INTERDEPARTMENTAL PROGRAMME OF POSTGRADUATE STUDIES
(I.P.P.S.) IN INFORMATION SYSTEMS

MSc Dissertation

JOINING LINKED OPEN DATA CUBES

by

IOANNIS M. ZISIS

Submitted as a prerequisite in fulfillment of the requirements for the acquisition of the postgraduate degree in Information Systems

February 2016

Abstract

The subject of this thesis is “Joining Linked Open Data Cubes” and deals with the investigation of applying traditional join types found in relational algebra to Linked Data Cubes. The investigation is based on a review of the multidimensional structures, of Linked Data and the join types defined in relational algebra. All the details of these three areas are used to conceptualize the way a join can happen between two Linked Data Cubes. The concepts that are established to achieve the join of Cubes are proven by implementing a program and using it against a dataset with Linked Open Data Cubes. The results of the program execution are analyzed revealing some interesting findings.

Περίληψη

Το θέμα αυτής της εργασίας λέγεται “Συνδυάζοντας Κύβους Ανοιχτών Διασυνδεδεμένων Δεδομένων” και ασχολείται με την έρευνα για την εφαρμογή παραδοσιακών τύπων συνδυασμού της σχεσιακής άλγεβρας σε Κύβους Διασυνδεδεμένων Δεδομένων. Η έρευνα στηρίζεται σε ανασκόπηση των πολυδιάστατων δομών, των Διασυνδεδεμένων Δεδομένων και της σχεσιακής άλγεβρας. Οι λεπτομέρειες αυτών των τριών περιοχών χρησιμοποιούνται για να οριστούν οι έννοιες για την πραγμάτωση συνδυασμού μεταξύ δύο Κύβων Διασυνδεδεμένων Δεδομένων. Οι έννοιες αυτές αποδεικνύονται με την ανάπτυξη ενός προγράμματος και τη χρήση του σε ένα σύνολο Κύβων Ανοιχτών Διασυνδεδεμένων Δεδομένων. Τα αποτελέσματα της εκτέλεσης του προγράμματος αυτού αναδεικνύουν μερικά ενδιαφέροντα ευρήματα.

Contents

1 Introduction

1.1 Problem statement

Open data become available in larger volumes lately by many organizations and government agencies. The provision of open data though is not supervised by strict rules, allowing various formats and structures. Examples of such formats are text documents, spreadsheets and web pages. These structures may be well suited for consumption and study by humans but have proven to be poor for machine processing.

In order to cope with the limits posed by the lack of strict rules, there is a trend to structure open data by using the linked data concept. This concept is based on the usage of common vocabularies for describing resources. Additionally it provides the ability to frame the data in a generic model using RDF [1] and the data can be easily available online since they are stored by using HTTP URI [2].

A large number of collections of open data are of statistical nature, with historical significance, justifying their storage in data warehouses. More specifically it is natural to store such data using the multidimensional data model which allows performing OLAP on them. The multidimensional data model is also known as multidimensional cube, OLAP cube, hypercube, data Cube or even just Cube. The terms data Cube or Cube will be used throughout the rest of the text. The concept of multidimensional data structure is attributed by many to Kimball, but Kimball himself has the opinion that probably this concept cannot be credited to a single person [3, p. 15]. OLAP (On-Line Analytical Processing) on the other hand, provides a set of operations which can be applied on a multidimensional Cube giving better insight to the data. Although the operations were used by a lot of people implicitly, the term was coined by Codd et al. [4].

The concepts of linked data and data Cube are combined in a framework described with the RDF Data Cube Vocabulary [5]. This framework provides the means to model and log data that need to be in the form of a multidimensional Cube, while being linked data. Consequently a multidimensional Cube consisting of open data is named a linked open data Cube.

The data Cube concept is mature enough with many studies in this area. The linked data concept and even more the linked data Cube concept is newer, providing space for more

studies. Considering also the diversity of information which exists in linked open data Cubes, there is motivation to investigate for operations on these Cubes which will reveal additional value.

1.2 Research scope and objective

Having available various linked open data Cubes which store different observed or measured information, interest arises to combine this information as an operation between the Cubes. The objective of this thesis is to achieve the combination of linked open data Cubes by studying the various join types taken from the relational algebra. The scope is focused on how the natural join can be applied both theoretically and practically, on two sample linked open data Cubes.

1.3 Structure of the thesis

The thesis is structured in a way to provide all the concepts needed and to pinpoint all the details that need attention, up to the point where a working example is in place.

In chapter 2 the multidimensional data model and the linked data Cubes are described. It is important to present a clear view on the forms of multidimensional data in order to succeed in the join operations. Chaudhuri et al. [6] split the multidimensional data models into two major categories. One category is called ROLAP, coming from relational OLAP and the other category is MOLAP, coming of multidimensional OLAP. According to Kimball and Ross [3] a ROLAP multidimensional data model is best described as a “star schema” with a fact table in the core which contains all observations or measured values that need to be stored. The fact table is surrounded by dimensions and every tuple in the fact table is connected to all dimensions. Each dimension is a relation with attributes which enrich the information stored in each fact. Additionally each dimension may have a hierarchy which places every tuple of the dimension under a different level of granularity. An extension of the star schema is the “snowflake schema” which normalizes further the dimensions by forming separate relations out of them. On the other hand MOLAP applies the multidimensional rational in the implementation using special data structures like multidimensional arrays to construct the data store. Whether a multidimensional data model is based on ROLAP or MOLAP, conceptually the key components are the *dimensions*, the *measures* (corresponding to fact tables in ROLAP) and the *attributes* on dimensions. The attributes may also be used to support the hierarchy levels for the domain of a dimension. In the context of OLAP

and multidimensional data models the work and the studies about combining/integrating data Cubes will be presented. As an example Datta and Thomas [7] provide algebraic formulations on the operations that can be performed on a multidimensional such as Cartesian product, and join.

Although the conceptual description of the multidimensional data model is generic, it is equally important to have a clear definition of how a linked data Cube is structured. Linked data Cubes are highly based on linked data [1], [8] and the RDF Data Cube Vocabulary [5]. According to this vocabulary a linked data Cube is structured with a set of *dimensions*, a set of *measures* and a set of *attributes*. Each dimension and attribute can usually take values from code lists with finite elements, avoiding free text or arbitrary values. This is feasible by reusing other vocabularies of linked data or defining custom code lists. Additionally the values of a dimension may have a hierarchy scheme providing levels of granularity on its values. The actual placeholder for data is the *observation*. Each observation has values for each dimension, measure and attribute, in a way that the values of each dimension act as coordinates and point to the this observation which stores measured or observed values for all measures. One important difference between the multidimensional data model and linked data Cubes is the concept of attributes. In the multidimensional data model the attributes are bound to dimensions. On the other hand in linked data cube model the attributes have equal weight to dimensions or measures and they are bound either to observations or to the entire Cube, but not to dimensions. This difference between the multidimensional data model and the linked data Cube model pose the first challenge in the attempt to join linked data Cubes.

In chapter 3 a methodology is described to achieve the objective. The methodology contains the clear steps that are followed to fulfill the goal of this thesis. As the scope of this work is the investigation of applying join types from relational algebra to the linked open data Cubes, it becomes essential to provide documentation around them in chapters 4 and 5. The concepts of relational algebra were first offered in a publication by E. F. Codd [9], and later on they were analyzed in more detail and enriched with findings and studies by newer publications in a book by the same author, called “The Relational Model for Database Management” [10] and another book by C. J. Date, called “An Introduction to Database Systems” [11]. These texts provide all the needed information regarding relational algebra, going into details like relational databases and SQL which could be considered beyond the scope of this thesis. Surely they are not the

exhaustive list of information but they cover the need to source the documentation and study on the various join types of the relational algebra. The next step in the methodology is to apply the join types on linked data Cubes. Since the linked data Cubes are multidimensional constructions another challenge appears on how to operate in multiple dimensions while the join operation is defined only for the two-dimensional relations in relational algebra. This is achieved by flattening the Cube and fitting its structure and contents in a two-dimensional array, looking like a relation. Each join type is evaluated and natural join is chosen for further investigation.

In chapter 6 the natural join is analyzed in detail and is chosen as the type of join for further study. The study is done in a depth which can establish a solid framework to be used during any attempt to perform natural join between two linked data Cubes. The framework consist of some constraints that must hold for the input Cube, a structural consideration of the result Cube and a metric that can be useful to for the join evaluation.

Chapter 7 and 8 apply the findings of chapter 6 on real linked open data Cubes. The available linked open data Cubes are stored in RDF stores and the way to extract data from graphs in such stores is by querying them using the SPARQL [12] language. So at this point queries are formed through a program and applied on the linked open data Cubes using the facilities of Java and SPARQL. These queries aim to extract data which can tell whether two Cubes are joinable and to provide the metric values for evaluation of their possible join. This programmatic implementation on real data will prove and support the theoretical assertion.

2 Literature review

2.1 Introduction

This chapter presents the work on the related fields of interest. Initially the multidimensional data model is described. Also a review is done about the existing work on integrating multidimensional structures. Then the linked data Cube model is described completing the information needed for the rest of the work.

2.2 Multidimensional Data Cubes

Data not being current in a business activity needed to be stored and kept, creating the Data Warehouse idea. Data stock in Data Warehouses soon revealed the need to invent structures that would provide both data integrity but also the ability to view them under various perspectives for continuous study, serving the business intelligence goals.

Chaudhuri et al. [6] in a review describe that there are 2 basic categories of Data Warehouse implementations. One is based on relational OLAP (ROLAP) and the other is based on multidimensional OLAP (MOLAP). ROLAP is described in great detail in Kimball and Ross's book [3] and the relational model is used to build the Data Warehouse schema. In the center of such a relational schema is one relation, the fact table, which as the name implies holds all the measured or observed facts that need to be stored in the Data Warehouse. This schema comes in two forms; one is the star schema and the other is the snowflake schema. The star schema has relations around the fact table which constitute the dimensions of the facts. The surrounding dimensions connect to the central fact table with foreign keys as the relational model orders. An example of such a star schema is shown in Figure 1. The Snowflake schema is based on the star schema and refines further the dimension relations. It follows the relational model requirements for normalized schemas, so in the example of Figure 2 the Customer relation could connect to another relation called Location in place of Region and City attributes, and Date could provide the day of week, week and quarter information in a separate relation called Dateinfo. The normalization of dimension attributes, which is also described as "snowflaking" by Kimball and Ross [3], allows for more accurate presentation of hierarchical data.

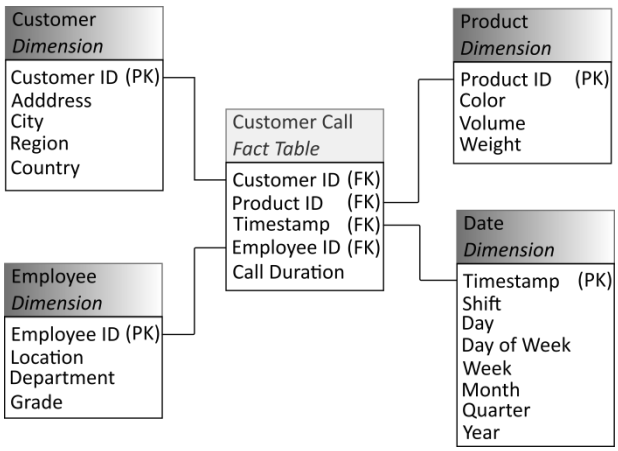

Figure 1: Star schema

Figure 2 shows a snowflake schema example.

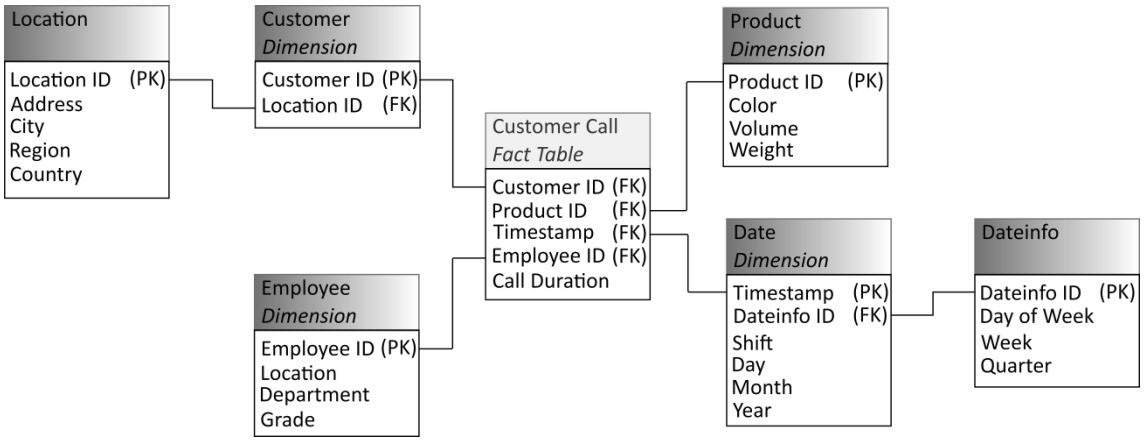

Figure 2: Snowflake schema

MOLAP is the other alternative to store data in a multidimensional manner. MOLAP does not lie on a higher level of abstraction like the relational model, but instead it exploits low level implementation details like sparse arrays, advanced indexing and hashing techniques for fast data acquisition. Additionally MOLAP implementations achieve faster response time because they apply precomputation or pre-aggregation of information which is ready available for delivery as it is described by Pedersen and Jensen in [13]. MOLAP is closer to the Cube concept, meaning a multidimensional structure as the one shown in Figure 3.

Regardless of the internal implementation, both ROLAP and MOLAP are multidimensional models providing as output a conceptual multidimensional Cube with common key components:

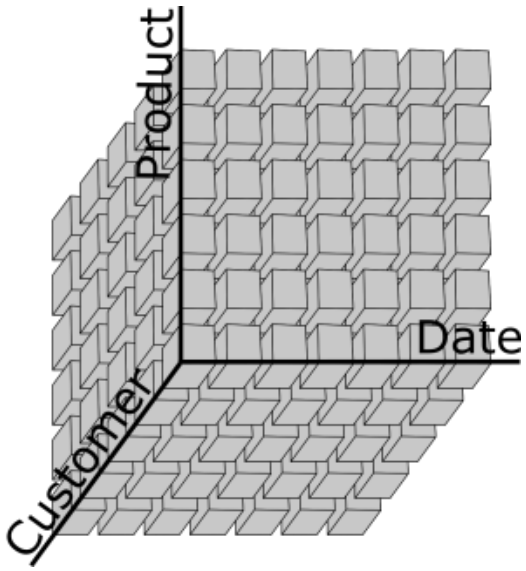

Figure 3: Data cube

- Dimensions
- Measures
- Attributes on dimensions
- Hierarchies

The key components of the multidimensional cube concept are organized as it is shown in the diagram in Figure 4.

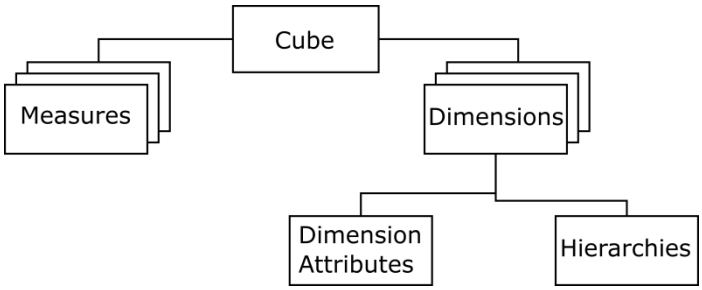

Figure 4: Data cube key components

The final goal of the multidimensional model is to provide a means to the user to perform analytical operations on the data, which are known as OLAP. The main operations of OLAP are:

- *Roll-up*: This operation aggregates measured values on a requested level in a hierarchy of a dimension.

- *Drill-down*: This operation looks into measured values which correspond to more fine grained level in a hierarchy of a dimension, being the opposite pf roll-up.
- *Slicing / dicing*: This operation slices a part of the data Cube, by keeping the measured values, which correspond to a subset of one dimension. Dicing occurs by viewing the slice (or the whole Cube) from different perspectives focusing on few dimensions each time.

The multidimensional model with its key components and the OLAP operations performed on multidimensional stores are the foundations of higher level operations such as data mining and business intelligence, which aim at extracting information out of big stock of data and drawing conclusions by analysis on raw numbers.

After describing the concepts around the multidimensional model, it is feasible to provide also the related work in the field of joining data cubes.

One of the earliest definitions of joining multidimensional data structures is found in [14] by Agrawal et al. In this work the join operation relates two Cubes and it is based on a number of dimensions which are called the *join dimensions*. The join dimensions do not need to be same as there is the possibility to perform transformation on them. Another key concept in this join is the need for a function called f_{elem} which is applied between the observed values at the input Cubes and the result is written in the relevant cell in the output Cube. This f_{elem} function is the one that relates the joined Cubes. This kind of join is not very clear how it applies on the non-join dimensions. The example provided in the paper considers two Cubes *C* and *C1* and f_{elem} is the division of the value in a cell in *C* by the value in *C1* for all the cells that are filtered and kept from *C* as the join dimension *D1* states. A similar example is provided in Figure 5. In case there are no dimensions taken as *join dimensions* then this type of join is considered to be a Cartesian product between the cubes. Another special case appears if all dimensions of one Cube are joined with some of the dimensions the other Cube. This is called an associate join.

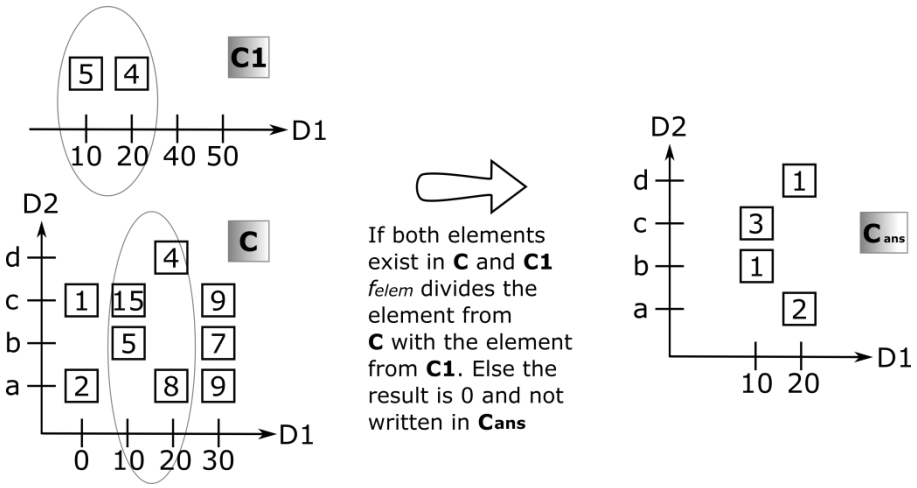

Figure 5: Join according to Agrawal et al.

Cabibbo and Torlone in [15] have defined the Cartesian product and the natural join as operators of the *MD* (stands for MultiDimensional data model) algebra. Their *MD* is based on the concept of f-tables which are described by schemes in the form $[A_1, \ldots A_n] \rightarrow [M_1, \ldots M_m]$, where $A_1, \ldots A_n$ are dimensions drilled-in at specific hierarchy levels (called *attributes* in the paper) and $M_1, \ldots M_m$ are measures. Cartesian product and natural join as it is described in the following two paragraphs are quite generic and pose clearly the concepts of common or non-common *attributes*.

Cartesian product according to [15]: Having two f-tables E and E' with their respective schemes $[A_1, \ldots A_n] \rightarrow [M_1, \ldots M_m]$ and $[A'_1, \ldots A'_{n'}] \rightarrow [M'_1, \ldots M'_{m'}]$, then the Cartesian product $E \times E'$ has a scheme of attributes and measures: $[A_1, \ldots A_n, A'_1, \ldots A'_{n'}] \rightarrow [M_1, \ldots M_m, M'_1, \ldots M'_{m'}]$. It occurs if all *attributes* and measures are disjoint. The result is produced by creating an entry for each pair of entries of E and E' and this entry includes the juxtaposition of the corresponding measures.

Natural join according to [15]: Natural join occurs if there are common *attributes* (defined on the same level of hierarchy) and no common measures between the two f-tables. The common *attributes* are $A_1, \ldots A_k$ and the schemes for E and E' are $[A_1, \ldots A_k, A_{k+1}, \ldots A_n] \rightarrow [M_1, \ldots M_m]$ and $[A_1, \ldots A_k, A'_{k+1}, \ldots A'_{n'}] \rightarrow [M'_1, \ldots M'_{m'}]$ respectively. The natural join $E \bowtie E'$ has a scheme $[A_1, \ldots A_k, A_{k+1}, \ldots A_n, A'_{k+1}, \ldots A'_{n'}] \rightarrow [M_1, \ldots M_m, M'_1, \ldots M'_{m'}]$ and the result contains entries by the pairs of entries of E and E' which have equal values at their attributes. The result entries include the juxtaposition of the corresponding measures.

Another description of Cartesian product and natural join is found in [7] by Datta and Thomas. In order to describe these operations they define a materialized Cube as a 6-tuple $<D, M, A, f, V, g>$ and call it a cube-instance or simply cube. The characteristics in the 6-tuple are:

- $D = \{d_1, d_2 \dots d_n\}$ is a set of dimensions with every d_i being a dimension name.
- $M = \{m_1, m_2 \dots m_k\}$ is a set of measures with every m_i being a measure name.
- $D \cap M = \emptyset$, so there are no common names between dimensions and measures.
- $A = \{a_1, a_2 \dots a_t\}$ is a set of attributes with every a_i being an attribute name.
- $f : D \rightarrow A$ meaning there is a mapping function f which maps subsets of attributes from A to dimensions in D with a restriction that an attribute may appear only in one dimension: $f(d_i) \cap f(d_j) = \emptyset, \forall i, j, i \neq j$.
- V is a set of values that materialize the cube. Thus each $v_i \in V$ is a cell of the cube and contains a k-tuple $<\mu_1, \mu_2 \dots \mu_k>$ with μ_i being a literal value for every measure m_i.
- $g : dom_{dim(1)} \times dom_{dim(2)} \times \dots \times dom_{dim(n)} \rightarrow V$ which is the function that provides the coordinates of e each value $v_i \in V$ in the cube space.

Cartesian product according to [7]: Having two Cubes $C_1 = <D_1, M_1, A_1, f_1, V_1, g_1>$ and $C_2 = <D_2, M_2, A_2, f_2, V_2, g_2>$, then the Cartesian product $C_O = C_1 \times C_2$ is the 6-tuple $C_O = <D_O, M_O, A_O, f_O, V_O, g_O>$ such as: $D_O = D_1 \cup D_2$, $M_O = M_1 \cup M_2$, $A_O = A_1 \cup A_2$, $V_O = V_1 \times V_2$, $|V_O| = |V_1| \times |V_2|$. f_O is a mapping function for attributes existing in A_O towards dimensions existing in D_O. g_O is a function that provides the coordinates for all the values in V_O. The cells in C_O occur by the product of $V_1 \times V_2$ which really means that all combinations of V_1 with V_2 are taken, and the result is a large cube containing all information from the source Cubes.

Join according to [7]: The join operator is a special case of Cartesian product. This operation is feasible only if there are common dimensions between the source cubes: $D_1 \cap D_2 \neq \emptyset$ and is noted as: $D_1 \cap D_2 = \{cd_1, cd_2 \dots cd_l\}$. Additionally the attributes on each of the common dimensions must coincide: $\forall d_i \in (D_1 \cap D_2), f_1(d_i) = f_2(d_i)$. The join of two Cubes $C_O = C_1 \bowtie C_2$ is obtained from the Cartesian product if the cells kept have equal values on their common dimensions. This is expressed mathematically with the restriction (σ) operator as it is defined in [7]. According to this operator there can be a selection of entries from a Cube according to a predicate P which is a formula

checking on dimension values. In the case of join this predicate acts on the common dimensions according to the formula:
$P = [(C_1.cd_1 = C_2.cd_1) \wedge (C_1.cd_2 = C_2.cd_2) \wedge \ldots \wedge (C_1.cd_l = C_2.cd_l)]$ and the join is also written as: $C_1 \bowtie C_2 = \sigma_P(C_1 \times C_2)$. The description of join by Datta and Thomas [7] has a clear mathematical notation and is identical to the natural join described by Cabibbo and Torlone in [15]. Despite the fact that it is clear in mathematical terms, there is no consideration of the semantic impact on the data occurring in the joined Cube.

Pedersen et al. [16] provide a description of a generic type of join, called "identity-based join". This work is based on fact-types and dimension types. Having two multidimensional objects (MOs) M_1 and M_2 which consist of fact-types F_1 and F_2 and dimension types D_1 and D_2 respectively, then identity-based join can be applied based on a predicate $p(f_1, f_2) \in \{f_1 = f_2, f_1 \neq f_2, true\}$. It is noted as: $M_1 \bowtie_{[p]} M_2$ and the result is a new fact type occurring by the pairs of the old fact types F_1 and F_2. The new set of dimension types is the union of the old sets D_1 and D_2.The predicate on the join, can specialize the operation. If $p(f_1, f_2)$ is $f_1 = f_2$ then the join becomes an equi-join. If $p(f_1, f_2)$ is *true* then the join becomes a Cartesian product. In the same paper there is another specialization of identity-based called value-based join which joins multidimensional objects based on common dimensions. In case the value-based join uses $f_1 = f_2$ as a predicate then it becomes a natural join, as long as any duplicate entries are excluded from the result. This work provides a generic framework to handle joins on multidimensional structures but with a fuzzy description. It is notable though that it provides descriptions on specializations of the general case and natural join is one of the considered specializations.

Franconi and Kamble [17] in their work introduce the *GMD*, generalizing the *MD* which first was proposed by Cabibbo and Torlone [15]. *GMD* has a schema for *fact* definitions in the form: $F \doteq E \{D_{1|L_1} \ldots D_{n|L_n}\}: \{M_{1|V_1} \ldots M_{m|V_m}\}$. This form is explained as F is a defined fact, based on E. $D_{1|L_1} \ldots D_{n|L_n}$ is the set of dimensions restricted to specific level L_i and $M_{1|V_1} \ldots M_{n|V_n}$ is the set of measures restricted to specific domains V_j. Based on the *fact* definition, they provide also the definition of join. Given F and G:
$F \doteq E\text{-}1 \{D_{1|L_1} \ldots D_{n|L_n}\}: \{M_{1|V_1} \ldots M_{m|V_m}\}$; $G \doteq E\text{-}2 \{D_{1|L_1} \ldots D_{n|L_n}\}: \{N_{1|W_1} \ldots N_{q|W_q}\}$
then their join can be noted as $H \doteq F \bowtie G$ and it is defined by the following:
$\forall f, g, h, l_1, \ldots l_n, v_1, \ldots v_m, w_1, \ldots w_q.$
$(H(h) \wedge D_1(h) = l_1 \wedge \ldots \wedge D_n(h) = l_n \wedge M_1(h) = v_1 \wedge \ldots \wedge M_m(h) = v_m \wedge N_1(h) = w_1 \wedge \ldots$

$\wedge N_q(h) = w_q) \leftrightarrow (F(f) \wedge D_1(f) = l_1 \wedge ... \wedge D_n(f) = l_n \wedge M_1(f) = v_1 \wedge ... \wedge M_m(f) = v_m \wedge G(g) \wedge D_1(g) = l_1 \wedge ... \wedge D_n(g) = l_n \wedge N_1(g) = w_1 \wedge ... \wedge N_q(g) = w_q)$. Although it is not explicitly mentioned in the paper, it is considered a natural join and moreover it requires all dimensions of the source cubes to be common and on the same level of hierarchy. This restriction of having all dimensions common and in the same level may provide a result which is clearly meaningful, since it does not allow the introduction of a dimension that can spoil the semantics of the result, but the analysis in the paper does not offer any details around this concern.

All related work reveals that the join operation is considered in the multidimensional world. Natural join seems to be also highlighted among the total set of join operations although it is seen mostly mathematically and is explained how it can stand in a multidimensional space.

2.3 Linked Data Cubes

The World Wide Web was thriving, when Tim Berners-Lee et al. [18] envisioned a new aspect of the web, where data stored in it would be meaningful to machines besides human beings. So, the *Semantic Web* emerged and spawned a new ecosystem with infrastructure and concepts.

As Heath and Bizer in [8] notice, the web is full of pages which can be consumed by humans but it is not easy to be consumed by machines in a way to extract information from them, chain it logically and conclude to a result. The main reason is that the web pages are written using vocabularies that can be understood only by people. There are also web APIs which allow machines to communicate and exchange information through web services, but they are quite strict, proprietary and not do not allow connection with other data that exist in other domains. *Linked Data* is the new concept that tries to provide the field for a global space of data, in the World Wide Web, usable by machines.

One of the basic Linked Data infrastructure parts is the Resource Description Framework (RDF) [1] which provides the framework both to model and log data. Data modeled with RDF can be understood and retrieved by the computers, they can be spread around the web and they can even evolve. RDF is considered a stable framework since it is supported by a suite of W3C Recommendations [19]. Linked Data has as foundation four principles, which are known as the *Linked Data Principles*, written by

Tim Berners Lee in [2] and referred a lot in literature as in [1] and [8]. These are the following:

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

RDF sees all information as resources which need to be connected somehow. Resources can be anything, including documents, people, physical objects, and abstract concepts [20]. There is an abstract syntax for putting these resources into a "sentence". The core structure of the abstract syntax is a set of triples, each consisting of a subject, a predicate and an object. A set of such triples is called an RDF graph. An RDF graph can be visualized as a node and directed-arc diagram, in which each triple is represented as a node-arc-node link.

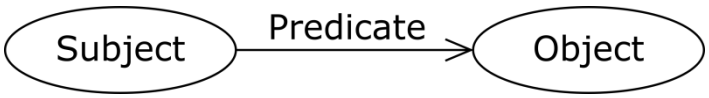

Figure 6: An RDF graph with two nodes (Subject and Object) and a triple connecting them (Predicate)

There can be three kinds of nodes in an RDF graph: IRIs, literals, and blank nodes. This description is taken as is from [21], since it is as an important part of the RDF foundation as the four principles. A triple (RDF statement) relates two resources. The Subject and the Object are these two resources and the Predicate is the kind of relation they have. Predicates are also mentioned as properties in contrast to resources, as they are used to connect resources and cannot become resources themselves.

IRI stands for "International Resource Identifier" and is a generalization of URI which stands for "Uniform Resource Identifier". Thus if a resource is named with an IRI it can become a global resource allowing other users to refer to it or reuse it through its IRI. Such IRIs become more useful when they are documented in vocabularies or they follow well established conventions. For example one IRI used a lot in Predicates is the *http://xmlns.com/foaf/0.1/knows* when a triple is made to describe that a person *knows* another person. In this IRI, the part *http://xmlns.com/foaf/0.1* points to a vocabulary. A relevant initiative is DBpedia which provides data modeled with RDF. An example of

an IRI from DBpedia is *http://dbpedia.org/page/Greece* which can be used to refer to the resource Greece. IRIs can appear in any position of a triple.

Literals are final, actual values. Usually they are based on primitive data types such as string, integer, date, boolean, decimal and so on. String literals may have also a language tag, to indicate the human language of the characters in the string itself. Literals can appear only as an Object in a triple. Literals are considered final because when an Object is a literal the graph stops at this point and the literal cannot become a Subject connecting to something further.

Blank nodes are a facility in RDF which allows to stroll a graph without using every single IRI or literal of each node when describing a path. Blank nodes provide also the ability to define placeholders in a graph, without the need to invent an IRI for every node when populating the graph. So they imply that something exists in a relationship, without naming it explicitly. Blank nodes are used only as Subjects or Objects in a triple.

RDF offers the ability to model real world concepts and relations. The example in Figure 7 models teachers and classes they teach.

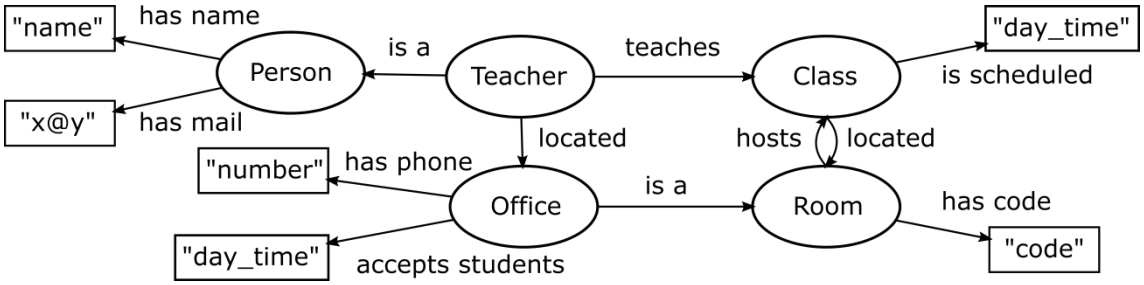

Figure 7: RDF model graph

Nodes in the graph are either ellipses if they are named with an IRI or rectangles if they are named with a literal. Apart from the general model, RDF offers also the mechanism, using the same concepts, to store real data under this model and populate graphs. This is achieved by creating copies of the above graph using different instances for the nodes which exist in the real world, such as real teachers, real rooms and real classes.

According to the Linked Data principles, the predicates and nodes that are not literals should be replaced by http URIs. Some of the predicates used in the example have already some URIs which are commonly accepted:

```
is a → http://www.w3.org/1999/02/22-rdf-syntax-ns#type
has name → http://xmlns.com/foaf/0.1/name
has mail → http://xmlns.com/foaf/0.1/mbox
```

There are also predicates or non-literal nodes that are specific to the model. In this case new URIs can be invented:

```
located → http://uom.gr/lod/location
has code → http://uom.gr/lod/rcode
Room → http://uom.gr/lod/Room
```

RDF builds on the concept of vocabularies and encourages the reuse them. Some of them are provided in the list in Figure 8. RDF Schema [22] is a standard which proposes the form of vocabularies. According to RDF Schema the terms of a vocabulary which can be used as resources in RDF statements (triples), are separated into Classes and Properties. Instances of Classes offer *Subjects* or *Objects* to be used in triples. Properties are used as predicates in triples. Each Property is has a *domain* which depicts the set of Classes that can provide *Subjects* to the triple that contains this Property. Additionally each Property has a *range* which depicts the set of Classes that can provide *Objects* to a triple with this Property.

- Friend of a friend (*foaf)* [28] is used for describing social networks. URI used: *http://xmlns.com/foaf/0.1/*
- Dublin Core (*dcterms*) [26] maintains a metadata element set describing a wide range of resources. Example properties usually used are “creator”, “publisher” and “title”. URI used: *http://purl.org/dc/terms*
- SKOS (*skos*) [27] provides terms for publishing on the web classification schemes like terminologies and thesauri. URI used: *http://www.w3.org/2004/02/skos/core#*
- Data cube (*qb*) [5]. This vocabulary allows multi-dimensional data, such as statistics, to be published in RDF. URI used: *http://purl.org/linked-data/cube#*
- SDMX (*sdmx*) [33]. This vocabulary extends the data cube vocabulary to support publication of statistical data in RDF, using an information model

Figure 8: Sample vocabulary list

It is seen that the URIs may have a large part in their string which is repeated. To improve readability, this can be handled by using a sort of shortcut, called a prefix. Usually vocabularies as the ones given in Figure 8 have a common part in their URI and can be replaced by a prefix being the name of the dictionary. Thus the foaf vocabulary may form the *foaf* prefix for the URI *http://xmlns.com/foaf/0.1/* and the RDF standard may form the *rdf* prefix for the URI *http://www.w3.org/1999/02/22-rdf-syntax-ns#*. This ability applies to any set of URIs that appear in a graph and have a common part in their string. For example the URI *http://uom.gr/lod/* may form a prefix called *duom*. Common parts of URIs are also mentioned as *namespace URIs*. The syntax of a URI with a prefix is in the form of *[prefix]:[specific:term]*. By using these prefixes the predicates and nodes from the graph in Figure 7 would be written as:

```
is a → rdf:type
has name → foaf:name
has mail → foaf:mbox
located → duom:location
has code → duom:rcode
Room → duom:Room
```

Graphs which model data as the one in Figure 7 or graphs which represent actual data can be written (serialized is the formal term) in text, using a high level language. According to [20] there are several options to use for graph serialization:

1. Turtle family of RDF languages (N-Triples, Turtle, TriG and N-Quads);
2. JSON-LD (JSON-based RDF syntax);
3. RDFa (for HTML and XML embedding);
4. RDF/XML (XML syntax for RDF).

Turtle [23] (and TriG which is actually Turtle enhanced) is the one which is most human friendly. RDF/XML is also popular since XML is widely accepted.

Close to Turtle, and inheriting elements from the Relational Databases, emerged another language for accessing Linked Data stored in datasets, called SPARQL [12]. SPARQL combines the triples notion of Turtle, adding variables as value holders or as result bearers with functions for filtering, ordering and modifying data as can be seen also in SQL. Thus SPARQL is the state of the art facility to access the Linked Data which have been serialized using one of the mentioned methods. Although SPARQL inherits many elements from SQL, there are no explicit JOIN operations available.

The features of Linked Data as described in the previous paragraphs provide the means to model the structure of data Cubes. There is already a vocabulary defined for describing data Cubes [5] and it is a W3C Recommendation. The graph that models a generic data Cube is shown in Figure 9 and is taken from the recommendation document.

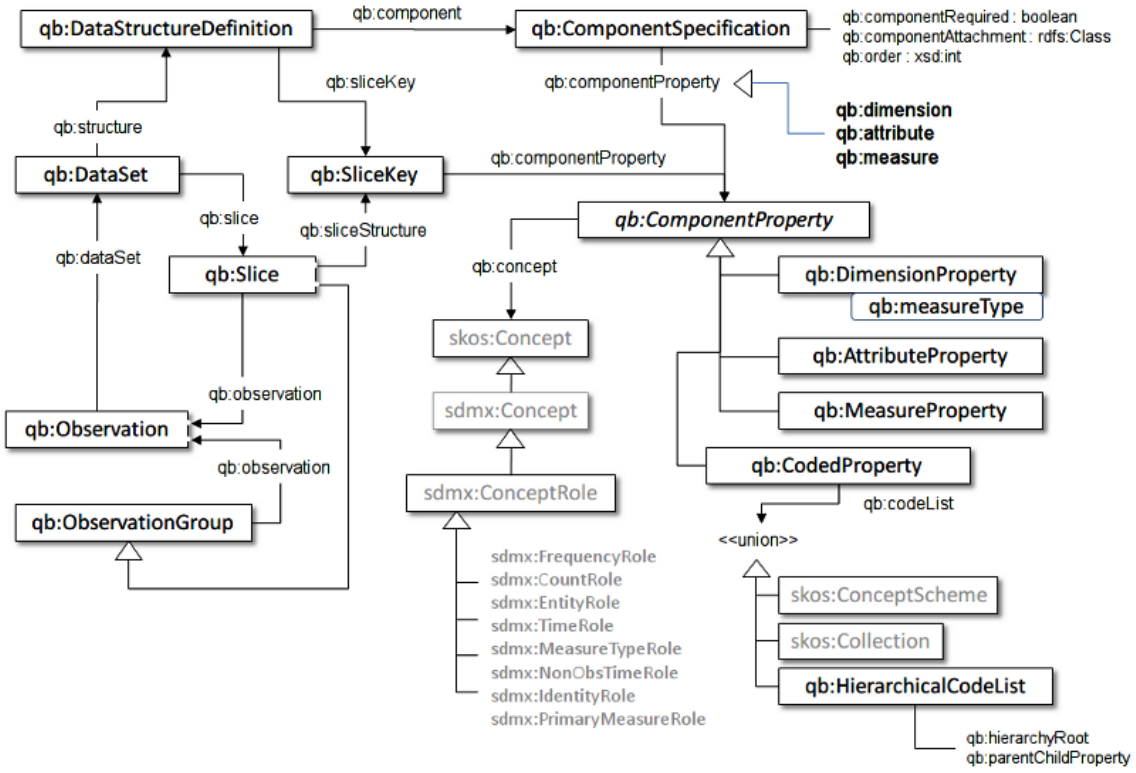

Figure 9: Pictorial summary of key terms and their relationship

All nodes in the graph are Classes, some of them being abstract and have to be instantiated in order to be used in a graph of specific Cube.

A complete Cube consists of data that define its structure and of data that populate the cells. The populated cells are referred as observations. The structure of a Cube is organized by a set of dimensions, attributes and measures. All of them are called components. The dimension components serve as the coordinates towards an observation. A set of specific values for all the dimension components point to a single observation. The measure components are used to record values of the phenomena being observed. The attribute components are used to provide additional information on the observations, like unit of measure, scaling factors or status of the observation (e.g. estimated, forecast, provisional).

Each of the components must be created by instantiating the corresponding Class. So a dimension is created by instantiating the *qb:DimensionProperty* Class, an attribute is created by instantiating the *qb:AttributeProperty* Class and a measure is created by instantiating the *qb:MeasureProperty* Class. All these Classes are Subclasses of the abstract Class *qb:ComponentProperty*. Since *qb:ComponentProperty* is an abstract class there is no instantiation of it in a Linked Data Cube, but it offers an optional property called *qb:concept*. This property is the connection point of the component instances with known and defined concepts in other dictionaries. So, through this property the designer can link to *skos* and *sdmx* vocabularies which contain controlled lists of terms or thesauri and reuse them. A few examples are currency (*sdmx-concept:currency*) and education level (*sdmx-concept:educationLev*) concepts. In case a component instance is limited to a specific cube and cannot be linked to a concept on another vocabulary, the *qb:concept* can be omitted as it is optional.

Besides the *qb:concept* the abstract Class *qb:ComponentProperty* offers another Subclass which can be used to provide hierarchical order of code lists. This Class is called *qb:CodedProperty* and has the property *qb:codeList* which links to exactly one instance of the following Classes: *skos:ConceptScheme, skos:Collection, qb:HierarchicalCodeList*. Classes *skos:ConceptScheme* and *skos:Collection* contain code lists organized in hierarchies of general real world concepts that can be reused. In case *skos* is not sufficient, the designer can define an instance of Class *qb:HierarchicalCodeList*. Instances of *qb:HierarchicalCodeList* can use two available properties: *qb:hierarchyRoot* and *qb:parentChildProperty*; *qb:hierarchyRoot* links all values of a code list that belong to the same level of hierarchy; *qb:parentChildProperty* points to the next narrower level of hierarchy.

The RDF Data Cube vocabulary allows defining multiple measures in the same Cube. With multiple measures, a data Cube can store observed values from different phenomena. There are two approaches in defining multiple measures; one is the *measure dimension* approach and the other is the *multi-measure* approach. The *measure dimension* approach is mostly used by OLAP Cubes and the *measure dimension* is used by SDMX representations. These two approaches cannot be mixed in the same Cube and it is important to consider this fact when joining two Cubes.

In *multi-measure* approach there are more than one measures defined in the cube, meaning there are more than one instances of the Class *qb:MeasureProperty*. When all

measurements are collected and available to add an entry in the Cube, then one cell is created, which has all measured values stored in triples in this same cell.

In *measure dimension* approach there are similarly more than one instances of the Class *qb:MeasureProperty*. The difference with *multi-measure* is that besides the normal cube's dimensions there is an additional, special dimension, the *measure dimension*, which is used to point to one of the measures each time. Thus when all measurements are collected, there are multiple cells created to enter them into the Cube. One cell is created per measurement and in each cell the *measure dimension* declares which measure is used. Thus there is a separate entry in the cube for every observed phenomenon. The *measure dimension* is implemented by using the *qb:measureType* property in a triple, which links to the appropriate measure.

At this point the structure of a Cube can be described. Each data Cube is considered to be a complete dataset in terms of Linked Data. Thus the whole Cube is declared with an instance of the Class *qb:DataSet*. This Cube instance needs an instance of the Class *qb:DataStructureDefinition* which as the name implies holds the information of which are the structural components of the Cube. The instance of the data structure definition has many instances of the *qb:ComponentSpecification* Class, which is a wrapper class of any one of the components, such as a measure, a dimension or an attribute. Usually there is no explicit instance of this wrapper class, since it can be written using a blank node. At the same time it offers a few useful properties:

- *qb:componentRequired*: may be true or false and indicates whether the wrapped component is optional or not. This is applicable only for components of type attribute, because measures and dimensions are mandatory.
- *qb:componentAttachement*: indicates a Class to which the wrapped component is attached. This property is applicable only for components of type attribute. An attribute may be attached to the whole *qb:DataSet*, to a *qb:Slice*, to a *qb:MeasureProperty* or to a *qb:Observation*. Components of type measure and dimension are always attached to *qb:Observation*s.
- *qb:order*: indicates a relative order among the components of the Cube. It is an integer value, which does not affect the Cube structure in a way, but may be useful for visualization purposes.

A conclusion drawn from the usage of *qb:componentAttachement* property is that the attributes are not connected to the dimensions as in the relational modeling of multidimensional data structures. Instead an attribute can be attached either to the entire Cube, or to a slice of the Cube or to one of the measure components. This feature of the attribute components also affects the join process of two cubes.

The data that materialize the Cube are instances of the Class *qb:Observation*. All instant phenomena that are observed, measured and need to be logged, are stored as cells in the cube. Each cell is an instance of the *qb:Observation* Class, which contains distinct values for every dimension, every measure (in a *multi-measure* approach) and every attribute (if there are attributes attached in *qb:Observation* level). Observations are autonomous regarding data containment, but the data they carry are meaningful only with a data structure definition. The collection of all observations is the body of the Cube.

An additional feature offered by the RDF Data Cube vocabulary is the slicing of a cube to parts, according to the OLAP *slice* concept. Thus one or more dimensions can have fixed values and then all observations that are available through these fixed values are gathered into a group, which is an instance of the Class *qb:Slice*. This feature offers precomputed slices on a Linked Data Cube and may be created on top of a populated Cube. They do not replace the fundamental feature of the data structure definition which can constantly offer slices of the Cube on the fly by making appropriate SPARQL queries to the complete dataset. Figure 10 shows schematically the way the components of a Linked Data Cube are related.

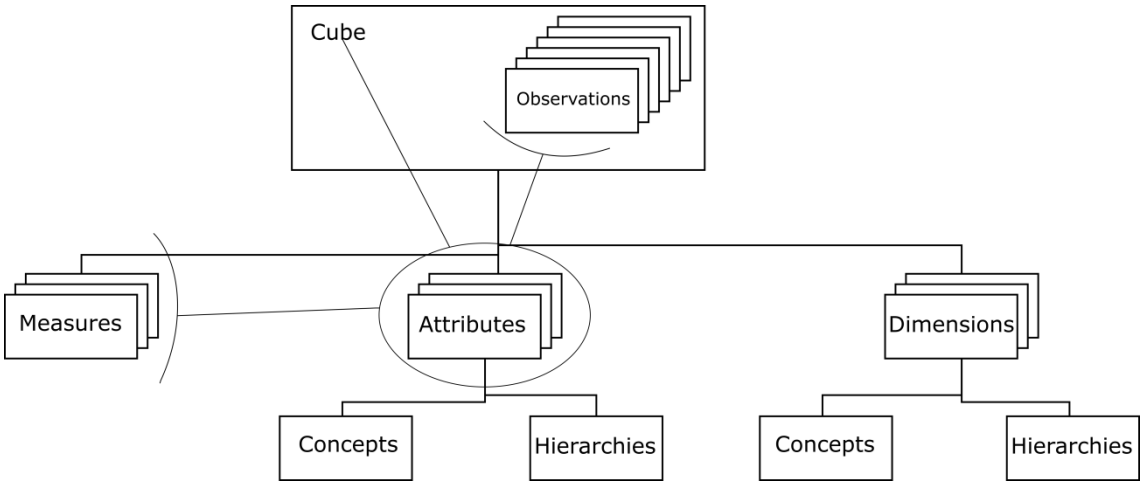

Figure 10: Linked Data Cube key components

2.4 Summary

This chapter presents relevant work and literature regarding the scope of the thesis. Initially some papers are presented and analyzed at the part where they contain description and information about Cubes and the join operation on them. The findings in the papers conclude that the theoretical and mostly mathematical description of joining Cubes has been provided, but no practical implementation has been reported. Additionally most the papers that are found focus on data Cubes coming from the multidimensional structures area and not from the linked data area. Besides the join operation, this chapter presents the literature and information about linked data. The theory of linked data is essential to the work of this thesis requiring the description of many details in it.

3 Methodology

3.1 Introduction

The description of core concepts around multidimensional data structures are described in the previous chapter along with the evolution of the multidimensional model from its conception up to the standardized recommendation of an RDF representation. It is also presented that the idea of joining multidimensional data structures is provided in related literature, focusing on few types of join. In this chapter there is a list of the steps, which construct a methodology to be followed for investigation in more detail of the join types in multidimensional data structures.

3.2 Step1: Review the join types in Relational Algebra

This step of the methodology reviews the join types defined in Relational Algebra. Joining data structures to combine the information carried in their data was conceived and well established in the relational model. The join types as they are defined by Codd [10] are seen in detail, using an example in parallel to highlight the practicalities of applying each one on relations.

3.3 Step2: Apply the join types on Linked Data Cubes

This step analyzes the way to apply they join types on Linked Data Cubes. Join types in relational algebra are well suited for relations which are two dimension tables. The practicalities as they are seen in relational algebra are adapted to the multidimensional model. Moreover any hypothesis that is needed to be taken into account is described at this step of the methodology.

3.4 Step3: Choose of a join type for detailed analysis

Each join type produces a different result when it is applied on two source Cubes and each result has a different meaning for the data. This step evaluates all the investigated types of join and natural join is chosen for detailed analysis, revealing more prerequisites and constraints that need to be considered when it is applied on Linked Data Cubes.

3.5 Step4: Analyzing the natural join on the Flemish Government dataset

This step includes the practical implementation of findings in the previous steps. It builds a proof of concept using real Linked Data Cubes provided by the Flemish authorities by applying SPARQL queries on them. The goal is to identify which of them can be joined and what is the result of their join. Results that are obtained by the execution of the proof of concept program are analyzed and presented.

3.6 Summary

The methodology described at this point provides concrete steps to be followed to achieve a transition of the theoretical study of joining Linked Data Cubes to a working proof of concept. The next chapters elaborate on these steps and make use of this methodology.

4 Join types in Relational Algebra

4.1 Introduction

This chapter describes the types of join as they are defined in the relational model. The relational model defines a relation to be a form of a table consisting of columns (also called attributes) which have distinct names and the table is populated with data grouped in tuples. Each tuple is an entry in the table, viewed as a line and groups together the values of each column for the specific entry. Relations do not have duplicate tuples and the order either of tuples or of columns has no matter.

Relational algebra defines many operations that can be performed on a relation, or between relations, and among these operators there are the various types of join which will be described in the rest of the chapter. To accommodate these descriptions, there will be two relations ***E*** and ***D*** defined and used throughout the examples.

E has columns E (EMP#, NAME, SALARY, CITY) and contains the tuples shown in Figure 11.

EMP#	NAME	SALARY	CITY
A1010	Adams	10000	San Francisco
A890	Bishop	12000	Atlanta
A1103	Chips	12000	Atlanta
A903	Duncan	11000	Dallas
A950	Emerson	15000	San Francisco
A990	Forester	11000	Chicago
A1001	Grover	11000	San Jose

Figure 11: Example relation E

D has columns D (DEP#, CITY) and contains the tuples shown in Figure 12.

DEP#	CITY
D10	San Francisco
D12	Chicago
D07	Atlanta
D05	Dallas
D04	San Diego

Figure 12: Example relation D

4.2 Cartesian product

The Cartesian product (also known as cross join) is an operator that is rarely used in computations over relations. The reason that it is not really used is that mathematically

it is defined as all combinations of tuples, but conceptually it may produce unusable results. On the other hand the Cartesian product contains all the data that can appear in the rest of the join types. The Cartesian product between relations E and D is denoted E × D, is the same as D × E and is formed by concatenating each and every tuple from one relation to each and every tuple from the other relation. Any columns having the same name get a prefix from the relation they come from, to keep them distinct. Cartesian product $R = E \times D$ is show in Figure 13.

EMP#	NAME	SALARY	E.CITY	DEP#	D.CITY
A1010	Adams	10000	San Francisco	D10	San Francisco
A1010	Adams	10000	San Francisco	D12	Chicago
A1010	Adams	10000	San Francisco	D07	Atlanta
A1010	Adams	10000	San Francisco	D05	Dallas
A1010	Adams	10000	San Francisco	D04	San Diego
A890	Bishop	12000	Atlanta	D10	San Francisco
A890	Bishop	12000	Atlanta	D12	Chicago
A890	Bishop	12000	Atlanta	D07	Atlanta
A890	Bishop	12000	Atlanta	D05	Dallas
A890	Bishop	12000	Atlanta	D04	San Diego
A1103	Chips	12000	Atlanta	D10	San Francisco
A1103	Chips	12000	Atlanta	D12	Chicago
A1103	Chips	12000	Atlanta	D07	Atlanta
A1103	Chips	12000	Atlanta	D05	Dallas
A1103	Chips	12000	Atlanta	D04	San Diego
A903	Duncan	11000	Dallas	D10	San Francisco
A903	Duncan	11000	Dallas	D12	Chicago
A903	Duncan	11000	Dallas	D07	Atlanta
A903	Duncan	11000	Dallas	D05	Dallas
A903	Duncan	11000	Dallas	D04	San Diego
A950	Emerson	15000	San Francisco	D10	San Francisco
A950	Emerson	15000	San Francisco	D12	Chicago
A950	Emerson	15000	San Francisco	D07	Atlanta
A950	Emerson	15000	San Francisco	D05	Dallas
A950	Emerson	15000	San Francisco	D04	San Diego
A990	Forester	11000	Chicago	D10	San Francisco
A990	Forester	11000	Chicago	D12	Chicago
A990	Forester	11000	Chicago	D07	Atlanta
A990	Forester	11000	Chicago	D05	Dallas
A990	Forester	11000	Chicago	D04	San Diego
A1001	Grover	11000	San Jose	D10	San Francisco
A1001	Grover	11000	San Jose	D12	Chicago
A1001	Grover	11000	San Jose	D07	Atlanta
A1001	Grover	11000	San Jose	D05	Dallas
A1001	Grover	11000	San Jose	D04	San Diego

Figure 13: Cartesian product R = E × D

4.3 Theta join – the Boolean extension.

Theta (θ) join is the most generic type of join, which filters data in a meaningful way. The rest types of join can be seen as a special case theta join. The name of this type of join comes from the "theta" operator which can be any one of the following:

1. EQUALITY
2. INEQUALITY
3. LESS THAN
4. LESS THAN OR EQUAL TO
5. GREATER THAN
6. GREATER THAN OR EQUAL TO
7. GREATEST LESS THAN
8. GREATEST LESS THAN OR EQUAL TO
9. LEAST GREATER THAN
10. LEAST GREATER THAN OR EQUAL TO

The general representation of theta joins between two relations E and D is: $E \bowtie_\theta D$. In its Boolean extension, theta operator can be a function consisting of comparisons combined with logical operators such as AND, OR, NOT and IMPLIES. The comparisons can be between the values of two columns of the two relations, as long as they draw values from the same domain, or they may be comparisons between a column's values and a constant, while these comparisons can be combined with logical operators in between. Theta join between two relations is achieved if the theta function is applied on all tuples of the Cartesian product of relations. Assuming that the theta function is $\theta = SALARY \leq 11000\ AND\ E.CITY = D.CITY$, then theta join between relations E and D is denoted as $R = E \bowtie_{SALARY \leq 11000\ AND\ E.CITY = D.CITY} D$ and the table with the result can be seen in Figure 14.

EMP#	NAME	SALARY	E.CITY	DEP#	D.CITY
A1010	Adams	10000	San Francisco	D10	San Francisco
A903	Duncan	11000	Dallas	D05	Dallas
A990	Forester	11000	Chicago	D12	Chicago

Figure 14: Theta join R = E $\bowtie_{SALARY \leq 11000\ AND\ E.CITY = D.CITY}$ D

4.4 Equijoin

Equijoin is the first specialization of theta join. It depends only on the EQUALITY operator from the list in subchapter 4.3. The equality can be applied to more than one

column between the joined relations, using the logical AND, as long as they draw their values from the same domain. Equijoin does not have a separate symbol; it uses the theta notation, with detail on the columns that will be checked. Equijoin between two relations is achieved by checking all tuples of their Cartesian product, for equality of the values between the specified columns. Equijoin of relations E and D on columns E.CITY and D.CITY is denoted as $R = E \bowtie_{E.CITY = D.CITY} D$ and the result can be seen in Figure 15.

EMP#	NAME	SALARY	E.CITY	DEP#	D.CITY
A1010	Adams	10000	San Francisco	D10	San Francisco
A890	Bishop	12000	Atlanta	D07	Atlanta
A1103	Chips	12000	Atlanta	D07	Atlanta
A903	Duncan	11000	Dallas	D05	Dallas
A950	Emerson	15000	San Francisco	D10	San Francisco
A990	Forester	11000	Chicago	D12	Chicago

Figure 15: Equijoin R = E $\bowtie_{E.CITY = D.CITY}$ D

It can be seen that in Figure 15 columns E.CITY and D.CITY have exactly the same data, making one of them redundant. In case there were more than one column pairs in the equijoin function, there would be more redundant columns in the result.

4.5 Natural join

Natural join is a specialization of equijoin, and deals with the redundant columns that occur by the latter. In detail natural join checks on equality of values between columns that have the same name in the two relations, but keep only one of these columns into the result. Since it involves equality of columns implicitly, it does not need a formula like theta, so the general representation of natural joins between two relations E and D is: $R = E \bowtie D$. Figure 16 shows the result relation as it formed by the natural join of relations E and D.

EMP#	NAME	SALARY	CITY	DEP#
A1010	Adams	10000	San Francisco	D10
A890	Bishop	12000	Atlanta	D07
A1103	Chips	12000	Atlanta	D07
A903	Duncan	11000	Dallas	D05
A950	Emerson	15000	San Francisco	D10
A990	Forester	11000	Chicago	D12

Figure 16: Natural join R = E $\bowtie$ D

Natural join is probably the most useful type in the relational database design. In SQL, when the plain “JOIN” keyword is used between two relations, it implies a natural join of them.

4.6 Semijoin

Semijoin is based on the previous types of join. The main difference in the result is that it excludes the elements of one of the operands. A join is a binary operation, meaning it consists of two operands (relations) one left and one right of the join operator. So in the case of $R = E \bowtie D$, E is the left operand and D is the right operand. Semijoin splits in two branches, the left semijoin and the right semijoin depending on the operand that will form the result. Theoretically a semijoin, either left or right, can be applied on top of a theta join which is the most generic type and consequently to all specializations of theta join. This subchapter focuses on semijoin of a natural join.

Left semijoin

Left semijoin is a natural join which contains in the result only the tuples of the left operand. So the equality comparison on the common columns is applied, but the concatenation of the tuples is not applied. The left semijoin of relations E and D is denoted as $R = E \ltimes D$ and the result is shown in Figure 17.

EMP#	NAME	SALARY	CITY
A1010	Adams	10000	San Francisco
A890	Bishop	12000	Atlanta
A1103	Chips	12000	Atlanta
A903	Duncan	11000	Dallas
A950	Emerson	15000	San Francisco
A990	Forester	11000	Chicago

Figure 17: Left semijoin R = E ⋉ D

Right semijoin

Right semijoin, in the same logic as left join, is a natural join, keeping only the tuples from the right operand. It is denoted as $R = E \rtimes D$ and the result is shown in Figure 18.

DEP#	CITY
D10	San Francisco
D12	Chicago
D07	Atlanta
D05	Dallas

Figure 18: Right semijoin R = E ⋊ D

If the right semijoin is applied on a result of a natural join as the one shown in Figure 16, then as the right operand tuples are kept, duplicate values occur. Since the result is a relation all duplicate values are removed and the result becomes the one in Figure 18.

4.7 Outer join

All the types of join presented up to this point are strict regarding a function, called theta in the most general form. The result relation keeps at most the data which conform to the function, or less as in semijoin. These types of join are part of a category called *inner joins*. The other category is called *outer joins* and complements the inner joins by expanding the result relation with data that do not conform to the checking function. Outer join has three forms: full (or symmetric) outer join, left outer join and right outer join. Additionally the generic form of outer join is the outer equijoin. This description will focus on outer natural join, which is a specialization of the outer equijoin, eliminating the redundant columns as the inner natural join.

Full outer natural join

Full outer natural join is a natural join, extended with the tuples of both operands that do not conform to the equality check. The result relation has missing values in the tuples that occur by this extension and they are written as NULL. Full outer natural join between two relations E and D is denoted as $R = E \bowtie D$ and the result is seen in Figure 19.

EMP#	NAME	SALARY	CITY	DEP#
A1001	Grover	11000	San Jose	NULL
A1010	Adams	10000	San Francisco	D10
A890	Bishop	12000	Atlanta	D07
A1103	Chips	12000	Atlanta	D07
A903	Duncan	11000	Dallas	D05
A950	Emerson	15000	San Francisco	D10
A990	Forester	11000	Chicago	D12
NULL	NULL	NULL	San Diego	D04

Figure 19: Full outer natural join R = E ⋈ D

Left outer natural join

Left outer natural join is based on the full outer natural join, by keeping in the result the tuples that origin from the left operand, including the tuples that introduce NULL

values. The left outer natural join between two relations E and D is denoted as $R = E ⟕ D$ and the result is shown in Figure 20.

EMP#	NAME	SALARY	CITY	DEP#
A1001	Grover	11000	San Jose	NULL
A1010	Adams	10000	San Francisco	D10
A890	Bishop	12000	Atlanta	D07
A1103	Chips	12000	Atlanta	D07
A903	Duncan	11000	Dallas	D05
A950	Emerson	15000	San Francisco	D10
A990	Forester	11000	Chicago	D12

Figure 20: Left outer natural join R = E ⟕ D

Right outer natural join

Right outer natural join is based on the full outer natural join, by keeping in the result the tuples that origin from the right operand, including the tuples that introduce NULL values. The right outer natural join between two relations E and D is denoted as $R = E ⟖ D$ and the result is shown in Figure 21.

EMP#	NAME	SALARY	CITY	DEP#
A1010	Adams	10000	San Francisco	D10
A890	Bishop	12000	Atlanta	D07
A1103	Chips	12000	Atlanta	D07
A903	Duncan	11000	Dallas	D05
A950	Emerson	15000	San Francisco	D10
A990	Forester	11000	Chicago	D12
NULL	NULL	NULL	San Diego	D04

Figure 21: Right outer natural join R = E ⟖ D

4.8 Summary

This chapter presents the join types that are found in relational algebra. The list of them presented here is not exhaustive but it is considered complete, since only few special cases are omitted to avoid details that are not useful. Each join type is complemented with an example, which shows how it can be applied practically and helps to understand the way to transfer it in Linked Data Cubes.

5 Applying join types on Linked Data Cubes

5.1 Introduction

At this point all the needed concepts and theory are in place and can be used to study the join of Linked Data Cubes. Initially a general description of a data cube *C* is provided to highlight the structural parts. As in the previous chapter two specific Cubes *X* and *Y* are built and used as examples in the rest of the chapter. Then Cubes *X* and *Y* are flattened for easier handling throughout the examples.

Let *C* be a Cube which has dimensions $Dc = \{d_{C1}, d_{C2} \dots d_{Cn}\}$ and measures $M_C = \{m_{C1}, m_{C2} \dots m_{Ck}\}$. Additionally let Cube *C* have object values for each dimension $d_i \in D_C$, such as $O_i = \{o_{i,1}, o_{i,2} \dots o_{i,m}\}$, and the size *m* of O_i vary depending on the dimension. An observation contains actual values for the measures of the Cube and it is coordinated by one object value per dimension. The set of actual measured values of each *k-th* observation can be defined as $v_k = \{\mu_{1,k}, \mu_{2,k} \dots \mu_{n,k}\}$ for all *n* dimensions. The total set of v_k sets for Cube C can be defined as $V_C = \bigcup v_k$ and materialize the Cube with data. Each dimension may be connected to a concept through the *qb:concept* property according to the description in chapter 2.3. According to the same chapter, each dimension may have also a connection to a specific code list through the *qb:codeList* either with a hierarchy or not. These characteristics are considered inherent to the dimension and they are handled implicitly during the join operations. This means that when two values of two dimensions are checked, they belong to the same code list and under the same level of hierarchy. Additionally the attributes as they are defined for Linked Data Cubes are irrelevant to the dimensions. During studying the join operations they are left aside and are considered at a point where all other details are settled.

Let *X* be a specific Cube with the following dimensions and measures: $D_X = \{d_{X1}, d_{X2}, d_{X3}\}$ and $M_X = \{m_{X1}, m_{X2}\}$. The object values for each dimension $d_i \in D_X$ are: $O_{X1} = \{t_1, t_2, t_3, t_4\}$, $O_{X2} = \{l_1, l_2, l_3, l_4\}$, $O_{X3} = \{s_1, s_2, s_3, s_4\}$. The measured values for each measure per coordinate are: for m_{X1}:{f, g, f, e} and for m_{X2}:{u, v, w, q}. Thus $V_X = \{\{f, u\}, \{g, v\}, \{f, w\}, \{e, q\}\}$ and $|V_X| = 4$. A drawing of cube *X* is provided in Figure 22.

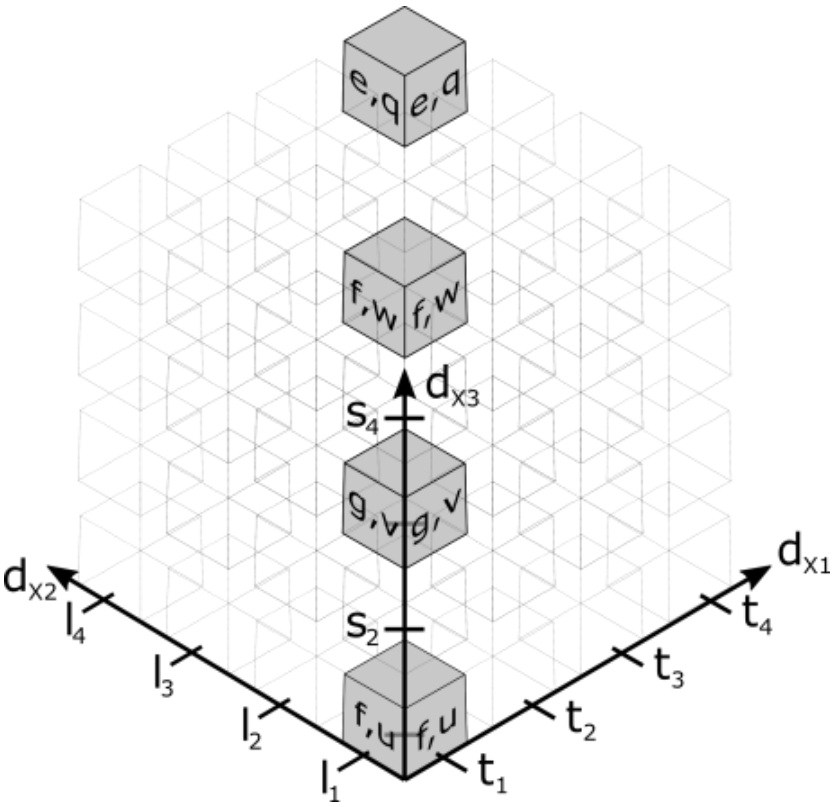

Figure 22: Example Cube X

An attempt to flatten Cube *X* and present it in the two dimension space of the screen could be seen in Figure 23. In more detail, a multidimensional Cube can be flattened for visual purposes, if all observations become tuples in a kind of a special relation. This special relation has the dimensions as columns. Each tuple contains values for all dimensions. At the end there is an additional column, related to the measures of the Cube, which may contain more than one element as the measured values for each measure.

d_{X1}	d_{X2}	d_{X3}	V_X
t_1	l_1	s_1	{f, u}
t_2	l_2	s_2	{g, v}
t_3	l_3	s_3	{f, w}
t_4	l_4	s_4	{e, q}

Figure 23: Example flattened Cube X

Let *Y* be a specific Cube with the following dimensions and measures: $D_Y = \{d_{Y1}, d_{Y2}, d_{Y3}\}$ and $M_Y = \{m_{Y1}\}$. The object values for each dimension $d_i \in D_Y$ are: $O_{Y1} = \{b_1, b_2, b_3\}$, $O_{Y2} = \{l_1, l_3, l_5\}$, $O_{Y3} = \{c_1, c_2, c_3\}$. The measured values for the measure are: m_{Y1}:{q, a, z}. Thus $V_Y = \{q, a, z\}$ and $|V_Y| = 3$. A drawing of Cube *Y* is shown in Figure 24.

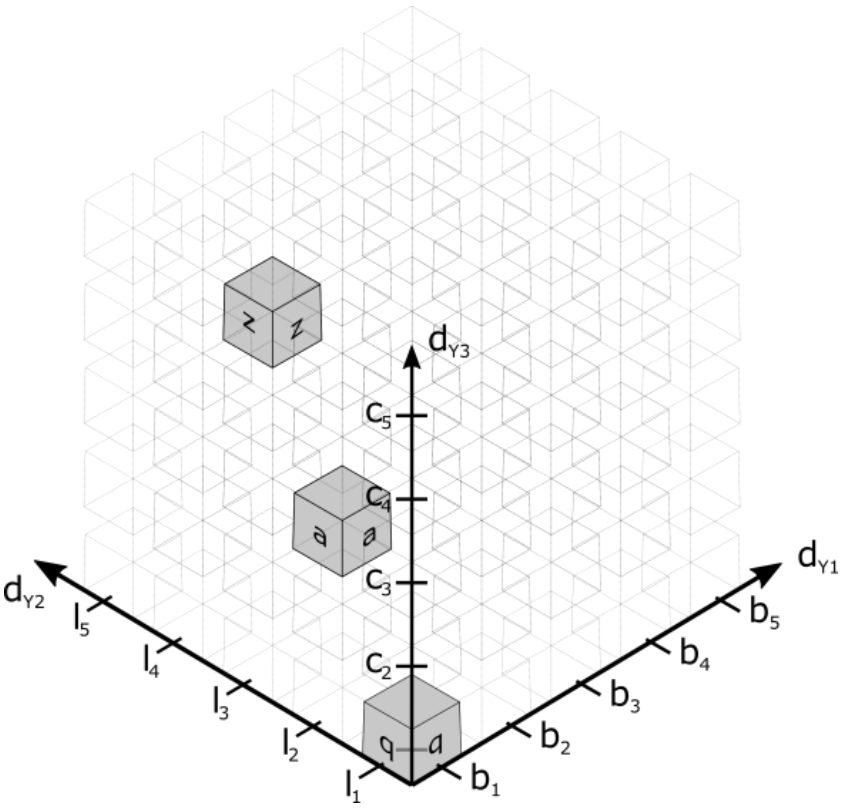

Figure 24: Example Cube Y

Similarly Cube *Y* can be flattened as it is show in Figure 25.

d_{Y1}	d_{Y2}	d_{Y3}	V_Y
b_1	l_1	c_1	q
b_2	l_3	c_2	a
b_3	l_5	c_3	z

Figure 25: Example flattened Cube Y

In the rest of this chapter the Cube materializations in examples will be in flat form and columns except the rightmost one, will imply dimensions and the other way around.

5.2 Cartesian product

Cartesian product has its origin in set theory and is a rarely used type of join, because the result contains combinations produced by all inputs mixing all the values. The description of the Cartesian product is useful though, because it stands as the basis for the rest of the join operations.

In data Cubes model, according to Datta and Thomas [7] the Cartesian product between two cubes *X* and *Y* is defined as: $Z = X \times Y$, where: $D_Z = D_X \cup D_Y$, $M_Z = M_X \cup M_Y$, $A_Z = A_X \cup A_Y$, $V_Z = V_X \times V_Y$ and $|V_Z| = |V_X| \times |V_Y|$. Applying this operation to the sample Cubes the result is shown with Figure 26.

d_{X1}	d_{X2}	d_{X3}	d_{Y1}	d_{Y2}	d_{Y3}	V_Z
t_1	l_1	s_1	b_1	l_1	c_1	{f, u, q}
t_1	l_1	s_1	b_2	l_3	c_2	{f, u, a}
t_1	l_1	s_1	b_3	l_5	c_3	{f, u, z}
t_2	l_2	s_2	b_1	l_1	c_1	{g, v, q}
t_2	l_2	s_2	b_2	l_3	c_2	{g, v, a}
t_2	l_2	s_2	b_3	l_5	c_3	{g, v, z}
t_3	l_3	s_3	b_1	l_1	c_1	{f, w, q}
t_3	l_3	s_3	b_2	l_3	c_2	{f, w, a}
t_3	l_3	s_3	b_3	l_5	c_3	{f, w, z}
t_4	l_4	s_4	b_1	l_1	c_1	{e, q, q}
t_4	l_4	s_4	b_2	l_3	c_2	{e, q, a}
t_4	l_4	s_4	b_3	l_5	c_3	{e, q, z}

Figure 26: Cube Cartesian product Z = X × Y

All the rest of join types can be produced by the Cartesian product relation, by selecting specific tuples.

5.3 Theta Join

Theta join (θ-join) is described first because it is the most generic type of join and from this one various other types of join can be defined. Theta join relies on a formula which poses operations to the columns. The formula is based on an operator called θ which can be one of the following operators as it is proposed by Codd in [10].

1. EQUALITY
2. INEQUALITY
3. LESS THAN
4. LESS THAN OR EQUAL TO
5. GREATER THAN
6. GREATER THAN OR EQUAL TO
7. GREATEST LESS THAN
8. GREATEST LESS THAN OR EQUAL TO
9. LEAST GREATER THAN
10. LEAST GREATER THAN OR EQUAL TO

An extension of the theta join allows θ to be also a logical operator, such as AND, OR, NOT and IMPLIES which allow building complex formulas. The generic notation of a theta join between cube X and Y is written as: $X \bowtie_f Y$, where $f = d_X \, \theta \, d_Y$. The operands can also be constants.

Considering cube Z which is produced by the Cartesian product in the previous chapter, and assuming that for object values $\{l_1, l_2, l_3, l_4\}$ also stands: $l_1 < l_2 < l_3 < l_4$. A theta join could use a formula like: $d_{X2} < d_{Y2}$. Then theta join between Cube X and Z is defined as: $Z = X \bowtie_{dX2 < dY2} Y$ and the result is shown in Figure 27.

d_{X1}	d_{X2}	d_{X3}	d_{Y1}	d_{Y2}	d_{Y3}	V_Z
t_1	l_1	s_1	b_1	l_1	c_1	{f, u, q}
t_1	l_1	s_1	b_2	l_3	c_2	{f, u, a}
t_1	l_1	s_1	b_3	l_5	c_3	{f, u, z}
t_2	l_2	s_2	b_2	l_3	c_2	{g, v, a}
t_2	l_2	s_2	b_3	l_5	c_3	{g, v, z}
t_3	l_3	s_3	b_2	l_3	c_2	{f, w, a}
t_3	l_3	s_3	b_3	l_5	c_3	{f, w, z}
t_4	l_4	s_4	b_3	l_5	c_3	{e, q, z}

Figure 27: Cube theta join Z = X $\bowtie_{dX2 < dY2}$ Y

Operations with the θ operand on dimensions are possible, if the dimensions are checked to take values of the same type or code list.

5.4 Equijoin

Equijoin is the case of a theta join, where θ is the equality operator between two columns. This kind of join does not have a special symbol, but shows explicitly the columns that are checked for equality.

Application of equijoin between Cubes X and Y on dimensions d_{X2} and d_{Y2} is defined as: $Z = X \bowtie_{dX2 = dY2} Y$ and the result is shown in Figure 28.

d_{X1}	d_{X2}	d_{X3}	d_{Y1}	d_{Y2}	d_{Y3}	V_Z
t_1	l_1	s_1	b_1	l_1	c_1	{f, u, q}
t_3	l_3	s_3	b_2	l_3	c_2	{f, w, a}

Figure 28: Cube equijoin Z = X $\bowtie_{dX2 = dY2}$ Y

5.5 Natural join

Theta join and equijoin as they are described, they are depended on the values in the columns, which are actually object values of the dimensions. Thus the result contains all the columns on which the filtering operations are performed. Especially in the case of equijoin the result contains at least two columns with the same contents exactly. In data Cube terms, this result contains two dimensions with exactly the same object values.

Natural join is the type of join that takes into account the column names and joins the columns into one, while keeping only equal values. Thus, natural join can be performed only when there are at least two common dimensions.

Assuming that the sample cubes *X* and *Y* have a common dimension d_{X2} and d_{Y2} respectively and it is called "geo", the natural join of cubes *X* and *Y* is defined as: $Z = X \bowtie Y$ and the result is shown in Figure 29.

d_{X1}	**geo**	d_{X3}	d_{Y1}	d_{Y3}	V_Z
t_1	l_1	s_1	b_1	c_1	{f, u, q}
t_3	l_3	s_3	b_2	c_2	{f, w, a}

Figure 29: Cube natural join Z = X ⋈ Y

5.6 Semijoin

Semijoin is based on natural join, meaning it is derived from the natural join. It is distinguished in two kinds, the left semijoin and right semijoin. Semijoin follows the same rules as the natural join, but the result excludes any columns that come from the "open" side of the semijoin. In the case of data Cubes, the dimensions and the measures that come from the "open" side of the semijoin are excluded from the result.

Applying left semijoin on sample Cubes *X* and *Y* is defined as: $Z = X \ltimes Y$ and the result is shown in Figure 30.

d_{X1}	**geo**	d_{X3}	V_Z
t_1	l_1	s_1	{f, u}
t_3	l_3	s_3	{f, w}

Figure 30: Cube left semijoin Z = X ⋉ Y

On the other hand applying right semijoin on sample Cubes *X* and *Y* is defined as: $Z = X \rtimes Y$ and the result is shown in Figure 31.

d_{Y1}	**geo**	d_{Y3}	V_Z
b_1	l_1	c_1	q
b_2	l_3	c_2	a

Figure 31: Cube right semijoin Z = X ⋊ Y

5.7 Outer join

Outer join is a type of join that extends the aforementioned traditional types by incrementing the values that are selected from the common column to be used for joining. This allows NULL values to exist into the result. Outer join takes into account

the existence of at least two common columns between the joined relations. Outer join type is separated to three categories: Full outer join, left outer join and right outer join.

Full outer join

Full outer join is achieved by joining on all distinct values of the common column appearing in both relations. The full outer join operation on data Cubes allows NULL values, both in object values and in observations of the result. It is easily induced that the observations that contain NULL values occur from the NULL object values.

To apply a full outer join to the sample Cubes *X* and *Y* it is assumed that there is a common dimension between the two, d_X and d_Y respectively, which is called "geo". The distinct object values which form the common dimension are the following set: $O_{geo} = \{l_1, l_2, l_3, l_4, l_5\}$. The full outer join between Cubes *X* and *Y* is defined as: $Z = X \bowtie Y$ and the result is shown in Figure 32.

d_{X1}	**geo**	d_{X3}	d_{Y1}	d_{Y3}	V_Z
t_1	l_1	s_1	b_1	c_1	{f, u, q}
t_2	l_2	s_2	NULL	NULL	{g, v, NULL}
t_3	l_3	s_3	b_2	c_2	{f, w, a}
t_4	l_4	s_4	NULL	NULL	{e, q, NULL}
NULL	l_5	NULL	b_3	c_3	{NULL, NULL, z}

Figure 32: Cube full outer join Z = X ⋈ Y

Left outer join

Left outer join is achieved by joining on all distinct values of the common column seen only in the relation in the left-hand side. These values that do not exist to the right-hand side relation introduce NULL in the result.

To apply a left outer join on the sample Cubes *X* and *Y* the set of the common dimension "geo" is formed as: $O_{geo} = \{l_1, l_2, l_3, l_4\}$. The left outer join between Cubes *X* and *Y* is defined as: $Z = X \bowtie Y$ and the result is shown in Figure 33.

d_{X1}	**geo**	d_{X3}	d_{Y1}	d_{Y3}	V_Z
t_1	l_1	s_1	b_1	c_1	{f, u, q}
t_2	l_2	s_2	NULL	NULL	{g, v, NULL}
t_3	l_3	s_3	b_2	c_2	{f, w, a}
t_4	l_4	s_4	NULL	NULL	{e, q, NULL}

Figure 33: Cube left outer join Z = X ⋈ Y

Right outer join

Right outer join is achieved by joining on all distinct values of the common column seen only in the relation in the right-hand side. These values that do not exist to the left-hand side relation introduce NULL in the result.

To apply a right outer join on the sample Cubes *X* and *Y* the set of the common dimension "geo" is formed as: $O_{geo} = \{l_1, l_3, l_5\}$. The right outer join between Cubes *X* and *Y* is defined as: $Z = X \rtimes Y$ and the result is shown in Figure 34.

d_{X1}	**geo**	d_{X3}	d_{Y1}	d_{Y3}	V_Z
t_1	l_1	s_1	b_1	c_1	{f, u, q}
t_3	l_3	s_3	b_2	c_2	{f, w, a}
NULL	l_5	NULL	b_3	c_3	{NULL, NULL, z}

Figure 34: Cube right outer join Z = X ⋊ Y

5.8 Evaluation of join types

The next step is to select the join types for combining data Cubes, by reviewing how the result data Cube is formed.

The Cartesian product, as already described, is not used in general. The complete combinations of object values and measured values may have mathematical value, but a data Cube resulting from the Cartesian product does not provide observations from the real world.

Theta join is the next more generic type of join after the Cartesian product. The formula with θ operators makes it a flexible type of join, with filtering capabilities on the source data Cubes. Yet, it is quite complex to get a result data Cube that can be evaluated and provide useful information.

Equijoin restrains the flexibility of theta join and provides more control to the result data Cube, but the fact that two or more dimensions contain a set of same object values, can still leave room for ambiguity if the result is valuable. It is probably worth looking into this type of join. If this ambiguity is cleared by the user or by a process which can identify that the equality is performed on dimensions that draw values from the same domain, then this type of join provides the ability to combine Cubes which at first glance seem incompatible.

Natural join is more restrictive than the equijoin, since it eliminates the ambiguity of the result, because there are common dimensions between the source Cubes. Thus there is certainty that the Cubes can be joined based on the common dimensions. This type of join is also worthy to look deeper.

Semijoins either left or right, although they derive from the natural join, they keep observations from only one of the source Cubes. So these types of join do not offer expansion and could not be studied further.

The outer join types are the only types which can increase the volume of the result Cube. It is inevitable that the volume increase is achieved by bringing in NULL values. Comparing the outer join with the Cartesian product which also increases the volume of the result Cube, it is visible that the NULL values put a clear sign that some piece of the data is not available. On the other hand the Cartesian product provides data which may not have any meaning at all due to the full meshed combinations. In addition, NULL values can actually be exploited to create queries for the missing data, such as to collect all observations that have NULL values in a specific measure.

5.9 Summary

All join types defined in relational algebra are transferred to the Linked Data Cubes. This is achieved by flattening the multidimensional structures and bringing them into a two dimension shape which allows reducing the operation as a join between relations. The examples that are used throughout this chapter offer a better understanding, used to evaluate the types of join and lead to the selection of one for further study and implementation.

6 Natural join as a choice

6.1 Introduction

According to the evaluation of join types, equijoin, natural join and outer joins could be candidates for further investigation and experimentation. Natural join will be chosen in this thesis because it is cleaner in description. Datta and Thomas [7] define that a natural join of two data Cubes is feasible as long as they have at least one common dimension with matching attribute sets among the two Cubes regarding the common dimension. Their definition is mathematically complete but there are some limitations which will be highlighted by identifying three constraints that must be considered.

6.2 Natural join in depth

The study of natural join in depth is feasible with the use of examples. Let Cubes *X* and *Y* be the source Cubes and *Z* be the outcome of their natural join. Then it is already described that for *Z* holds: $D_Z = D_X \cup D_Y$; $M_Z = M_X \cup M_Y$; $O_{Common} = O_{X2} \cap O_{Y2}$; $O_{Z1} \subseteq O_{X1}$; $O_{Z2} \subseteq O_{X3}$; $O_{Z3} \subseteq O_{Y1}$; $O_{Z4} \subseteq O_{Y3}$. The subsets of object values of the non-common dimensions occur due to the restriction of tuples driven by the equality of object values on the common dimensions. This means that in the result Cube, the size of each dimension will be equal or smaller than the size of the source Cubes, in terms of object values.

Regarding the dimensions of the result Cube *Z*, they are at least as many as the number of the common dimensions of the source Cubes, which occurs from the following union: $D_Z = D_X \cup D_Y$. This union poses some concerns, although at first glance it may be seen that it helps to increase the size of the Cube in general with more data. Looking once more at the example in chapter 5.5 the result Cube *Z* appears in Figure 35.

d_{X1}	**geo**	d_{X3}	d_{Y1}	d_{Y3}	V_Z
t_1	l_1	s_1	b_1	c_1	{f, u, q}
t_3	l_3	s_3	b_2	c_2	{f, w, a}

Figure 35: Choosing natural join Z = X ⋈ Y

The set of dimensions becomes: $D_Z = D_X \cup D_Y = \{d_{X1}, geo, d_{X3}, d_{Y1}, d_{Y3}\}$. The first observation in the result Cube consists of the measured values in set $v_1 = \{f, u, q\}$. This set has a coordinate in the Cube space, defined by the object values in tuple $<t_1, l_1, s_1, b_1, c_1>$. A careful inspection reveals that the measured values *f, u* have now additional coordinates, which are $<b_1, c_1>$. On the other hand the measured value *q* has

additional coordinates $<t_1, s_1>$. Using abstract dimension names and hypothetical values mitigates the actual effect, but the result Cube contains new coordinates to measured values which may not correspond to reality. In a more stretched situation expanding the set of dimensions may lead to a result Cube with contradicting data, such as the case of Cube *X* describing something provided for free and Cube *Y* describing cost in one of its dimensions. Once two such Cubes are joined it is impossible to distinguish which object value from a dimension refers to what measured value.

This concern generates the first constraint for using natural join between two Cubes: *The Cubes X and Y to be joined must have all dimensions in common:* $D_X \equiv D_Y$*. This identity includes the equality of any corresponding concepts, code lists or hierarchy levels on the dimensions.*

The result Cube *Z* will contain new measures which occur from the union of measures of source Cubes: $M_Z = M_X \cup M_Y$. This union is adding new data to the result Cube and increases its value. There are cases though at which the measures in the result Cube do not add data. This may happen either if the set of measures M_X is a subset of the set of measures M_Y or the other way around. This observation generates the second constraint during joining two Cubes: *The sets of measures of the source Cubes must not be subsets of one another:* $M_X \nsubseteq M_Y$ *and* $M_Y \nsubseteq M_X$.

At this point it is needed to bring in the role of attributes that are defined on an RDF Cube. This is achieved by a new more concrete example.

Let Cube *A* be one of the source Cubes. Cube *A* has three dimensions: D_A *= {geo, sex, time}*, one measure: M_A *= {"people practicing lifelong learning"}* and one mandatory attribute on this measure, A_{M_A} *= {"% population"}*. In addition, Cube *A* has some object values per dimension: O_{Ageo} *= { BE, BG, CZ, DK, DE, EE, IE, EL, ES, FR, HR, IT, CY};* O_{Asex} *= {T, F, M};* O_{Atime} *= {1992, Q2_1993, 1993, Q2_1994, 1994, Q2_1995, 1995, Q2_1996, 1996, Q2_1997, 1997, Q2_1998, 1998, Q2_1999, 1999, Q2_2000, 2000, Q2_2001, 2001, Q2_2002, 2002, Q2_2003, 2003, Q2_2004, 2004}* and one attribute on observation level with indicators on each observation A_{V_A} *= {break in time series, confidential, estimated, forecast, provisional, revised, low reliability}*. This attribute is optional, so some observations may include it and others not. Some sample observations are shown in Figure 36.

geo	sex	time	V_A	% population
FR	F	Q2_1993	10	forecasted
DE	M	2003	15	
CZ	M	2002	17	estimated
FR	T	1999	14	revised
IT	T	1997	9	
FR	F	1997	14	revised
DE	F	Q2_2002	15	

Figure 36: Example flattened RDF Cube A

Let Cube *B* be the second of the source Cubes, having the same three dimensions: $D_B = \{geo, sex, time\}$, one measure: $M_B = \{\text{“people at tertiary education level”}\}$ and one mandatory attribute on the measure, $A_{M_B} = \{\text{“percentage”}\}$. Each dimension has some object values per dimension: $O_{Bgeo} = \{DE, EE, IE, EL, ES, FR, HR, IT, CY, LV, LT, LU, HU, MT, NL, AT\}$; $O_{Bsex} = \{T, F, M\}$; $O_{Btime} = \{1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009\}$. This Cube does not have any attributes on observation level. Some sample observations are shown in Figure 37.

geo	sex	time	V_B	percentage
EL	F	2009	40	
DE	M	2003	35	
LV	M	2008	30	
FR	T	1999	32	
IT	T	1997	30	
FR	F	1996	31	
DE	F	2002	37	

Figure 37: Example flattened RDF Cube B

The object values of each dimension in the result Cube are also produced by the intersection of the object values sets of the source Cubes, for each dimension: $O_{Adi} \cap O_{Bdi}, \forall\, d_i \in D_A$ *or* $d_i \in D_B$. Since dimensions are common, this intersection is sufficient to provide the object values sets for all these dimensions. The process of applying intersection on the above mentioned sets draws attention to the fact that none of the result sets should be an empty set. Consequently it is not possible to join two Cubes if one of the common dimensions does not have any object values or if the following relation does not stand: $O_{Adi} \cap O_{Bdi} \neq \emptyset, \forall\, d_i \in D_A$ *or* $d_i \in D_B$.

This observation generates the third constraint for using natural join between two Cubes: *The object values sets of the result Cube C must not be empty sets:* $O_{Cdi} \neq \emptyset, \forall d_i \in D_C$.

The attributes attached to observations, which in parallel may be optional, pose a small challenge during natural join. Having a characteristic coupled so tight to an observation forces the use of "measure dimension observations" in the result Cube. The measure dimension observation approach literally keeps separate cells in terms of contents, although they belong to the same observation as it is pointed by the dimension values. So each separate cell can have the needed attributes which are tightly attached to the observation of the source cube. In case one of the source Cubes is built using the "multi-measure observations" approach and needs to be naturally joint with another Cube which has attributes attached on observations, then the first Cube must be transformed using the "measure dimension observations" approach during the join operation. This is unavoidable because the RDF Data Cube vocabulary [5] does not allow mixing the two approaches.

The attributes attached to measures (or to the data set in the same sense) need to be handled also during natural join. According to RDF Data Cube vocabulary [5], when attributes are attached to measures and multiple measures per Cube exist, it is not possible to associate a distinct attribute to a distinct measure. Thus when a natural join happens and multiple measures appear in the result Cube the attribute loses its correlation to the measure. This can be solved if the attribute is moved from the measure to the observation level by replicating it in all observations. Then by using the "measure dimension observations" approach the attribute will transferred and located only to the cells that occurred from the cube which had a measure attached attribute.

Handling of attributes leads to define a structural directive: *Bringing attributes from source Cubes to the result cube during natural join requires the result Cube to be structured using "measure dimension observations".*

Applying this directive to the sample Cubes *A* and *B* transforms them accordingly as it shown in Figure 38 and Figure 39 respectively.

geo	sex	time	V_A		
FR	F	Q2_1993	10	% population	forecasted
DE	M	2003	15	% population	
CZ	M	2002	17	% population	estimated
FR	T	1999	14	% population	revised
IT	T	1997	9	% population	
FR	F	1997	14	% population	revised
DE	F	Q2_2002	15	% population	

Figure 38: Example flattened Cube A, measure attribute transferred to observations

geo	sex	time	V_B	
EL	F	2009	40	percentage
DE	M	2003	35	percentage
LV	M	2008	30	percentage
FR	T	1999	32	percentage
IT	T	1997	30	percentage
FR	F	1996	31	percentage
DE	F	2002	37	percentage

Figure 39: Example flattened Cube B, measure attribute transferred to observations

The natural join of data Cubes *A* and *B* in the last concrete example will have as a result data Cube *C*, which is described next. The dimensions are identical in the result Cube, thus $D_C \equiv D_A \equiv D_B$ *and* $D_C = \{geo, sex, time\}$. The measures of the result Cube occur from the union of the measures sets of the source Cubes, thus $M_C = M_A \cup M_B \Rightarrow M_C =$ *{"people practicing lifelong learning", "people at tertiary education level"}*. The object values sets occur from the intersection of the corresponding dimensions of the source Cubes, thus $O_{Ci} = O_{Ai} \cap O_{Bi}, \forall\, d_i \in D_C \Rightarrow O_{Cgeo} = \{DE, EE, IE, EL, ES, FR, HR, IT, CY\}$; $O_{Csex} = \{T, F, M\}$; $O_{Ctime} = \{1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004\}$. The result Cube *C* is seen in Figure 40 having 3 observations and a measure dimension clarifying which measure is stored in each cell.

geo	sex	time	measure	V_C		
DE	M	2003	M_A	15	% population	
DE	M	2003	M_B	35	percentage	
FR	T	1999	M_A	14	% population	revised
FR	T	1999	M_B	32	percentage	
IT	T	1997	M_A	9	% population	
IT	T	1997	M_B	30	percentage	

Figure 40: Cube C as the natural join of cubes A and B

This concrete example provided a new data Cube *C* which has a number of object values in every dimension and more specifically: $|O_{Cgeo}| = 9$; $|O_{Csex}| = 3$; $|O_{Ctime}| = 10$. Cube *C* with such dimensions is ready to undergo any OLAP. There is possibility though that not all dimensions get a sufficient population of object values. The third restriction ensures that there will not be a dimension without object values, but the intersection operation which provides the result sets can provide the smallest of sets, with one element. Taking into account these facts, it becomes necessary to establish a metric which will allow evaluating the merging process of two Cubes. This metric will use the size of the object values sets and will indicate the percentage of overlapping between them. In the last example, the object values of dimension *sex*, O_{Csex} have 100% overlap dimension. On the other hand the object values of dimension *geo* have overlap

of (9/20)*100 = 45%. A formula to calculate the overlapping percentage is based on the size of the intersection of sets, divided by the size of the union of sets:

$$\frac{|O_{Adi} \cap O_{Bdi}|}{|O_{Adi} \cup O_{Bdi}|} = \frac{|O_{Cdi}|}{|O_{Adi}| + |O_{Bdi}| - |O_{Cdi}|}$$

The example in Figure 41 shows two sets of size 6, intersecting to a result set of size 4. The overlap of this intersection is calculated to be 50%.

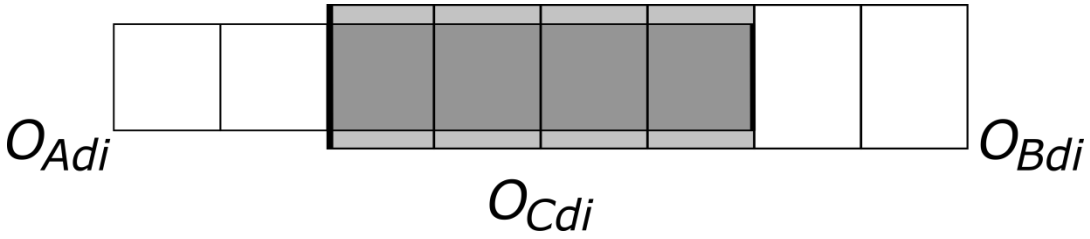

Figure 41: Overlapping of object values

6.3 Summary

Natural join of data Cubes is an operation which is established theoretically. This chapter identifies three constraints that need to apply on top, to provide a result Cube that can become an item for further study and analysis. The three constraints are provided as a list:

- *Cubes X and Y to be joined must have all dimensions in common:* $D_X \equiv D_Y$.
- *The sets of measures of the source Cubes must not be subsets of one another:* $M_X \nsubseteq M_Y$ *and* $M_Y \nsubseteq M_X$.
- *The object values sets of the result Cube Z must not be empty sets:* $O_{Zdi} \neq \emptyset$, $\forall\ d_i \in D_Z$.

There is also a structural directive which provides the means to handle attributes in an RDF Cube:

- *If there are attributes in Cubes X and Y, the result Cube Z needs to be structured using "measure dimension observations".*

Finally there is a metric which can be observed and evaluated and refers to the percentage of object values overlapping, between the source and result Cubes. This overlapping is observed per dimension. It is calculated by the following formula:

$$\frac{|O_{Zdi}|}{|O_{Xdi}| + |O_{Ydi}| - |O_{Zdi}|}$$

Practical application of natural join between Linked Data Cubes needs to follow the constraints. Additionally the metric on object value overlapping should be studied and evaluated. The lack of conformance to these constraints or poor evaluation of the overlapping leads to a result Cube which misses the goal, to combine information in a meaningful and valuable way.

7 Natural join of Linked Data cubes, an implementation.

7.1 Introduction

This chapter describes a program implementation which presents a proof of concept around the theoretical study that has been presented for natural join of Linked Data Cubes. The implementation uses the Java programming language, integrating the Apache Jena API [24] which is "A free and open source Java framework for building Semantic Web and Linked Data applications." and it is licensed under the Apache License, Version 2.0.

The Java program performs SPARQL queries towards a dataset through a SPARQL endpoint. It collects data, stores them into internal structures and performs the checks and calculations that are described theoretically in the previous chapter. This chapter provides technical details of the program's structure later on.

7.2 Program requirements

Taking into account what are the constraints which need to be met to join two Cubes according to chapter 6 and the number of Linked Data Cubes that may reside in a dataset, the implementation of the program is based on the following requirements:

1. The program must be able to connect to a SPARQL endpoint and perform SPARQL queries to obtain all needed information. The existence of a SPARQL endpoint is mandatory and it should be configurable at program start.
2. The program scans the dataset for all available Linked Data Cubes.
3. The program checks which Cubes are eligible for join by checking their dimensions and the measures.
4. The program calculates the metric defined in 6.2 for the overlapping of object values between the same dimensions of the eligible to join cubes.
5. The program is able to scan either for original Cubes or for original and aggregated Cubes.
6. The program is eligible to filter out Cubes from the calculation process according to the needs of the user.
7. The program attempts to use parallel processing to decrease the overall execution time, due to the large number of combinations and due to the HTTP latency period.

8. The program prints the calculated result on screen.
9. The program parametrization is achieved by using command line arguments.

Filtering of the Cubes is achieved by providing a file with the URIs of the Cubes which will not be part of the processing. In order to avoid additional syntactic terms, this file will be a Turtle file containing triples of the form:

<cubeURI> rdf:type qb:DataSet .

The <cubeURI> resource is used as a pattern during filtering out Cubes, thus each line in the filter will take out all Cubes that have a URI starting with the pattern *cubeURI*.

7.3 Program design

7.3.1 Use cases

The requirements that are listed in the previous subchapter drive intuitively to prepare the use case diagram of the program. Obviously the program has a single user who can request execution of the program guiding it to different behavior each time. The requirements pose that the behavior is guided with command line arguments at the startup of the program. The diagram in Figure 42 shows the use cases.

- The gray shaded use case *Process Original Cubes* is the default use case, and it is the minimum that the program can execute by not providing any arguments besides the SPARQL endpoint URL which is mandatory.
- The *Help* use case which is invoked by using *-h* or *--help* in the command line and supports the user on the syntax that can be used at program invocation. If this use case is invoked, any other arguments in the command line are ignored.
- The *Process Original and Aggregated Cubes* use case extends the default one by including into the calculation both the original Cubes and the aggregated ones. This use case is invoked by using *-a* or *--aggregated* in the command line and as it already described, if omitted the *Process Original Cubes* will be triggered.

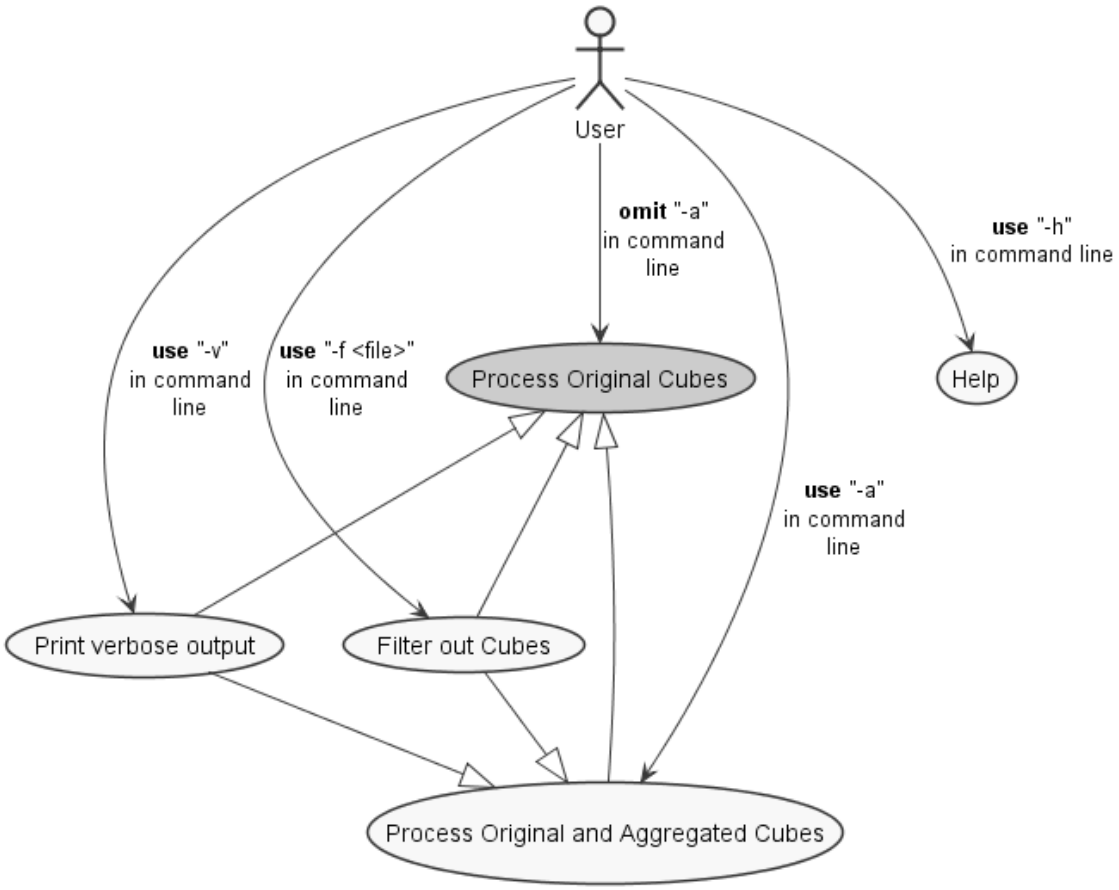

Figure 42: Program use cases

- The *Filter out Cubes* use case is invoked by using *-f <file>* or *--filter <file>* in the command line interface. This argument needs additionally a value which is the path to the file that contains the filter to be applied. This argument is optional and if it is omitted no filter is applied during the process. The *Filter out Cubes* use case extends both the *Process Original Cubes* and *Process Original and Aggregated Cubes*, meaning that the filter is applied on top of either use case, in relation to whether *-a* was used or not.
- The *Print verbose output* use case is invoked by using *-v* or *--verbose* in the command line interface. This argument is optional and when it is used it prints to the output all the results (regardless if the Cubes are eligible to join) and details about the dimensions, the measures and the attributes of the processed Cubes. The *Print verbose output* use case extends both the *Process Original Cubes* and *Process Original and Aggregated Cubes*, meaning that the verbose printing is performed on top of either use case, in relation to whether *-a* was used or not.

The screenshot in Figure 43 shows the help message printed when the *Help* use case is invoked. This screenshot shows also the way a user can invoke all use cases of the program.

```
usage: java -jar jcs.jar [-a] [-f <file>] [-h] -s <URL> [-v]

This program connects to a SPARQL end point and checks the cubes for
joining possibility.
 -a,--aggregated       Apply on aggregated cubes of dataset. If omitted
                       original cubes are used
 -f,--filter <file>    Path to a Turtle file with cubes to filter out
 -h,--help             Print this message
 -s,--sparqlEP <URL>   URL of a SPARQL Endpoint
 -v,--verbose          Verbose printing

Enjoy your cubes.
```

Figure 43: Help message

7.3.2 Activity

The program is invoked by passing the desired command line arguments. The execution phase consists of several activities until the result is available which are shown in the diagram in Figure 44 as a high level flow of the program. More details per activity are provided in the following list:

1. *Parse command line arguments*: Naturally this is the first activity performed by the program and according to the arguments that are used, the flow of the activities goes to the relevant direction. Parsing the command line is achieved by using the "CLI" component of the Apache Commons project [25]. According to this component's page: "The Apache Commons CLI library provides an API for parsing command line options passed to programs. It's also able to print help messages detailing the options available for a command line tool." The library is licensed under the Apache License, Version 2.0.
2. The next activity of the program is to decide if it applies only to original Cubes or if it must include the aggregated ones too.
3. *Collect both original and aggregated Cubes*: This activity builds a SPARQL query which asks for all the Cubes that exist in the dataset. The result is iterated and every Cube is stored in a separate list which will be used in later activities.
4. *Collect original Cubes*: This activity is similar to the previous one, with a difference in the prepared SPARQL query. In this case the query allows in the result only the URIs that have the shortest text length, since it is observed that the aggregated Cubes have a common part with the original Cubes in their URI

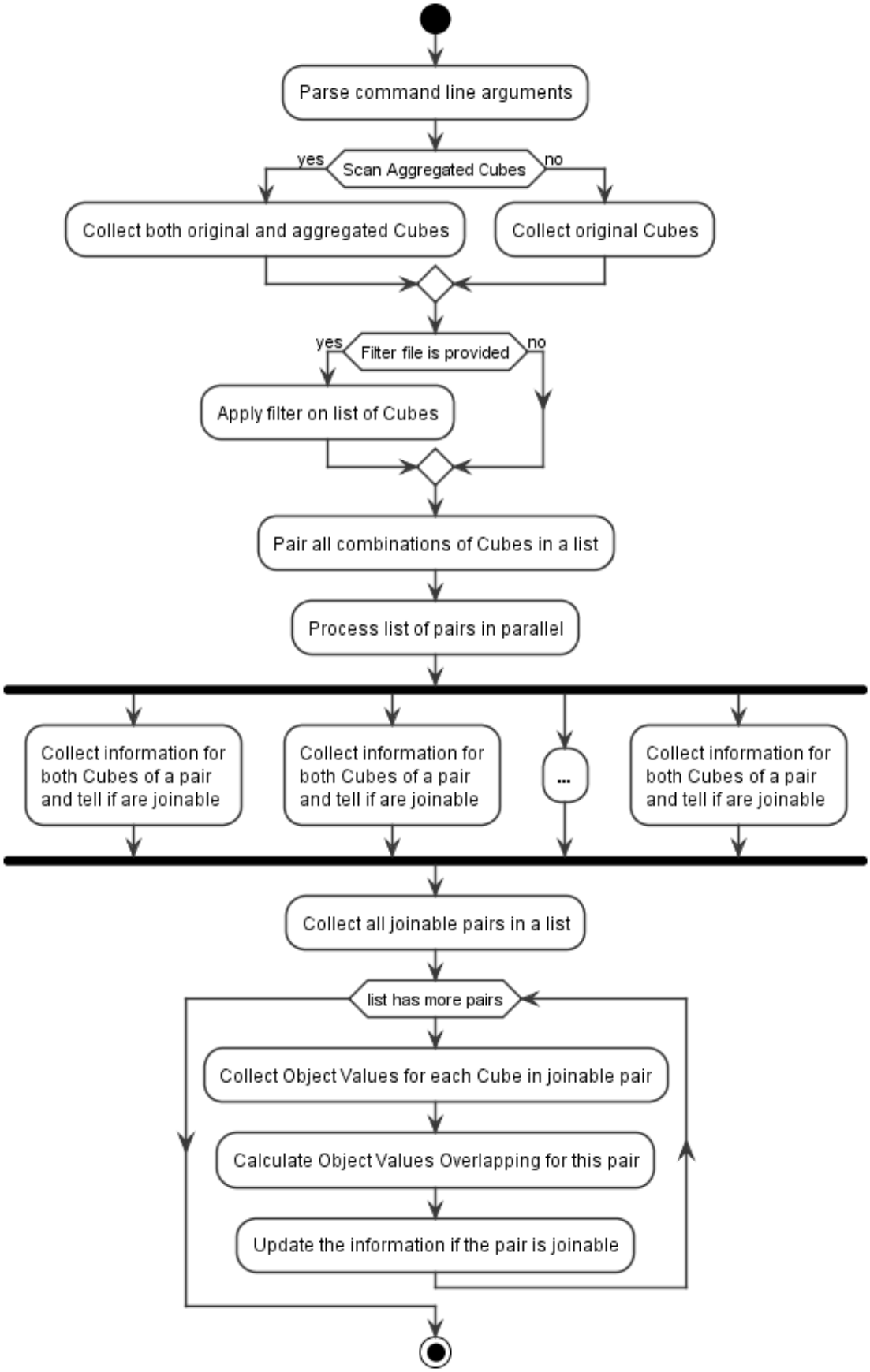

Figure 44: Activity diagram

and it is extended with some distinctive text. This observation is shown in the example in subchapter 8.2 while describing dataset in detail.

5. The next activity of the program is to apply or not a filter to the list of Cubes, according to the presence or not of a filter file. If there is such a filter file, then the list created in the previous activities is re-iterated and any Cube URI that matches the pattern of the Cubes provided in the filter, is removed from the list and is never taken into account in the next activities.
6. *Pair all combinations of Cubes in a list*: This activity acts again on the list of Cubes as it is formed so far, by combining all of the Cubes in pairs and creates a new list with them. At this point the program provides a printout of the initial list of Cubes that will be processed, the size of the initial list and the size of the list of pairs. The initial list is provided so the user can pick possibly some Cubes which need to be excluded in another run of the program, by placing them in a filter file. The sizes of the two lists are useful because they give a sense of the magnitude of the process that will continue.
7. *Process list of pairs in parallel*: The list of pairs that is created in the previous activity can become quite large. In detail if the initial list of Cubes has size *n* then the size of the list of pairs is calculated as:

$$\frac{(n-1)\,n}{2}$$

   For each pair, there will be need to perform SPARQL queries over HTTP adding the network latency period for each operation.
8. *Collect information for both Cubes of a pair and tell if are joinable*: During these activities information about the dimensions and the measures is collected for each Cube of the pair. The constraints regarding the dimensions and the measures are checked and the pair is marked if it is joinable or not. These activities are executed in parallel. This is achieved by getting a "parallelstream" from the list of pairs and iterate on this parallel stream. There is care to avoid need for synchronization on shared resources and Java performs the parallel processing implicitly without use of explicit multithreading. Even with parallel processing the execution time of these activities takes quite long if the number of pairs is large.
9. *Collect all joinable pairs in a list*: This activity collects in a list all pairs that are found to be joinable and will be used in the next one.

10. *Iterate over the list of joinable pairs*: The next activities are part of a loop which iterates sequentially the list of joinable pairs.
    - All object values for each dimension of each Cube of the joinable pair are collected.
    - The object values are checked on the same dimensions of the pair of Cubes and the overlapping is calculated.
    - The fact if the pair is joinable or not is updated according to the third constraint on joining, by checking if overlapping is calculated to be zero for at least one dimension.

7.3.3 Classes

This subchapter presents the classes and their relationship through diagrams which construct the program and provide the behavior described in 7.3.2.

When the application decides to collect the list of Cubes it needs to send the appropriate SPARQL query to the dataset. In order to handle this query in a transparent manner there is a factory which creates an object that will perform the correct query to the dataset. The class diagram shown in Figure 45 describes this factory method implementation.

The factory when it is created by the program, it takes as an argument an enumerated value (*ORIGINAL, AGGREGATED*) which drives the creation of the appropriate object by one of the concrete classes, either *CubeScannerAggregated* or *CubeScannerOriginal*. Each of the concrete classes connects to the SPARQL endpoint and performs the query to fetch the relevant list of Cubes. At the same time the object applies a filter file if it is provided during construction. The outcome of this procedure is to have a list of Cubes containing either only the original or all of them, excluding any Cube which matches the patterns found in the filter file.

At this point it is worth describing how the SPARQL queries are built and sent to the endpoint. Every class that needs to perform a SPARQL query contains a template text of the query, using variables for the unknown parts of the graph that is queried.

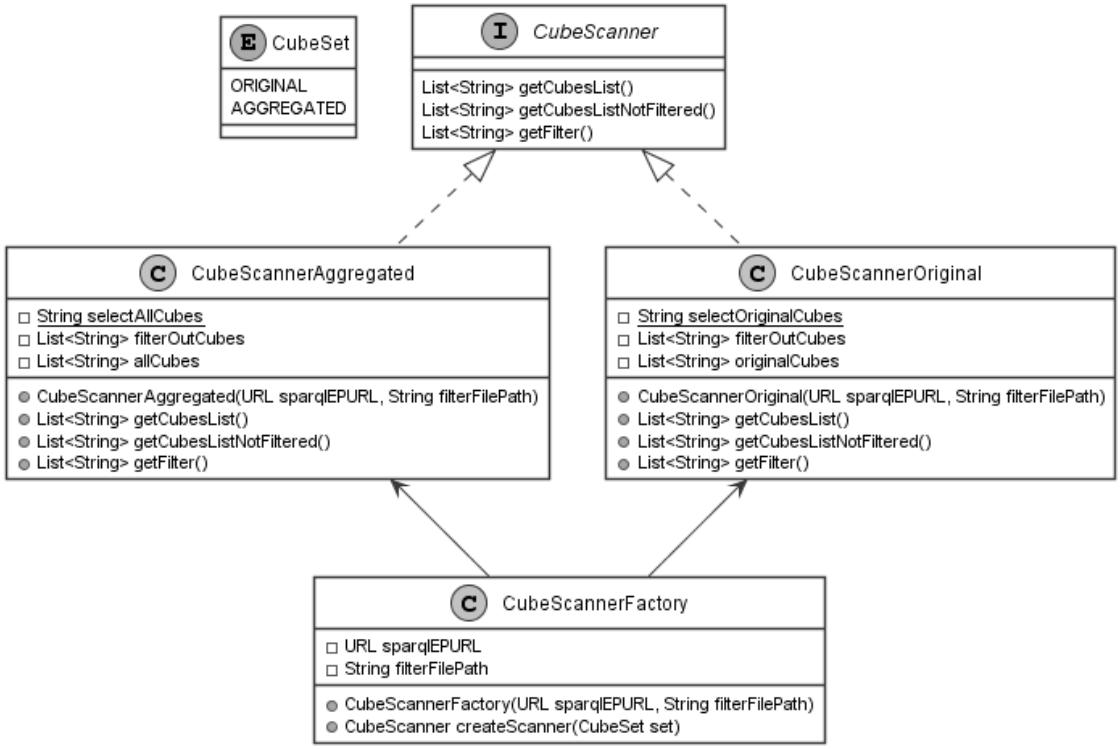

Figure 45: Cube scanner class diagram

This template text with the query contents is stored in a static Java String because the template as such is not modified at runtime. In the class diagram shown in Figure 45 the concrete classes have respectively the static Strings *selectAllCubes* and *selectOriginalCubes* with the template queries in. The template query is used to create an object of the *ParameterizedSparqlString* class. This new object is able to modify the variables received from the template, using objects of the *QuerySolutionMap* class. A variable can be modified to become a specific value which may be either a resource or a literal as they are defined in RDF. The variables of the *ParameterizedSparqlString* object which are not assigned a specific value, they remain variables in the final query, attracting bindings of values on them from the data being queried. The query is sent over HTTP by using a *QueryEngineHTTP* object which connects to the SPARQL endpoint transparently. The result of the query is a *ResultSet* object which contains all the value bindings of all the solutions that exist in the dataset. The *ResultSet* object can be iterated to provide the distinct solution bindings to the query's variables. The classes *ParameterizedSparqlString, QuerySolutionMap, QueryEngineHTTP* and *ResultSet* are a small sample provided by the Jena library which can be used to query, model and construct Linked Data. The Jena library though does not provide something specific for Cubes, so the facilities it has are wrapped in new classes of the program. The

description of building and performing a SPARQL query in this paragraph is followed in all the classes of the program that need to collect data from the dataset.

The “graph” is a basic concept in Linked Data and it is found also in the RDF model for Cubes. So there may be different graphs for storing the structures or the data of Cubes. Using a graph to separate data provides better overview and maintenance of the dataset but may have optimized response time to the queries. In order to make use of these potential optimized response times, the program finds the relevant graph and uses it into the query. Finding the graph is performed by knowing which part of the Cube model will be queried. This program queries the *qb:DataSet* part of the Cube, or the observations (data) part, or the structure part of the Cube. Subsequently the factory method pattern is used again, to create an object which contains the relevant graph URI. Figure 46 shows the class diagram of the graph scanner factory.

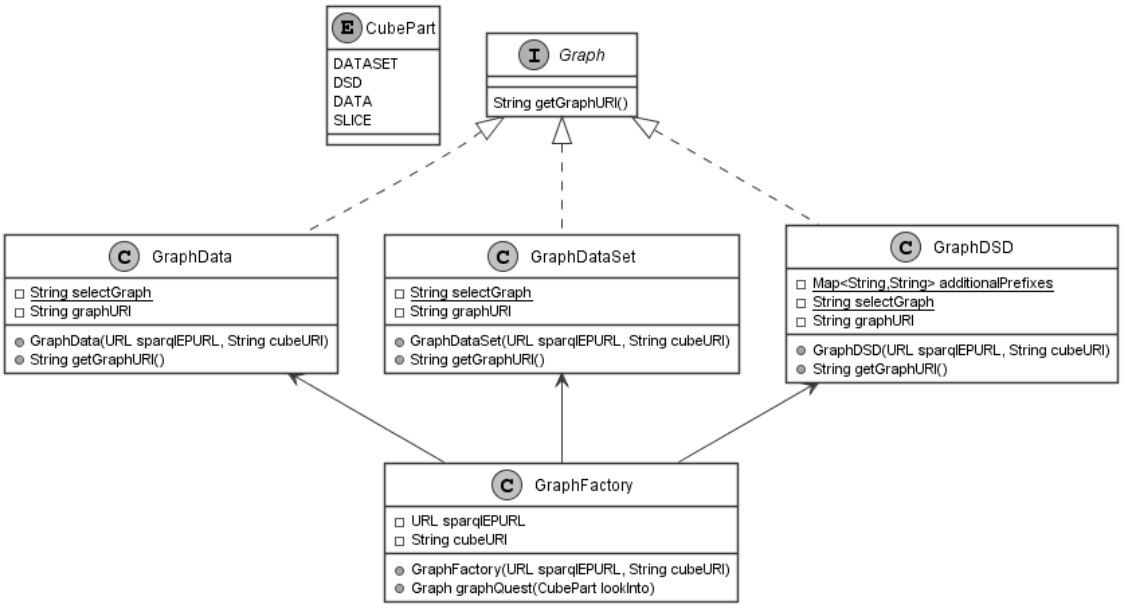

Figure 46: Graph scanner class diagram

The factory that creates the concrete scanner is driven by the value of the *CubePart* enumeration. There are four possible values in the enumeration (*DATASET, DSD, DATA, SLICE*) but currently the factory method does not take into account *SLICE*. This one is ignored and defaults to *DATASET* since there is no use case that needs a query in the *qb:Slice* part of the cube. Having a factory that hides the implementation, would make easy the support scanning the graph of the *qb:Slice* part when a use case arises, by creating a concrete class and modifying the creator method of the factory.

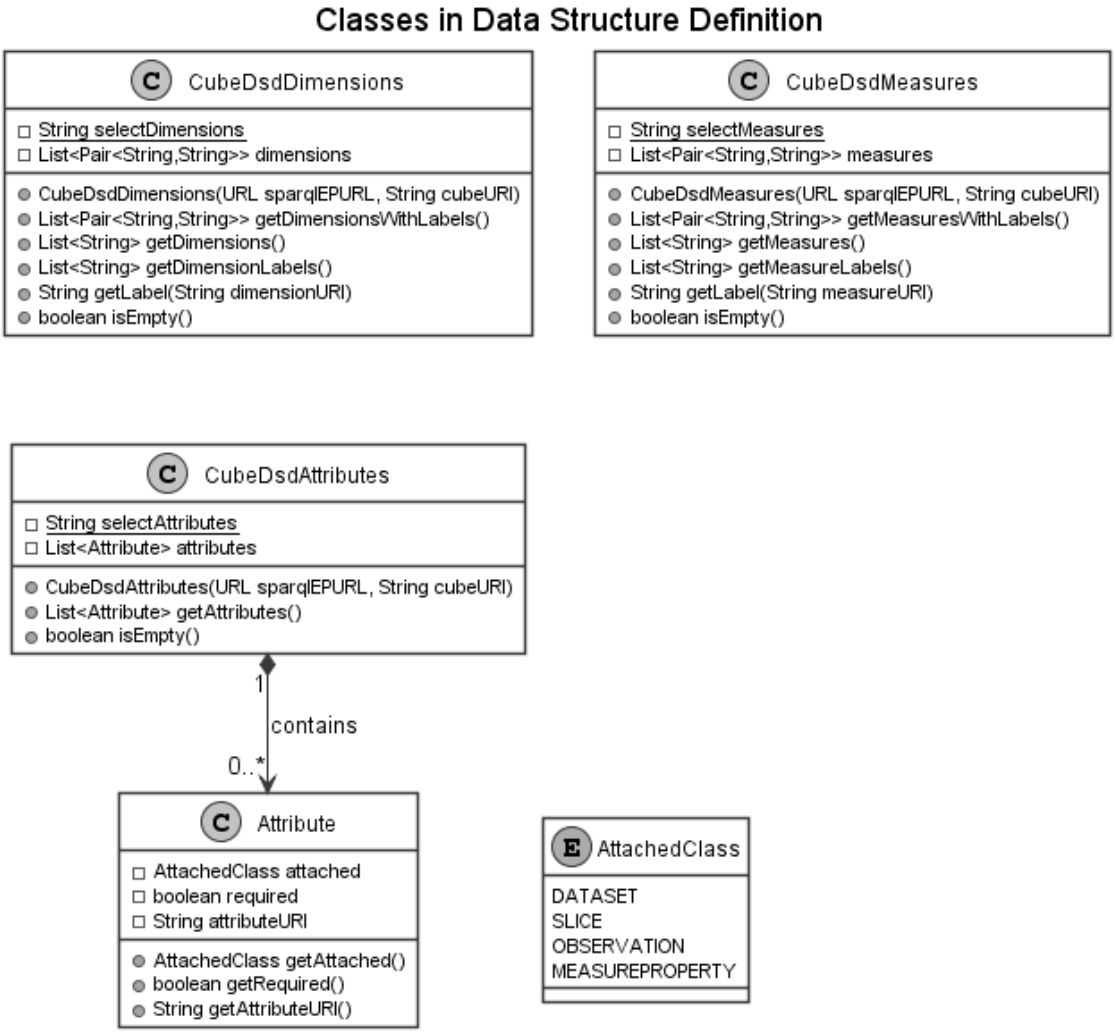

Figure 47: Basic classes

There are some basic classes that are used during the program, and their goal is to keep information of the core structural elements of a Cube, such as the dimensions the measures and the attributes. Figure 47 shows these basic classes. They are implemented in correspondence to the classes that are present in RDF Cube model, which are *qb:DimensionProperty*, *qb:MeasureProperty* and *qb:AttributeProperty*. Each class as it is shown in Figure 47 is actually a collection of the *qb:DimensionProperty*, *qb:MeasureProperty* or *qb:AttributeProperty* instances of the Cube. This is visible by the fact that each class holds a list of the set of dimensions or measures or attributes. This list is built by querying the dataset when an object of one of the classes is created. For example when an object of *CubeDsdDimensions* is created, a query is performed and all dimensions of a Cube are collected and stored in a list. The list contains the URI of the dimension and its label literal and it is maintained during the lifetime of the object.

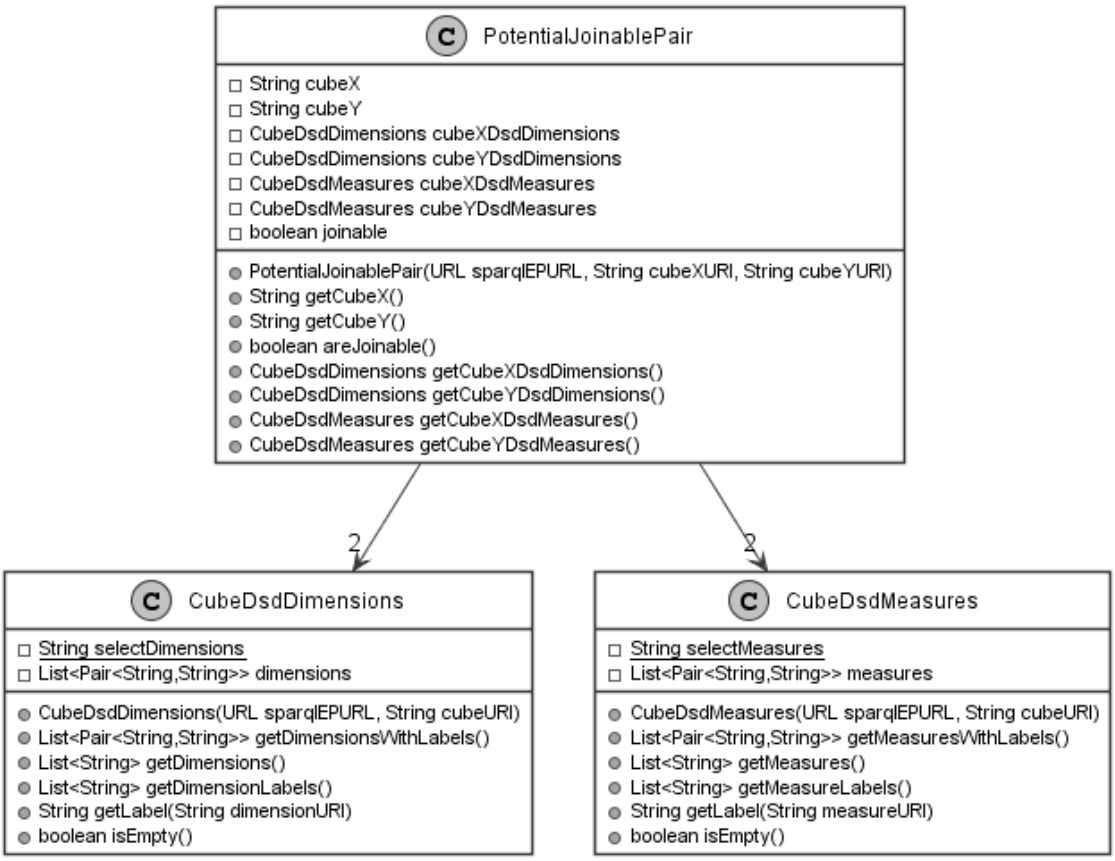

Figure 48: Potential joinable pair class diagram

During iteration on the list of pairs the program creates an object of class *PotentialJoinablePair*. This class is shown in the diagram in Figure 48 and it is visible that at creation of this object, two more objects are created coming from the basic classes, which are the *CubeDsdDimensions* and *CubeDsdMeasures*. These two objects collect through SPARQL queries the information about the dimensions and the measures of the pair of Cubes and this information is used by the caller object which applies the first two constraints and verdicts if the pair is joinable. The verdict is stored in the object and is available through a getter method. The multiplicity on the association is **"2"** because every *PotentialJoinablePair* object contains two *CubeDsdDimensions* objects and two *CubeDsdMeasures* objects, one for each Cube of the pair. The creation of a *PotentialJoinablePair* and the production of the verdict is a time consuming process because it involves four queries to the SPARQL endpoint, and has to take the impact of the HTTP latency.

When the parallel iteration on the list of all Cube pairs is complete, a new list with objects of *PotentialJoinablePair* that are found to be joinable is built. This list is iterated sequentially seeking for the object values of each dimension of each Cube in

order to perform calculations for the overlapping of the collected object values. Figure 49 shows the classes that are used to achieve this functionality of the program.

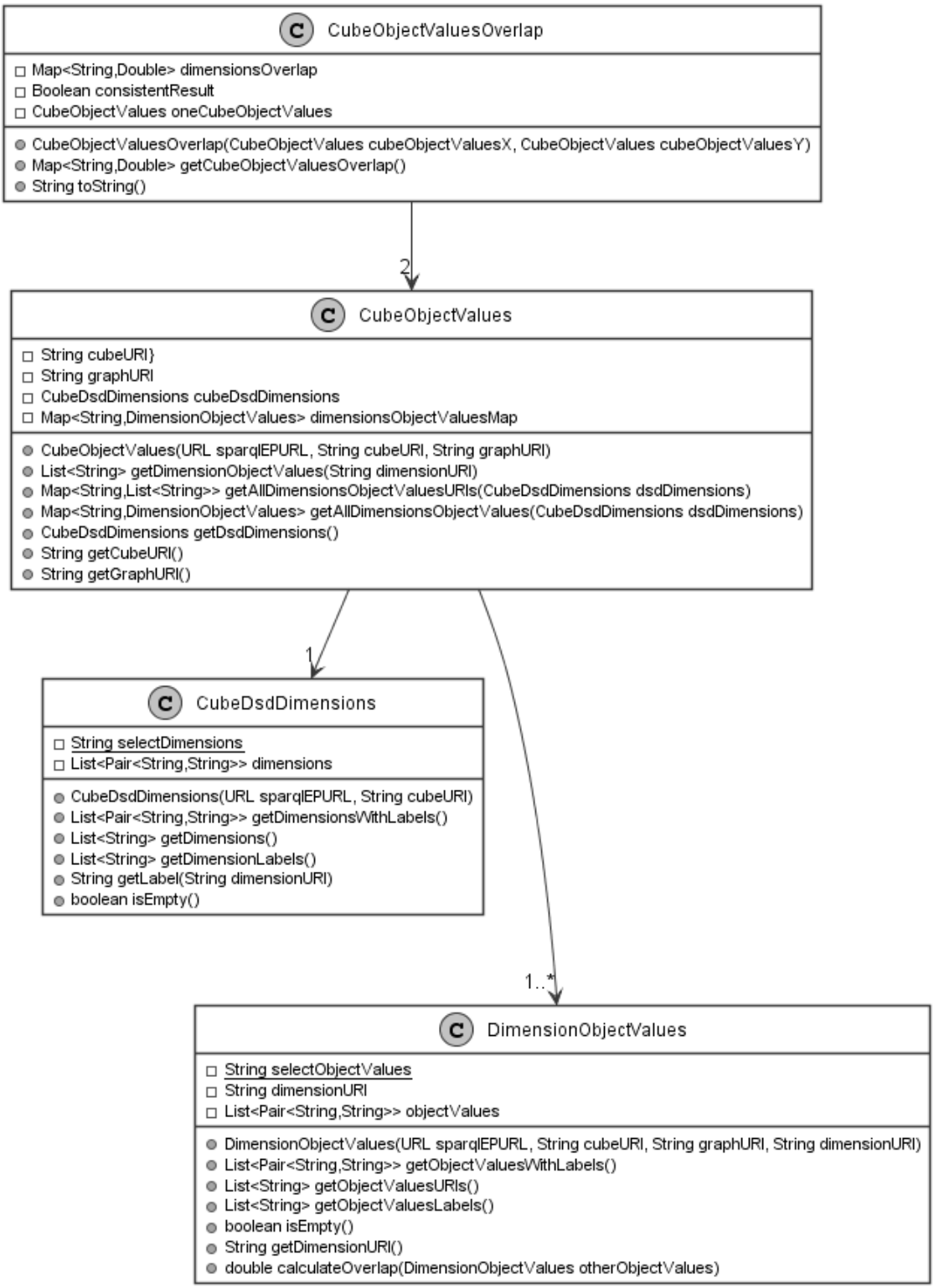

Figure 49: Object values overlap class diagram

For each joinable pair, an object of type *CubeObjectValuesOverlap* is created which is the root of every calculation between two cubes. This object retains a Java Map

structure, which contains the final overlapping result. The Map relates one dimension URI with a decimal value showing what is the overlapping percentage according to the formula presented in chapter 6. Each object of type *CubeObjectValuesOverlap* creates two objects of type *CubeObjectValues* explaining the multiplicity of **"2"** in their association and these two objects correspond to the two Cubes of the pair which will undergo the object values collection and overlap calculation.

The *CubeObjectValues* object needs one object of the basic class *CubeDsdDimensions* which contains all the dimensions of the Cube. Thus the multiplicity in the relation between classes *CubeObjectValues* and *CubeDsdDimensions* is **"1"**. The dimensions stored in the latter object, are used by the higher one in order to collect the object values for each dimension. This is achieved by creating new objects of type *DimensionObjectValues* as many as the number of dimension, found in the *CubeDsdDimensions* object. This justifies the multiplicity of **"1.."** between *CubeObjectValues* and *DimensionObjectValues*. At creation of a *DimensionObjectValues* object, the dataset is queried for all the object values of a specific dimension of a cube. These object values are stored in a list. Additionally the class *DimensionObjectValues* provides a method which takes as an argument an object of the same kind. It uses both objects –the current and the one passed as an argument to the method– to calculate what is the overlap percentage between the common dimension of two Cubes.

7.4 Summary

This chapter brings into practice the theoretical analysis that is presented in the previous chapters. The practical application is materialized with a program that runs against a dataset containing Linked Data Cubes. The program is generic but assumes that a dataset consists of original and aggregated Cubes with the latter having a URI which extends the original one. The implementation is based on a set of requirements. These high level requirements drive the design of the use cases of the program, some activity diagram to keep track of the flow and finally the design of the classes that do the actual work. The classes that are designed are described to a degree, using UML class diagrams to visualize the relationship between them. The presented classes are not an exhaustive list of all that are created in the program code, but a selection of the most representative. Additionally the program makes use of two widely accepted APIs to minimize the risk of faults and the re-implementation of widely used frameworks. These

are the Jena and the Commons CLI, both provided and licensed by the Apache foundation.

8 Data analysis and findings assimilation

8.1 Introduction

The program that is developed and described in chapter 7 can be executed according to the supported use cases. This chapter presents the results of the program execution towards the Flemish Government dataset, which is described in 8.2.

During all program executions, a basic filter file is used, to keep out of the calculations some Cubes of the dataset which are known to be created as temporary Cubes for other studies that were done using the same dataset. The contents of the basic filter file are shown in Figure 50 and they are written in Turtle as it is already described.

```
PREFIX qb: <http://purl.org/linked-data/cube#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX oc: <http://opencube-project.eu/>
PREFIX scot: <http://statistics.gov.scot/data/>
PREFIX fl-cube: <http://id.vlaanderen.be/statistieken/dq/>

oc:cube_5568892507773617436 a qb:DataSet .
oc:cube_5425538940707008831 a qb:DataSet .
fl-cube:kubus-arbeidsmarkt-swse-employment#id a qb:DataSet .
fl-cube:kubus-gemiddelde-prijs-1993#id a qb:DataSet .
scot:housing/house-sales-prices a qb:DataSet .
scot:economic-activity-benefits-and-tax-credits/employment a qb:DataSe
```

Figure 50: Basic filter TTL

The results of the program execution are presented either as a complete printout of the program or as plots with statistical view on them. Finally some execution characteristics are provided, which were collected while monitoring the program during runtime.

8.2 The Flemish Government dataset

Although the implementation has a goal to be generic enough, it is needed to understand the dataset internals as the program will interact with it. In this subchapter there is a description of the Flemish Government dataset.

This dataset is fully accessible through a single SPARQL endpoint in the following URL: http://188.166.18.242:8890/sparql. This endpoint offers a form to insert SPARQL queries by hand but it also responds to any HTTP interactions. The dataset as such contains many Linked Data Cubes which are modeled according to the description

provided in 2.3 and the overview shown in Figure 9. Thus each Cube is a different *qb:DataSet*. At this point it is needed to clarify that the dataset which is accessible through the SPARQL endpoint is different from the *qb:DataSet* concept which dictates the way to model distinct Linked Data Cubes. Each Cube (*qb:DataSet*) in the dataset is stored in a different graph, which is also a concept of RDF and allows to separate (group) the stored triples.

The dataset contains Linked Data Cubes with data collected by the Flemish Government, but some additional Cubes were stored during various research activities containing special versions of the initial Cubes or containing data from other authorities, such as the Scottish regional authorities. These additional Cubes need to be excluded during the evaluation of the dataset from the program. All these Cubes which contain native data are called original Cubes at the rest of the text. Besides the original Cubes, there are also other Cubes which have emerged by aggregation process on dimensions. For each original Cube aggregation has been applied to one of its dimensions, and the aggregated Cube is used to apply aggregation to another dimension, until the process reaches a single dimension.

The following example explains how the aggregated Cubes are stored in the dataset: Two prefixes are considered for common parts of the URIs which are used in the dataset.

```
PREFIX fl-cube: <http://id.vlaanderen.be/statistieken/dq/>
PREFIX fl-def: <http://id.vlaanderen.be/statistieken/def#>
```

Then one can see the original "Cube land register": **fl-cube: kubus-kadaster#id** with the following dimensions:

dimensions	labels
fl-def:refArea	The country or geographic area to which the measured statistical phenomenon relates.
fl-def:timePeriod	The period of time or point in time to which the measured observation refers.
fl-def:oppervlaktetype	Type of surface area.

Figure 51: "Cube land register" dimensions

When aggregation is applied to all combinations of the dimensions in Figure 51 then the following aggregated Cubes occur:

- fl-cube:kubus-kadaster#id_refArea_timePeriod_6181226772279384784
- fl-cube:kubus-kadaster#id_refArea_oppervlaktetype_562767933852780600
- fl-cube:kubus-kadaster#id_timePeriod_oppervlaktetype_6812170408054366028
- fl-cube:kubus-kadaster#id_refArea_7094041710006777789
- fl-cube:kubus-kadaster#id_timePeriod_5971220929863271458
- fl-cube: kubus-kadaster#id_oppervlaktetype_8101972862609043143

All Cubes that emerged through aggregation process belong to the same aggregation set and this is stated in the dataset using the RDF property: <http://opencube-project.eu/aggregationSet>. In the rest of the text we refer to them as the aggregated Cubes. All the aggregated cubes are pre-calculated and stored in the dataset.

The program is executed towards the Flemish Government dataset with the following three variations, which are analyzed further in the next subchapters.

1. Collect information for original Cubes
2. Collect information for original and aggregated Cubes
3. Collect information for original and aggregated Cubes excluding the aggregated ones that have ended up with a single dimension.

8.3 Execution on original Cubes

The first execution of the program is performed seeking which of the original Cubes in the Flemish Government dataset are joinable. Figure 52 shows the result, as well the complete printout of the program. Most the different phases of the program execution can be detected, as they are described in subchapter Activity 7.3.2.

The program prints information on various calculations as execution is moving on. When the list of Cubes is acquired the total number of Cubes and the total number of combined pairs of Cubes are printed. Subsequently the number of joinable pairs is printed, followed by the pair's Cubes URIs. Finally all the dimensions are printed with the overlap percentage on the object values. The complete list with this information is printed for all the joinable Cubes.

```
Found ORIGINAL cubes in: 1603 ms.

http://id.milieuinfo.be/kubus/luchtemissies#id
LABEL: Cube air emissions, source: milieuinfo.be

http://id.vlaanderen.be/statistieken/dq/kubus-rvz-bedrijfstak#id
LABEL: Cube type of industry self-employed

http://id.vlaanderen.be/statistieken/dq/kubus-gemiddelde-prijs#id
LABEL: Cube average price real estate

http://id.vlaanderen.be/statistieken/dq/kubus-kadaster#id
LABEL: Cube land register

http://id.vlaanderen.be/statistieken/dq/kubus-rsvz-aard-van-bezigheid#id
LABEL: Cube type of activity self-employed

http://id.vlaanderen.be/statistieken/dq/kubus-bouwvergunningen#id
LABEL: Cube building permits

http://id.vlaanderen.be/statistieken/dq/kubus-nationaliteit-nwwz#id
LABEL: Cube nationality non-working jobseekers

http://id.vlaanderen.be/statistieken/dq/kubus-arbeidsmarkt-swse#id
LABEL: Cube labourmarket

http://id.vlaanderen.be/statistieken/dq/kubus-studieniveau-nwwz#id
LABEL: Cube educationlevel non-working jobseekers

http://id.vlaanderen.be/statistieken/dq/wonen-sociale-huisvesting-
kubus#id LABEL: Cube social housing

http://id.vlaanderen.be/statistieken/dq/Kubus-voorkeursregeling-in-de-
ziekteverzekering#id
LABEL: Cube preference scheme in the health insurance

Got labels for cubes in: 2390 ms.

Processing cubes, identifying joinable...

Number of all cubes: 11

Number of all pairs: 55

----------

Built potential joinable pairs in: 15437 ms.

Number of good to join pairs: 1

----------

http://id.vlaanderen.be/statistieken/dq/kubus-arbeidsmarkt-swse#id vs.
http://id.vlaanderen.be/statistieken/dq/Kubus-voorkeursregeling-in-de-
ziekteverzekering#id

Cube labourmarket vs. Cube preference scheme in the health insurance

ARE JOINABLE: TRUE

http://purl.org/linked-data/sdmx/2009/dimension#sex - The state of being
male or female. OVERLAP 100.

http://id.vlaanderen.be/statistieken/def#refArea - The country or
geographic area to which the measured statistical phenomenon relates.
OVERLAP 100.

http://id.vlaanderen.be/statistieken/def#leeftijdsgroep - The length of
time that a person has lived in age groups. OVERLAP 72.4

http://id.vlaanderen.be/statistieken/def#timePeriod - The period of time
or point in time to which the measured observation refers. OVERLAP 45.4

----------

Analyzed good to join pairs in: 21486 ms.

Overall processing time: 21486 ms.
```

Figure 52: Original Cubes execution printout

This information is considered useful since it allows the user to decide which of the joinable Cubes would make sense to really join, according to the Cube's label (indicating the contents) and the overlapping percentage on object values. In the case of few joinable Cubes, the complete list is manageable. In between the lines an approximation of the processing time is printed, which highlights the time consuming activities.

8.4 Execution on original and aggregated Cubes

The next execution of the program is performed scanning the whole Flemish Government dataset, which includes both the original Cubes and the aggregated ones. Since the number of Cubes taken into account during this run is much bigger than the run with the original Cubes, this subchapter provides a statistical analysis of the result, and not the complete printout of the program. Some parts of the program output are provided though in Figure 53, as they contain essential information.

```
Found AGGREGATED cubes in: 1563 ms.

http://id.vlaanderen.be/statistieken/dq/kubus-studieniveau-nwwz
#id_refArea_timePeriod_leeftijdsgroep_sex_2076382561997635220
LABEL: Cube educationlevel non-working jobseekers

http://id.vlaanderen.be/statistieken/dq/kubus-nationaliteit-nwwz
#id_refArea_leeftijdsgroep_4346448899998041737
LABEL: Cube nationality non-working jobseekers

• • •

Processing cubes, identifying joinable...

Number of all cubes: 199

Number of all pairs: 19701

----------

Built potential joinable pairs in: 4652000 ms.

Number of good to join pairs: 239

----------

http://id.vlaanderen.be/statistieken/dq/kubus-nationaliteit-
nwwz#id_timePeriod_leeftijdsgroep_sex_5870456696435121336 vs.
http://id.vlaanderen.be/statistieken/dq/kubus-arbeidsmarkt-swse#id_timeP

• • •
```

Figure 53: Original and aggregated Cubes execution printout

In Figure 53 it is visible that all Cubes which will be scanned are 199 and their pairing results to a total of 19701 pairs. After all the calculation work, the result is that there are 239 joinable pairs, and each pair has its own overlapping percentages on every common

dimension. This information is useful to decide whether a specific pair of Cubes is worth a natural join, but having such a long list of results does not allow seeing the big picture of the dataset.

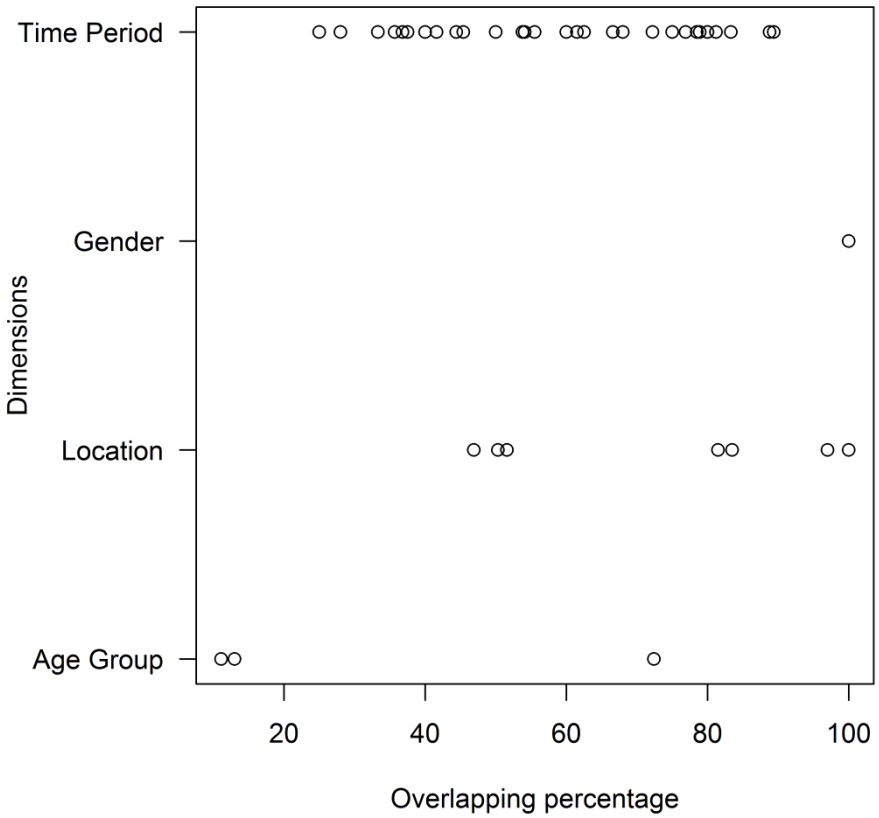

Figure 54: Scatterplot for overlapping on all Cubes

Statistical view of the overlapping results can provide the big picture of the dataset and also allow a user to understand where the overlapping percentage is moving for each dimension. This statistical overview is achieved by creating scatterplots and boxplots on various observations of the overlapping results. Initially the whole result received from the program is plotted in Figure 54 and Figure 55. The scatterplot shows how the various calculated overlapping percentages are spread for each dimension. Each circle in the plot is not showing a unique occurrence of an overlapping value but may cover multiple occurrences of the same value. This form of scatterplot does not differentiate in some way if there are one, two or thirty occurrences of overlapping value “100” on the Gender dimension. To visualize better the statistics there is a boxplot with the same data, which shows the minimum, and maximum values that occur, but also highlights the density of the values in ranges. Moreover the boxplot provides a view of the median of the data, as the bold dash in each box. This helps to identify towards where the

population of data is leaning. Finally the boxplot provides the notion of "outliers" which are values standing away from the generic concentration of the population of values.

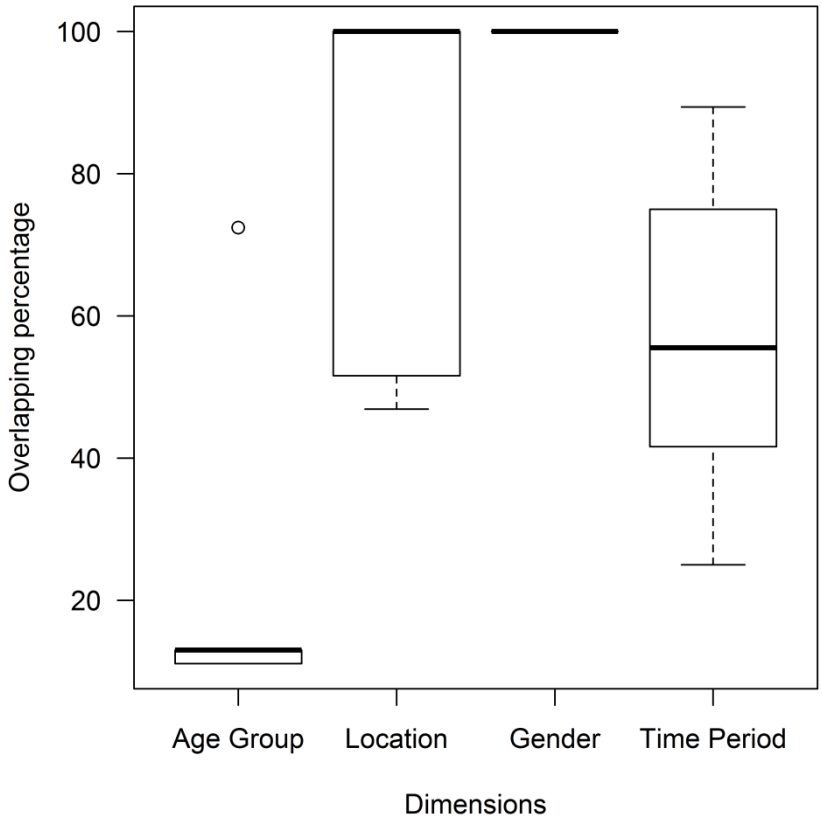

Figure 55: Boxplot of overlapping on all Cubes

In Figure 55 it is visible that there is an outlier indication at the Age Group dimension. Additionally the Location dimension shows that there are overlapping percentages that have gone as low as 50%, but the median shows that the population of values is concentrated very high, at 100%. This is an indication that in the dataset there may be a few Cubes which have poor overlapping. This could call for enhancing the filter to exclude them from the calculation in order to keep the 100% overlapping on this dimension.

Analysis on groups of Cubes, according to the number of their dimensions

The statistical analysis is further broken down by separating the data, into groups which contain either four dimensioned Cubes, three dimensioned Cubes, two dimensioned Cubes or unidimensional Cubes.

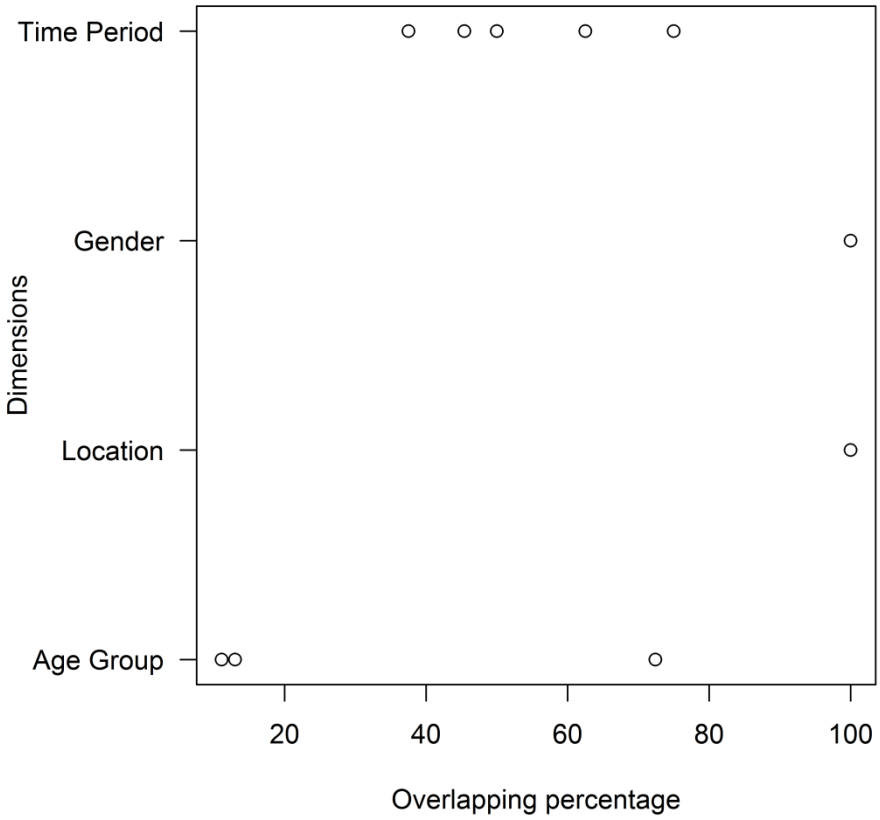

Figure 56: Scatterplot for overlapping on four dimensional Cubes

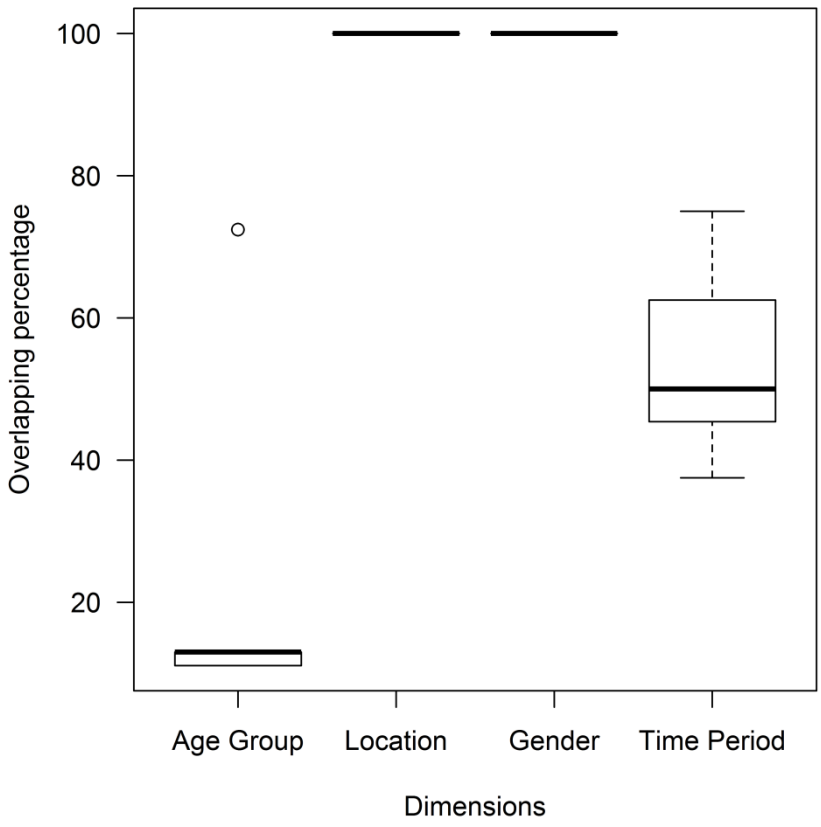

Figure 57: Boxplot for overlapping on four dimensional Cubes

Plotting the overlapping percentages of joinable Cubes which have four dimensions is shown in Figure 56 and Figure 57. It is observed that the Location dimension has overlapping of 100% and the Time Period dimension has more density around the median. The Age Group dimension retains its behavior in the case of four dimensional Cubes.

Overlapping percentages of three dimensional Cubes are shown in Figure 58 and Figure 59. Location dimension retains a 100% overlap and Age Group dimension presents a steady behavior, with the same outlier value. Time Period dimension has more expansion towards the minimum and the maximum but it can be seen that the expansion is larger towards the maximum overlap values. The median though seems to remain steady at almost 50%.

Overlapping percentages of two dimensional Cubes are shown in Figure 60 and Figure 61. Location dimension has various overlapping values expanding from 50% to 100%. The concentration though seems to be near 100% since the median is between 95% and 100%. Age Group dimension is still steady. Time Period dimension shows a slight shift upwards, both for the box and the median, which passes 50%.

Overlapping percentages of unidimensional Cubes are shown in Figure 62 and Figure 63. Location dimension expands a little further towards 50% but the median remains high between 95% and 100%. Age Group dimension continues its steady behavior. Time Period dimension shows a slight shift of the median, which reaches 60%.

In all plots the Gender dimension remains steady at 100% which can be naturally explained by the fact that it is a dimension with only two values (Female, Male) and all Cubes, either original or aggregated use both of these values in their data.

The persistence though of the Age Group dimension in all the above cases, at a steady distribution of overlap, with an outlier high at 75% cannot be explained easily. Detailed investigation of the printout of the program could reveal the Cubes that drive this behavior. Additionally focused evaluation check on these cubes may highlight if it is valid to have this overlapping values, or if there is any issue in the program reading the values or even if there is some problem in the dataset in the way it has the values stored.

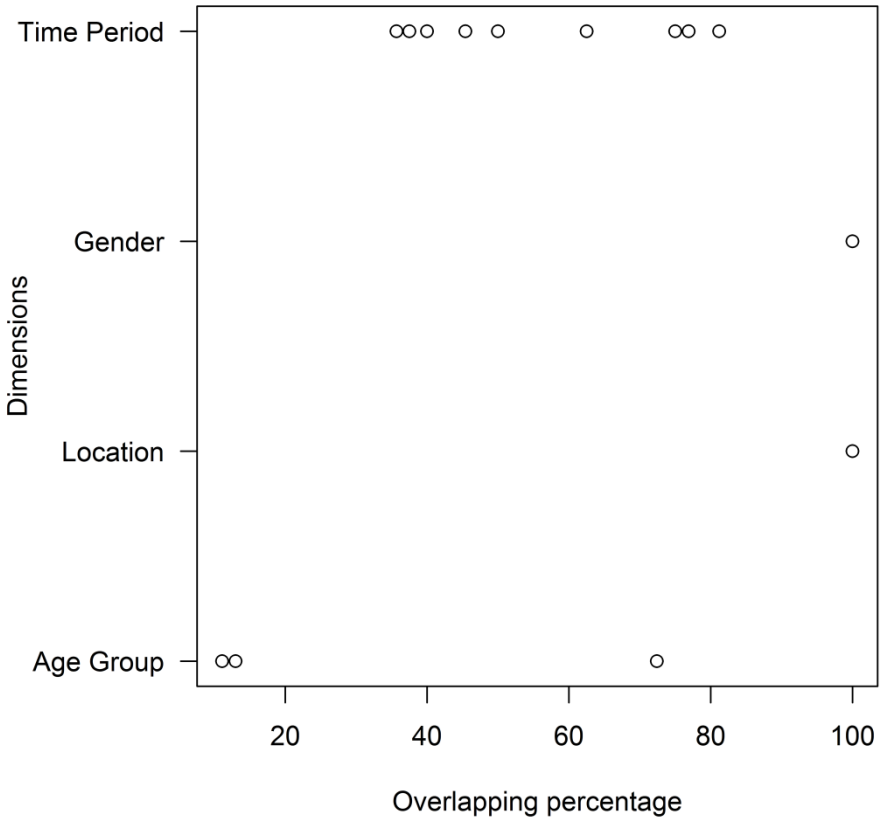

Figure 58: Scatterplot for overlapping on three dimensional Cubes

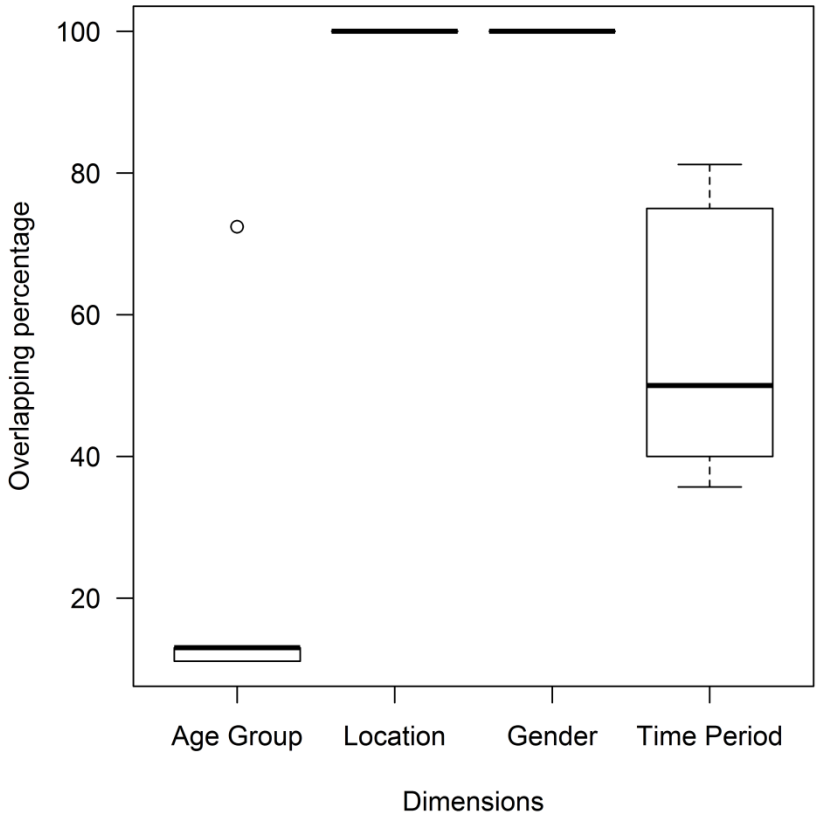

Figure 59: Boxplot for overlapping on three dimensional Cubes

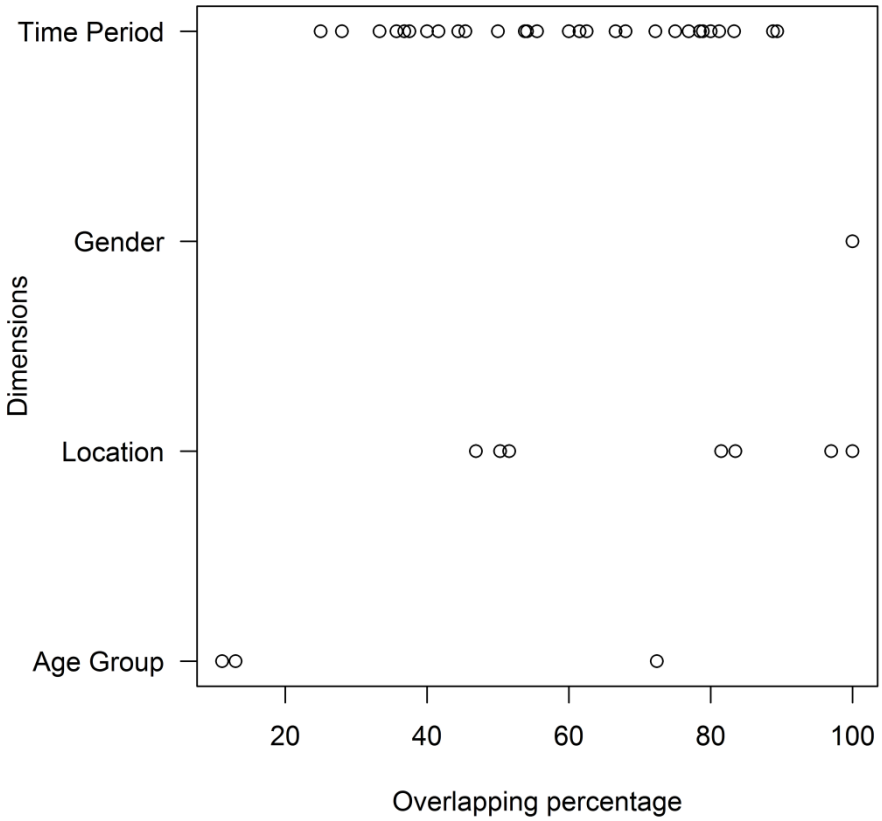

Figure 60: Scatterplot for overlapping on two dimensional Cubes

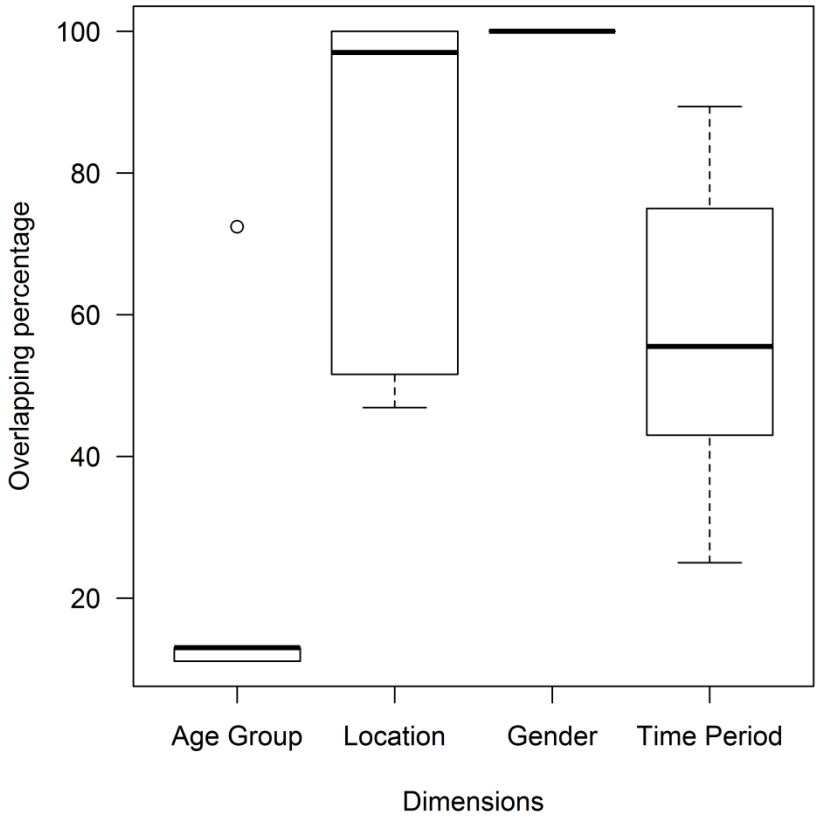

Figure 61: Boxplot for overlapping on two dimensional Cubes

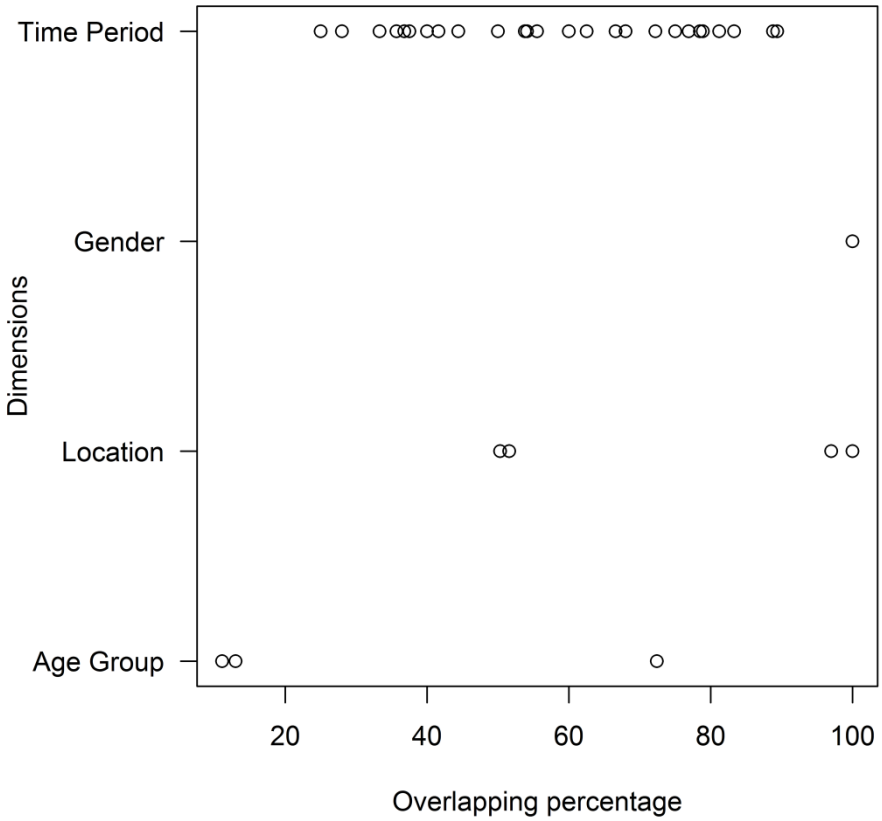

Figure 62: Scatterplot for overlapping on unidimensional Cubes

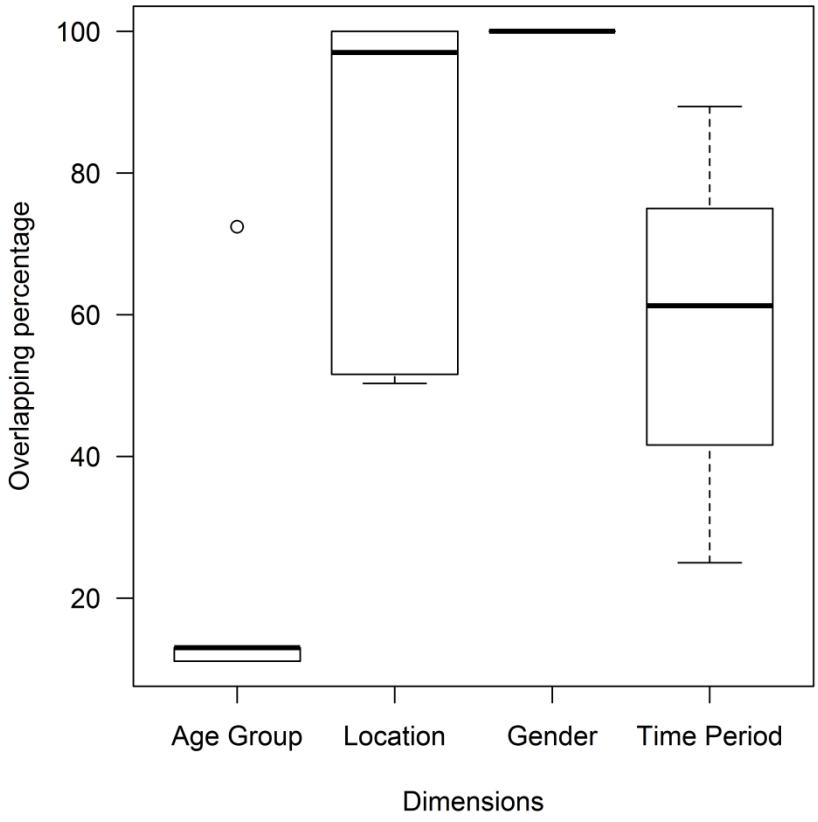

Figure 63: Boxplot for overlapping on unidimensional Cubes

8.5 Execution on original and aggregated Cubes excluding unidimensional

The statistical analysis in the previous subchapter shows that two dimensional Cubes and unidimensional Cubes have similar overlapping behavior regarding the values of their dimensions. Having a Cube with a single dimension though seems awkward since Cubes are multidimensional structures, unless this unidimensional Cube has a special meaning. Specifically in the Flemish Government dataset all unidimensional Cubes emerged through the aggregation process, so they are special in that sense.

The implemented program provides the tools to exclude unidimensional Cubes from the calculation, by enhancing the filter file with the unidimensional Cubes URIs. Then it is executed again with the new filter file and the "-a" argument. Part of the execution printout is shown in Figure 64.

```
Found AGGREGATED cubes in: 1469 ms.
http://id.vlaanderen.be/statistieken/dq/kubus-studieniveau-
nwwz#id_timePeriod_sex_educationLev_751906361034961950
LABEL: Cube educationlevel non-working jobseekers
http://id.vlaanderen.be/statistieken/dq/kubus-rsvz-aard-van-
bezigheid#id_refArea_timePeriod_zelfstandigenstatuut_sex_184925197520316
0890 LABEL: Cube type of activity self-employed

• • •
Processing cubes, identifying joinable...
Number of all cubes: 156
Number of all pairs: 12090
----------
Built potential joinable pairs in: 2862635 ms.
Number of good to join pairs: 135
----------
http://id.vlaanderen.be/statistieken/dq/kubus-nationaliteit-
nwwz#id_timePeriod_leeftijdsgroep_sex_5870456696435121336 vs.
http://id.vlaanderen.be/statistieken/dq/Kubus-voorkeursregeling-in-de-
ziekteverzekering#id_timePeriod_leeftijdsgroep_sex_3238490060760410594

• • •
```

Figure 64: Original and aggregated Cubes excluding unidimensional, execution printout

It is visible that the number of cubes is reduced to 156, and accordingly the number of pairs to be checked is reduced to 12090 instead of 19701. Additionally the joinable pairs are 135 in contrast to 239 of the full dataset. The difference between this execution and

the one that included all Cubes is big as the numbers show, adding much to the overall execution time which needs to be considered.

The statistical analysis of this execution is not repeated, as the expected differences are small. The plots for the four dimensional, three dimensional and two dimensional Cubes are the same as the ones in the previous subchapter. The plots for unidimensional Cubes are not produced in this execution and the difference in the plot that includes all joinable pairs should be minimal.

8.6 Execution characteristics

The calculation of join eligibility and overlapping percentages is a time consuming process, because it involves queries towards a SPARQL endpoint. Since the object values are collected by querying the observations of a Cube, the returned result may be large, depending on the size of the Cube, extending the execution time of a query. Moreover, the size of the results may have an impact to the memory consumption of the program since there are lists with objects holding strings returned as results from the query. Currently the program does not take any precautions on memory management and relies on the available heap size that the Java virtual machine provides. The execution of this program towards the Flemish Government dataset was monitored using the "Oracle® Java Mission Control 5.5.0" tool which accompanies the Oracle JDK.

Figure 65: Program monitor at initialization

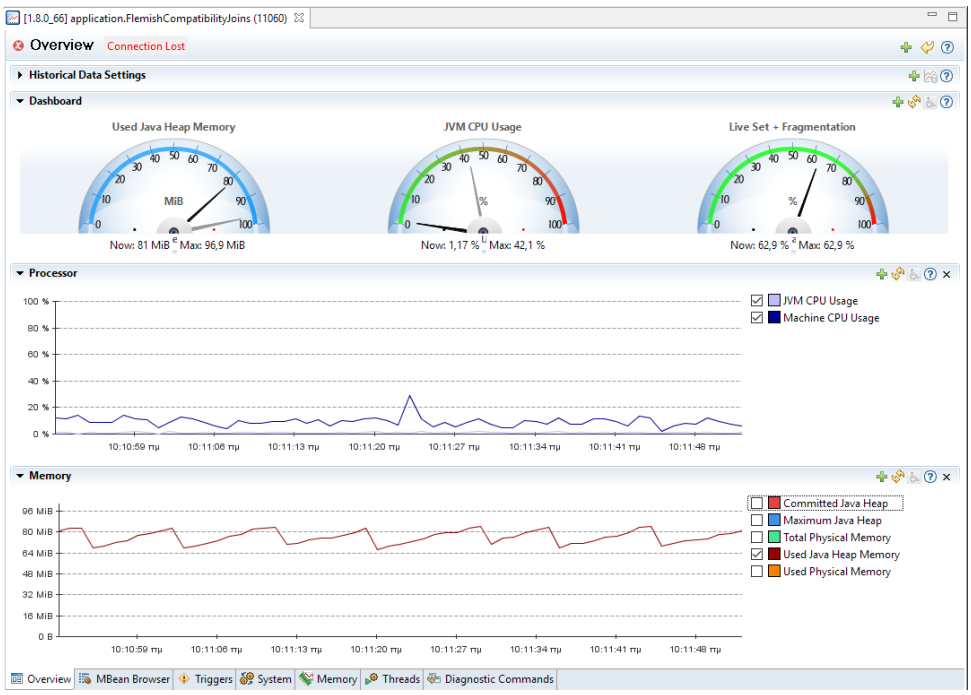

Figure 66: Program monitor at finish

Figure 65 and Figure 66 are two snapshots of the Mission Control tool at the initialization and finalization of the program respectively. It can be seen that the CPU consumption remains below 20% throughout the execution, which is expected since the program is bound to the network I/O. The memory consumption begins at around 20MiB and at the end it is approximately 80MiB, while during execution it has reached a maximum of 100MiB.

The table in Figure 67 summarizes the execution times of the program as they were measured during the three variations described in 8.2. The values presented in the cells are aggregated times.

	Collect Cubes	Calculate on dimensions	Calculate overlapping object values
Original Cubes	1.6 s	15.4 s	21.4 s
Original and aggregated Cubes	1.5 s	77.5 min	87.2 min
Original and aggregated excluding unidimensional Cubes	1.4 s	47.7 min	52.9 min

Figure 67: Execution times for three variations of the program

8.7 Summary

This chapter presents the results of executions of the program towards the Flemish Government dataset. There are three execution variations and the collected results are analyzed and presented. The first variation which scans only for original Cubes in the

dataset produces a single Cube pair which is joinable, allowing to present the complete execution printout. The next two variations of the program which use the aggregated Cubes, generate large printouts which have all the useful details for a user to know which Cubes worth joining. They do not provide though an overview of the results. For this reason a statistical analysis is applied on them which visualizes the overlapping percentages distribution by using charts. Finally some execution characteristics are presented, which must be taken into account by a user, since the execution of this program can take long time and the memory consumption may be high, depending on the size of the dataset that is scanned.

9 Conclusions and future work

The work described in this thesis is done to investigate how the join types found in relational algebra can be applied to Linked Data Cubes. While a theoretical background is established through publications and literature for multidimensional structures, the analysis and in depth investigation revealed that even the natural join of Cubes which is a clean form of join requires some constraints to be met and poses various other considerations such as the metric on object values overlapping and the structural impact of the result Cube in case there are *Attributes* in the source Cubes.

The findings during analysis of applying natural join to Linked Data Cubes drive the implementation of a program that can run against a dataset over a SPARQL endpoint and provide results on the eligibility to join Cubes that exist in the dataset. The execution of the program towards the Flemish Government dataset which contains Linked Open Data Cubes highlighted the Cubes that can be joined and also provided results on the metric of overlapping object values on the dimensions of the Cubes.

One limitation that was consciously used during the overlapping metric calculation is the fact that the levels of hierarchy on dimensions are not taken into account. This limitation does not make the results faulty or defective, but helped fulfill the proof of concept.

The program can be executed stand alone, but it is designed in the hope that the Class structure could offer an API usable also through other applications.

The execution of the program towards big datasets with many Cubes revealed the need for additional statistical analysis of the results. The statistical analysis is post calculated on the collected results, after the program end.

An enhancement of the program would be to use the hierarchy levels of the dimensions during the overlapping calculation. The current calculation of overlapping on the whole dimension stands, but it can also be broken down be performed on the hierarchy levels that are used in each dimension.

Another enhancement of the program would be to calculate the statistics during runtime and offer a use case to filter Cubes according to a desired range of overlapping percentage per dimension or per hierarchy level of a dimension.

An even further enhancement would be to integrate and adapt the API of the program with a Cube explorer and offer the ability to explore which Cubes are joinable and filter them according to desired overlapping percentages on their dimensions.

In this thesis a foundation for joining Linked Data Cubes is set and the concepts are proven so far. Further development can make this join an ordinary operation.

10 References

[1] C. Bizer, T. Heath and T. Berners-Lee, "Linked data-the story so far," *International Journal on Semantic Web and Information Systems 5.3*, pp. 1-22, 2009.

[2] T. Berners-Lee, "Linked Data," 27 July 2006. [Online]. Available: http://www.w3.org/DesignIssues/LinkedData.html. [Accessed Nov. 2015].

[3] R. Kimball and M. Ross, The data warehouse toolkit: The definitive guide to dimensional modeling, John Wiley and Sons, Inc., 2013.

[4] E. F. Codd, S. B. Codd and C. T. Salley, "Providing OLAP (on-line Analytical Processing) to User-analysts: An IT Mandate," *Codd and Date*, vol. 32, 1993.

[5] R. Cyganiak and D. Reynolds, "The RDF Data Cube Vocabulary," W3C Recommendation, 16 January 2014. [Online]. Available: http://www.w3.org/TR/vocab-data-cube/. [Accessed Nov. 2015].

[6] S. Chaudhuri and D. Umeshwar, "An overview of data warehousing and OLAP technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65-74, 1997.

[7] A. Datta and E. Thomas, "The cube data model: a conceptual model and algebra for on-line analytical processing in data warehouses," *Decision Support Systems*, vol. 27, no. 3, pp. 289-301, 1999.

[8] T. Heath and C. Bizer, Linked Data: Evolving the Web into a Global Data Space (1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool, 2011.

[9] E. F. Codd, ""A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, 1970.

[10] E. F. Codd, The Relational Model for Database Management : version 2, Addison-Wesley Publishing Company, Inc., 1990.

[11] C. J. Date, An Introduction to Database Systems (eighth edition), Pearson Education, Inc., 2004.

[12] W3C SPARQL Working Group, "SPARQL 1.1 Overview," W3C Recommendation, 21 March 2013. [Online]. Available:

http://www.w3.org/TR/sparql11-overview/. [Accessed Nov. 2015].

[13] T. B. Pedersen and C. S. Jensen, "Multidimensional database technology," *Computer*, vol. 34, no. 12, pp. 40-46, 2001.

[14] R. Agrawal, A. Gupta and S. Sarawagi, "Modeling multidimensional databases," in *Proceedings of 13th IEEE International Conference on Data Engineering*, 1997.

[15] L. Cabibbo and R. Torlone, "From a procedural to a visual query language for OLAP," in *Proceedings of 10th IEEE International Conference on Scientific and Statistical Database Management*, 1998.

[16] T. Pedersen, C. Jensen and C. Dyreson, "A foundation for capturing and querying complex multidimensional data," *Information Systems*, vol. 26, no. 5, pp. 383-423, 2001.

[17] E. Franconi and A. Kamble, "The GMD Data Model and Algebra for Multidimensional Information," in *Proceedings of 16th International Conference on Advanced Information Systems Engineering*, 2004.

[18] T. Berners-Lee, J. Hendler and O. Lassila, "The Semantic Web," *Scentific American*, 2001.

[19] W3C, "RDF CURRENT STATUS," W3C, 2015. [Online]. Available: http://www.w3.org/standards/techs/rdf#w3c_all. [Accessed Nov. 2015].

[20] G. Schreiber and Y. Raimond, "RDF 1.1 Primer," W3C, 2014. [Online]. Available: http://www.w3.org/TR/rdf11-primer/. [Accessed Dec. 2015].

[21] R. Cyganiak, D. Wood and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," W3C, [Online]. Available: http://www.w3.org/TR/rdf11-concepts/. [Accessed Dec. 2015].

[22] D. Brickley and R. Guha, "RDF Schema 1.1," W3C Recommendation, 2014. [Online]. Available: http://www.w3.org/TR/rdf-schema/. [Accessed Dec. 2015].

[23] D. Beckett and T. Berners-Lee, "Turtle - Terse RDF Triple Language," W3C Team Submission, 2011. [Online]. Available: http://www.w3.org/TeamSubmission/turtle/. [Accessed Nov. 2015].

[24] "Apache Jena," The Apache Software Foundation, [Online]. Available: https://jena.apache.org/. [Accessed Jan. 2016].

[25] "Apache Commons CLI," The Apache Software Foundation, [Online]. Available: http://commons.apache.org/proper/commons-cli. [Accessed Jan. 2016].

[26] F. Ravat, O. Teste, R. Tournier and G. Zurfluh, "Algebraic and graphic languages for OLAP manipulations," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 4, no. 1, pp. 17-46, 2008.

[27] O. Romero and A. Abelló, "A survey of Multidimensional Modeling Methodologies," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 5, no. 2, pp. 1-23, 2009.

[28] DCMI Usage Board, "DCMI Metadata Terms," DCMI, 2012. [Online]. Available: http://dublincore.org/documents/dcmi-terms/. [Accessed Nov. 2015].

[29] Semantic Web Deployment Working Group, "SKOS Simple Knowledge Organization System - Home Page," W3C, 2009. [Online]. Available: http://www.w3.org/2004/02/skos/core.html. [Accessed Nov. 2015].

[30] D. Brickley and L. Miller, "FOAF Vocabulary Specification," 2014. [Online]. Available: http://xmlns.com/foaf/spec/. [Accessed Nov. 2015].

[31] B. Kämpgen, S. O'Riain and A. Harth, "Interacting with statistical linked data via OLAP operations," in *The Semantic Web: ESWC 2012 Satellite Events*, Springer Berlin Heidelberg, 2012.

[32] L. Etcheverry and A. Vaisman, "Enhancing OLAP analysis with web cubes," in *The Semantic Web: Research and Applications*, Springer Berlin Heidelberg, 2012.

[33] E. Tambouris, E. Kalampokis and K. Tarabanis, "Processing Linked Open Data Cubes," *Electronic Government*, pp. 130-143, Springer International Publishing, 2015.

[34] L. Etcheverry, A. Vaisman and E. Zimányi, "Modeling and querying data warehouses on the semantic web using QB4OLAP," *Data Warehousing and Knowledge Discovery*, pp. 45-56, Springer International Publishing, 2014.

[35] R. Cyganiac, C. Dollin and D. Raynolds, "Expressing Statistical Data in RDF with SDMX-RDF," SDMX.org, 2010. [Online]. Available: http://publishing-statistical-data.googlecode.com/svn/trunk/specs/src/main/html/index.html. [Accessed Dec. 2015].