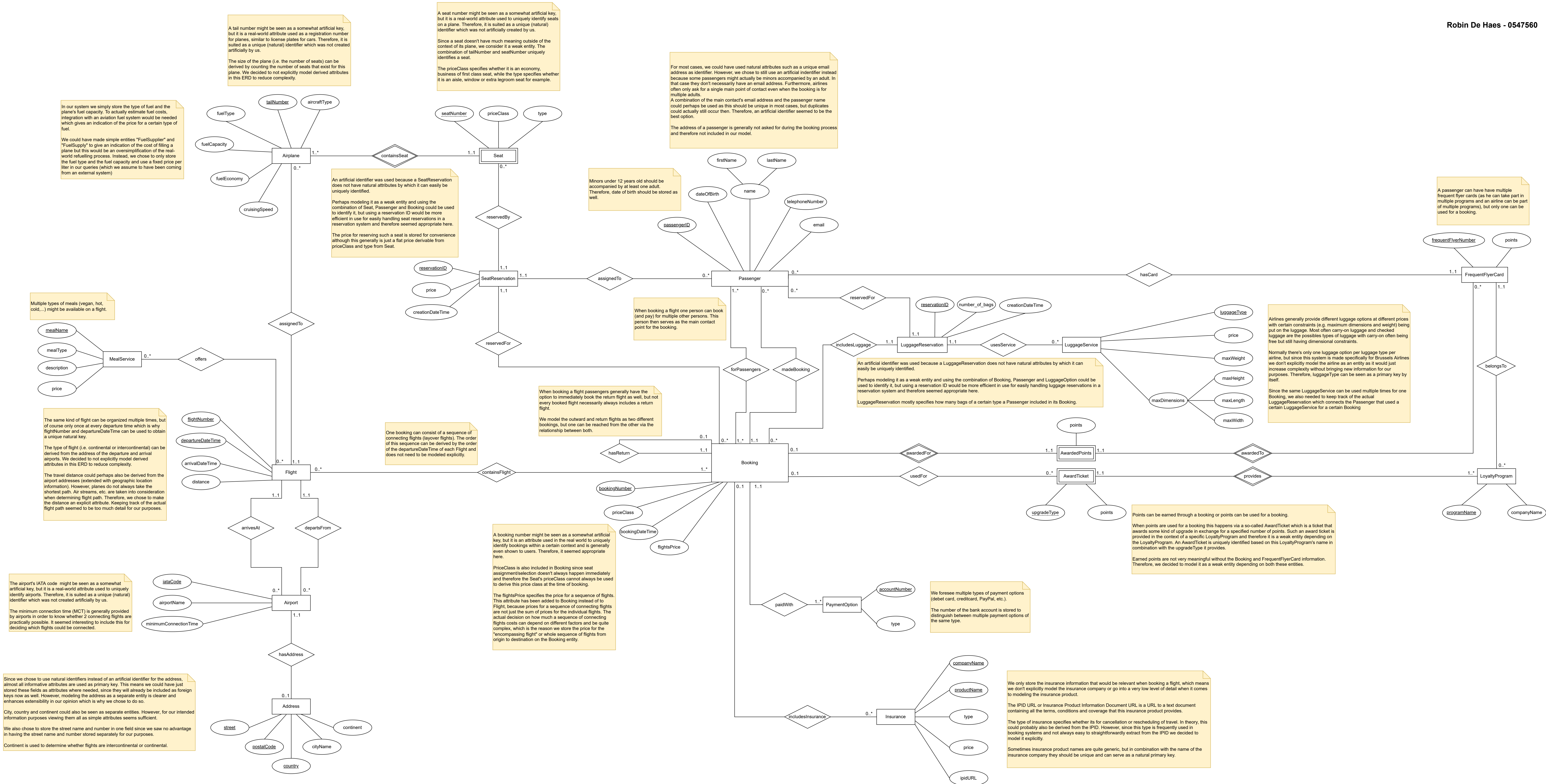
## References

- Bergman, M. K., Bergman, M. K., & Lagerstrom-Fife. (2018). *Knowledge representation practicality*. Springer.
- Brussels Airlines. (2022). Brussels airlines. Retrieved July 26, 2022, from <https://www.brusselsairlines.com>
- Calvanese, D., Carroll, J., De Giacomo, G., Hendler, J., Herman, I., Parsia, B., Patel-Schneider, P., Ruttenberg, A., Sattler, U., & Schneider, M. (2012). OWL 2 Web Ontology Language Profiles (Second Edition). Retrieved August 1, 2022, from <https://www.w3.org/TR/owl2-profiles/>
- Chaves, M., Freitas, L., & Vieira, R. (2012). Hontology: A multilingual ontology for the accommodation sector in the tourism industry.
- Fikes, R., Hayes, P., & Horrocks, I. (2004). OWL-QL — a language for deductive query answering on the semantic web. *Journal of Web Semantics*, 2(1), 19–29. <https://doi.org/https://doi.org/10.1016/j.websem.2004.07.002>
- Firat, A., Kaleem, M., Lee, P., Madnick, S., Moulton, A., Siegel, M., & Zhu, H. (2003). Context interchange (coin) system demonstration.
- Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5-6), 907–928.
- Hayes, P., & Welty, C. (2006). Defining n-ary relations on the semantic web. Retrieved August 1, 2022, from <https://www.w3.org/TR/swbp-n-aryRelations/>
- Hepp, M. (2013). Accommodation Ontology Language Reference. Retrieved August 1, 2022, from <http://ontologies.sti-innsbruck.at/acco/ns.html>
- Lohmann, S., Negru, S., Haag, F., & Ertl, T. (2016). Visualizing ontologies with VOWL. *Semantic Web*, 7(4), 399–419. <https://doi.org/10.3233/SW-150200>
- M. Keller, R. (2019). Building a Knowledge Graph for the Air Traffic Management Community. *Companion Proceedings of The 2019 World Wide Web Conference*, 700–704. <https://doi.org/10.1145/3308560.3317706>
- Mockaroo. (2022). Mockaroo - Random Data Generator and API Mocking Tool. Retrieved July 28, 2022, from <https://www.mockaroo.com/>
- Noy, N., & Musen, M. (1999). Smart: Automated support for ontology merging and alignment.
- Patokallio, J. (2022). OpenFlights.org: Flight logging, mapping, stats and sharing. Retrieved July 28, 2022, from <https://openflights.org/>
- Simpson, J., & Weiner, E. (Eds.). (1989). *The Oxford English Dictionary* (Second Edition). Oxford University Press.
- Stefanidis, D., Christodoulou, C., Symeonidis, M., Pallis, G., Dikaiakos, M., Pouis, L., Orphanou, K., Lampathaki, F., & Alexandrou, D. (2020). The icarus ontology: A general aviation ontology developed using a multi-layer approach. *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*, 21–32. <https://doi.org/10.1145/3405962.3405983>
- Studer, R., Benjamins, V. R., & Fensel, D. (1998). Knowledge engineering: Principles and methods. *Data & knowledge engineering*, 25(1-2), 161–197.
- Vukmirovic, M., Szymczak, M., Ganzha, M., & Paprzycki, M. (2006). Utilizing ontologies in an agent-based airline ticket auctioning system. *28th International Conference on Information Technology Interfaces, 2006.*, 385–390.
- W3C. (2022). R2RML: RDB to RDF Mapping Language. Retrieved August 7, 2022, from <https://www.w3.org/TR/r2rml/>

## Appendices

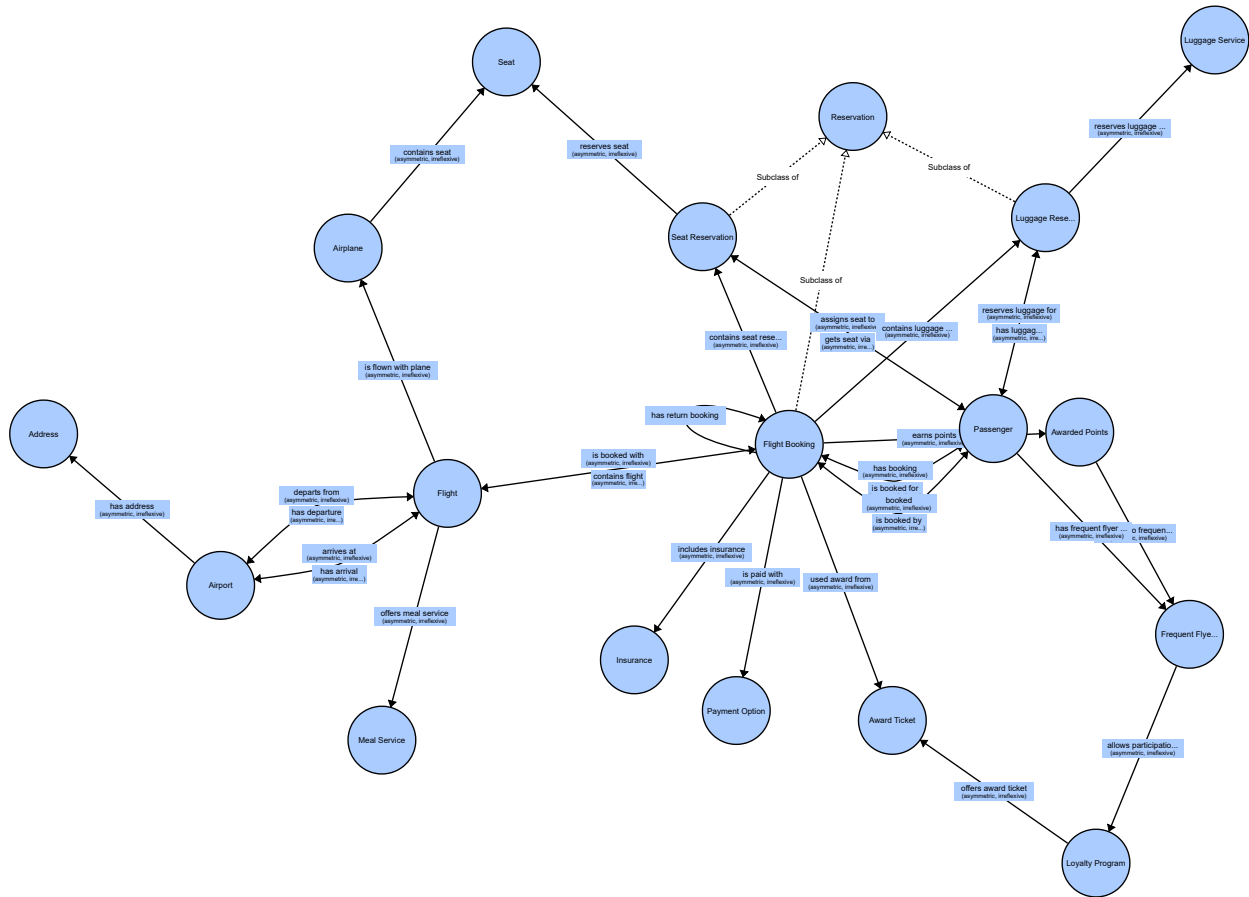
## A Entity Relationship Diagram







## **B WebVOWL visualization**



## C Query Results

Data Output	Explain	Messages	Notifications
out_flight text	out_flight_nr character varying (32)	out_departure timestamp without time zone	out_arrival timestamp without time zone
1 BRU->JFK	SN5555	2022-10-03 10:00:00	2022-10-04 01:30:00
2 BRU->JFK	SN5555	2022-10-03 10:00:00	2022-10-04 01:30:00
3 BRU->JFK	SN5555	2022-10-10 10:00:00	2022-10-11 01:30:00

Figure 1: Results of the ‘Flight Searches’ SQL query.

Table

Response

3 results in 0.073 seconds

Simple view

☒ Ellipse

☐ Filter query results

Page size: 50

outFlightDesc...	outFlight...	outDepartureDateTime	outArrivalDateTime	outFlightDuration	returnFlightDesc...	returnFlight...	returnDepartureDateTime	returnArrivalDateTime	returnFlightDuration
BRU -> JFK	SN5555	2022-10-03T10:00:00	2022-10-04T01:30:00	0D15H30M0S	JFK -> BRU	SN5556	2022-10-10T08:00:00	2022-10-10T23:30:00	0D15H30M0S
BRU -> JFK	SN5555	2022-10-03T10:00:00	2022-10-04T01:30:00	0D15H30M0S	JFK -> BRU	SN5556	2022-10-17T08:00:00	2022-10-17T23:30:00	0D15H30M0S
BRU -> JFK	SN5555	2022-10-10T10:00:00	2022-10-11T01:30:00	0D15H30M0S	JFK -> BRU	SN5556	2022-10-17T08:00:00	2022-10-17T23:30:00	0D15H30M0S

Showing 1 to 3 of 3 entries

<

1

>

Figure 2: Results of the ‘Flight Searches’ SPARQL query.







Data Output	Explain	Messages	Notifications	
 passenger_id character varying (32) 	email character varying (128) 	loyalty_program_name character varying (64) 	points integer 	booking_date_time timestamp without time zone 
1	P00000074324	tdhae@outlook.com	Miles & More	25000 2022-08-01 14:44:23
2	P00000074323	lsimco1@dmoz.org	Miles & More	25000 2022-08-02 13:55:23
3	P00000074315	sclewlw2@google.co.uk	Miles & More	15000 2022-08-07 20:29:11
4	P00000074314	rdh91@msn.com	Miles & More	10000 2022-08-11 18:13:29
5	P00000074321	hpardon8@oracle.com	Miles & More	5000 2022-07-15 09:21:23
6	P00000074331	Helgaaah@outlook.com	Miles & More	5000 2022-08-10 11:33:26
7	P00000074313	arustion6@wunderground.com	Miles & More	3000 2022-08-07 22:16:53
8	P00000074501	lsimco1@dmoz.org	Miles & More	1000 2022-08-02 13:55:23

Figure 3: Results of the ‘Overbooking Mitigation’ SQL query.

Table

Response

8 results in 0.017 seconds

Simple view

Ellipse

Filter query results

Page size: 50

passengerID	email	bookingNumber
P00000074324	tdhae@outlook.com	B00000000006
P00000074323	lsimco1@dmoz.org	B00000000007
P00000074315	sclewlw2@google.co.uk	B00000000012
P00000074314	rdh91@msn.com	B00000000014
P00000074321	hpardon8@oracle.com	B00000000004
P00000074331	Helgaaah@outlook.com	B00000000013
P00000074313	arustion6@wunderground.com	B00000000009
P00000074501	lsimco1@dmoz.org	B00000000007

Showing 1 to 8 of 8 entries

<1>

Figure 4: Results of the ‘Overbooking Mitigation’ SPARQL query.

Data Output		Explain	Messages	Notifications			
	<div>booking</div> <div>character varying (32)</div>		<div>booker</div> <div>character varying (32)</div>	<div>total_price</div> <div>numeric</div>	<div>payment</div> <div>character varying (64)</div>	<div>loyalty_program</div> <div>character varying (64)</div>	<div>awarded_points</div> <div>integer</div>
1	B00000000017		P00000074333	1636.13	maestro	Miles & More	4513

Figure 5: Results of the ‘Booking Price & Awarded Points’ SQL query.

	Table	Response	1 result in 0.012 seconds	Simple view	Ellipse	Filter query results	Page size: 50		
	<b>bookingNumber</b>	<b>bookerID</b>	<b>totalPrice</b>	<b>paymentType</b>	<b>loyaltyProgramName</b>	<b>points</b>			
	B00000000017	P00000074333	1636.13	maestro	Miles & More	4513			
Showing 1 to 1 of 1 entries									
								<	>

Figure 6: Results of the ‘Booking Price & Awarded Points’ SPARQL query.

	Data Output	Explain	Messages	Notifications
	<b>tail_number</b> [PK] character varying (32)	<b>aircraft_type</b> character varying (64)	<b>fuel_cost</b> numeric	
1	O-BAJJ	Airbus A330-300	5172.31	
2	O-BAHG	Airbus A330-300	5603.33	
3	O-BAJE	Airbus A330-300	5603.33	
4	O-BAJF	Airbus A330-300	5603.33	

Figure 7: Results of the ‘Cost Computations’ SQL query.

	Table	Response	4 results in 0.014 seconds	Simple view	Ellipse	Filter query results	Page size: 50		
	<b>tailNumber</b>	<b>aircraftType</b>	<b>fuelCost</b>						
	O-BAJJ	Airbus A330-300	5172.308						
	O-BAHG	Airbus A330-300	5603.3335						
	O-BAJE	Airbus A330-300	5603.3335						
	O-BAJF	Airbus A330-300	5603.3335						
Showing 1 to 4 of 4 entries									
								<	>

Figure 8: Results of the ‘Cost Computations’ SPARQL query.





	Data Output	Explain	Messages	Notifications
	 flight text	 flight_type text	 has_layovers boolean	
1	BRU -> HAM	continental	false	
2	BRU -> LED	continental	false	
3	BRU -> SOF	continental	false	
4	BRU -> STN	continental	false	
5	BRU -> TLS	continental	false	
6	BRU -> HAM -> JFK	intercontinental	true	
7	BRU -> HAM -> MEL	intercontinental	true	
8	BRU -> JFK	intercontinental	false	
9	BRU -> MTY	intercontinental	false	

Figure 9: Results of the ‘Flight Overview’ SQL query.

Table

Response

9 results in 0.024 seconds

Simple view ☒ Ellipse ☐

Filter query results

Page size: 50

Download

Help

flightDescription	flightType	hasLayovers
BRU -> HAM	continental	false
BRU -> LED	continental	false
BRU -> SOF	continental	false
BRU -> STN	continental	false
BRU -> TLS	continental	false
BRU -> HAM -> JFK	intercontinental	true
BRU -> HAM -> MEL	intercontinental	true
BRU -> JFK	intercontinental	false
BRU -> MTY	intercontinental	false

Showing 1 to 9 of 9 entries

< 1 >

Figure 10: Results of the ‘Flight Overview’ SPARQL query.