



UNIVERSIDADE FEDERAL DE PERNAMBUCO  
CENTRO DE INFORMÁTICA  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MÁRCIO ROBÉRIO DA COSTA FERRO

**AStar:** A Modeling Language for Document-oriented Geospatial Data Warehouses

Recife

2022

MÁRCIO ROBÉRIO DA COSTA FERRO

**AStar:** A Modeling Language for Document-oriented Geospatial Data Warehouses

Tese de Doutorado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Doutor em Ciência da Computação.

**Área de Concentração:** Banco de dados

**Orientador:** Robson do Nascimento Fidalgo

Recife

2022

Catálogo na fonte  
Bibliotecária Nataly Soares Leite Moro, CRB4-1722

F395a      Ferro, Márcio Robério da Costa  
              *AStar: a modeling language for document-oriented geospatial data  
              warehouses* / Márcio Robério da Costa Ferro. – 2022.  
              146 f.: il., fig., tab.

              Orientador: Robson do Nascimento Fidalgo.  
              Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da  
              Computação, Recife, 2022.  
              Inclui referências e apêndices.

              1. Banco de dados. 2. Data warehouse geoespacial. 3. Bancos de dados  
              orientados a documentos. 4. Esquema lógico. 5. DSML. I. Fidalgo, Robson do  
              Nascimento (orientador). II. Título

              025.04                      CDD (23. ed.)                      UFPE - CCEN 2022 – 97

**Márcio Robério da Costa Ferro**

**“AStar: A Modeling Language for Document-oriented Geospatial Data Warehouses”**

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Banco de Dados.

Aprovado em: 18/02/2022.

---

**Orientador: Prof. Dr. Robson do Nascimento Fidalgo**

**BANCA EXAMINADORA**

---

Profa. Dra. Ana Carolina Brandão Salgado  
Centro de Informática /UFPE

---

Prof. Dr. Luciano de Andrade Barbosa  
Centro de Informática /UFPE

---

Prof. Dr. Vinicius Cardoso Garcia  
Centro de Informática /UFPE

---

Prof. Dr. Cláudio de Souza Baptista  
Centro de Engenharia Elétrica e Informática, Sistemas e Computação/UFCEG

---

Prof. Dr. Vítor Estêvão Silva Souza  
Departamento de Informática/UFES

*I dedicate this work to all who supported and encouraged me.*

## **ACKNOWLEDGMENTS**

To God, for the opportunities for learning and growth.

To my parents, Cicero and Francisca, and my wife, Bruna, who supported me along this journey.

To Robson, for this dedication as an advisor for this work. In addition to being an advisor, he became my friend.

To Stenio, who referred me to the doctoral program and to select Robson as my advisor.

To the friends who shared this journey, Edson, Natalia, Thiago, and Rogerio, with whom I shared knowledge, travel, coffee, beer, and fun.

To the friends who supported me to begin this journey, Raquel and Eunice.

To the teaches and workers at the Informatics Center of the Federal University of Pernambuco.

To the Federal Institute of Alagoas, for supporting this thesis.

To all those who, in some way, contributed to my achievement of this goal.

## ABSTRACT

A Geospatial Data Warehouse (GDW) is an extension of a traditional Data Warehouse that includes geospatial data in the decision-making processes. Several studies have proposed the use of document-oriented databases in a GDW as an alternative to relational databases. This is due to the ability of non-relational databases to scale horizontally, allowing for the storage and processing of large volumes of data. In this context, modeling the manner in which facts and dimensions are structured is important in order to understand, maintain, and evolve the Document-oriented GDW (DGDW) through visual analysis. However, to the best of our knowledge, there are no modeling languages that support the design of facts and dimensions as referenced or embedded documents, partitioned into one or more collections. To overcome this lack, we propose Aggregate Star (AStar), a Domain-Specific Modeling Language for designing DGDW logical schemas. AStar is defined from a concrete syntax (graphical notation), an abstract syntax (metamodel), and static semantics (well-formedness rules). In order to describe the semantics of the concepts defined in AStar, translational semantics map the graphical notation to the metamodel and the respective code, to define the schema in MongoDB (using JSON Schema). We evaluate the graphical notation using Physics of Notations (PoN), which provides a set of principles for designing cognitively effective visual notations. This evaluation revealed that AStar is in accordance with eight of the nine PoN Principles, an adequate level of cognitive effectiveness. As a proof of concept, the metamodel and well-formedness rules were implemented in a prototype of Computer-Assisted Software Engineering tool, called AStarCASE. In its current version, AStarCASE can be used to design DGDW logical schemas and to generate their corresponding code in the form of JSON Schemas. Furthermore, we present a guideline that shows how to design schemas that have facts, conventional dimensions, and geospatial dimensions related as referenced or embedded documents, and partitioned into one or more collections. The guidelines also present good practices to achieve low data volume and low query runtime in a DGDW.

**Keywords:** geospatial data warehouse; document-oriented databases; logical schema; DSML.

## RESUMO

Um Data Warehouse Geoespacial (DWG) é uma extensão de um Data Warehouse tradicional que inclui dados geoespaciais nos processos de tomada de decisão. Diversos estudos propõem o uso de bancos de dados orientados a documentos em um DWG como alternativa aos bancos de dados relacionais. Isso se deve à capacidade dos bancos de dados não relacionais de escalar horizontalmente, permitindo o armazenamento e o processamento de grandes volumes de dados. Nesse contexto, modelar por meio da análise visual a maneira como fatos e dimensões estão estruturados é importante para entender, manter e evoluir o DWG Orientado a Documentos (DWGD). No entanto, até onde sabemos, não há linguagens de modelagem que deem suporte ao design de fatos e dimensões como documentos referenciados ou embutidos, particionados em uma ou mais coleções. Para superar essa lacuna, propomos Aggregate Star (AStar), uma linguagem de modelagem específica de domínio para projetar esquemas lógicos de DWGD. AStar é definida por uma sintaxe concreta (notação gráfica), uma sintaxe abstrata (metamodelo) e semântica estática (regras de boa formação). Para descrever a semântica dos conceitos definidos em AStar, semântica translacional é usada para mapear a notação gráfica para o metamodelo e o respectivo código que define o esquema no MongoDB (usando JSON Schema). Avaliamos a notação gráfica usando *Physics of Notations* (PoN), que fornece um conjunto de princípios para projetar notações visuais cognitivamente eficazes. Essa avaliação revelou que AStar está de acordo com oito dos nove Princípios PoN, um nível adequado de eficácia cognitiva. Como prova de conceito, o metamodelo e as regras de boa formação foram implementados em um protótipo de ferramenta de Engenharia de Software Assistida por Computador, denominado AStarCASE. Nesta versão atual, AStarCASE pode ser usada para projetar esquemas lógicos de DWGD e gerar seu código correspondente na forma de esquemas JSON. Além disso, apresentamos uma guia que mostra como projetar esquemas que possuem fatos, dimensões convencionais e dimensões geoespaciais relacionadas como documentos referenciados ou incorporados, particionados em uma ou mais coleções. O guia também apresenta boas práticas para obter baixo volume de dados e baixo tempo de execução de consulta em um DWGD.

**Palavras-chave:** data warehouse geoespacial; bancos de dados orientados a documentos; esquema lógico; DSML.

## LIST OF FIGURES

Figure 1 – Inserting examples of JSON documents “customer” and “order” into MongoDB: (a) two collections with referenced documents; (b) one collection with referenced documents; and (c) one collection with one embedded document. . . . .	23
Figure 2 – Defining a logical schema with JSON schemas and unique constraints, in MongoDB: two collections with referenced documents. . . . .	25
Figure 4 – Defining a logical schema with JSON schemas and unique constraints, in MongoDB: one collection with one embedded document. . . . .	25
Figure 3 – Defining a logical schema with JSON schemas and unique constraints, in MongoDB: one collection with referenced documents. . . . .	26
Figure 5 – Characteristics of a graphical notation symbol. . . . .	31
Figure 6 – The method used to search for related studies in the literature. . . . .	35
Figure 7 – Schema example of P01. . . . .	39
Figure 8 – Schema example depicted with the graphical notation proposed in P02. . . . .	41
Figure 9 – Schema example depicted with the graphical notation proposed in P03. . . . .	42
Figure 10 – On the left, the graphical notation to represent (a) embedded documents and (b) referenced documents. On the right, (c) a schema example for the proposal of P04. . . . .	43
Figure 11 – On the right, (a) the geospatial data representation using pictograms in documents. On the left, (b) an example of a schema modeled with the proposal from P05. . . . .	44
Figure 12 – The (a) graphical notation, (b, c) example schemas, and (d) mapping of symbols to JSON Schemas in P06. . . . .	47
Figure 13 – Example schema depicted using the proposal from P08. . . . .	48
Figure 14 – Metamodel that defines a generic logical model for NoSQL databases in P09. . . . .	50
Figure 15 – Example of hybrid databases modeled with the proposal from P10. . . . .	51
Figure 16 – Example schema depicted as a graph in P11. . . . .	53
Figure 17 – Metamodel for document-oriented databases proposed in P12. . . . .	54

Figure 18 – A (a) conceptual model example and the (b) metamodel for document-oriented documents proposed in P13. . . . .	56
Figure 19 – An (a) example diagram and its (b) document-oriented structuring alternatives - P14. . . . .	57
Figure 20 – Metamodel for document-oriented databases proposed in P16. . . . .	59
Figure 21 – (a) The Falling Star schema, and (b) its implementation in MongoDB - P07. . . . .	60
Figure 22 – Mapping the symbols of a DGDW for the document-oriented data model in P15. . . . .	62
Figure 23 – AStar graphical notation. . . . .	66
Figure 24 – Two ways to model the date dimension: (a) using package or (b) fully-qualified name in the DocumentType. . . . .	67
Figure 25 – Using an Embed to embed a dimension into a fact. . . . .	68
Figure 26 – Using a Reference N:1 to form a relationship between DocumentTypes from the same collection. . . . .	69
Figure 27 – Using a Reference N:1 to form a relationship between DocumentTypes from different collections. . . . .	69
Figure 28 – Modeling a role-play dimension. . . . .	70
Figure 29 – Use of Reference 1:1 and Reference M:N. . . . .	71
Figure 30 – AStar metamodel. . . . .	73
Figure 31 – AStar nodes and their implementation code in MongoDB. . . . .	80
Figure 32 – Field types and their implementation code in MongoDB. . . . .	81
Figure 33 – Data Types and their implementation code in MongoDB. . . . .	82
Figure 34 – An Embed and its implementation code in MongoDB. . . . .	82
Figure 35 – Reference 1:1 and its implementation code in MongoDB. . . . .	83
Figure 36 – Reference N:1 and its implementation code in MongoDB. . . . .	84
Figure 37 – Reference N:M and its implementation code in MongoDB. . . . .	85
Figure 38 – Semiotic clarity of AStar. . . . .	88
Figure 39 – AStarCASE overview. . . . .	93
Figure 40 – Errors and warnings issued by AStarCASE. . . . .	94
Figure 41 – Generation of JSON Schema using AStarCASE. . . . .	94
Figure 42 – Relationships between facts and dimensions. . . . .	97
Figure 43 – Relationships between CD and GD. . . . .	97

Figure 44 – Relationships between GDs. . . . .	97
Figure 45 – Steps performed to generate the DGDW logical schemas. . . . .	99
Figure 46 – RGDW schema used to generate the DGDW schemas. . . . .	99
Figure 47 – DGDW logical schemas depicted with AStar. The symbols (!), (+), (-), [+], and [-] are addressed in the experimental results. . . . .	100
Figure 48 – Process to build the 36 DGDWs from one RGDW. . . . .	101
Figure 49 – Volume vs. arithmetic mean of query execution times for the 34 evalu- ated schemas. . . . .	105
Figure 50 – Volume vs. arithmetic mean of query execution times for the 18 schemas having similar results. . . . .	105

## LIST OF TABLES

Table 1 – Selected studies. . . . .	36
Table 2 – Overview of related studies. . . . .	63
Table 3 – Associations that are allowed or prohibited between the graphical notation symbols in a logical schema. The numbers in the columns correspond to the enumerated items in the rows. C=Containment, L=Linkage, p=Prohibited. . . . .	77
Table 4 – AStar compliance with the PoN principles. . . . .	90
Table 5 – RGDW used in numbers. . . . .	102
Table 6 – Results of the experimental evaluation. Size in GB and query execution time in seconds. Symbols used to highlight the DGDW schemas: (+) Best results in volume size; (-) Worst results in volume size; [+] Best results in query performance; [-] Worst results in query performance; (!) Schemas discarded from experimental evaluation. . . . .	104

## LIST OF ABBREVIATIONS AND ACRONYMS

<b>CASE</b>	<i>Computer Assisted Software Engineering</i>
<b>DGDW</b>	<i>Document-oriented GDW</i>
<b>DSML</b>	<i>Domain-Specific Modeling Language</i>
<b>DW</b>	<i>Data Warehouse</i>
<b>EMF</b>	<i>Eclipse Modeling Framework</i>
<b>ER</b>	<i>Entity-Relationship</i>
<b>EVL</b>	<i>Epsilon Validation Language</i>
<b>GDW</b>	<i>Geospatial Data Warehouse</i>
<b>GMP</b>	<i>Graphical Modeling Project</i>
<b>GPML</b>	<i>General-Purpose Modeling Language</i>
<b>ICCS</b>	<i>International Classification of Crime for Statistical Purposes</i>
<b>M2M</b>	<i>Model-to-Model</i>
<b>M2T</b>	<i>Model-to-Text</i>
<b>MDD</b>	<i>Model-Driven Development</i>
<b>MOF</b>	<i>Meta-Object Facility</i>
<b>MQL</b>	<i>MongoDB Query Language</i>
<b>OCL</b>	<i>Object Constraint Language</i>
<b>OGC</b>	<i>Open Geospatial Consortium</i>
<b>OLAP</b>	<i>On-Line Analytical Processing</i>
<b>PoN</b>	<i>Physics of Notations</i>
<b>RGDW</b>	<i>Relational GDW</i>
<b>SFA</b>	<i>Simple Feature Access</i>
<b>SOLAP</b>	<i>Spatial On-Line Analytical Processing</i>
<b>SSB</b>	<i>Star Schema Benchmark</i>
<b>T2M</b>	<i>Text-to-Model</i>

**UML**

*Unified Modeling Language*

## CONTENTS

<b>1</b>	<b>INTRODUCTION . . . . .</b>	<b>16</b>
1.1	CONTEXT AND MOTIVATION . . . . .	16
1.2	RESEARCH OBJECTIVES . . . . .	18
1.3	SCOPE . . . . .	19
1.4	METHODOLOGY . . . . .	19
1.5	DOCUMENT ORGANIZATION . . . . .	20
<b>2</b>	<b>BACKGROUND . . . . .</b>	<b>22</b>
2.1	DOCUMENT-ORIENTED DATABASES . . . . .	22
2.2	LOGICAL DESIGN OF GEOSPATIAL DATA WAREHOUSE . . . . .	26
2.3	MODEL-DRIVEN DEVELOPMENT AND DOMAIN-SPECIFIC MODELING LANGUAGES . . . . .	29
2.4	GRAPHICAL DESIGN OF MODELING LANGUAGES . . . . .	30
2.5	CHAPTER FINAL CONSIDERATIONS . . . . .	33
<b>3</b>	<b>RELATED STUDIES . . . . .</b>	<b>34</b>
3.1	SELECTION OF STUDIES . . . . .	34
3.2	DISCUSSION . . . . .	36
<b>3.2.1</b>	<b>General-purpose . . . . .</b>	<b>38</b>
<b>3.2.2</b>	<b>DGDW schemas . . . . .</b>	<b>59</b>
3.3	OVERALL ANALYSIS . . . . .	62
3.4	CHAPTER FINAL CONSIDERATIONS . . . . .	64
<b>4</b>	<b>ASTAR . . . . .</b>	<b>65</b>
4.1	CONCRETE SYNTAX . . . . .	65
4.2	ABSTRACT SYNTAX . . . . .	72
<b>4.2.1</b>	<b>Metamodel definition . . . . .</b>	<b>72</b>
<b>4.2.2</b>	<b>Nodes Associations . . . . .</b>	<b>76</b>
4.3	STATIC SEMANTICS . . . . .	77
4.4	TRANSLATIONAL SEMANTICS . . . . .	78
4.5	CHAPTER FINAL CONSIDERATIONS . . . . .	85
<b>5</b>	<b>EVALUATION AND IMPLEMENTATION OF ASTAR . . . . .</b>	<b>87</b>
5.1	ASTAR EVALUATION . . . . .	87

5.2	ASTARCASE . . . . .	91
5.3	CHAPTER FINAL CONSIDERATIONS . . . . .	95
<b>6</b>	<b>GUIDELINES TO DESIGN LOGICAL SCHEMAS WITH ASTAR .</b>	<b>96</b>
6.1	MODELING FACTS AND DIMENSIONS . . . . .	96
6.2	EXPERIMENTAL EVALUATION . . . . .	98
<b>6.2.1</b>	<b>Experimental Results . . . . .</b>	<b>103</b>
<b>6.2.2</b>	<b>Data volume analysis . . . . .</b>	<b>105</b>
<b>6.2.3</b>	<b>Query runtime analysis . . . . .</b>	<b>107</b>
6.3	BEST PRACTICES . . . . .	108
6.4	CHAPTER FINAL CONSIDERATIONS . . . . .	109
<b>7</b>	<b>CONCLUSIONS . . . . .</b>	<b>110</b>
7.1	FINAL CONSIDERATIONS . . . . .	110
7.2	CONTRIBUTIONS . . . . .	112
7.3	THREAT TO VALIDITY, LIMITATIONS, AND FUTURE STUDIES . . . .	113
	<b>REFERENCES . . . . .</b>	<b>115</b>
	<b>APPENDIX A – JSON SCHEMAS FOR GEOSPATIAL DATA TY-</b>	
	<b>PES . . . . .</b>	<b>123</b>
	<b>APPENDIX B – EXAMPLES OF JSON SCHEMAS . . . . .</b>	<b>132</b>
	<b>APPENDIX C – IMPLEMENTATION OF ASTAR METAMODEL .</b>	<b>142</b>
	<b>APPENDIX D – ASTARCASE TRANSFORMATION M2T . . . . .</b>	<b>144</b>

# 1 INTRODUCTION

This chapter introduces the thesis and highlights its context and motivation, research objectives, scope, and methodology. At the end of this chapter, the structure of the remaining chapters is presented.

## 1.1 CONTEXT AND MOTIVATION

A *Geospatial Data Warehouse* (GDW) is a database used in decision-making processes that can store large volumes of historical, non-volatile, and well-structured conventional and geospatial data (FIDALGO et al., 2004). In other words, a GDW is a *Data Warehouse* (DW) (KIMBALL; ROSS, 2013) with additional support to store geospatial data that can represent localizations, describe trajectories, or delimit areas (LEE; MINSEO, 2015). These kinds of databases are commonly structured using the Star schema, which is composed of facts (quantitative data) and dimensions (descriptive data) (O'NEIL; O'NEIL; CHEN, 2007). As an alternative to relational databases, several studies have proposed the use of document-oriented databases to build DWs and GDWs (YANGUI; NABLI; GARGOURI, 2016; CHEVALIER et al., 2015; CHEVALIER et al., 2016; CHEVALIER et al., 2017; BENSALLOUA; BENAMEUR, 2021; FERRAHI et al., 2017). This is because of the ability of non-relational databases to scale horizontally using commodity hardware, allowing for cheaper storage and processing of large volumes of data (HAN et al., 2011; DAVOUDIAN; CHEN; LIU, 2018). However, given the peculiarities of document-oriented databases, the design of a *Document-oriented GDW* (DGDW) involves a different manner of structuring data than the one used by a *Relational GDW* (RGDW).

Document-oriented databases manipulate aggregated data, which correspond to collections of documents stored in a database with a flexible schema (DAVOUDIAN; CHEN; LIU, 2018; EVANS, 2004; BUGIOTTI et al., 2014; GESSERT et al., 2017). This flexibility makes it possible to use document-oriented databases in schema-on-read or schema-on-write applications (MILOSLAVSKAYA; TOLSTOY, 2016). Schema-on-read applications do not require a predefined schema, delegating to the users the creation of read techniques to extract values from large raw data sets (e.g., Data Lake (HAI; GEISLER; QUIX, 2016)). However, schema-on-write applications have predefined schemas designed to fit a specific

domain (e.g., GDW), which can be defined using JSON Schemas and database constraint functions (WRIGHT et al., 2020; DYNAMODB, 2021; MONGODB, 2021b; INDEXES, 2021). In short, defining schemas in a document-oriented database is useful to validate documents before inserting them, as well as to allow retrieval of schema metadata from an existing repository.

Database schema modeling can be divided into three levels (ELMASRI; NAVATHE, 2010): (i) conceptual, which defines the contents of the database at a high level of abstraction; (ii) logical, which specifies how to implement the database within a specific data model; and (iii) physical, which describes how to implement and optimize the database using a particular technology, addressing its specific aspects and dialects. The logical schema has more details (e.g., data type and constraints for uniqueness, key, or not null) than conceptual modeling and is the basis for defining the physical schema. In the context of a DGDW, a logical schema describes facts and dimensions as either referenced or embedded documents, which can be partitioned into one or more collections (CHEVALIER et al., 2016). Furthermore, a collection may be either homogeneous or heterogeneous, with the former meaning that documents have the same field structure while the latter can contain documents with different field structures.

Modeling languages are used to define models at a high level of abstraction, enabling users to focus on semantic issues instead of syntactic ones (CATARCI et al., 1997; DEMARCO, 1979; HAREL, 1988). In the database context, they can reduce the complexity of schema definition by using visual symbols instead of textual code. Modeling languages can be classified into *General-Purpose Modeling Language* (GPML) or *Domain-Specific Modeling Language* (DSML) (BRAMBILLA; CABOT; WIMMER, 2017). GPMLs are designed to solve problems in various domains (e.g., *Unified Modeling Language* (UML)), but commonly provide poor support for the depiction of specific features or the prevention of incorrect constructions addressed in a specific domain. In contrast, DSMLs are customized to define models with expressiveness focused on a particular domain (e.g., modeling DGDW schemas). In addition, DSMLs can be implemented by *Computer Assisted Software Engineering* (CASE) tools, which commonly provide mechanisms to generate (forward engineering) or interpret (reverse engineering) code (SILVA, 2015). A DSML is composed of a concrete syntax (i.e., graphical notation), an abstract syntax (i.e., grammar or metamodel), and static semantics (i.e., well-formedness rules) (BRAMBILLA; CABOT; WIMMER, 2017). It can be designed from a new metamodel or extend an existing one. The

first approach defines the concepts (and relationships between them) behind its graphical notation symbols from scratch. The second extends the syntax and semantics from an existing metamodel, which requires that a set of rules to remove or to adapt the symbols to the given domain be coded, increasing the complexity of the implementation (e.g., UML Profile (KELLY; TOLVANEN, 2008)). In other words, defining a DSML with a new metamodel provides a more custom-tailored syntax and semantics, as well as a more expressive set of symbols for the domain.

Although there are many modeling languages available for designing RGDW logical schemas (AGUILA; FIDALGO; MOTA, 2011; CUZZOCREA; FIDALGO, 2012; MALINOWSKI; ZIMÁNYI, 2004; GLORIO; TRUJILLO, 2008; BOULIL; BIMONTE; PINET, 2015), the definition of modeling languages for DGDW logical schemas is an area still rarely explored. On the one hand, the existing proposals for the design of DGDW schemas do not cover some features of document-oriented databases, such as the definition of relationships using embedded or referenced documents, or the partitioning of documents into one or more collections. In addition, these proposals include *Spatial On-Line Analytical Processing* (SOLAP) concepts in the DGDW schema modeling, though a GDW (relational or document-oriented) can be used for applications other than SOLAP, such as data mining, reports, and what-if-analysis (BREAUULT; GOODALL; FOS, 2002; SRAI et al., 2017; LI, 2021). On the other hand, proposals for the design of general-purpose document-oriented databases do not cover GDW modeling concepts, such as defining facts and dimensions, nor do they prevent the modeling of constructions that could impair the performance of the DGDW. This means that there are no proposals that define a concrete syntax, an abstract syntax, and static semantics that capture and cover the GDW modeling concepts and document-oriented database features, nor that prevent or alert the user about invalid constructions.

## 1.2 RESEARCH OBJECTIVES

The main objective of this thesis is to propose a DSML for DGDW logical schemas with support for the different types of relationships between facts and dimensions, as well as support for the partitioning of documents among collections. The following specific objectives are necessary to reach this main objective:

- Define a concrete syntax, which consists of a graphical notation composed of a set

of symbols that specify logical schemas;

- Establish an abstract syntax, which consists of a metamodel that describes the concepts of the modeling language and the relationships allowed between them;
- Define the static semantics, which consist of a set of well-formedness rules that prevent or alert the user when incorrect constructions that the metamodel does not prohibit are created;
- Present translational semantics, which correlate the syntax and semantics of the modeling language.

To evaluate the proposal and show how to use it in practice, the following specific objectives were also defined:

- Evaluate the graphical notation;
- Implement the metamodel and well-formedness rules as a prototype of CASE tool;
- Define guidelines to help design logical schemas.

### 1.3 SCOPE

The prototype of the CASE tool presented in this thesis is shown as a proof of concept to demonstrate that implementing the proposed modeling language is a viable task. Therefore, a usability evaluation of the CASE tool prototype is not part of the scope of this work.

### 1.4 METHODOLOGY

The first step was to specify a DSML composed of a graphical notation (concrete syntax), a metamodel (abstract syntax), and a set of well-formedness rules (static semantics), called AStar. Translational semantics, which correlate the syntax and semantics of AStar, were also presented. To define a cognitively efficient graphical notation, inspiration was drawn from the UML class diagram notation. The UML graphical symbol representation was chosen because it is well-known in both industry and academics, and many CASE tools based on their notation exist. In this way, designers can model DGDW logical schemas

using either an AStar implementation or UML tools, increasing the number of computational tools that support AStar. It is important to note that AStar symbols do not extend the concepts of the UML metamodel. AStar’s syntax and semantics are custom-tailored to the modeling of DGDW logical schemas. Therefore, only CASE tools based on AStar can prevent or alert users to design errors.

To evaluate the graphical notation of AStar, its cognitive effectiveness was assessed. This is a concept related to how humans understand a set of graphical symbols. For this, *Physics of Notations* (PoN) (MOODY, 2009) was used, which consists of a set of principles for cognitively designing effective visual notations. Furthermore, as a proof of concept, AStar was implemented as a prototype of CASE tool called AStarCASE. This prototype was implemented with Eclipse Epsilon, which has resources to generate code from a valid metamodel and implement the static well-formedness rules. In the current version, AStarCASE provides support for the design of DGDW logical schemas and the generation of their corresponding JSON Schemas (WRIGHT et al., 2020) (*Model-to-Text* (M2T) transformation).

To demonstrate the design of DGDW logical schemas with AStar, a guideline that addresses the depiction of logical schemas with different levels of conventional and geospatial data redundancy is presented. This guideline defines models that establish relationships between facts, conventional dimensions, and geospatial dimensions as either referenced or embedded documents, partitioned into one or more (homogeneous or heterogeneous) collections. In addition, best practices that can help to achieve low-cost storage and low query runtime are presented, based on experimental evaluation that compares the data volume and query performance of 36 DGDWs whose schemas were designed with AStar.

## 1.5 DOCUMENT ORGANIZATION

The remaining chapters of this thesis are organized as follows:

**Chapter 2 - Background:** contextualizes the theoretical foundation needed to understand the proposal.

**Chapter 3 - Related Studies:** presents studies related to this proposal.

**Chapter 4 - AStar:** presents the graphical notation, metamodel, well-formedness rules, and translational semantics of AStar.

**Chapter 5 - Evaluation and Implementation of AStar:** evaluates the cognitive

effectiveness of the AStar graphical notation and shows AStarCASE, a prototype of CASE tool that implements AStar.

**Chapter 6 - Guidelines to Design Logical Schemas with AStar:** shows how to design logical schemas with AStar, and points out some best practices for achieving low data volume and low query runtime.

**Chapter 7 - Conclusions:** presents final considerations on the main topics covered in this thesis, highlighting the contributions and publications, also indicating possibilities for future studies.

Finally, this document has the following appendixes:

**Appendix A - JSON Schemas for geospatial data types:** presents JSON Schemas for implementing geospatial data types.

**Appendix B - Examples of JSON Schemas:** shows the JSON Schema for the examples used to explain the AStar concrete syntax.

**Appendix C - Implementation of AStar metamodel:** presents the implementation code for the AStar metamodel.

**Appendix D - AStarCASE transformation M2T:** shows a JSON Schema code generated by a DGDW logical schema modeled in AStarCASE.

## 2 BACKGROUND

This chapter presents the theoretical foundation underlying the AStar proposal. Section 2.1 presents an overview of document-oriented databases, section 2.2 discusses techniques and concepts related to the logical design of GDW, section 2.3 covers the components of a DSML, section 2.4 addresses the graphical design of modeling languages, and section 2.5 presents the final considerations of this chapter.

### 2.1 DOCUMENT-ORIENTED DATABASES

Document-oriented databases are the most commonly used non-relational databases in the industry (DB-ENGINES, 2020). MongoDB, Amazon DynamoDB, Couchbase, Microsoft Azure Cosmos DB, and CouchDB are some examples of this database type, which store aggregate data as JSON and GeoJSON documents (BRAY, 2017; BUTLER et al., 2016).

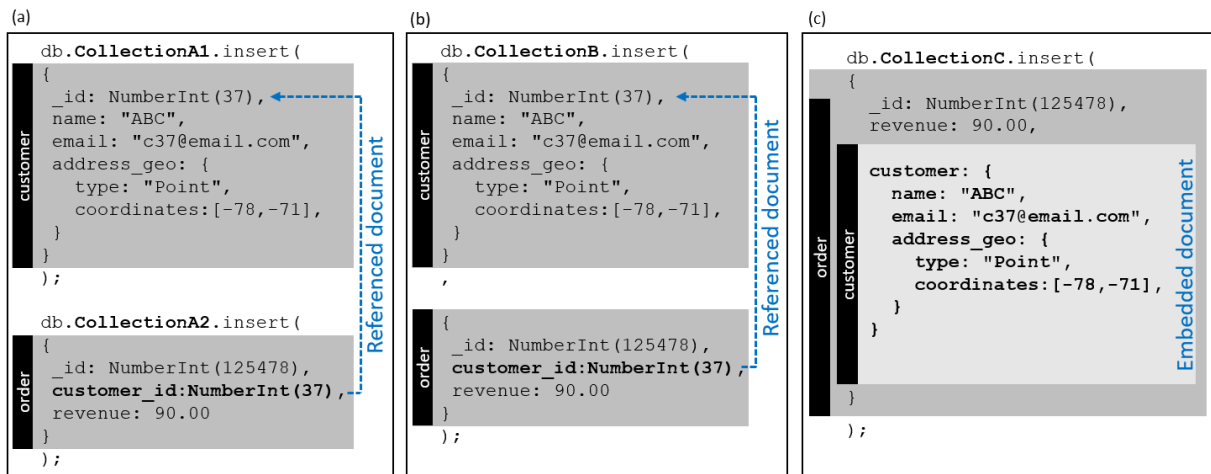
Documents are semi-structured data composed of a set of key-value pairs representing a field name and its respective value (GESSERT et al., 2017). In a document, a field can store conventional or geospatial data, and can be defined as regular, identifier, or unique. A field can also be multivalued, storing values of one or more data types as an array. Documents are stored in collections, which can be classified as homogeneous or heterogeneous. A homogenous collection contains documents that have the same field structure, while a heterogeneous collection contains documents with different field structures.

Relationships can be implemented using referenced or embedded documents (KANADE; GOPAL; KANADE, 2014; CHEVALIER et al., 2015; CHEVALIER et al., 2016; COPELAND, 2013; YANGUI; NABLI; GARGOURI, 2016). In the first approach, a document contains a reference to another document, enabling one-to-one, one-to-many, or many-to-many relationships between documents that belong to the same or to different collections. In the second approach, a document embeds another document within it, enabling one-to-one relationships, such that it does not require a join to retrieve related documents. In this type of relationship, both documents belong to the same collection, and the embedded document has no identifier field because it is identified by the document in which it is embedded. For this reason, documents related as embedded documents cannot belong to different collections. It is important to highlight that a document can be embedded within only

one document. Although copies of a document can be embedded into other documents, these copies are independent, meaning that updating an embedded document does not update its copies. Multivalued fields can store many embedded documents, implementing one-to-many or many-to-many relationships. However, this practice is discouraged because it can produce arrays with complex data and documents with a very high data volume, which can negatively impact database performance (MONGODB, 2022; AZURE, 2020; COACHDB, 2020). Many-to-many relationships can also produce cyclic dependency because, given two documents named A and B related as embedded documents, A must contain B and B must contain A.

Figure 1 shows how to insert (a,b) referenced documents and (c) embedded documents into MongoDB, which is the most popular document-oriented database in the industry (DB-ENGINES, 2020). Note that Figure 1a shows the JSON documents “customer” and “order” inserted into distinct collections (CollectionA1 and CollectionA2), producing homogeneous collections; Figure 1b shows “customer” and “order” inserted into the same collection (CollectionB), producing a heterogeneous collection; and Figure 1c also shows “customer” and “order” inserted into the same collection (Collection C), but in this case, “customer” is embedded within “order”, producing a homogeneous collection.

Figure 1 – Inserting examples of JSON documents “customer” and “order” into MongoDB: (a) two collections with referenced documents; (b) one collection with referenced documents; and (c) one collection with one embedded document.



Source: The Author

The schema of a document-oriented database can be flexible or previously defined with JSON schemas and constraint functions (DYNAMODB, 2021; MONGODB, 2021b; INDEXES, 2021). The JSON schema is the base format that defines the structure of JSON docu-

ments and is commonly used to validate documents before inserting them into a database (WRIGHT et al., 2020). A JSON schema defines the required fields and their respective data types, as well as data enumerations that can be used to specify geospatial data types (BUTLER et al., 2016) (cf. Appendix A). In other words, a JSON schema defines a data structure for the documents, which will be called `DocumentType`. Constraint functions complement a JSON schema, as they define, for example, which fields in a collection must be a unique (INDEXES, 2021).

Figure 2 shows how to define a schema in MongoDB, with the JSON schemas and constraint functions highlighted with a gray background. This example creates collections “CollectionA1” and “CollectionA2”, each one with a JSON schema that defining the field structures titled “customer” and “order”, respectively. In other words, in this example, “CollectionA1” and “CollectionA2” have DocumentTypes “customer” and “order”. As the JSON schema does not have a mechanism to represent referenced documents, the field “description” is used for this purpose. This approach does not guarantee referential integrity, but can be used at the application level to validate or identify document relationships. Thus, as can be seen in Figure 2, the field “customer\_id” of DocumentType “order” stores a reference for a 1:N relationship with “customer” (description:“REF\_1N:customer”)<sup>1</sup>. An example of the insertion of documents for “customer” and “order” into “CollectionA1” and “CollectionA2” is shown in Figure 1a.

Similar to the previous figure, Figure 3 establishes a relationship between the JSON schemas (i.e., DocumentTypes) “customer” and “order” using references. However, in this example, the DocumentTypes are defined into only one collection, named “CollectionB”. An example of the insertion of documents into this schema is shown in Figure 1b.

Finally, Figure 4 shows a schema with one collection named “CollectionC”, whose DocumentType “customer” is embedded into “order”. An example showing the insertion of documents into this schema is shown in Figure 1c.

<sup>1</sup> REF\_11 mean a 1:1 relationship, REF\_1N mean a 1:N relationship, and REF\_NM mean a N:M relationship

Figure 2 – Defining a logical schema with JSON schemas and unique constraints, in MongoDB: two collections with referenced documents.

JSON schema for customer	<pre>db.createCollection("CollectionA1",{   validator: { \$jsonSchema:     {       title: "customer",       bsonType:"object",       required:["_id","name", "email","address_geo"],       properties: {         _id: { bsonType: "int" },         name: { bsonType: "string" },         email: { bsonType: "string" },         address_geo: {           bsonType : "object",           required: ["type", "coordinates"],           properties: {             type:{               bsonType:"string",               enum:["Point"]             },             coordinates: {               bsonType : "array",               minItems: 2,               items: {bsonType:"number"}             }           }         }       }     }   } });</pre>	<pre>db.createCollection("CollectionA2",{   validator: { \$jsonSchema:     {       title: "order",       bsonType: "object",       required:["_id","customer_id", "revenue"],       properties: {         _id: { bsonType: "int" },         customer_id: {           description: "REF_1N:customer",           bsonType: "int"         },         revenue: { bsonType: "number" }       }     }   } });</pre>
constraint	<pre>db.CollectionA1.createIndex(   {"email":1},{unique:true} );</pre>	

Source: The Author

Figure 4 – Defining a logical schema with JSON schemas and unique constraints, in MongoDB: one collection with one embedded document.

JSON schema for order	<pre> db.createCollection("CollectionC", {   validator:{\$jsonSchema:     {       title: "order",       bsonType:"object",       required:["_id","customer", "revenue"],       properties: {         _id: { bsonType: "int" },         revenue: { bsonType: "number" },         customer:           {             title:"customer",             bsonType: "object",             required: ["name","email", "address_geo"],             properties:{               name: { bsonType: "string" },               email: { bsonType: "string" },               address_geo : {                 bsonType : "object",                 required:["type", "coordinates"],                 properties: {                   type: {                     bsonType : "string",                     enum: ["Point"]                   },                   coordinates: {                     bsonType : "array",                     minItems: 2,                     items: {bsonType : "number"}                   }                 }               }             }           }       }     }   } }); </pre>	<pre> db.CollectionC.createIndex(   {"customer.email":1},{unique:true} ); </pre>
-----------------------	---	--

Source: The Author

Figure 3 – Defining a logical schema with JSON schemas and unique constraints, in MongoDB: one collection with referenced documents.

<div data-bbox="247 414 279 817" style="writing-mode: vertical-rl; transform: rotate(180deg);">JSON schema for customer</div> <pre> db.createCollection("CollectionB",{   validator: { \$jsonSchema: {     anyOf: [       {         title: "customer",         bsonType:"object",         required:["_id","name", "email","address_geo"],         properties: {           _id: { bsonType: "int" },           name: { bsonType: "string" },           email: { bsonType: "string" },           address_geo: {             bsonType : "object",             required: ["type", "coordinates"],             properties: {               type:{                 bsonType:"string",                 enum:["Point"]               },               coordinates: {                 bsonType : "array",                 minItems: 2,                 items: {bsonType:"number"}               }             }           }         }       }     ]   } }) </pre>	<div data-bbox="885 414 917 683" style="writing-mode: vertical-rl; transform: rotate(180deg);">JSON schema for order</div> <pre> {   title: "order",   bsonType: "object",   required:["_id","customer_id", "revenue"],   properties: {     _id: { bsonType: "int" },     customer_id: {       bsonType: "int",       description: "REF_1N:customer",     },     revenue: { bsonType: "number" },   } } ]}}}); </pre> <div data-bbox="885 806 917 907" style="writing-mode: vertical-rl; transform: rotate(180deg);">constraint</div> <pre> db.CollectionB.createIndex(   {"email":1},{unique:true} ); </pre>
--	---

Source: The Author

## 2.2 LOGICAL DESIGN OF GEOSPATIAL DATA WAREHOUSE

A Data Warehouse (DW) is a subject-oriented, integrated, non-volatile, and time-varying database that supports the decision-making process (INMON, 2005). These characteristics can be summarized as follows: (i) subject-oriented - they store data that corresponds to facts and not to the transactions that generated the facts; (ii) integrated - they integrate data from databases located across different computer systems in an organization, consolidating data from different sources; (iii) non-volatile - the data is rarely modified, essentially limited to loads and queries; and (iv) time-variant - the data is stored as it varies along a timeline, maintaining its history. The construction of a DW consists of transforming transactional data from different sources into consolidated strategic information to support decision-making processes. From a DW, users can analyze the data using SQL query tools. Among these tools, *On-Line Analytical Processing* (OLAP) stands out, but other tools may also be used, such as reporting, data mining, and what-if-analysis systems (CHAUDHURI; DAYAL, 1997; BREAUULT; GOODALL; FOS, 2002; SRAI et al., 2017; LI, 2021).

When built in a relational database, a DW schema is structured as tables of facts and dimensions. Dimension tables store business descriptions, while fact tables contain all of the foreign keys for the dimensions and store business's numerical measures. The tables of a DW are commonly designed according to Star or Snowflake approaches, with the former being more popular and having better query performance (KIMBALL; ROSS, 2013). In the first approach, the dimension tables are denormalized to optimize the performance of complex queries, because no joins are required. The second approach normalizes all data, reducing its volume, but requiring a high number of joins in queries.

A Geospatial DW (GDW) is a combination of features from a DW with the functionality of a geospatial database in order to manage large amounts of historical data that have a geospatial context (e.g., localization, trajectories, areas) (FIDALGO et al., 2004; LEE; MINSEO, 2015). In other words, a GDW is an extension of the traditional DW approach, adding a geospatial component. Basically, this extends the DW schema by inserting geospatial information into its dimension (geospatial attributes) or fact tables (geospatial measures). It is important to highlight that some authors include the concepts of hierarchy and level in the description of a GDW schema (MALINOWSKI; ZIMÁNYI, 2004; GLORIO; TRUJILLO, 2008; BOULIL; BIMONTE; PINET, 2015). However, this can be understood as a mixing of concepts, because hierarchy and level are concepts specific to data cube modeling used by OLAP or SOLAP tools (BERSON; SMITH, 1997; TSOIS; KARAYANNIDIS; SELLIS, 2001). If hierarchies and levels were intrinsic GDW concepts, any GDW query tool would be able to execute multilevel queries (e.g., drill-down and roll-up), but only SOLAP tools are able to do this. In other words, because a GDW is a database that can be used by applications other than SOLAP (e.g., data mining, reports, and what-if-analysis), the GDW schema must be more general, and not contain these SOLAP specific concepts. Therefore, in this study, only fields, dimensions, and facts are considered as part of the design of a logical GDW schema.

Fields are properties that can be used to describe, quantify, or reference dimensions or facts. While dimension fields essentially describe a business process, fact fields quantify the measures of a business process. Fields can also be used to reference two dimensions, two facts, or a dimension and a fact (KIMBALL; ROSS, 2013). Furthermore, fields have a data type that can be conventional (e.g., string or number) or geospatial (e.g., point and polygon); can have a uniqueness constraint; and can be defined as an identifier (i.e., key). Lastly, although a field of a document-oriented database can be null, multivalued,

or address a complex data type, these properties of fields are discouraged in DW schemas. Null values have a negative impact on data quality, as they may confuse users with regard to interpretation of query results (e.g., nonexistent value versus unknown value) (THORNTHWAITE, 2003), while multivalued and complex data type fields make the schema unstructured and complex to manage (TING, 2010).

Dimensions arrange information related to the business process, such as what, who, where, and when. Dimensions are identified by a single key (i.e., surrogate key) that is typically a meaningless integer (i.e., auto-incremented key). Dimensions usually are denormalized, however dimensions with geometric data are often normalized because this reduces the data volume and increases query execution performance (FIDALGO et al., 2004; MATEUS et al., 2010; MATEUS et al., 2016).

Facts capture the events of a business process and are identified by a set of keys to their dimensions or to other facts. Facts can contain conventional (e.g., quantity and amount) or geospatial (e.g., planting area or crime location) measures, as well as descriptive information (e.g., flags). In other words, facts are commonly identified by a set of references to their dimensions or to other facts and can contain measures as well as descriptive information.

There are many techniques for modeling dimensions and facts (KIMBALL; ROSS, 2013). These techniques are used to capture business semantics (e.g., factless fact table, degenerate dimensions, role-playing dimensions, and heterogeneous dimensions), to reduce data volume (e.g., mini dimensions, junk dimensions, and outriggers dimensions), or to resolve limitations of relational databases (e.g., bridge table). It is important to highlight that a bridge table is commonly used to implement an M:N relationship in relational databases. However, the document-oriented data model supports M:N relationships using arrays (COPELAND, 2013), making the use of a bridge table in a DGDW a design decision.

Taking into consideration the peculiarities of document-oriented databases (cf. section 2.1), and that a GDW is essentially a well-structured database (KIMBALL; ROSS, 2013), a DGDW logical schema represents facts and dimensions as documents, which can be related as referenced or embedded documents (CHEVALIER et al., 2016). Therefore, a DGDW schema (i) organize documents into homogeneous or heterogeneous collections; (ii) express the cardinality of the relationship (i.e., 1:1, 1:N, or M:N); (iii) distinguish whether a field is conventional or geospatial; and (iv) symbolize whether a field is unique (used to support business rules), identifier (used to reference documents), or regular

(neither unique nor identifier).

### 2.3 MODEL-DRIVEN DEVELOPMENT AND DOMAIN-SPECIFIC MODELING LANGUAGES

Modeling Languages are used in the *Model-Driven Development* (MDD) paradigm to define software artifacts (e.g., interpreted or executable code) from models (SILVA, 2015). Modeling languages tend to offer higher-level abstractions than textual languages because their graphic notations are human-oriented, facilitating communication and problem-solving (DEMARCO, 1979; HAREL, 1988). In the database area, modeling languages are commonly employed to assist in creating logical or conceptual schemas (FIDALGO et al., 2004; FIDALGO et al., 2012; FIDALGO et al., 2013; FERRAHI et al., 2017). Furthermore, modeling Languages can be implemented in CASE tools, taking advantage of various features: (i) automatic validation, such as validating a database logical schema; (ii) *Text-to-Model* (T2M) transformations, such as reverse engineering; (iii) M2T transformations, such as forward engineering from diagrams; and (iv) *Model-to-Model* (M2M) transformations, such as transforming an *Entity-Relationship* (ER) diagram into a UML class diagram.

A Modeling Language created and custom-tailored for a given domain is classified as a Domain-Specific Modeling Language (DSML), which can be built as an extension of an existing GPML or be entirely new (BRAMBILLA; CABOT; WIMMER, 2017; LOPES et al., 2016). The first approach has the advantage of using an existing already and well-known syntax, such as the UML profiles (FONTOURA; PREE; RUMPE, 2000; MOHAGHEGHI; DEHLEN; NEPLE, 2009). However, extending an existing GPML has the following drawbacks: (i) new symbols may demand changes that compromise the standardization of the approach, and (ii) rules may be created to remove some symbols or to change the semantics of others that, besides being labor-intensive, can introduce inconsistencies into the language. Creating a new DSML has the advantage of defining a smaller and more expressive set of symbols for the domain, with the drawback of having to build everything from scratch.

A DSML is defined by its concrete syntax, abstract syntax, and static semantics (BRAMBILLA; CABOT; WIMMER, 2017). The concrete syntax corresponds to the graphical notation that defines the symbols used by designers when creating diagrams. The abstract syntax is the grammar of the language, represented by a metamodel that defines the set

of concepts within a specific domain, as well as their attributes and relationships. Meta-models can be defined using the *Meta-Object Facility* (MOF) and are commonly shown as UML class diagrams (GROUP, 2020; GUY et al., 2012). Thus, modeling language concepts can be described as metaclasses that are related through generalizations and associations. Static semantics consist of a set of well-formedness rules that validate syntactically correct constructions (i.e., those allowed by the metamodel), but that are semantically contradictory (e.g., cardinality conflicts). In other words, the rules contain constraints on the metamodel concepts that determine how their instances must be defined. Static semantics are commonly implemented in CASE tools using *Object Constraint Language* (OCL) (OMG, 2014), *Epsilon Validation Language* (EVL) (KOLOVOS et al., 2017), or the programming language of a specific framework.

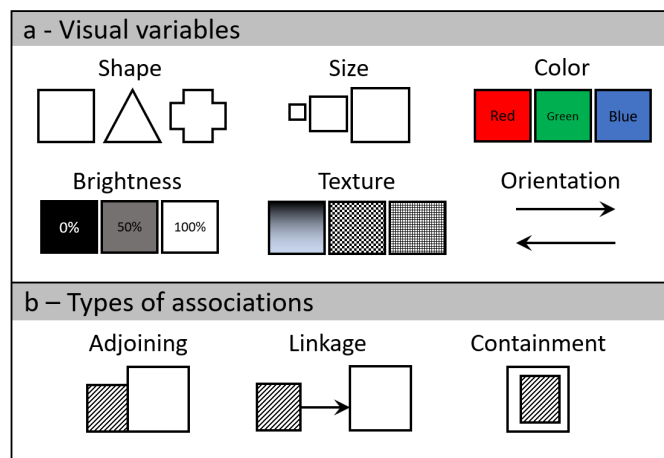
In order to understand the concrete and abstract syntax better, a semantic description for defined concepts is commonly addressed in a DSML project. The semantic description is commonly defined using translational semantics, which map the concepts of the DSML syntax to an existing formal language (KLEPPE, 2007; BRYANT et al., 2011; FLECK; TROYA; WIMMER, 2016).

## 2.4 GRAPHICAL DESIGN OF MODELING LANGUAGES

The graphical design of a modeling language defines its set of symbols, which encode information as a graphical representation, although it can also have textual elements to represent any additional information (e.g., in UML, the class name) (MOODY, 2009). The visual appearance of a symbol uses one or more visual variables, such as shape, size, color, brightness, texture, or orientation (Figure 5a) (BERTIN, 1983). Out of all of the visual variables, shape allows for the greatest ability to discriminate between symbols, as it represents the primary basis that human beings uses to identify objects (MOODY, 2009). In a diagram, symbols can be topologically associated to represent a relationship between concepts. These associations can be classified as adjoining, linkage, or containment (NICKERSON, 1994), as shown in Figure 5b. In an adjoining association, the symbols share a side. This type of relationship has a connectivity limit and, therefore, it is usually only used in simple diagrams. Attempting to represent many associations (e.g., a fact related to many dimensions) with this type of relationship could result in complex diagrams. In a linkage association, the symbols are linked by lines that can convey addi-

tional information, such as labels, direction, or navigability. Linkages are usually used for more extensive and complex diagrams (e.g., logical schemas of databases) due to the large amount of information they can transmit. Finally, in a containment association, a symbol lies within another symbol. This type of relationship is used to represent structures such as “A belongs to B” or “A is a subset of B”, but there may be a limit to how they can be represented. Recursive structures or long chains of subsets (e.g., A embeds B, B embeds C, and C embeds D) can be difficult to represent using containment (NICKERSON, 1994).

Figure 5 – Characteristics of a graphical notation symbol.



**Source:** The Author

The design of a modeling language has the challenge of defining a graphical notation that has cognitive effectiveness, a concept related to the speed, ease, and accuracy with which its symbols can be understood by the human mind (LARKIN; SIMON, 1987). In order to validate the cognitive effectiveness of a graphical notation, concepts such as simplicity, aesthetics, expressiveness, and naturalness are commonly used. However, these concepts are considered subjective, as they can be easily biased in the modeling language design process or in user evaluation. To minimize this issue, scientific approaches are commonly used, such as PoN (LINDEN; HADAR, 2019; TEIXEIRA et al., 2016). PoN is a set of principles based on a synthesis of theories (e.g., the psychology and cognitive sciences) to analyze and improve a graphical notation’s ability to be cognitively effective (MOODY, 2009). PoN defines nine principles that should be addressed by a graphical notation, which are not necessarily complementary, as they can produce conflict or synergy. The principles follow:

**P1** - Semiotic clarity: there should be a 1:1 correspondence between metamodel concepts and graphical notation symbols. Otherwise, there will be an anomaly that can

be classified as: multiple symbols that represent one concept (redundancy), multiple concepts represented by the same symbol (overload), a symbol that does not correspond to a concept (excess), or a concept that not is represented by a symbol (deficit).

- P2** - Perceptual discriminability: symbols should be distinguishable from each another. The number of visual variables (e.g., shape) and their variants (e.g., edge weights) can be used to measure the differences among symbols. If differences among them are too subtle, interpretation errors may be made. On the other hand, the greater the visual distance between symbols, the faster and more accurately they will be recognized.
- P3** - Semantic transparency: the appearance of the symbols should suggest their meaning. Appearances can be (i) semantically immediate if a novice user would be able to infer its meaning alone; (ii) semantically opaque, if the relationship with its meaning is arbitrary; (iii) semantically perverse, if a novice user would be likely to infer a different meaning.
- P4** - Complexity management: the graphical notation should have mechanisms for dealing with the complexity of diagrams with a high number of elements (i.e., symbol instances). To reduce complexity, techniques such as modularization should be supported. This technique consists of dividing a diagram into smaller parts, such as representing UML packages individually.
- P5** - Cognitive integration: a graphical notation should include mechanisms to support information integration, where multiple diagrams are used to represent a system.
- P6** - Visual expressiveness: a graphical notation should use the full range and capacities of visual variables. While perceptual discriminability measures pairwise visual variation, visual expressiveness measures the visual variation across the entire graphical notation. It is measured by the number of visual variables.
- P7** - Dual coding: text can be used to complement the symbols. Therefore, text such as labels or annotations can be placed near the symbols to reinforce and clarify their meaning.

**P8** - Graphic economy: the number of different graphical symbols should be cognitively manageable, because there are cognitive limits on the number of visual categories that can be effectively recognized. There are main three strategies for reducing excessive graphical complexity in a graphical notation: (i) reduce the semantic complexity of symbols or partition it to reduce the diagrammatic complexity; (ii) introduce symbol deficit by replacing same symbols with text; (iii) increase the visual expressiveness, and thereby increase human discriminatory ability, such as by using multiple visual variables to differentiate between symbols.

**P9** - Cognitive fit: use different visual dialects for different tasks and audiences. As a graphical notation may not be simultaneously effective for both novice and experienced users, it can be helpful to varying the symbol depending on the audience. Furthermore, the challenges of hand drawing should be taken into account for a graphical notation. The hand drawing requirement discourages the use of some visual variables (e.g., color) or 3D shapes, as the user may have a condition where colors are misunderstood (e.g., color blindness) or poor artistic ability.

## 2.5 CHAPTER FINAL CONSIDERATIONS

This chapter presented a brief theoretical foundation that approaches the main concepts necessary to understand the rest of this thesis. This chapter consists of four subjects: document-oriented databases, logical design of GDW, model-driven development and DSML, and graphical design of modeling languages.

### 3 RELATED STUDIES

This chapter presents proposals related to our study. Section 3.1 describes the method used to select the studies from the literature, section 3.2 discusses them, section 3.3 provides an overall analysis, and section 3.4 presents the final chapter considerations.

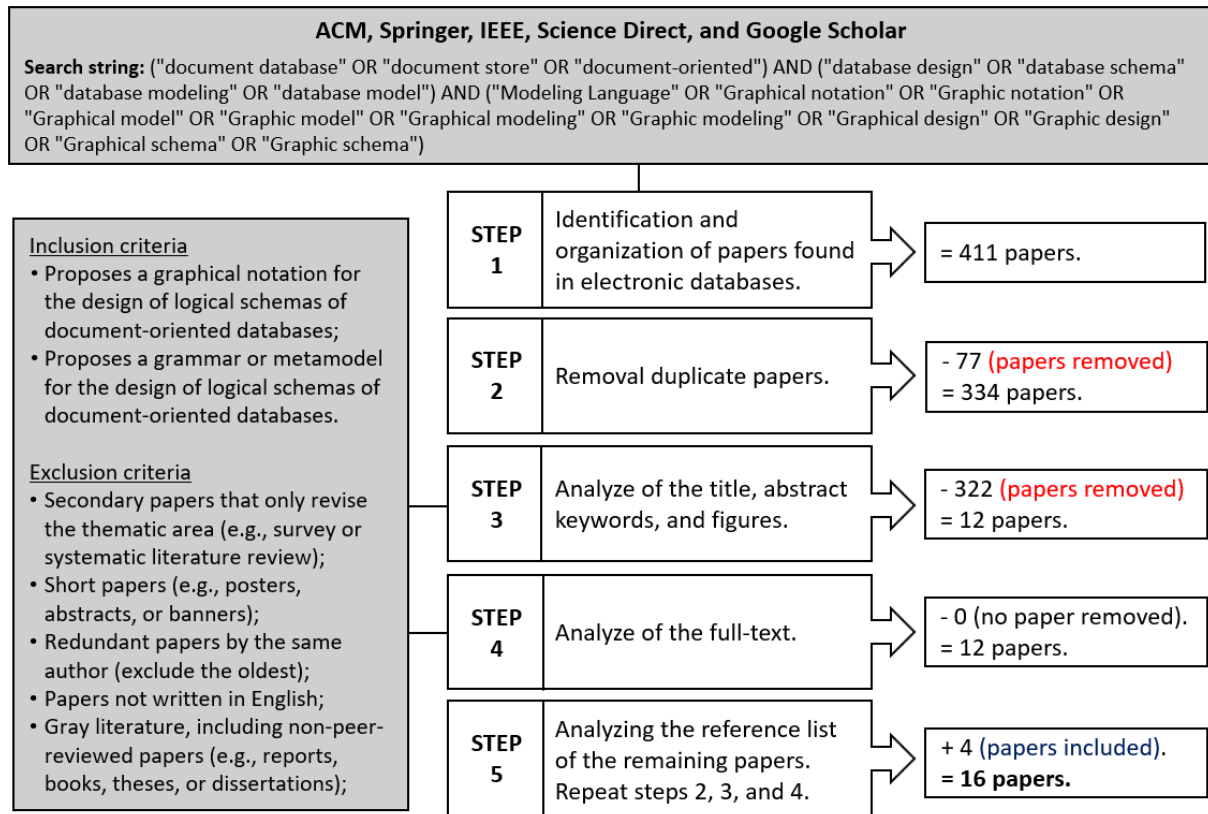
#### 3.1 SELECTION OF STUDIES

Figure 6 illustrates the method that was adopted to search through the literature for studies related to this proposal. The method sought to find proposals of modeling languages or graphical notations for schemas of DGDW or general-purpose document-oriented databases. The reason behind searching for graphical notation proposals (even though a graphical notation is part of a modeling language) is because some of them adopt a subset of symbols from an existing graphical notation to represent a schema (e.g., UML Profile). It is common for these proposals to show their symbols and how they were adapted for a specific domain. Furthermore, general-purpose proposals (beyond the DGDW domain) were included in the search to point out resources that can be used to model DGDW schemas, as well as to highlight the advancement of the state-of-the-art with regard to modeling document-oriented databases.

As can be seen in Figure 6, the method consists of 5 steps, which are described below:

- Step 1** - A search string was defined and used to search papers in research libraries containing high-quality computer science journals and conferences, as well as in Google Scholar, which indexes a wide content of scientific papers.
- Step 2** - Papers with the same title were removed to avoid duplicated studies.
- Step 3** - After analyzing the title, abstract, keywords, and figures (to identify, for example, a graphical notation or a metamodel), the papers that did not meet at least one inclusion criteria or met at least one exclusion criteria (cf. Figure 6) were removed.
- Step 4** - After analyzing the full text of the remaining papers, the papers that did not meet at least one inclusion criteria or met at least one exclusion criteria were also removed.

Figure 6 – The method used to search for related studies in the literature.



Source: The Author

**Step 5** - The references from the remaining papers were analyzed to identify and include works of potential interest to this study. This analysis applied steps 2, 3, and 4 to exclude duplicated studies and those unrelated studies to this proposal.

Table 1 presents the purpose (i.e., general-purpose or DGDW) and title of the selected papers, ordered by their publishing year (last column). The tag in the first column is to identify the papers in the next section.

Table 1 – Selected studies.

#	Purpose	Title	Year
P01	General-purpose	Aggregate data modeling style	2013
P02	General-purpose	Modeling and querying data in NoSQL databases	2013
P03	General-purpose	A Workload-Driven Logical Design Approach for NoSQL Document Databases	2015
P04	General-purpose	Data modeling for NoSQL document-oriented databases	2015
P05	General-purpose	Geographic data modeling for NoSQL document-oriented databases	2015
P06	General-purpose	Towards logical level design of Big Data	2015
P07	DGDW	Design and Implementation of Falling Star - A Non-Redundant Spatio-Multidimensional Logical Model for Document Stores	2017
P08	General-purpose	NoSQL database design using UML conceptual data model based on peter chen's framework	2017
P09	General-purpose	MDA-Based Approach for NoSQL Databases Modeling	2017
P10	General-purpose	Model driven development of hybrid databases using lightweight metamodel extensions	2018
P11	General-purpose	A Graph Based Knowledge and Reasoning Representation Approach for Modeling MongoDB Data Structure and Query	2019
P12	General-purpose	Extraction process of conceptual model from a document-oriented NoSQL database	2019
P13	General-purpose	Mortadelo: Automatic generation of NoSQL stores from platform-independent data models	2020
P14	General-purpose	Analysis and evaluation of document-oriented structures	2021
P15	DGDW	Towards NoSQL-based Data Warehouse Solution integrating ECDIS for Maritime Navigation Decision Support System	2021
P16	General-purpose	A unified metamodel for NoSQL and relational databases	2022

**Source:** The Author

### 3.2 DISCUSSION

This section discusses the papers shown in Table 1. In each paper, evidence of resources that can be used to model DGDW schemas was sought, taking into account the particularities of the document-oriented data model and the GDW logical design. Resources

referring to modeling language definition and implementation are also checked for each paper.

Regarding the data model of document-oriented databases (cf. section 2.1), the ability to support the design of homogeneous or heterogeneous collections; to specify geospatial or conventional data types for fields; to define fields as identifier or unique<sup>1</sup>; and to draw relationships using embedded documents or referenced documents was verified. For embedded documents, it is important to note that relationships with 1:N cardinality are implemented using arrays of embedded documents, producing documents with a very high data volume, which can impair the database performance; relationships with M:N cardinality (in addition to the issue regarding 1:N cardinality) can produce cyclic dependence; they should belong to the same collection and have no identifier field.

For the design of GDW schemas (cf. section 2.2), support for the ability to distinguish facts from dimensions and to prevent disconnected facts or dimensions was verified.

Concerning the modeling language resources (cf. section 2.3), it was verified if the proposal is specific to the DGDW domain; proposed a graphical notation; defined a meta-model or grammar; presented translational semantics; was evaluated; and provided computational support, such as a CASE tool.

In order to systematically search for these resources within the papers, they were structured as follows:

a) Document-oriented data model:

- D1 - Defines homogeneous collections;
- D2 - Defines heterogeneous collections;
- D3 - Depicts the field structure of documents;
- D4 - Specifies the conventional data types;
- D5 - Specifies the geospatial data types;
- D6 - Represents identifier fields;
- D7 - Represents unique fields;
- D8 - Establishes relationships using embedded documents;
- D9 - Depicts the 1:1 cardinality for embedded documents;

---

<sup>1</sup> Fields not defined as identifiers or unique are assumed to be regular fields. Therefore, support for specifying regular fields is not addressed.

- D10 - Depicts the 1:N cardinality for embedded documents;
- D11 - Depicts the M:N cardinality for embedded documents;
- D12 - Specifies documents from different collections as embedded documents;
- D13 - Defines identifier field in an embedded document;
- D14 - Establishes relationships using referenced documents;
- D15 - Depicts the 1:1 cardinality for referenced documents;
- D16 - Depicts the 1:N cardinality for referenced documents;
- D17 - Depicts the M:N cardinality for referenced documents.

b) Design of GDW schemas

- G1 - Distinguishes facts and dimensions;
- G2 - Prevents disconnected facts or dimensions.

c) Modeling language

- M1 - Proposes a graphical notation;
- M2 - Defines a metamodel or grammar;
- M3 - Presents translational semantics;
- M4 - Evaluates the proposal;
- M5 - Provides computational support, such as a CASE tool.

The studies dealing with the design of general-purpose document-oriented schemas are covered in section 3.2.1, while studies dealing with the design of DGDW schemas are covered in section 3.2.2.

### 3.2.1 General-purpose

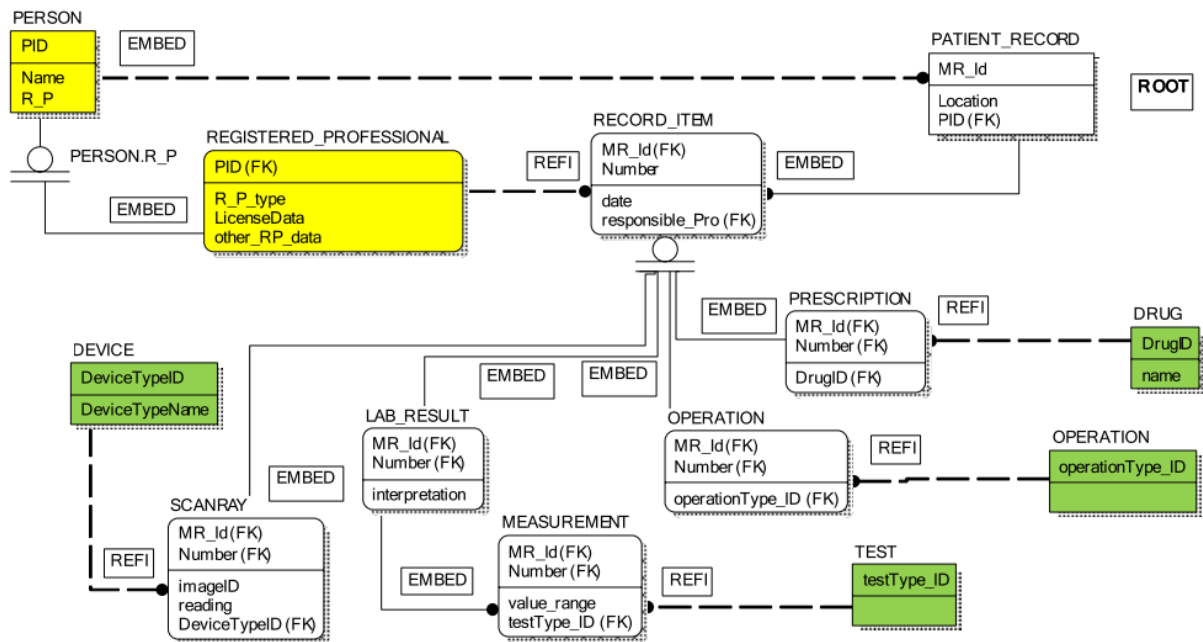
#### P01

Reference (JOVANOVIĆ; BENSON, 2013) proposes a graphical notation based on IDEF1X (BRUCE, 1992) for the design of general-purpose document-oriented database schemas. In this graphical notation, an entity represents the field structure of the documents. In an

entity, fields can be defined as identifiers, unique, or regular, but there is no definition of data types. Entities can be related using links, whose labels distinguish relationships using embedded documents (EMBED) from those using referenced documents (REFI). For both relationship approaches, 1:1 or 1:N cardinality can be set. As there is no mention of a symbol that represents a collection in this proposal, no evidence that homogeneous or heterogeneous collections can be defined was found. Furthermore, no evidence was found with regard to the drawing of embedding documents that are in different collections or about the definition of identifier fields in embedded documents.

Figure 7 shows an example depicted with P01, in which entities are related using referenced or embedded documents.

Figure 7 – Schema example of P01.



Source: JOVANOVIĆ; BENSON (2013)

In short, evidence was found in this proposal for the following resources: D3 - Depicts the field structure of documents; D6 - Represents identifier fields; D7 - Represents unique fields; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D14 - Establishes relationships using referenced documents; D15 - Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the 1:N cardinality for referenced documents; and M1 - Proposes a graphical notation.

## P02

Reference (KAUR; RANI, 2013) defines a graphical notation based on the UML class diagram to model schemas for relational and NoSQL databases (column-oriented, document-oriented, key-value, and graph-oriented). For document-oriented database design, a class represents a field structure for documents. In a class, the attributes represent fields, but with no indication of their data types, nor any way to define them as identifiers or unique. Classes can be related using association or composition links, representing relationships using referenced or embedded documents, respectively. There is no cardinality definition in either relationship approach. In a relationship using referenced documents, the fields that store the references are graphically represented inside a rectangle with rounded corners. As there is no graphical representation for identifier fields, no evidence was found regarding the definition of identifier fields in embedded documents. As with P01, there is no mention of a symbol to represent collections. Therefore, no evidence was found that this proposal allows for the design of homogeneous or heterogeneous collections, or draws relationships between documents of different collections as embedded documents.

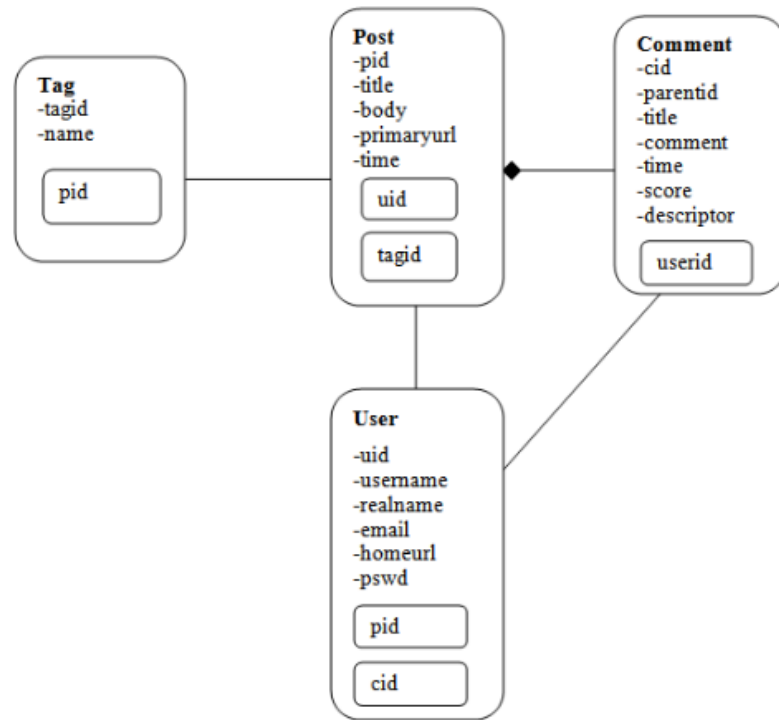
Figure 8 shows an example depicted with P02, which defines document field structures named Tag, Post, Comment, and User. These field structures are related using embedded documents (Post embeds Comment) and referenced documents (Tag to Post, Post to User, and User to Comment).

In short, evidence was found in this proposal for the following resources: D3 - Depicts the field structure of documents; D8 - Establishes relationships using embedded documents; D14 - Establishes relationships using referenced documents; and M1 - Proposes a graphical notation.

## P03

Reference (LIMA; MELLO, 2015) proposes a graphical notation for the design of aggregated database schemas (key-value, document-oriented, and column-oriented). In the document-oriented context, a rectangle represents a homogeneous collection. A rectangle is composed of blocks that describe the field structure of documents (named root blocks) or the field structure of embedded documents (named hierarchical blocks). Both of these types of blocks describe the names of the fields, which can be single-value or multiva-

Figure 8 – Schema example depicted with the graphical notation proposed in P02.



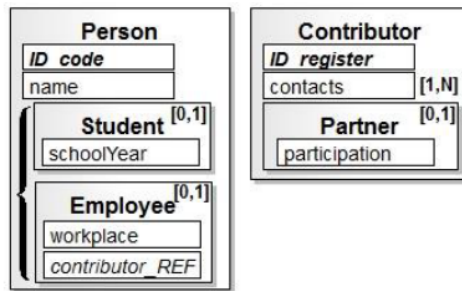
**Source:** KAUR; RANI (2013)

lued, but without a data type indication. An identifier field is represented by displaying its name in bold and italic font. The hierarchical blocks are shown with their minimum and maximum occurrences in the documents, which makes it possible to define relationships using embedded documents with 1:1 or 1:N cardinality. As embedded documents are graphically represented using the containment association (i.e., a block within a collection), this proposal avoids relating documents that belong to different collections using this relationship approach. No evidence was found regarding the definition of identifier fields in hierarchical blocks. This proposal also provides support for defining relationships using referenced documents, but without representing their cardinality. A case study that models a database in the e-commerce domain is shown as the proposal evaluation.

Figure 9 shows an example of schema depicted with P03. In this example, Person and Contributor are root blocks that define two collections with the field structure of their documents. Person has one identifier field in its root block, named *ID\_code*, and two hierarchical blocks, named Student and Employee. Contributor has one identifier field in its root block, named *ID\_register*, and one hierarchical block, named Partner. The hierarchical blocks (embedded documents) have minimum and maximum occurrences of 0 and

1, respectively  $([0,1])$ , which indicates a relationship having 1:1 cardinality. A relationship using references is established between Employee and the root block of Contributor. In this relationship, the field `contributor_REF` of Employee references `ID_register`, the identifier field of Contributor. This example also defines a multivalued field, named `contacts`, in which the label  $[1,N]$  after its name suggests that a contributor can have one or more contacts.

Figure 9 – Schema example depicted with the graphical notation proposed in P03.



Source: LIMA; MELLO (2015)

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D3 - Depicts the field structure of documents; D6 – Represents identifier fields; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D11 - Depicts the M:N cardinality for embedded documents; D14 - Establishes relationships using referenced documents; M1 - Proposes a graphical notation; and M4 - Evaluates the proposal.

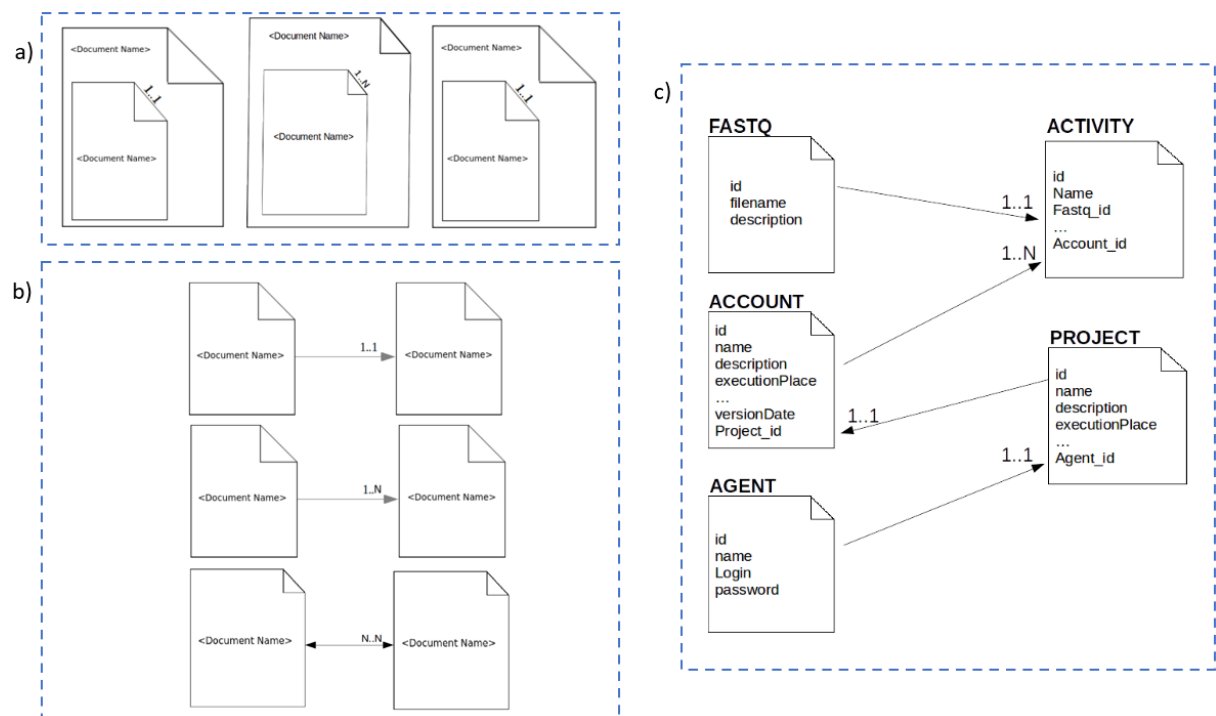
## P04

Reference (VERA et al., 2015) proposes a graphical notation for the design of document-oriented database schemas. Rectangles are used to represent homogeneous collections and the field structure of their documents. The fields are represented by only their names, as there is no representation for data types nor any way to set them as identifiers or unique. A rectangle within another rectangle represents a relationship using embedded documents, which can assume a 1:1, 1:N, or M:N cardinality. As the embedded document is represented using a containment association (i.e., within a collection), the proposal avoids relating documents that belong to different collections by using this relationship

approach. However, this graphical representation can seem ambiguous, as the same symbol (rectangle) represents a collection with the field structure of its documents or a field structure of embedded documents. Because there is no representation that defines a field as an identifier, no evidence was found regarding the definition of identifier fields in embedded documents. Relationships using referenced documents are represented by a link that relates two collections, whose cardinality can be 1:1, 1:N, or M:N. This proposal is evaluated through a case study that models a database containing genomic data.

Figure 10a and Figure 10b show how to represent embedded documents and referenced documents, respectively. Figure 10c shows an example schema in which documents are related using referenced documents having different cardinalities.

Figure 10 – On the left, the graphical notation to represent (a) embedded documents and (b) referenced documents. On the right, (c) a schema example for the proposal of P04.



**Source:** VERA et al. (2015)

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D3 - Depicts the field structure of documents; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D11 - Depicts the M:N cardinality for embedded documents; D14 - Establishes relationships using referenced documents; D15 - Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the

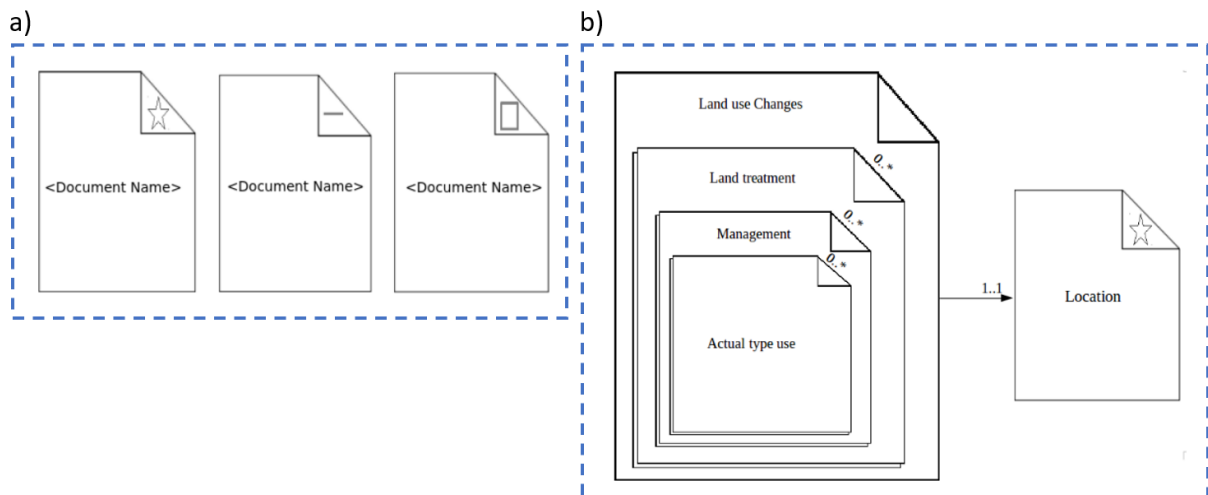
1:N cardinality for referenced documents; D17 - Depicts the M:N cardinality for referenced documents; M1 - Proposes a graphical notation; and M4 - Evaluates the proposal.

## P05

Reference (FILHO et al., 2015) extends the graphical notation of P04 to support the modeling of geospatial data. The data types point, line, and polygon are represented with the pictograms star, line, and square, respectively. These pictograms are drawn in the top right of the collection, limiting the depiction to only one geospatial field per document. This proposal also differs from P04 by the relationship cardinality allowed in the embedded or referenced documents. While P04 allows for the definition of 1:1, 1:N, or M:N cardinality, P05 permits only 1:1 or 1:N. A case study that models a database with biomes graphically identified by points (i.e., a latitude and longitude pair) is presented as the proposal evaluation.

Figure 11a shows how to represent geospatial data using this proposal, while Figure 11b presents an example of a schema that models a nesting of embedded documents and a relationship with referenced documents. Note that the document named Location has a star in the upper right corner, indicating a point-type field in this document, but without indicating which field has that data type.

Figure 11 – On the right, (a) the geospatial data representation using pictograms in documents. On the left, (b) an example of a schema modeled with the proposal from P05.



Source: FILHO et al. (2015)

In short, evidence was found in this proposal for the following resources: D1 - Defines

homogeneous collections; D3 - Depicts the field structure of documents; D5 - Specifies the geospatial data types; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D14 - Establishes relationships using referenced documents; D15 - Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the 1:N cardinality for referenced documents; M1 - Proposes a graphical notation; and M4 - Evaluates the proposal.

## P06

Reference (BANERJEE et al., 2015) defines a systematic transformation from the graph-semantic based conceptual data model (GOOSSDM)(SARKAR; ROY, 2011) to JSON Schemas. Its graphical notation uses the vertices and edges of GOOSSDM to model a document-oriented database schema. Vertices can assume the shape of a rectangle or circle, in which the first represents an object (i.e., a field structure of documents) or an array (i.e., a multivalued field), while the second denotes a field. The fields are graphically represented with their names below the vertices and can be set as identifiers (using a gray background within the vertices). However, the graphical notation does not allow for the definition of the data type of fields nor set them as unique. Edges are graphically represented with (i) one arrow, (ii) a filled arrow, or (iii) no arrow. The first is used to relate a circle to a rectangle, indicating that a field belongs to a field structure of documents, or that a field belongs to a multivalued field. The second is used to relate two rectangles and represents a relationship using embedded documents, which is implemented using JSON Schema inheritance, a legacy resource that existed up until Draft 3 of the JSON Schema (ZYP; COURT, 2010) but that is not used in the current version (WRIGHT et al., 2020). The third is also used to relate two rectangles, defining a nested array or a relationship using embedded documents having 1:1, 1:N, or M:N cardinality. Although the paper shows an edge named Reference, there are no examples or mentions of how this proposal supports relationships using referenced documents. No evidence was found regarding the definition of identifier fields in embedded documents. Furthermore, because there is no symbol to represent a collection, no evidence was found that this proposal can define homogeneous or heterogeneous collections or draw relationships between documents from different collections as embedded documents. A case study that models and generates JSON Schema

for a project management database is presented to evaluate the proposal.

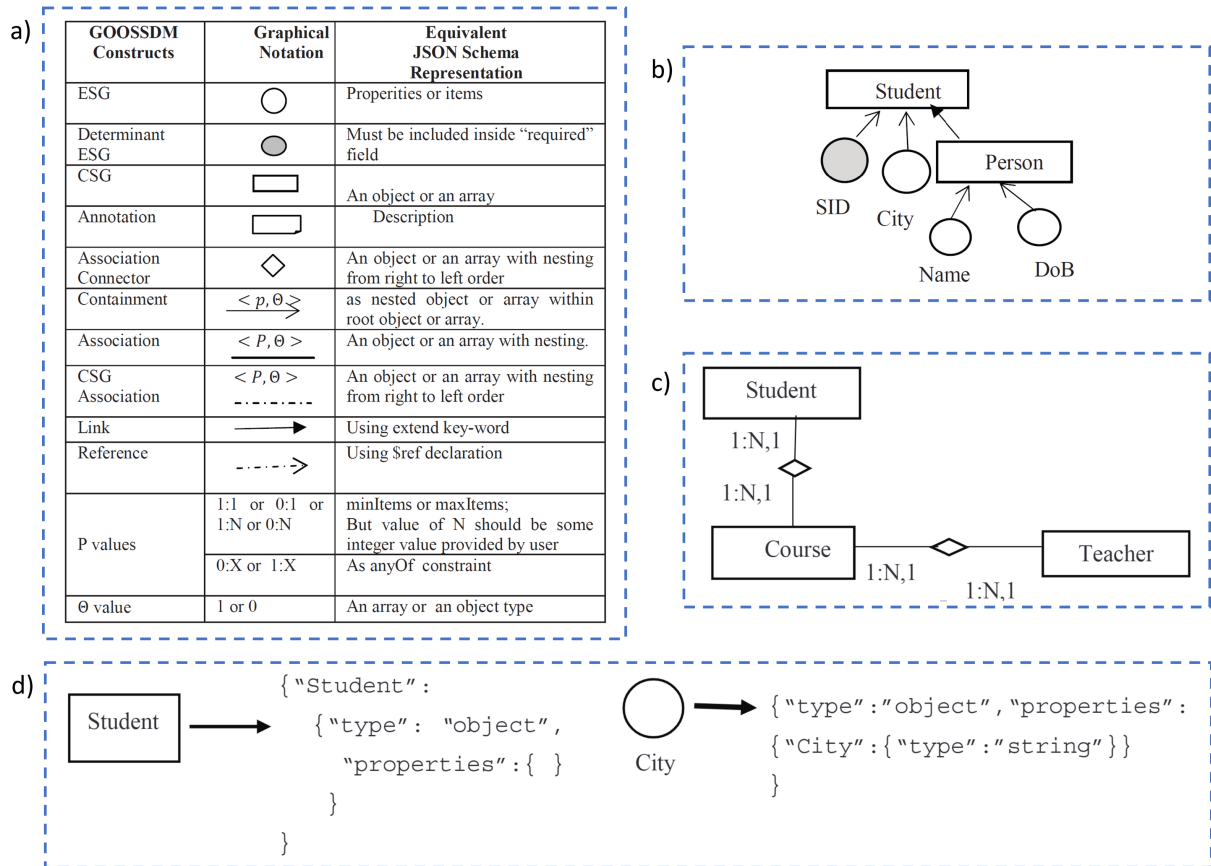
Figure 12a shows the graphical notation proposed in P06. Figure 12b illustrates an example that defines a field structure of documents named Student, composed of an identifier field named SID and a regular field named City. Student embeds Person (relationship represented by the edge with a filled arrow), a field structure of documents composed of two fields called Name and DoB. Note in this example that linkage associations (represented by edges with one arrow) are used to indicate that fields (circles) belong to Person or Student (rectangles), which can create a polluted diagram when many fields must be represented. Figure 12c illustrates an example where Student, Course, and Teacher are related using edges without arrows. The value 1 after the cardinality (i.e., 1:N, 1) means that the rectangle represents a multivalued field because, as defined in the graphical notation (cf.  $\Theta$  value in Figure 12a), a value of 1 indicates an array, while a value of 0 indicates an object (i.e., a field structure of documents). The association connector indicates how these multivalued fields are nested: one or more Teacher instances are nested into one Course instance (right to left), and one or more Course instances are nested into one Student instance (bottom to top). Note in this example that if the  $\Theta$  value was 0, this would imply a nesting of embedded documents (i.e., complex data). Finally, Figure 12d shows the translation of a rectangle (that models a field structure of documents) and a circle (that models a field) to their respective JSON Schemas.

In short, evidence was found in this proposal for the following resources: D3 - Depicts the field structure of documents; D6 – Represents identifier fields; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D11 - Depicts the M:N cardinality for embedded documents; M1 - Proposes a graphical notation; M3 - Presents translational semantics; and M4 - Evaluates the proposal;

## P08

Reference (SHIN; HWANG; JUNG, 2017) proposes a NoSQL database design method based on the UML class diagram notation. In the document-oriented database context, classes represent collections and the field structure of their documents. Fields can be set as identifier (with the “«PK»” stereotype), unique (with the “«AK»” stereotype), or regular (with no stereotype). However, their data types are not indicated. Associations

Figure 12 – The (a) graphical notation, (b, c) example schemas, and (d) mapping of symbols to JSON Schemas in P06.

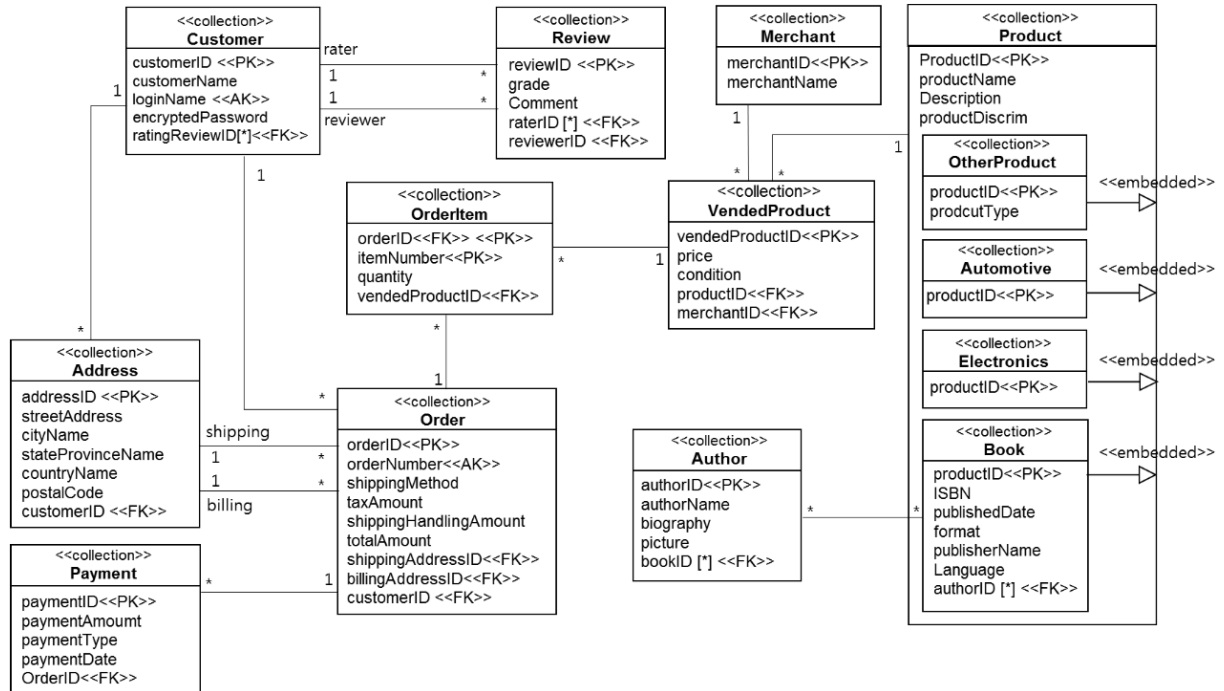


Source: BANERJEE et al. (2015)

represent relationships using referenced or embedded documents. An association between two classes without tail decoration represents referenced documents, whose cardinality can be 1:1, 1:N, or M:N. The field that stores the reference is marked with “«FK»”. An association with an arrow (similar to a generalization link) on one tail and the “«embedded»” stereotype between two classes indicates a relationship using embedded documents, but without representation of its cardinality. In this relationship approach, the class that represents the embedded document is depicted within another, i.e., using a containment association. This means that this proposal allows for the definition of homogeneous collections only, as a class cannot have another class within it that does not belong to an embedded document relationship. However, embedding one collection (a class) into another seems to be a mistake, as a collection cannot be embedded into another, and documents from different collections cannot be related as embedded documents. The proposal is evaluated through a case study that models an e-commerce business database in which a customer orders products through the internet or a smartphone.

Figure 13 shows a schema modeled with this proposal. Note that the class Product embeds four classes, named OtherProduct, Automotive, Electronics, and Book, which is a mistake, because a collection cannot be embedded into another. Furthermore, there are no mechanisms to prevent the definition of identifier fields in embedded documents because, as shown in this example, identifier fields are found in the embedded documents.

Figure 13 – Example schema depicted using the proposal from P08.



Source: SHIN; HWANG; JUNG (2017)

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D3 - Depicts the field structure of documents; D6 – Represents identifier fields; D7 – Represents unique fields; D8 - Establishes relationships using embedded documents; D12 - Specifies documents from different collections as embedded documents; D13 - Defines identifier field in an embedded document; D14 - Establishes relationships using referenced documents; D15 - Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the 1:N cardinality for referenced documents; D17 - Depicts the M:N cardinality for referenced documents; M1 - Proposes a graphical notation; and M4 - Evaluates the proposal.

## P09

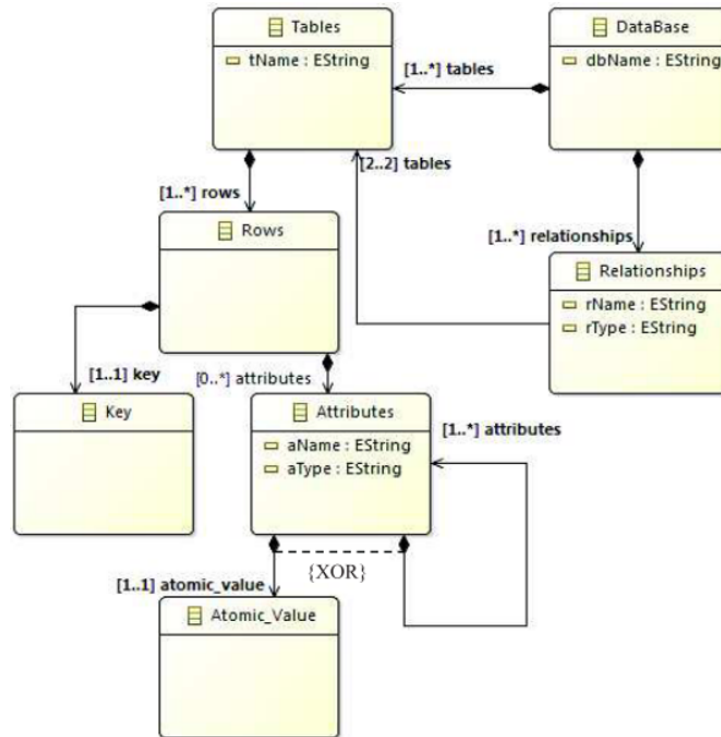
Reference (ABDELHEDI et al., 2017b) proposes UMLtoNoSQL<sup>2</sup>, an automated process to transform conceptual models to physical models of NoSQL databases (document-oriented, column-oriented, or graph-oriented). In other words, this proposal consists of a UML profile that uses class diagram notation to represent a schema, which is then used to generate documents. In a class diagram, classes represent homogeneous collections, and attributes represent the field structure of their documents. Fields are defined with conventional data types and can be set as identifiers. The composition and association links are used to define relationships that use embedded and referenced documents, respectively. For both approaches, cardinalities of 1:1, 1:N, or M:N are allowed. As classes can be related with a composite link, it seems that this proposal allows embedded documents to be drawn from documents belonging to different collections. No evidence was found regarding the definition of identifier fields in embedded documents. This proposal defines a set of rules used to transform the concepts of the UML metamodel into concepts from the document-oriented data model, which are defined in a generic metamodel. However, this generic metamodel does not capture the particularities of document-oriented databases, such as the definition of collections and relationships using embedded documents. An experiment that models a schema using a UML class diagram and analyzes the runtimes of some queries is used to evaluate the proposal.

Reference (ABDELHEDI et al., 2017b) does not depict a schema example depicted with its proposal. However, the metamodel for the generic data model is addressed, as shown in Figure 14. This metamodel defines a Database to be composed of Tables (collections) and Relationships between Tables. Tables should have one key (identifier field) and other attributes (regular fields), defined with a name and data type. Attributes can be multivalued or atomic.

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; D6 - Represents identifier fields; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D11 - Depicts the M:N car-

<sup>2</sup> This proposal is also addressed in the references (ABDELHEDI et al., 2016) and (ABDELHEDI et al., 2017a).

Figure 14 – Metamodel that defines a generic logical model for NoSQL databases in P09.



Source: ABDELHEDI et al. (2017b)

dinality for embedded documents; D12 - Specifies documents from different collections as embedded documents; D14 - Establishes relationships using referenced documents; D15 - Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the 1:N cardinality for referenced documents; D17 - Depicts the M:N cardinality for referenced documents; M1 - Proposes a graphical notation; M2 - Defines a metamodel or grammar (it was partially found, as the proposed metamodel adapts the UML metamodel for the design of document-oriented databases); and M4 - Evaluates the proposal.

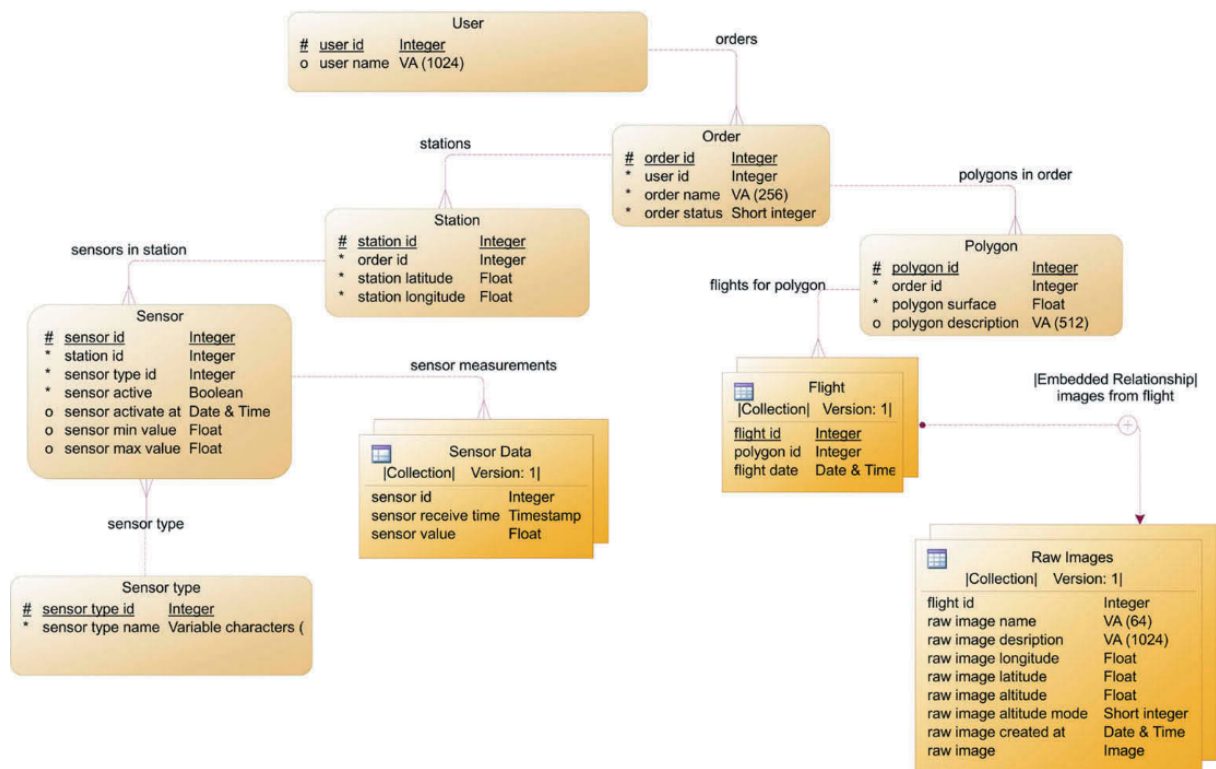
## P10

Reference (ZEČEVIĆ et al., 2018) proposes a graphical notation to model hybrid databases, which are databases containing both relational and NoSQL characteristics. In the document-oriented database context, entities represent homogeneous collections, and attributes make up the field structure of the documents. These fields are defined with either a conventional or geospatial data type, and can be set to be identifiers. Links are used to represent the relationships that use embedded or referenced documents. For both relationship approaches, cardinality can be 1:1, 1:N, or M:N, which are represented according

to the IDEF1X notation. Because entities represent collections, it seems to be a mistake to embed a collection into another, as well as relate documents from different collections as embedded documents. No evidence was found regarding the definition of identifier fields in embedded documents. A metamodel defines the concepts that can be used to implement a lightweight extension for modeling tools. These lightweight extensions define UML profiles, adding constraints and new elements without modifying the metamodel of an existing modeling language (BRUCK; HUSSEY, 2007). This proposal is evaluated by modeling a hybrid database (relational and document-oriented) that stores data about crops and farmers.

Figure 15 shows the database schema used in the proposal evaluation. The entities Sensor Data, Flight, and Raw Images (with corner edges and overlapping rectangles) represent collections for a document-oriented database, while other entities (with rounded edges) represent tables for a relational database. Note that the collection Flight embeds the collection Raw Image, but in a document-oriented database, a collection can not be embedded into another collection.

Figure 15 – Example of hybrid databases modeled with the proposal from P10.



Source: ZEČEVIĆ et al. (2018)

In short, evidence was found in this proposal for the following resources: D1 - Defines

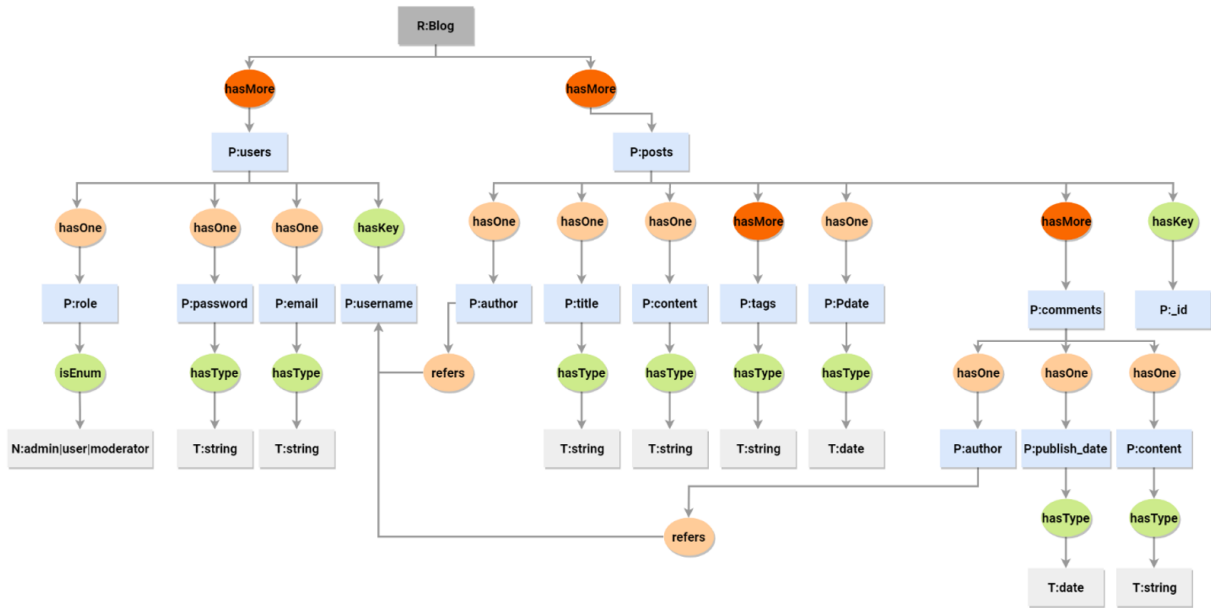
homogeneous collections; D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; D5 - Specifies the geospatial data types; D6 - Represents identifier fields; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D11 - Depicts the M:N cardinality for embedded documents; D12 - Specifies documents from different collections as embedded documents; D14 - Establishes relationships using referenced documents; D15 - Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the 1:N cardinality for referenced documents; D17 - Depicts the M:N cardinality for referenced documents; M1 - Proposes a graphical notation; M2 - Defines a metamodel or grammar; and M4 - Evaluates the proposal.

## P11

Reference (ANDOR; VARGA; SăCăREA, 2019) presents a modeling method based on conceptual graphs for MongoDB. In the graphical notation of this proposal, a schema is designed using nodes and edges. Nodes can represent more than one concept, as they are used to depict a homogeneous collection, a field structure for documents, a field, or a conventional data type. An edge is used to make a relationship between two nodes, and the description of this relationship uses JSON Schema keywords (e.g., `keyField`, `hasOne`, `hasMor`, `isOptional`) within an oval symbol. Two nodes related by an edge can represent, for example, that a document field structure has an identifier field (`keyField`), a regular field (`hasOne`), a multivalued field (`hasMore`), or an optional field (`isOptional`). Note that a field can be set as identifier or regular, but not as unique. Edges are also used to establish relationships using embedded or referenced documents, but without a cardinality definition. No evidence was found regarding relating documents from different collections as embedded documents, nor about the definition of identifier fields in embedded documents.

Figure 16 shows an example of a schema represented with this proposal, which models a database called Blog, containing documents that store users and posts. This schema models posts with one or more comments (array of embedded documents), where each comment is related to an author (referenced documents). In other words, although the relationship cardinality is not depicted, this proposal enables relationships to be defined as embedded documents with 1:1 or 1:N cardinality. A JSON Schema for this example is shown in the paper.

Figure 16 – Example schema depicted as a graph in P11.



Source: ANDOR; VARGA; SăCăREA (2019)

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; D6 – Represents identifier fields; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D14 - Establishes relationships using referenced documents; M1 - Proposes a graphical notation; and M3 - Presents translational semantics.

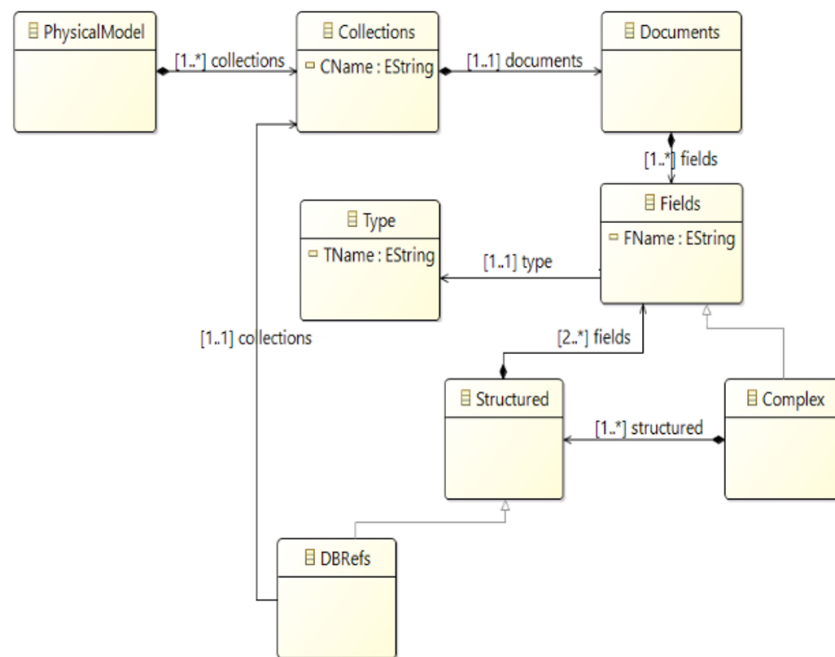
## P12

Reference (BRAHIM; FERHAT; ZURFLUH, 2019) presents a metamodel for document-oriented databases, which is used in a process that extends the one shown in the paper P09, in order to extract a conceptual model from a document-oriented database (i.e., reverse engineering). The schema retrieval is done using rules that map a metamodel with document-oriented concepts to the UML metamodel. The proposal is evaluated by a case study that extracts the conceptual model from a MongoDB database with medical data.

Figure 17 shows the proposed metamodel, which defines a database (PhysicalModel) composed of one or more collections (Collections), each one with only one document field structure (Documents), i.e., homogeneous collections. A field (Fields) has a name

and a data type (Type), which can be integer, string, or boolean (i.e., conventional data types). There is no support to set a field as identifier or unique. This metamodel also defines that documents can be related using embedded or referenced documents, but there is no mention of the relationship cardinality definition. Relationships using embedded documents are defined by the metaclass Complex, a specialization of Fields. Complex is composed of instances of the metaclass Structured, indicating that this type of relationship is established among documents that belong to the same collection. However, because Complex is composed of one or more instances of Structured, it means that embedded documents can be stored in arrays, which enables relationships with a cardinality of 1:1 or 1:N. Relationships using referenced documents are defined by the metaclass DBRefs, a specialization of Structure related to the Collections metaclass. This definition allows documents that belong to the same or different collections to be related as referenced documents. However, as there is no support for defining identifier fields in this metamodel, it is difficult to recognize which field is referenced in a relationship.

Figure 17 – Metamodel for document-oriented databases proposed in P12.



**Source:** BRAHIM; FERHAT; ZURFLUH (2019)

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for

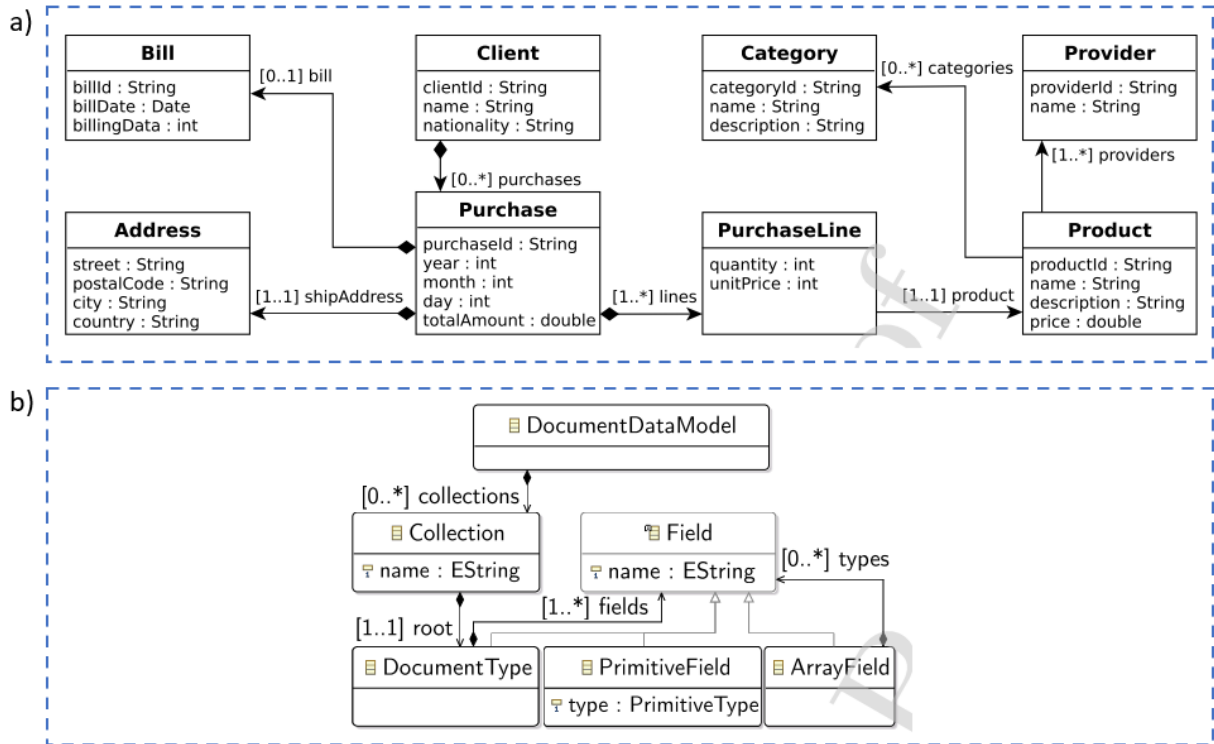
embedded documents; D14 - Establishes relationships using referenced documents; M2 - Defines a metamodel or grammar; and M4 - Evaluates the proposal.

## P13

Reference (DE LA VEGA et al., 2020) proposed a representation of NoSQL (document-oriented or column-oriented) database schemas using the UML class diagram notation. For the document-oriented databases, classes represent homogeneous collections, and attributes represent the document fields. These fields are defined with a conventional data type, but there is no mention of setting them as identifiers or unique. The association and composition links are used to represent relationships using referenced and embedded documents, respectively. The multiplicity of the relationship is represented for both relationship approaches, making it possible to define cardinality as 1:1 or 1:N. However, relating two classes with a composition seems to be a mistake, as there is no relationship between collections in a document-oriented database nor relationships among documents of different collections using embedded documents. This proposal also defines a metamodel for document-oriented databases, which is used to generate the physical schema (i.e., JSON Schema) from a conceptual model (i.e., a UML class diagram). This metamodel defines embedded documents as belonging to the same collection, although the graphical notation indicates that a collection can be embedded into another (i.e., two classes related with a composition link). This proposal was evaluated in a case study that transforms industrial data related to a cutting machine so it can be stored in different NoSQL databases, such as MongoDB. A prototype of this proposal is available for download.

Figure 18a shows a schema represented by this proposal, which deals with a database that stores e-commerce data. Figure 18b shows the proposed metamodel for document-oriented databases. This metamodel defines a database (DocumentDataModel) with one or more collections (Collection), each composed of one document field structure (DocumentType). A field can be single-value or multivalued, as its metaclass (Field) is specialized as either PrimitiveField or ArrayField. The relationships using embedded documents are captured by the metaclass Field, as this metaclass is also specialized in DocumentType, which is composed of one or more Fields. However, the metamodel does not define the graphical notation symbols that represent relationships, nor how relationships using embedded documents are established.

Figure 18 – A (a) conceptual model example and the (b) metamodel for document-oriented documents proposed in P13.



Source: DE LA VEGA et al. (2020)

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; D12 - Specifies documents from different collections as embedded documents; D14 - Establishes relationships using referenced documents; D15 - Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the 1:N cardinality for referenced documents; M1 - Proposes a graphical notation; M2 - Defines a metamodel or grammar; M4 - Evaluates the proposal; and M5 - Provides computational support, such as a CASE tool.

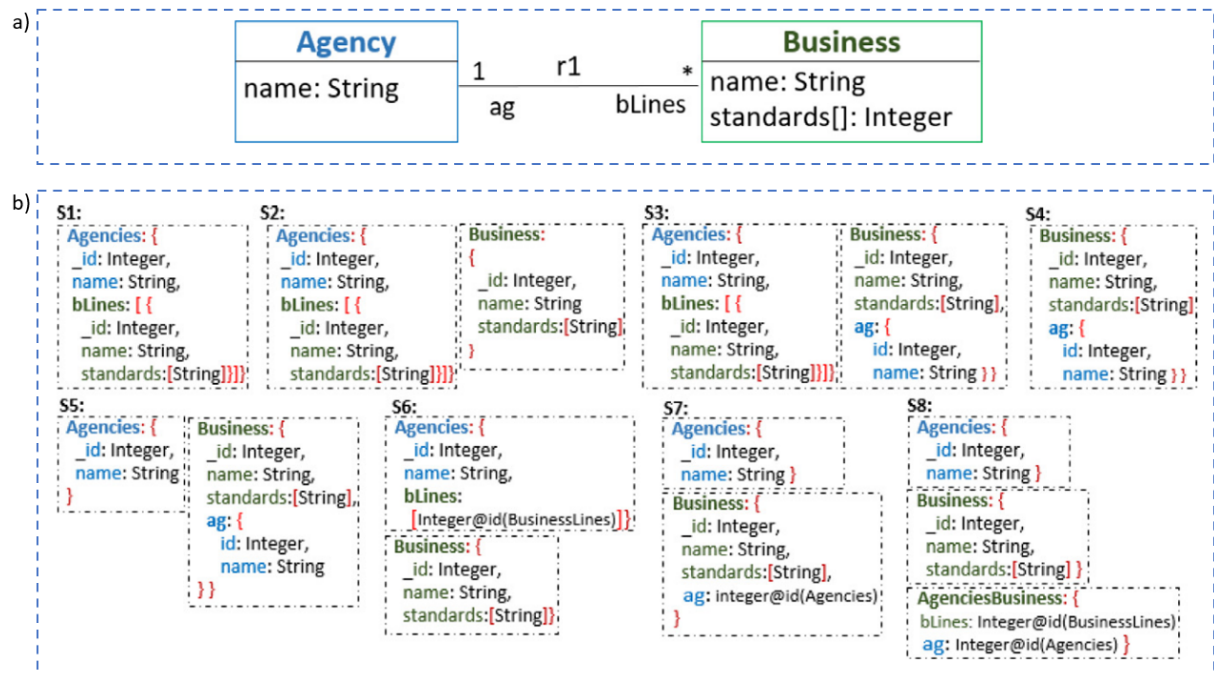
## P14

Reference (GÓMEZ; RONCANCIO; CASALLAS, 2021) addresses a method of automatically generating a set of document-oriented structuring alternatives (as text) from a schema modeled with a UML class diagram. In a diagram, a class represents a homoge-

neous collection, and the attributes compose the document field structure. The fields are represented by a name and a conventional data type, but there is no representation for setting them as identifier or unique. An association link is used to represent a relationship between documents, without distinguishing the relationship type (i.e., referenced or embedded documents). Alternatives for implementing the physical model are generated from two related classes, which include relating the documents using references or embedding them. In other words, the proposal has no resources to distinguish referenced or embedded documents, nor represent the relationship cardinality. The proposal is evaluated with a case study using MongoDB, where structuring alternatives from a schema are presented and compared.

Figure 19a shows a diagram in which two collections named Agency and Business are related. From this diagram, eight different structuring alternatives (S1 to S8) are shown in Figure 19b. The structuring alternatives are presented using AJSchema, an abstraction based on JSON code. Each rectangle with dotted edges represents a collection, with its document field structure. In this example, from two related classes, the proposal generates alternatives for implementing the relationship using referenced documents (S5, S6, S7, S8), embedded documents (S4), or arrays of embedded documents (S1, S2, S3).

Figure 19 – An (a) example diagram and its (b) document-oriented structuring alternatives - P14.



Source: G6MEZ; RONCANCIO; CASALLAS (2021)

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; M1 - Proposes a graphical notation; and M4 - Evaluates the proposal.

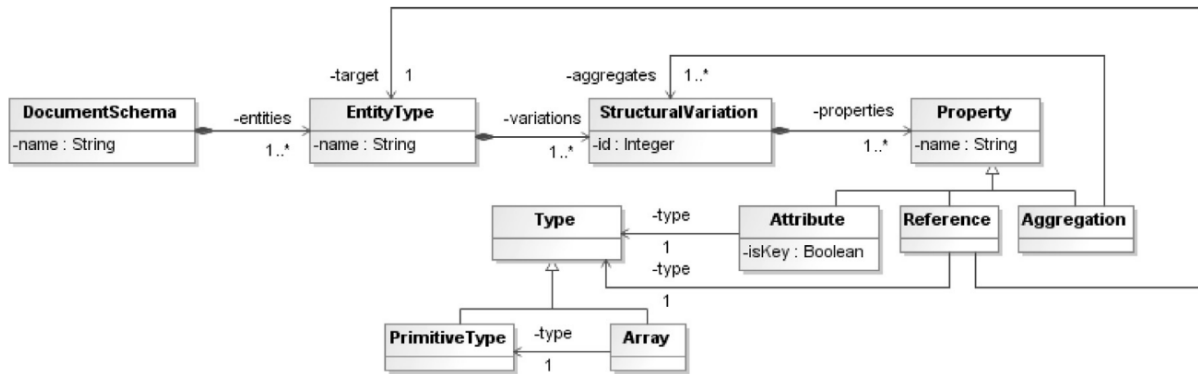
## P16

Reference (CANDEL; RUIZ; GARCÍA-MOLINA, 2022) proposes a unified metamodel for database schemas, which is then mapped to metamodels that capture the particularities of the different NoSQL databases categories (document-oriented, column-oriented, and graph-oriented). The proposal was evaluated with an experiment that generated databases of different categories from a data set.

Figure 20 shows the metamodel for the document-oriented databases. In this metamodel, a database (DocumentSchema) contains one or more collections (EntityType). The collections can have one or more document field structures (StructuralVariation), which means that a collection can be homogeneous or heterogeneous. The document field structure is composed of fields (Property), which can be specialized as conventional fields (Attribute - PrimitiveType), multivalued fields (Attribute - Array), relationship references (Reference), or fields that embed documents (Aggregation). Fields can also be set as identifiers in the documents (isKey in Attribute). Note that this metamodel provides support for defining relationships using embedded or referenced documents, but without a relationship cardinality definition. As the metaclass Aggregation is related to the metaclass StructuredVariation (which defines a document field structure in a collection), the metamodel prevents relationships between documents belonging to different collections from being defined. No evidence was found regarding the definition of identifier fields in embedded documents.

In short, evidence was found in this proposal for the following resources: D1 - Defines homogeneous collections; D2 - Defines heterogeneous collections; D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; D6 – Represents identifier fields; D8 - Establishes relationships using embedded documents; D14 - Establishes relationships using referenced documents; M2 - Defines a metamodel or grammar; and M4 - Evaluates the proposal.

Figure 20 – Metamodel for document-oriented databases proposed in P16.



Source: CANDEL; RUIZ; GARCÍA-MOLINA (2022)

### 3.2.2 DGDW schemas

#### P07

Reference (FERRAHI et al., 2017) proposes a UML Profile for the design of DGDW schemas. This proposal mixes SOLAP and GDW concepts in a schema, which is represented using the UML class diagram notation. A class represents a fact or an aggregation of levels (i.e., dimensions), and corresponds to a document field structure. Attributes represent levels or measures, which correspond to fields within the documents. These fields are defined with a conventional or geospatial data type and can be set as identifiers. Associations are used to establish relationships between facts and aggregations of levels using embedded documents, but without a relationship cardinality definition. Relationships using referenced documents are not mentioned. This proposal uses stereotypes (defined in a metamodel) and OCL rules to adapt the symbol semantics to the DGDW domain. In other words, this proposal defines a metamodel that adapts the UML metamodel to the DGDW domain. Because no graphical representation for collections is mentioned, there is no evidence if the proposal allows for the definition of homogeneous or heterogeneous collections, nor for the establishment of relationships using embedded documents between documents that belong to different collections. No evidence was found with regard to the definition of identifier fields in embedded documents or to the avoidance of facts disconnected from level aggregations. This proposal was evaluated through an experiment that modeled and generated a DGDW from an RGDW (SIQUEIRA et al., 2010), analyzing its data volume and query performance.

Figure 21a shows a schema depicted with this proposal, named Falling Star. This schema models geospatial fields into fact documents (class Lineorder\_), while the conventional fields from different dimensions are mixed in embedded documents (class fact). As can be seen in the implementation of this schema shown in Figure 21b, every fact document has an array of embedded documents (cf. row 11 in Figure 21b), each containing a combination of fields belonging to different dimensions. It means that, although it is not graphically represented, this proposal enables relationship cardinality to be defined as 1:1 (only one embedded document into the array) or 1:N (many embedded documents into the array). It is important to highlight that there is no graphical notation for representing dimensions, as these are represented by aggregations of levels. Furthermore, level is a SOLAP concept, but because DGDW can be used for applications other than SOLAP (e.g., data mining, reports, and what-if-analysis), SOLAP concepts are dispensable when modeling a DGDW schema.

Figure 21 – (a) The Falling Star schema, and (b) its implementation in MongoDB - P07.



Source: FERRAHI et al. (2017)

In short, evidence was found in this proposal for the following resources: D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; D5 – Specifies the geospatial data types; D6 – Represents identifier fields; D8 - Establishes relationships using embedded documents; D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents; G1 - Distinguishes facts and dimensions (it was partially found because there is no graphical representation for dimensions); M1 - Proposes a graphical notation; M2 - Defines a metamodel or grammar (it was partially found, as the proposed metamodel adapts the UML metamodel for the DGDW domain); and M4 - Evaluates the proposal.

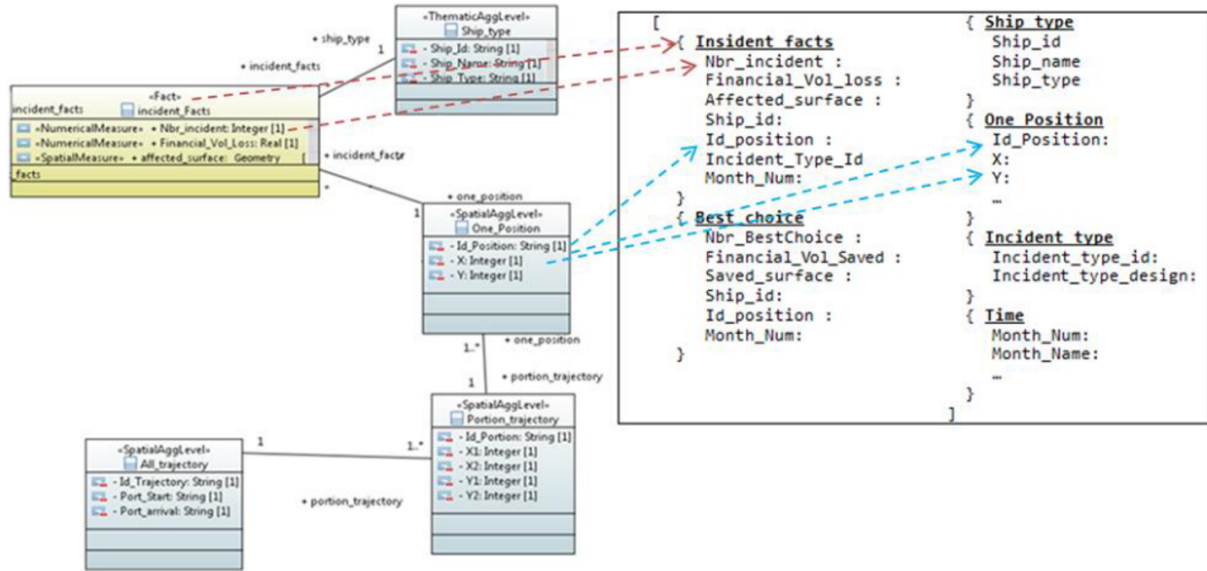
## P15

Reference (BENSALLOUA; BENAMEUR, 2021) proposed a UML profile for modeling maritime navigation DGDW schemas. This proposal was refined from that presented by the authors in (BOULIL; BIMONTE; PINET, 2015), which addressed a UML profile for modeling RGDW. The graphical notation uses the UML class diagram notation as follows: classes represent documents containing facts or level aggregations (i.e., dimensions), attributes represent the fields, and associations with 1:1 or 1:N cardinality represent relationships that use referenced documents. Fields are specified with a geospatial or conventional data type, but without any indication to set them as identifiers or unique. Similar to P07, a metamodel is proposed to define stereotypes that adapt the symbols of UML for the design of DGDW schemas. In addition, this proposal also mixes concepts from SOLAP and DGDW, as level aggregations compose a schema. There is no mention of a symbol to represent a collection (homogeneous or heterogeneous) in a diagram or to establish relationships using embedded documents. This proposal was evaluated in a case study that built a DGDW using MongoDB. This DGDW is designed to help decision-makers choose, for example, the best route a ship should take.

Figure 22 shows a diagram depicted with P15 and its mapping to JSON code. In this DGDW schema, the class “Incident\_Facts” models fact documents, while the other classes model level aggregations.

In short, evidence was found in this proposal for the following resources: D3 - Depicts the field structure of documents; D4 - Specifies the conventional data types; D5 – Specifies the geospatial data types; D14 - Establishes relationships using referenced documents; D15

Figure 22 – Mapping the symbols of a DGDW for the document-oriented data model in P15.



Source: BENSALLOUA; BENAMEUR (2021)

- Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the 1:N cardinality for referenced documents; G1 - Distinguishes facts and dimensions (it was partially found because there is no graphical representation for dimensions); M1 - Proposes a graphical notation; M2 - Defines a metamodel or grammar (it was partially found, as the proposed metamodel adapts the UML metamodel for the DGDW domain); and M4 - Evaluates the proposal.

### 3.3 OVERALL ANALYSIS

Table 2 summarizes the evidence found in the studies presented in Table 1 for the resources addressed in section 3.2. Each row corresponds to a proposal, and the columns correspond to the resources each one provided. An empty circle indicates that the item was not found (i.e., cannot be affirmed that it is addressed), a filled circle indicates that the item was found (i.e., there is evidence that it is addressed), and a half-filled circle indicates that the item was partially found (i.e., there is evidence that it is addressed, but with some caveats). A hyphen indicates that the discussion of the feature does not apply to the proposal, as it is designed to model document-oriented database schemas for general-purpose.

Regarding the features of the document-oriented data model, all proposals support

Table 2 – Overview of related studies.

	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	D16	D17	G1	G2	M1	M2	M3	M4	M5
P01	○	○	●	○	○	●	●	●	●	●	○	○	○	●	●	●	○	-	-	●	○	○	○	○
P02	○	○	●	○	○	○	○	●	○	○	○	○	○	●	○	○	○	-	-	●	○	○	○	○
P03	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	-	-	●	○	○	○	○
P04	●	○	●	○	○	○	○	●	●	●	●	○	○	●	●	●	●	-	-	●	○	○	●	○
P05	●	○	●	○	○	○	○	●	●	●	○	○	○	●	●	●	○	-	-	●	○	○	●	○
P06	○	○	●	○	○	○	○	●	●	●	●	○	○	○	○	○	○	-	-	●	○	●	●	○
P07	○	○	●	●	●	○	○	●	●	●	○	○	○	○	○	○	○	●	○	●	○	○	●	○
P08	●	○	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	-	-	●	○	○	●	○
P09	●	○	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	-	-	●	○	○	●	○
P10	●	○	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	-	-	●	○	○	●	○
P11	●	○	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	-	-	○	○	○	○	○
P12	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	-	-	○	○	○	○	○
P13	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	-	-	○	○	○	○	○
P14	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	-	-	○	○	○	○	○
P15	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
P16	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	-	-	○	○	○	○	○

○ feature not found   ● feature partially found   ● feature found   - not applied

D1 - Defines homogeneous collections; D2 - Defines heterogeneous collections; D3 - Depicts the field structure of documents;

D4 - Specifies the conventional data types; D5 - Specifies the geospatial data types; D6 - Represents identifier fields;

D7 - Represents unique fields; D8 - Establishes relationships using embedded documents;

D9 - Depicts the 1:1 cardinality for embedded documents; D10 - Depicts the 1:N cardinality for embedded documents;

D11 - Depicts the M:N cardinality for embedded documents; D12 - Specifies documents from different collections as embedded documents;

D13 - Defines identifier field in an embedded document; D14 - Establishes relationships using referenced documents;

D15 - Depicts the 1:1 cardinality for referenced documents; D16 - Depicts the 1:N cardinality for referenced documents;

D17 - Depicts the M:N cardinality for referenced documents; G1 - Distinguishes facts and dimensions;

G2 - Prevents disconnected facts or dimensions; M1 - Proposes a graphical notation; M2 - Defines a metamodel or grammar;

M3 - Presents translational semantics; M4 - Evaluates the proposal; M5 - Provides computational support, such as a CASE tool.

**Source:** The Author

the depiction of the field structure of documents (D3: 100%). More than half provide support to represent homogeneous collections (D1:68.8%), to specify conventional data types (D4: 56.3%), to set fields as identifiers (D6: 56.3%), to establish relationships using embedded documents (D8: 87.5%), to depict embedded documents with 1:1 cardinality (D9: 68.8%), and to establish relationships using referenced documents (D14: 81.3%). However, few proposals (half or less) support the representation of heterogeneous collections (D2: 6.3%), the specification of geospatial data types (D5: 25.0%), the setting of fields as unique (D7: 12.5%), and the depiction of relationships using referenced documents with 1:1, 1:N, or M:N cardinality (D15: 50.0%, D16: 50.0%, D17: 25.0%). Note that some proposals allow for the definition of constructions that can impair DGDW performance, such as the drawing of embedded documents with 1:N or M:N cardinality (D10: 68.8%, D11: 31.3%). Furthermore, some proposals allow incorrect constructions to be drawn, such as relations between documents from different collections using embedded documents (D12: 25.0%) and the definition of identifier fields in embedded documents (D13: 6.3%).

Because most proposals are designed to model general-purpose schemas, few cover

the particularities of a GDW schema. In other words, few proposals distinguish facts and dimensions (G1: 12.5%), and none prevent the design of disconnected facts or dimensions (G2: 0.0%). Note, in Table 2, that feature G1 was partially found in the proposals P07 and P15. The reason for this is that there is no representation for dimensions in the schema because, in those proposals, facts are related to level aggregations, which is a SOLAP application concept.

With respect to the characteristics related to the definition of modeling languages, almost all papers present a graphical notation (M1: 87.5%) and evaluate its proposal (M4: 81.3%). However, few proposals define a metamodel (M2: 43.8%), present translational semantics (M3: 12.5%), or provide computational support, such as a CASE tool (M5: 6.3%). Note that feature M3 was marked in Table 2 as partially found for P07, P09, and P15 because the metamodels of these proposals adapted the UML metamodel for the DGDW domain, which can limit the implementation of the modeling language as only a UML profile.

### 3.4 CHAPTER FINAL CONSIDERATIONS

This chapter presented studies related to this thesis. These studies were selected through a method of searching for proposals that address modeling languages for general-purpose document-oriented databases or DGDW. For each proposal found, evidence for their support of modeling features of the document-oriented databases and features of the GDW logical design were sought. Evidence for features regarding the definition of modeling languages was also sought. The evidence found was summarized in a table, in order to highlight the existing gaps regarding the definition of a modeling language for DGDW.

## 4 ASTAR

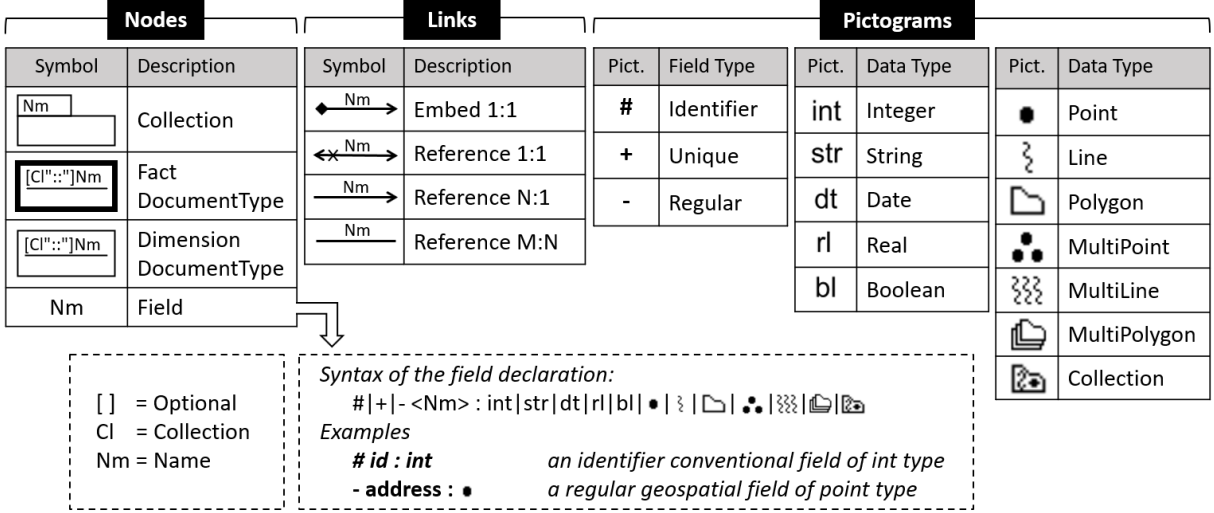
AStar is a *Domain-Specific Modeling Language* (DSML) whose graphical notation (concrete syntax) is inspired by the UML class diagram notation. AStar therefore uses symbols that are well known in both industry and academics, although it is not a UML profile, as its syntax and semantics are defined in a specific and more straightforward metamodel (abstract syntax) and set of well-formedness rules (static semantics). This chapter details the specifications for AStar, presenting its concrete syntax in section 4.1, its abstract syntax in section 4.2, its static semantics in section 4.3, and translational semantics in section 4.4. The chapter final considerations are presented in section 4.5.

### 4.1 CONCRETE SYNTAX

Figure 23 shows the AStar concrete syntax, which consists of a graphical notation whose symbols are classified as nodes, links, and pictograms. Nodes represent facts, dimensions, or fields; links model the relationships between facts and dimensions; and pictograms are representations for field types (i.e., identifier, unique, or regular) and data types (i.e., conventional or geospatial). The following paragraphs describe the AStar graphical notation using some examples<sup>1</sup>. Figure 24 through Figure 28 cover the main symbols of AStar, which are exemplified using the geospatial version of the *Star Schema Benchmark* (SSB) (SIQUEIRA et al., 2009). Figure 29 addresses a scenario that uses relationships having 1:1 and M:N cardinalities, which is based on the *International Classification of Crime for Statistical Purposes* (ICCS) (DRUGS; CRIME, 2015). The figures also present example code for inserting JSON documents into the MongoDB database. The JSON Schemas corresponding to these examples are available on Appendix B.

As can be seen in Figure 23, the AStar notation has four nodes, each one with a name (i.e., Nm). The first node (Collection) is graphically depicted using package notation and models a collection of documents in the database. The second and third nodes are graphically depicted by classes and correspond to an abstraction called *DocumentType*, which specifies the field structure of a fact or dimension. In other words, a *DocumentType* defines the field names and their data types, and also indicates which fields are identifier, unique,

<sup>1</sup> Colored texts are used in the examples to highlight details that are explored in the example explanation. Different colors do not represent different meanings.

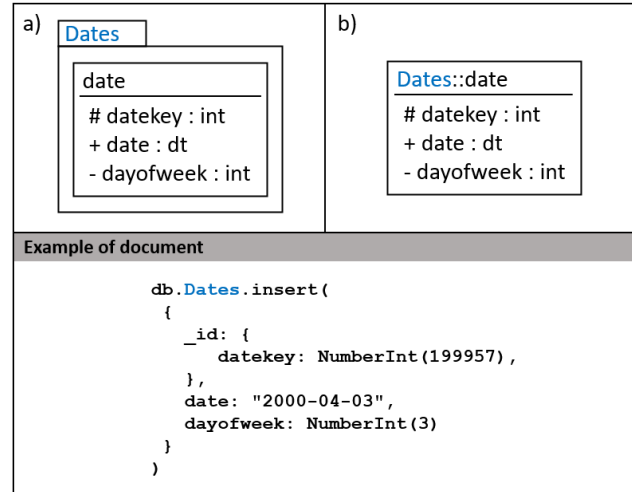


or regular. DocumentTypes can therefore be used to model a DGDW as a well-structured database, regardless of the document-oriented databases having flexible schemas. Facts are distinguished graphically from dimensions using thick edges to represent a Fact DocumentType and thin edges for a Dimension DocumentType. Both Fact and Dimension DocumentTypes can have a fully-qualified name using the collection name in which the DocumentType is contained (i.e., [Cl“::”]). When this fully-qualified name is used, it replaces the use of package notation to represent the collection. Note that the use of the package graphical representation highlights the collections, while fully-qualified names provide cleaner schemas and are easier to be defined by hand. The designer is, therefore, free to decide between modeling collections as packages (first node) or with fully-qualified names (textual representation). The last node is graphically depicted by a UML visibility symbol, a name, and a pictogram to model the fact and dimension fields. The #, +, and - pictograms before a field name define it as an identifier, unique, or regular, respectively. Furthermore, pictograms indicate the field’s data type.

The nodes of AStar can be related as follows: (i) a Collection can contain one or more Fact DocumentTypes or Dimension DocumentTypes, (ii) a DocumentType can contain one or more Fields, and (iii) a DocumentType can be related to another DocumentType by the Links shown in Figure 23. The first two constructions are based on containment associations (cf. Figure 5b), because they are used to draw graphical structures that do not produce multiple nesting of symbols. The third construction, however, is based on linkage associations (cf. Figure 5b), because it supports the nesting of DocumentTypes

to represent embedded documents, as well as relationships among facts and dimensions using references. Therefore, the nesting of embedded DocumentTypes is represented as a linkage, simplifying the graphical specification of this construction.

Figure 24 – Two ways to model the date dimension: (a) using package or (b) fully-qualified name in the DocumentType.



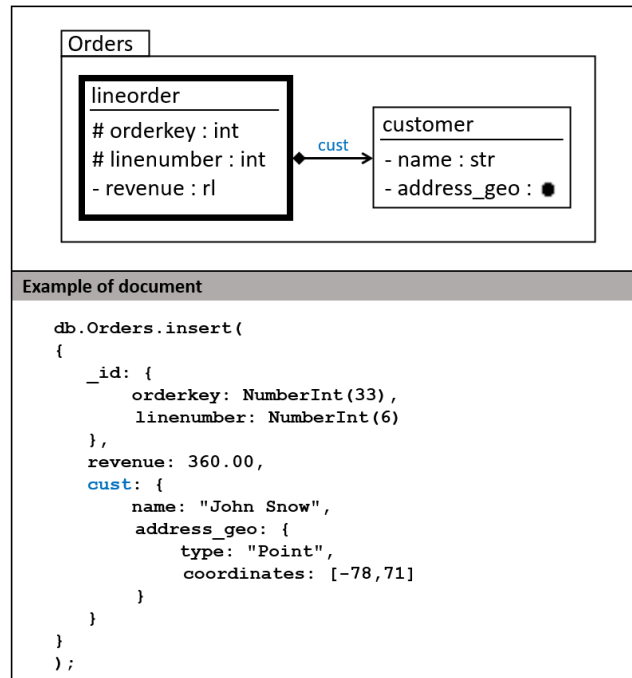
Source: The Author

Figure 24 presents two examples that demonstrate the containment association. Both Figure 24a and Figure 24b model only a small part of the date dimension of SSB, which covers three field types: identifier (datekey), unique (date), and regular (dayofweek). Note that Figure 24a models the collection Dates using a package, while Figure 24b uses a fully-qualified name. Figure 24 also presents a document example whose field structure corresponds to both graphical representations.

AStar graphical notation uses Embed and Reference links (cf. Figure 23) to specify relationships between DocumentTypes, either Fact to Fact, Fact to Dimension, or Dimension to Dimension. The difference between them is that an Embed represents a document within another document (i.e., embedded documents), while a Reference refers to a document referencing another document (i.e., referenced documents). We highlight that an Embed can only create a relationship between DocumentTypes in the same collection, while a Reference can connect DocumentTypes in the same or in different collections. The links have a name (i.e., Nm) that defines, in the documents, the field name that stores the embedded document in an Embed or the reference in a Reference. Link cardinality is represented as follows: the side with an arrow or diamond indicates the “one” cardinality, while the side with a simple tail indicates the “many” cardinality. The arrow

indicates the link navigability. Links with no arrow (i.e., Reference M:N) are bidirectional, while links with an arrow are unidirectional. Note that in Figure 23 Reference 1:1 has a small “x” indicating the direction that is not navigable.

Figure 25 – Using an Embed to embed a dimension into a fact.

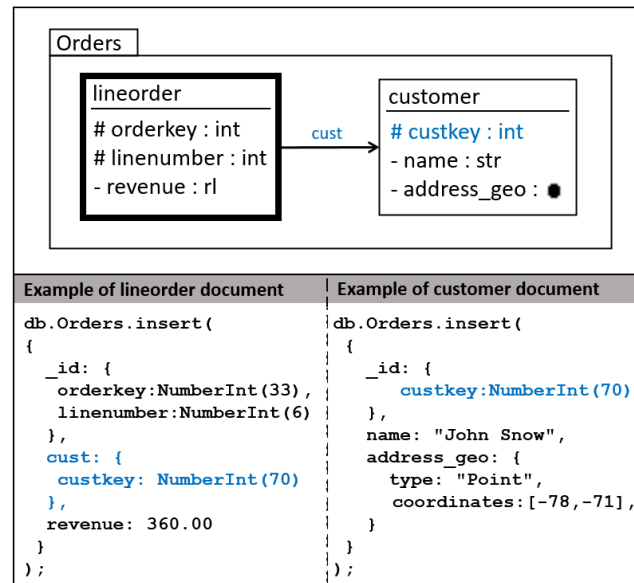


Source: The Author

Figure 25 illustrates how an Embed is used. It models parts of SSB’s lineorder fact table and customer dimension table, which are related as embedded documents. In other words, the DocumentType lineorder embeds the DocumentType customer. The Embed name (“cust”) defines the field name that embeds the document specified by the Embed. In turn, in the sample code, the fields of the customer document are embedded into the lineorder document (within “cust”), forming a single and uniform field structure. This means that, although the diagram presents two different DocumentTypes (i.e., lineorder and customer), they are embedded and, therefore, form a homogeneous collection.

Figure 26 shows the use of the Reference N:1 to create a relationship between the DocumentTypes from the previous example. Unlike an Embed, a Reference is mapped to a field storing a reference to another document. The name of this field is defined by the Reference name. For example, “cust” (Reference name) defines in the lineorder documents a field to store the reference for the customer documents. As can be seen in the sample code, the field “cust” of lineorder contains a field named “custkey”, which references the customer’s identifier field. This convention distinguishes references when

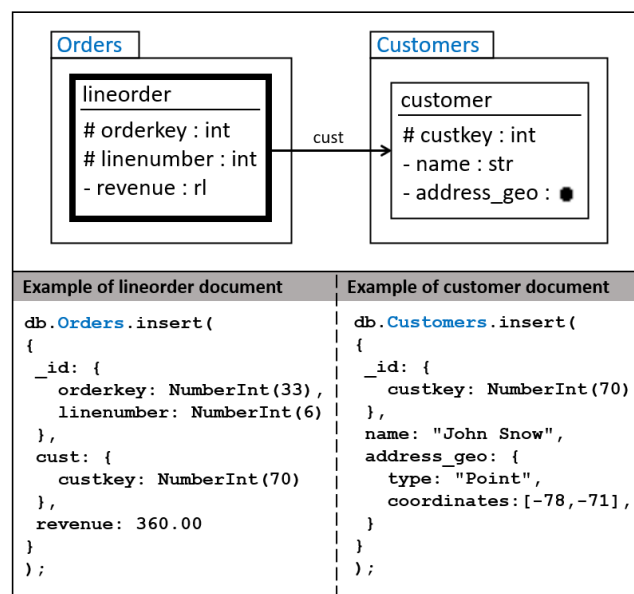
Figure 26 – Using a Reference N:1 to form a relationship between DocumentTypes from the same collection.



Source: The Author

there are multiple links between two DocumentTypes (this will be addressed in Figure 28). Observe in Figure 26 that, as the related DocumentTypes belong to the same collection, this collection has documents with two different field structures. Therefore, this example models Orders as a heterogeneous collection.

Figure 27 – Using a Reference N:1 to form a relationship between DocumentTypes from different collections.

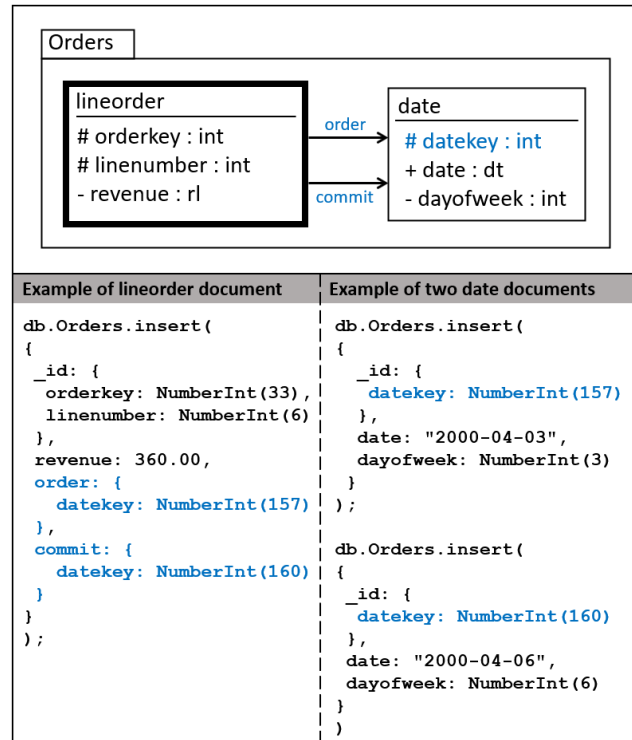


Source: The Author

Figure 27 shows the use of a Reference N:1 to relate DocumentTypes that belong to

different collections. Unlike Figure 26, Orders and Customers are homogeneous collections, because each has only one DocumentType.

Figure 28 – Modeling a role-play dimension.



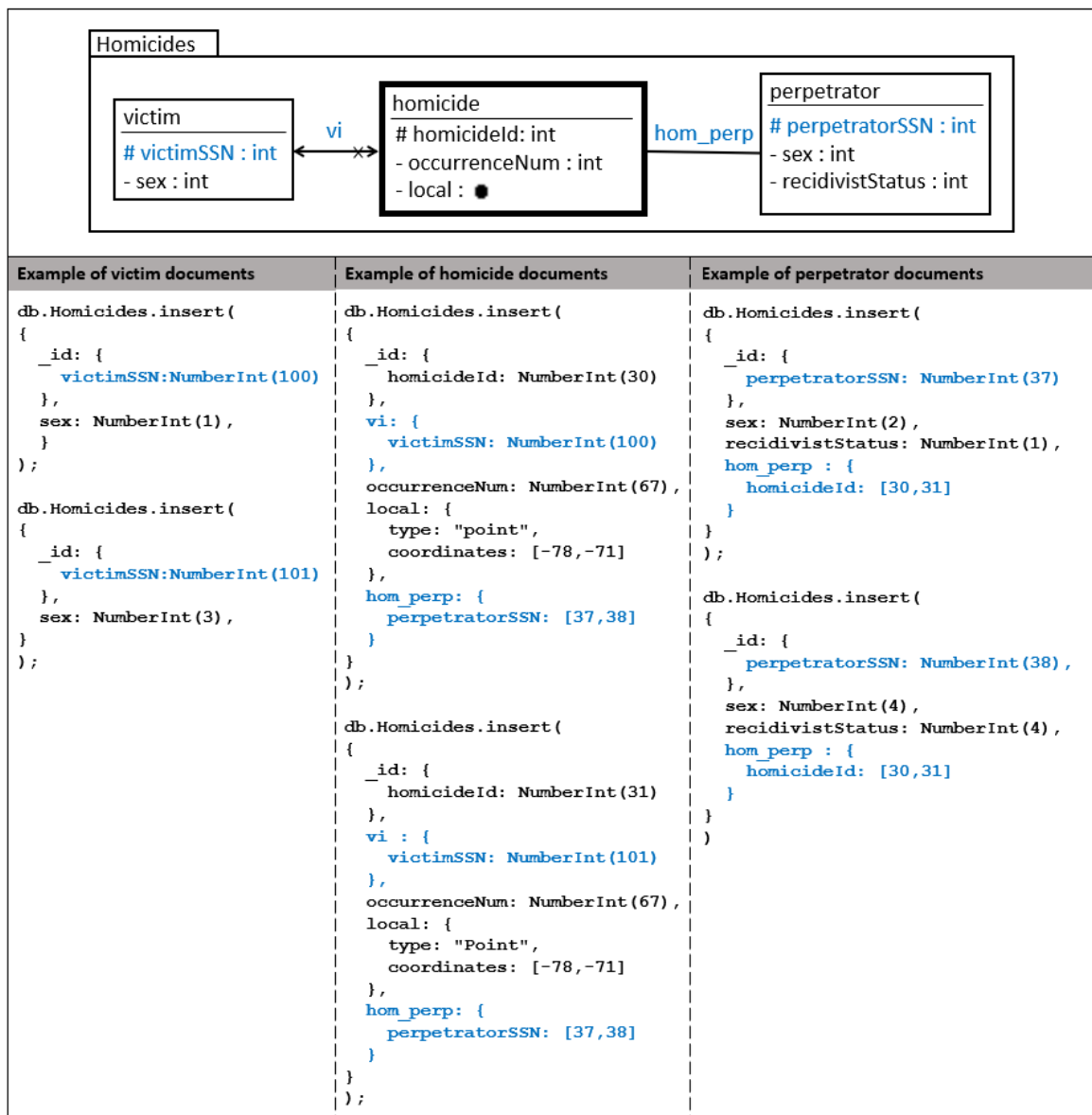
Source: The Author

It is possible to have more than one link between two DocumentTypes. This is used to create role-playing dimensions (KIMBALL; ROSS, 2013). An example is shown in Figure 28, where the DocumentType lineorder is doubly related to the DocumentType date. Note in the sample code that the fields “order” and “commit” correspond to the Reference names, distinguishing the references (“datekey”) for the order date and commit date.

Figure 29 shows part of a DGDW based on the ICCS. In this example, a Reference 1:1 relates the DocumentType homicide (fact) to the DocumentType victim (dimension). A Reference 1:1 also defines a field to store the reference, whose name uses the same convention as the Reference N:1. However, unlike a Reference N:1, this field is defined as unique. To respect navigability, the reference field is created in the documents whose DocumentType is on the “x” side of the link. Therefore, the field “vi” of a homicide document contains the reference field (“victimSSN”) for a victim document. This example also relates the DocumentType homicide to the DocumentType perpetrator (another dimension) using a Reference M:N. As this link is bidirectional, the field that stores the references is defined in both DocumentTypes (i.e., homicide and perpetrator). There-

fore, the field “hom\_perp” in the homicide documents contains the reference field for the perpetrator documents (“perpetratorSSN”), while “hom\_perp” in the perpetrator documents contains the reference field for the homicide documents (“homicideId”). Note in the sample code that a Reference 1:1 produces JSON code structures similar to a Reference N:1. A Reference M:N, however, uses arrays to store the references for the many-to-many relationship.

Figure 29 – Use of Reference 1:1 and Reference M:N.



Source: The Author

Finally, the AStar notation uses the pictograms defined by (CUZZOCREA; FIDALGO, 2012) to represent both conventional (Integer, String, Date, Real, Boolean) and geospatial (Point, Line, Polygon, MultiPoint, Multiline, MultiPolygon, Collection) fields. These

pictograms support the conventional and geospatial data types of the JSON and GeoJSON specification (BRAY, 2017; BUTLER et al., 2016). Note that there are no pictograms for multivalued fields (i.e., arrays). This is because arrays are rarely used in a GDW and searches for a specific value or group-by operations on these fields are difficult and slow. For these reasons, arrays are stipulated only for mapping references of Reference M:N.

## 4.2 ABSTRACT SYNTAX

This section presents the AStar abstract syntax, which consists of a metamodel whose metaclasses, attributes, relationships, and enumerations give meaning to the AStar graphical notation symbols. Subsection 4.2.1 details the metamodel definitions, while Subsection 4.2.2 addresses the associations types (e.g., containment, linkage) between nodes that are allowed by the metamodel.

### 4.2.1 Metamodel definition

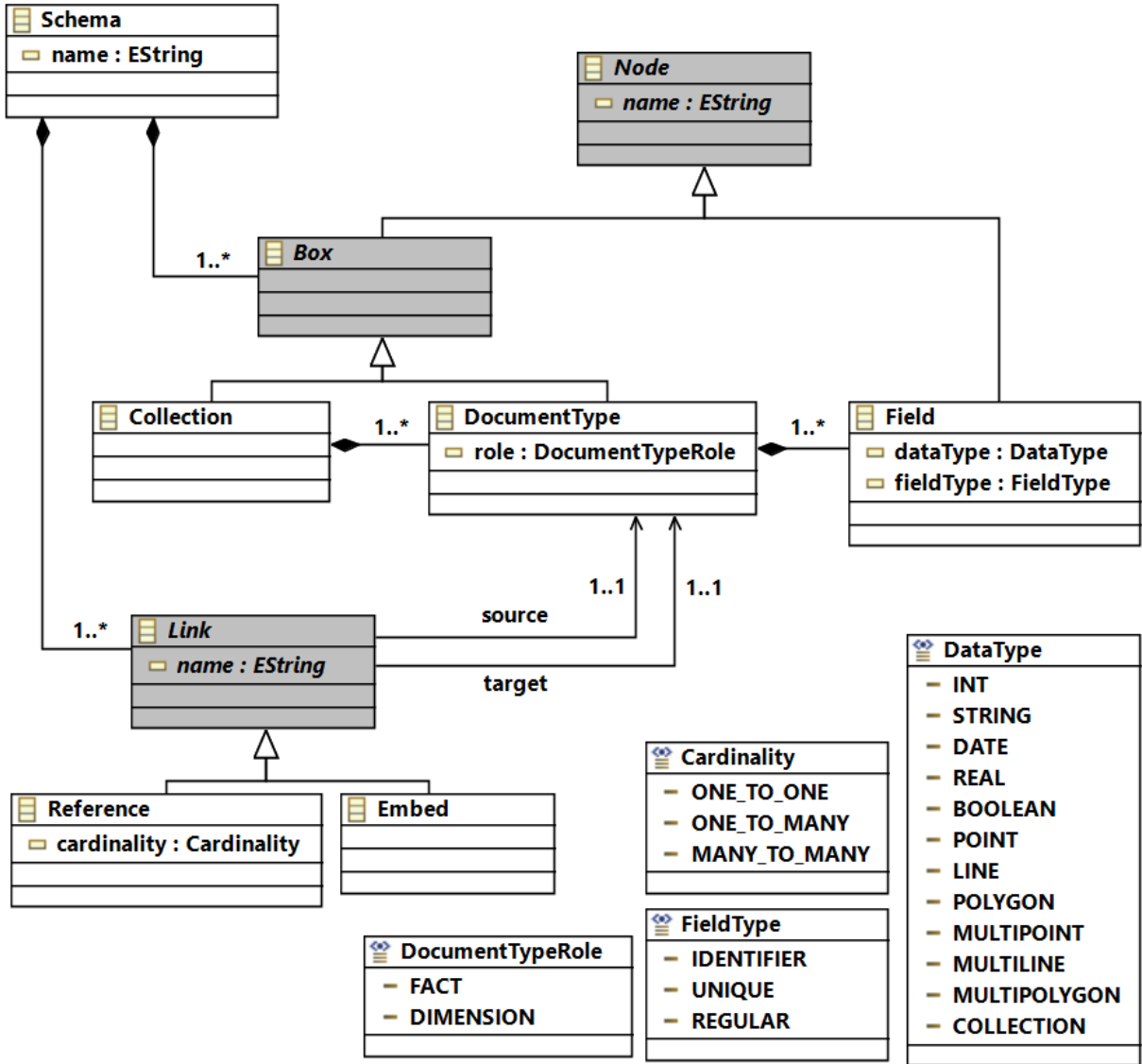
Figure 30 presents the AStar metamodel. The following paragraphs formalize each metamodel concept and correlate them to their graphical notation symbols (Figure 23).

The metamodel has four enumerations: *Cardinality*, *DocumentTypeRole*, *FieldType*, and *DataType*. These enumerations map the valid values for the attributes *cardinality*, *role*, *fieldType*, and *dataType* of the metaclasses *Reference*, *DocumentType*, and *Field*. Note that the *FieldType* and *DataType* items are shown by pictograms in the graphical notation. The geospatial data types are based on the *Simple Feature Access* (SFA) specification from the *Open Geospatial Consortium* (OGC) (OGC, 1999).

**Definition 1 - Enumerations.** Let *Cardinality*, *DocumentTypeRole*, *FieldType*, and *DataType* be enumerations, such that:  $Cardinality = \{ONE\_TO\_ONE, ONE\_TO\_MANY, MANY\_TO\_MANY\}$ ,  $DocumentTypeRole = \{FACT, DIMENSION\}$ ,  $FieldType = \{IDENTIFIER, UNIQUE, REGULAR\}$ , and  $DataType = \{INT, STRING, DATE, REAL, BOOLEAN, POINT, LINE, POLYGON, MULTIPOINT, MULTILINE, MULTIPOLYGON, COLLECTION\}$ .

Domains are used to state the primitive and symbolic data types of metaclass attributes. Definition 2 introduces these domains.

Figure 30 – AStar metamodel.



Source: The Author

**Definition 2 - Domain.** A domain is a set of values of type  $T$ . Primitive types include  $EString$  (the set of all strings) and  $EBoolean$  (the set formed by the values `True` and `False`). Symbolic types include the enumeration types `Cardinality`, `DocumentTypeRole`, `FieldType`, and `DataType`, as stated in Definition 1.

The metamodel has the abstract metaclasses `Node`, `Box`, and `Link`, which are highlighted in Figure 30 with a gray background. Note that `Box` is a specialization of `Node`. Definition 3 addresses them.

**Definition 3 - Abstract Metaclasses.** Let `Node` and `Link` be sets, such that  $(Node \cap Link = \emptyset) \wedge (Box \cap Link = \emptyset) \wedge (Node \cap Box = Box) \wedge (Node \cup Box = Node)$ .

Schema is the root metaclass of the metamodel that corresponds to the drawing area of a logical schema. A Schema must have at least one Box and at least one Link. Definition 4 formalizes the metaclass Schema.

**Definition 4 - Schema.** Let Schema be a set, such that:  $\forall s : \text{Schema}, sb : \text{Schema} \rightarrow \text{Box}$  and  $sl : \text{Schema} \rightarrow \text{Link} \therefore sb(s) \cap sl(s) = \emptyset \wedge \text{dom}(sb) \cup \text{dom}(sl) = \text{Schema}$ .

The abstract metaclasses are specialized in Collection, DocumentType, Field, Reference, and Embed, which implement the symbols with the same name. Node is specialized in Box and Field. The first is an abstract metaclass specialized in Collection and DocumentType, while the second is the metaclass that implements the symbol Field. Link is specialized in Embed and Reference. Note that, as a Schema must have one or more Box, a Schema can have both Collection and DocumentType. This enables defining into a Schema a collection with a package (Collection) or without a package, i.e., with a DocumentType whose name has a qualified name that corresponds to the collection name (cf. Figure 24). All inheritances are disjoint and complete. Note that a Collection must have at least one DocumentType (fact or dimension), and DocumentType must have at least one Field. Moreover, DocumentTypes can be created within Collections or Schema, and Fields within DocumentTypes. Although Embed and Reference represent relationships between DocumentTypes, they are different concepts and have distinct behavior. Embed and Reference represent an embedded and a referenced relationship, respectively. Definition 5 addresses the specialized metaclasses.

**Definition 5 - Specialized Metaclasses.** Let Collection, DocumentType, Field, Reference, and Embed be sets, such that:  $\text{Collection} \subseteq \text{Box} \wedge \text{DocumentType} \subseteq \text{Box} \wedge \text{Field} \subseteq \text{Node} \wedge (\text{Collection} \cap \text{DocumentType} \cap \text{Field} = \emptyset) \wedge (\text{Collection} \cap \text{DocumentType} = \emptyset) \wedge (\text{Collection} \cup \text{DocumentType} \cup \text{Field} = \text{Node}) \wedge (\text{Collection} \cup \text{DocumentType} = \text{Box}); \text{Embed} \subseteq \text{Link} \wedge \text{Reference} \subseteq \text{Link} \wedge (\text{Embed} \cap \text{Reference} = \emptyset) \wedge (\text{Embed} \cup \text{Reference} = \text{Link}); \forall cdt : \text{Collection} \rightarrow \text{DocumentType} \therefore \text{dom}(cdt) = \text{Collection}, \text{ and } \forall dtf : \text{DocumentType} \rightarrow \text{Field} \therefore \text{dom}(dtf) = \text{DocumentType}.$

In order to capture the facts or dimensions in a relationship, the associations named source and target between Link and DocumentType are defined. Definitions 6 and 7 specify these associations.

**Definition 6 - source.** Let  $source_L$  be an injective function from *Link* to *DocumentType* ( $source_L : Link \rightarrow DocumentType$ ).

**Definition 7 - target.** Let  $target_L$  be an injective function from *Link* to *DocumentType* ( $target_L : Link \rightarrow DocumentType$ ).

Functions formalize metaclass attributes to their domain. We start by specifying the attribute *name* of the metaclasses Schema, Node, and Link, which is used to name the instances of these metaclasses. Note that, as Node and Link are abstract metaclasses, their specializations inherit the attribute *name*. Definition 8 addresses the attribute *name* of the metaclass Schema and specializations of Node and Link.

**Definition 8 - name.** Let  $name_S$  be a function from the metaclass Schema to the domain EString ( $name_S : Schema \rightarrow EString$ );  $name_{NC}$  be a function from the metaclass Collection to the domain EString ( $name_{NC} : Collection \rightarrow EString$ );  $name_{ND}$  be a function from the metaclass DocumentType to the domain EString ( $name_{ND} : DocumentType \rightarrow EString$ );  $name_{NF}$  be a function from the metaclass Field to the domain EString ( $name_{NF} : Field \rightarrow EString$ );  $name_{LC}$  be a function from the metaclass Embed to the domain EString ( $name_{LC} : Embed \rightarrow EString$ ); and  $name_{LA}$  be a function from the metaclass Reference to the domain EString ( $name_{LA} : Reference \rightarrow EString$ ).

The metaclass DocumentType has an attribute named *role* that takes on one option from the enumeration DocumentTypeRole to set a DocumentType as either fact (i.e., a Fact DocumentType) or dimension (i.e., a Dimension DocumentType). Definition 9 addresses the attribute *role*.

**Definition 9 - role.** Let  $role_{DT}$  be a function from the metaclass DocumentType to the domain DocumentTypeRole ( $role_{DT} : DocumentType \rightarrow DocumentTypeRole$ ).

The metaclass Field has an attribute named *fieldType* that assumes one option from the enumeration FieldType to set the field to be identifier, unique, or regular. Definition 10 formalizes the attribute *fieldType*.

**Definition 10 - fieldType.** Let  $fieldType_F$  be a function from the metaclass Field to the domain FieldType ( $fieldType_F : Field \rightarrow FieldType$ ).

The metaclass Field has an attribute named *dataType* that assumes one option from

the enumeration `DataType` to set the field data type (conventional or geospatial). Definition 11 formalizes the attribute *dataType*.

**Definition 11 - dataType.** Let  $dataType_F$  be a function from the metaclass `Field` to the domain `DataType` ( $dataType_F : Field \rightarrow DataType$ ).

The metaclass `Reference` has an attribute named *cardinality* that assumes one option from the enumeration `Cardinality` to set the relationship cardinality. Definition 12 addresses the attribute *cardinality*.

**Definition 12 - cardinality.** Let  $cardinality_A$  be a function from the metaclass `Reference` to the domain `Cardinality` ( $cardinality_A : Reference \rightarrow Cardinality$ ).

#### 4.2.2 Nodes Associations

Relationships between the metaclasses of the AStar metamodel define the types of associations (cf. Figure 5) that can be established between the symbols of the AStar graphical notation. In short, (i) a composition (diamond association) between two metaclasses indicates a containment association, (ii) the double-relationship between `Link` and `DocumentTypes` indicates that two `DocumentTypes` can be related by a linkage association, and (iii) unrelated metaclasses imply that associations between their respective symbols are prohibited.

Table 3 should be read from left to right and shows the associations that are prohibited or allowed (i.e., Containment or Linkage) between AStar nodes. It is important to note that, because a collection can be graphically represented by a package or qualified name within the `DocumentType`, Table 3 distinguishes `DocumentTypes` with or without a qualified name. Note that AStar nodes do not use adjacent associations because they can limit the connectivity between facts and dimensions. For instance, an adjacent association would make it complex to associate a fact with many dimensions.

Table 3 – Associations that are allowed or prohibited between the graphical notation symbols in a logical schema. The numbers in the columns correspond to the enumerated items in the rows. C=Containment, L=Linkage, p=Prohibited.

	1	2	3	4	5	6	7
1 Logical Schema	p	C	C	C	C	C	p
2 Collection	p	p	C	C	C	C	p
3 Fact DocumentType with qualified name	p	p	L	L	L	L	C
4 Fact DocumentType without qualified name	p	p	L	L	L	L	C
5 Dimension DocumentType with qualified name	p	p	L	L	L	L	C
6 Dimension DocumentType without qualified name	p	p	L	L	L	L	C
7 Field	p	p	p	p	p	p	p

**Source:** The Author

### 4.3 STATIC SEMANTICS

Although a metamodel defines syntactically valid schemas, some constructions (e.g., associations between symbols) should be avoided because they are semantically wrong. The following paragraphs address these constructions, presenting well-formedness rules to avoid them. The well-formedness rules are presented in natural language, as the static semantics of a DSML can be implemented in a CASE tool using different languages (e.g., Java - cf. Listing 5.1).

As defined in the AStar metamodel, a logical schema must have at least one Box (Collection or DocumentType) and at least one Link (Embed or Reference). Although a Link is used to relate two Fact/Dimension DocumentTypes, this definition is not sufficient to ensure that a logical schema contains at least one fact and at least one dimension. Rule 1 addresses this.

**Rule 1** - A logical schema must contain at least one DocumentType set as Fact and one DocumentType set as Dimension.

All facts and dimensions of a logical schema must participate in at least one relationship. However, the AStar metamodel does not prevent the design of disconnected DocumentTypes. Rule 2 addresses this.

**Rule 2** - All DocumentTypes must be the *source* or *target* of at least one Reference

or Embed.

Embedded documents must belong to the same collection, but the AStar metamodel does not prevent using an Embed to relate two DocumentTypes that belong to different Collections. Rule 3 addresses this.

**Rule 3** - Embed should only establish relationships between Fact DocumentTypes or Dimension DocumentTypes that belong to the same Collection.

Embedded documents do not have identifier fields, as identifier fields already exist in the document in which they are embedded. Rule 4 addresses this.

**Rule 4** - Embedded DocumentTypes do not have identifier field.

A collection can be graphically represented by a Collection (composed by one or more DocumentType without a qualified name) or a DocumentType with a qualified name. Thus, a DocumentType inside a Collection does not have a qualified name because the Collection in which the DocumentType belongs is already defined. On the other hand, a DocumentType outside a Collection must have a qualified name to indicate to which collection it belongs, that can be a new collection or an existing collection. As the metamodel enables to create a DocumentType with a qualified name into a Collection (cf. rows 3 and 5 of Table 3) and a DocumentType without a qualified name into a logical schema (cf. rows 4 and 6 of Table 3), the rules 5 and 6 are defined.

**Rule 5** - A DocumentType outside a Collection must have a qualified name, which represents an existing collection or a new collection.

**Rule 6** - A DocumentType inside a Collection must not have a qualified name.

#### 4.4 TRANSLATIONAL SEMANTICS

The AStar concrete syntax has a set of human-readable symbols, whose meaning is defined in the AStar abstract syntax. To describe the semantics of these symbols, translational semantics is used, which will be addressed in this section. For this, the concrete syntax is mapped to the abstract syntax and its respective code to implement the schema

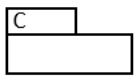

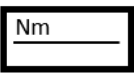

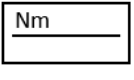

in the MongoDB database. This code consists of JSON Schemas and MongoDB code that creates collections and defines constraint functions. MongoDB was chosen because it is the most used document-oriented database in the industry (DB-ENGINES, 2020), and it is used in other examples of this work.

Figure 31 addresses the nodes of AStar. Because a collection can be graphically represented with a package or a qualified name within a `DocumentType`, these two representations have the same implementation code. The code for these two representations corresponds to the creation of a new collection in the database (A). Fact `DocumentType` and Dimension `DocumentType` correspond to JSON Schema code that defines the field structure for fact and dimension documents. In this JSON Schema, the values of the attributes “title” and “description” (B) are defined by the `DocumentType` name and `DocumentType` role (Fact or Dimension), respectively. Finally, a symbol Field corresponds to a new element in the attribute “required” of the JSON Schema, and its data type definition (C) in “properties”.

Figure 32 covers the field types. MongoDB uses a field named “\_id” to store the document identifier, which corresponds to a single value or set of embedded fields with their respective values (MONGODB, 2020). A field defined as an identifier in AStar corresponds to a field defined within “\_id” in the JSON Schema (A). As can be seen in the JSON Schema for the figure, the field named “Nm” is within “\_id” (B). This approach allows for the definition of more than one identifier field in a `DocumentType` (cf. Figure 25), as they correspond to elements in the attribute “required” (B) in “\_id”. A field defined as unique or regular corresponds to the same JSON Schema code (C). However, beyond the JSON Schema code, a unique field requires that a uniqueness constraint function (D) be applied to the collection.

Figure 33 covers the conventional and geospatial data types. As some primitive data types are supported in a JSON Schema, a field defined in AStar as int, string, real, and boolean will correspond to the JSON Schema data types int, string, number, and boolean, respectively (A). To implement the other data types of AStar, it is necessary to mix the native data types with regular expressions or data structures. Therefore, a field set as “date” in AStar corresponds to a definition in the JSON Schema of a string with a regular expression to guarantee that the date has a year, month, and day (B). For the

Figure 31 – AStar nodes and their implementation code in MongoDB.

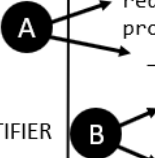

Concrete Syntax	Abstract Syntax	Implementation code
	Collection	 <code>db.createCollection( "C", {   validator: { \$jsonSchema:</code>
<code>[C::"]</code> <i>Qualified name within a DocumentType</i>		
	DocumentType role = FACT	<pre> {   title: "Nm",   description: "FACT",   bsonType:"object",   required:[&lt;&lt;fields&gt;&gt;],   properties:{     ...   } } </pre> 
	DocumentType role = DIMENSION	<pre> {   title: "Nm",   description: "DIMENSION",   bsonType:"object",   required:[&lt;&lt;fields&gt;&gt;],   properties:{     ...   } } </pre>
Nm	Field	<pre> ... required:["Nm",...], properties:{   Nm: {bsonType: &lt;&lt;DataType&gt;&gt;},   ... } </pre> 

Source: The Author

geospatial data types of AStar, a JSON Schema data structure must be defined to meet the specifications established by GeoJSON (BUTLER et al., 2016). The data structure for geospatial fields must ensure that geospatial data has a type (e.g., point, polygon) and coordinate pairs. Because the setup of the coordinate pairs (C) is different between the geospatial data types and demands many rows of code, the complete JSON Schema for each geospatial data type is presented in Appendix A.

Figure 34 shows the correlation between the link Embed and its corresponding JSON Schema. Because a link means that the DocumentType on the diamond side embeds the other DocumentType, the corresponding JSON schema has a field (A) for the Embed name, which contains the specifications for the data structure of the embedded DocumentType (B). It should be noted that, this link can only establish a relationship between

Figure 32 – Field types and their implementation code in MongoDB.

Concrete Syntax	Abstract Syntax	Implementation code
# nm	Field fieldType=IDENTIFIER	<pre> {   ...   required:["_id", ...],   properties:{     _id: {       bsonType: "object",       required: ["Nm"],       properties: {         Nm: {           bsonType: "&lt;&lt;DataType&gt;&gt;"         }       }     }   } } </pre> 
+ nm	Field fieldType=UNIQUE	<pre> required:["Nm",...], properties:{   Nm: {bsonType: &lt;&lt;DataType&gt;&gt;},   ... } </pre> <hr/> <pre> db.CollectionName.createIndex(   {"nm":1},{unique:true} ) </pre> 
- nm	Field fieldType=REGULAR	<pre> required:["Nm",...], properties:{   Nm: {bsonType: &lt;&lt;DataType&gt;&gt;},   ... } </pre>

Source: The Author

DocumentTypes that belong to the same collection, as detailed in section 4.1.

Figure 35 addresses the link Reference 1:1 and its corresponding JSON Schema. Unlike Embed, Reference (1:1, 1:N, or N:M) can establish relationships between DocumentTypes of either the same or of different collections. In other words, the JSON Schemas corresponding to the related DocumentTypes in Figure 35 (highlighted as A and B) can be defined into the same or into different collections. Note that the first JSON Schema (A) corresponds to the DocumentType which is on the “x” side of the Reference 1:1, which is the DocumentType that contains the reference for the relationship. The JSON Schema (A) has the field “Nm” (C), whose name matches the Reference name. This field contains a description of the cardinality<sup>2</sup> of the relationship (D), the referenced DocumentType (E), and the name and data type of the field that stores the reference for the other Do-

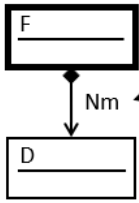
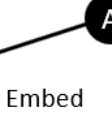
<sup>2</sup> “REF\_11” for a 1:1 relationship, “REF\_1N” for a 1:N relationship, and “REF\_NM” for a N:M relationship.

Figure 33 – Data Types and their implementation code in MongoDB.

Concrete Syntax	Abstract Syntax	Implementation code
nm : int	Field dataType=INT	<code>Nm:{bsonType: "int"},</code>
nm : str	Field dataType=STRING	<code>Nm:{bsonType: "string"},</code>
nm : rl	Field dataType=REAL	<code>Nm:{bsonType: "number"},</code>
nm : bl	Field dataType=BOOLEAN	<code>Nm:{bsonType: "boolean"},</code>
nm : dt	Field dataType=DATE	<code>Nm:{   bsonType: "string",   pattern: "^[1-9][0-9][0-9][0-9]-[0-1][0-9]-[0-3][0-9]\$", }</code>
nm : ●	Field dataType=POINT	<pre> Nm:{   bsonType: "object",   required: ["type", "coordinates"],   Properties: {     type: {       bsonType: "string",       enum: ["&lt;&lt;DataType&gt;&gt;"]     },     coordinates: {       bsonType: "array",       &lt;&lt;coordinates_setup&gt;&gt;     }   } } </pre> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">       Point        LineString        Polygon        MultiPoint        MultiLineString        MultiPolygon        GeometryCollection     </div>
nm : { }	Field dataType=LINE	
nm : ▭	Field dataType=POLYGON	
nm : ●●	Field dataType=MULTIPOINT	
nm : { }{ }	Field dataType=MULTILINE	
nm : ▭▭	Field dataType=MULTIPOLYGON	
nm : ▭▭▭	Field dataType=COLLECTION	

Source: The Author

Figure 34 – An Embed and its implementation code in MongoDB.

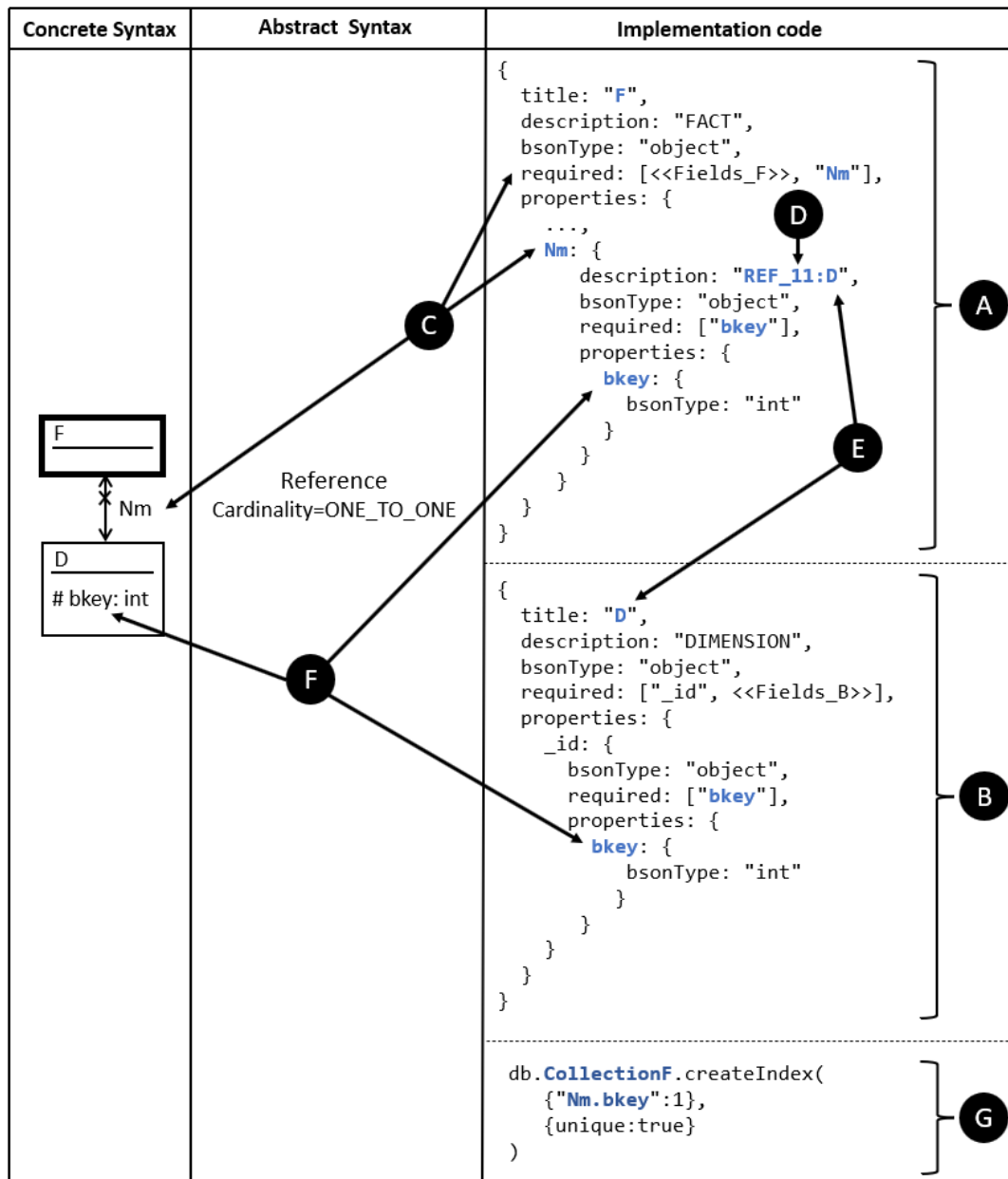
Concrete Syntax	Abstract Syntax	Implementation code
		<pre> {   title: "F",   description: "FACT",   bsonType: "object",   required: [&lt;&lt;Fields_F&gt;&gt;, "Nm"],   properties: {     ...,     Nm: {       title: "D",       description: "DIMENSION",       bsonType: "object",       required: [&lt;&lt;Fields_D&gt;&gt;],       properties: {         ...       }     }   } } </pre>

Source: The Author

cumentType (F). Note that in the figure, the name of the field that stores the reference is “bkey”, which is the name of the identifier field of the related DocumentType (B). As mentioned in section 4.1, in Reference 1:1, the reference is defined as unique, which

implies the definition of a uniqueness constraint for it (G).

Figure 35 – Reference 1:1 and its implementation code in MongoDB.

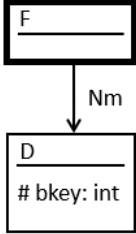


Source: The Author

Figure 36 addresses the link Reference 1:N, whose corresponding JSON Schema differs from Reference 1:1 in the way the reference field is defined, in addition to the description of the relationship. The description indicates a 1:N relationship (A) and there is no uniqueness constraint on the reference field (i.e., “bkey”).

Finally, Figure 37 shows the corresponding JSON Schema for a Reference N:M. In this link, both JSON Schemas corresponding to the related DocumentTypes have a field whose

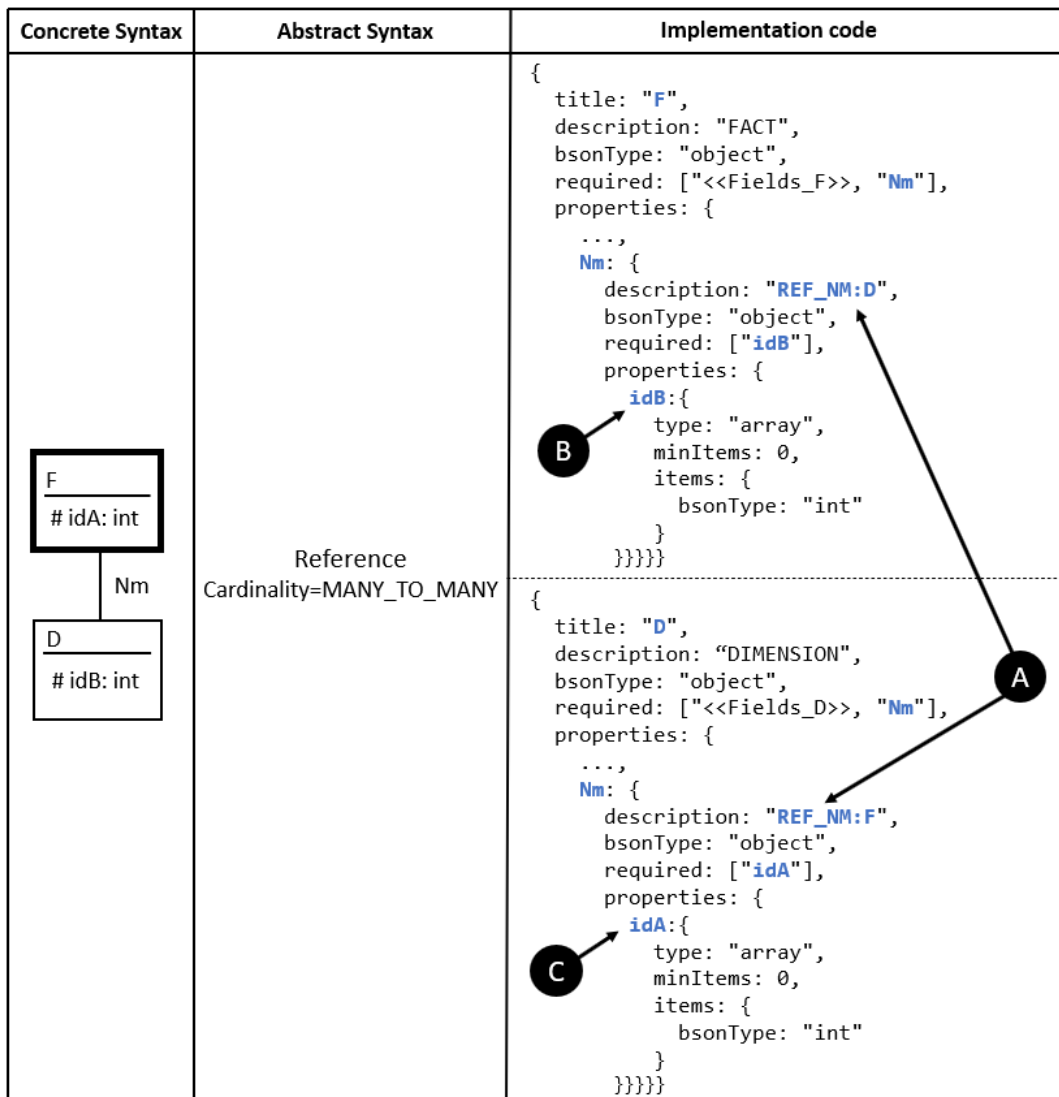
Figure 36 – Reference N:1 and its implementation code in MongoDB.

Concrete Syntax	Abstract Syntax	Implementation code
	<p>Reference</p> <p>Cardinality=ONE_TO_MANY</p>	<pre> {   title: "F",   description: "FACT",   bsonType: "object",   required: ["&lt;&lt;Fields_F&gt;&gt;", "Nm"],   properties: {     ...,     Nm: {       description: "REF_1N:D",       bsonType: "object",       required: ["bkey"],       properties: {         bkey: {           bsonType: "int"         }       }     }   } } </pre> <hr/> <pre> {   title: "D",   description: "DIMENSION",   bsonType: "object",   required: ["_id", "&lt;&lt;Fields_D&gt;&gt;"],   properties: {     _id: {       bsonType: "object",       required: ["bkey"],       properties: {         bkey: {           bsonType: "int"         }       }     }   } } </pre>

Source: The Author

name is the same as the Reference name. This field defines the relationship's cardinality in the attribute "description" (A), as well as the field that stores the references as an array (B and C).

Figure 37 – Reference N:M and its implementation code in MongoDB.



Source: The Author

## 4.5 CHAPTER FINAL CONSIDERATIONS

This chapter presented the graphical notation, metamodel, well-formedness rules, and translational semantics of AStar, a modeling language for the design of DGDW schemas. The AStar graphical notation is inspired by the UML class diagram notation, although the semantics of its symbols are defined in the AStar metamodel. The graphical notation is composed of nodes and links. Nodes consist of symbols that represent collections (homogeneous or heterogeneous), documents of facts or dimensions (i.e., DocumentType), and conventional or geospatial fields. Links consist of symbols that represent relationships using embedded documents that have 1:1 cardinality, or referenced documents that have

1:1, 1:N, or M:N cardinality. Well-formedness rules complement the metamodel, as they help avoid incorrect constructions that the metamodel does not prevent. Finally, translational semantics were presented to describe the meaning of the graphical notation symbols. The symbols are mapped to their corresponding concept in the metamodel and the respective code that implements the database schema in MongoDB.

## 5 EVALUATION AND IMPLEMENTATION OF ASTAR

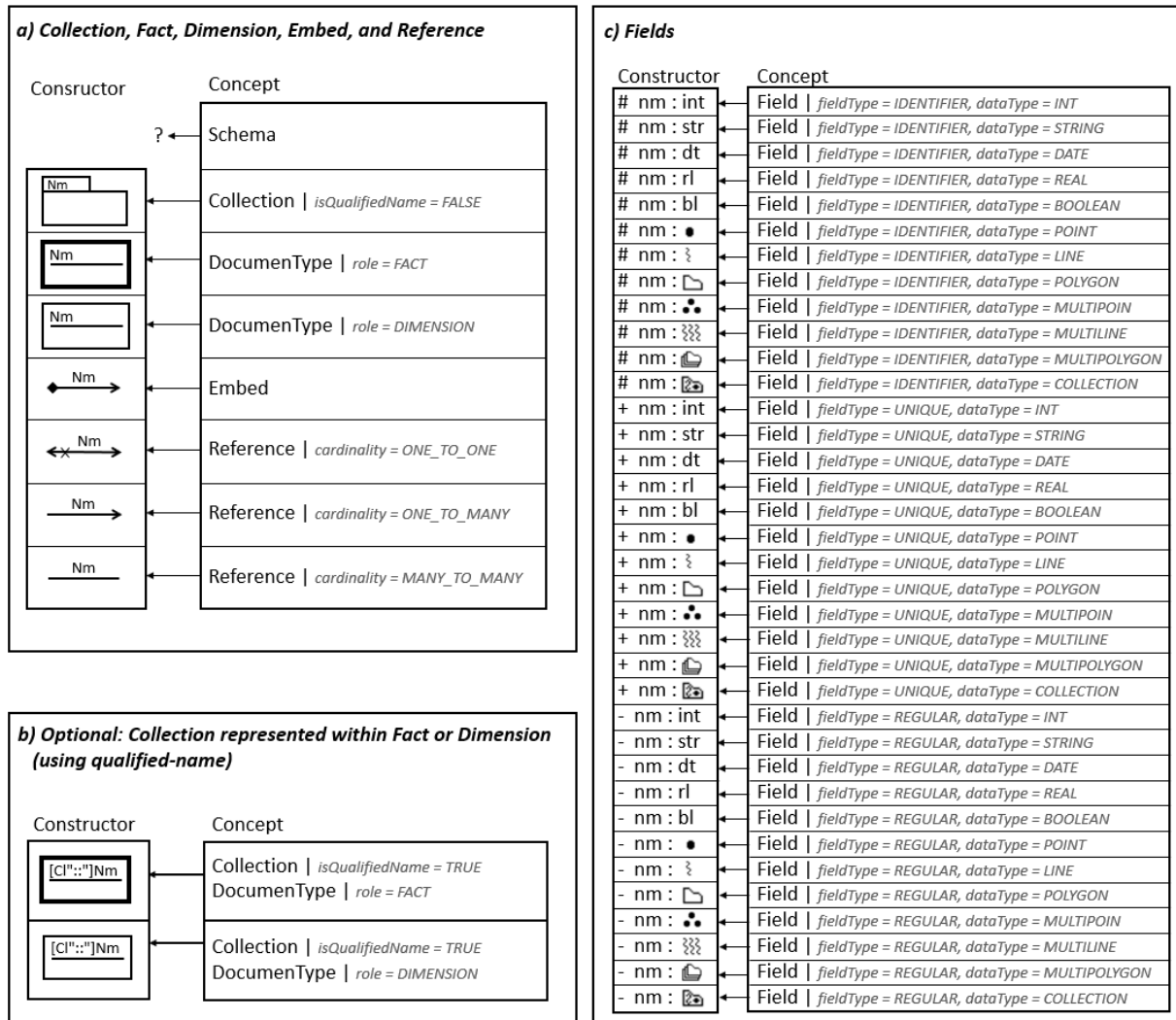
This chapter presents an evaluation of AStar. Section 5.1 discusses an evaluation of the AStar graphical notation, while section 5.2 shows AStarCASE, a prototype of CASE tool that implements the AStar metamodel and AStar well-formedness rules. Section 5.3 addresses the chapter final considerations.

### 5.1 ASTAR EVALUATION

This section presents an evaluation of AStar based on the principles of PoN. To do this, we analyzed how the principles were applied to the graphical notation of AStar in order to provide cognitive effectiveness. It is important to note that we do not evaluate with users, as the basic syntax of the UML class diagram is well known and accepted by both industrial and academic communities.

**Semiotic clarity** - Figure 38 illustrates all the symbols that can be depicted with AStar, correlating them with the respective concepts (i.e., metaclass of the AStar metamodel). Figure 38a shows the symbols Collection, DocumentType, Embed, and Reference; Figure 38b addresses the optional representation for Collection; and Figure 38c presents the symbol Field, with different field types (identifier, unique, or regular) and data types (conventional or geospatial). When analyzing Figure 38, it is possible to identify that AStar has two semiotic clarity anomalies. The first anomaly consists of a symbol deficit for the Schema concept, which does not have a corresponding symbol (Figure 38a). It is a symbol deficit that exists because a Schema instance represents the set of elements that make up a DGDW logical schema. In other words, in a CASE tool, a Schema instance is the drawing area where the designers create, customize, and establish relationships between the visual symbols. The second anomaly is symbol redundancy, as the concepts of Collection and DocumentType (Fact or Dimension) can be represented as illustrated in Figure 38a or Figure 38b. As shown in these figures, the designer can draw a Collection instance using a package or by inserting a qualified name within the DocumentType representation. This second anomaly is intentional and is addressed in the cognitive fit principle.

Figure 38 – Semiotic clarity of AStar.



Source: The Author

**Perceptual discriminability** - As shown in Figure 23, the symbols are grouped as nodes and links, and they are inspired by the symbols of the UML class diagram. The nodes graphically differ between themselves by the shape, as they are depicted as packages, classes, and class attributes. Furthermore, Fact DocumentType and Dimension DocumentType (i.e., a Document with role=FACT and role=Dimension, respectively) are distinguished graphically by the border thickness. The links are depicted by edges that differ at their ends: Embed has a diamond and an arrow, while Reference can have arrows and a “x” on one side to distinguish the 1:1, N:1, and M:N cardinalities. In short, the AStar symbols are distinguished by different visual variables.

**Semantic transparency** - As shown in Figure 23, the symbols of AStar graphical notation were arbitrarily defined by its authors. In this figure, it is possible to see that AStar

graphical notation is inspired by part of the UML class diagram notation. Because the meaning of AStar symbols was arbitrarily defined, there is an anomaly in their semantic transparency. In other words, the semantic transparency of AStar graphical notation is semantically opaque. On the one hand, this project decision contributes to increasing the number of tools that can be used to diagram DGDW schemas according to AStar, because of the expressive legacy of UML tools. On the other hand, this may cause confusion for users who are not familiar with AStar, but who know UML. To avoid this misinterpretation, novice users must have prior knowledge about the semantics of AStar symbols. Otherwise, the semantic transparency of AStar can be interpreted as semantically perverse.

**Complexity management** - AStar reduces the complexity of diagrams just like the UML class diagram does. Each collection (package) can be designed into distinct diagrams, modularizing a DGDW logical schema.

**Cognitive integration** - In this first version of AStar, there are no mechanisms to represent a cognitive integration between diagrams.

**Visual expressiveness** - The symbols use the visual variables of shape and orientation, as well as text (i.e., spatial-textual signals) to name the symbols. These variables are also used in the UML class diagram, from which the AStar graphical notation is inspired.

**Dual coding** - The nodes and links of the AStar graphical notation have dual encoding, as the text is used to name the symbols and reinforce their meaning.

**Graphic economy** - There are two strategies to reduce the number of graphical symbols. (i) Facts and dimensions are depicted as classes, but differ in the border thickness. In other words, the classes of a diagram represent a main concept (DocumentType), but the class border distinguishes the facts (DocumentType with role = FACT) from the dimensions (DocumentType with role = DIMENSION). (ii) As shown in Figure 38, 36 different configurations for fields can be depicted, but they make use of the same visual variables. The combination of two pictograms is what distinguishes the field type (identifier, unique, or regular) and data type (conventional or geospatial data types).

**Cognitive fit** - As addressed in the semiotic clarity principle, a Collection instance can be represented as a package or as a qualified name in the classes (DocumentType). For

a novice user, packages can be more intuitive, as they are easily perceived in a diagram. However, the use of a qualified name in the class eliminates the need to draw a package, making the diagram simpler for expert users. It is important to highlight that the symbols do not use complex, colored, or 3D shapes, reducing the difficulty for users with poor hand-drawing skills or misunderstood conditions. Furthermore, the AStar graphical notation symbols can be found in UML tools (e.g., Lucidchart<sup>1</sup>, VisualParadigm<sup>2</sup>, EdrawMax<sup>3</sup>). Therefore, users can use an implementation of AStar (e.g., AStarCASE) or UML tools to design DGDW logical schemas.

Table 4 summarizes AStar’s compliance with the PoN principles. Note that the cognitive integration and semantic transparency principles were not complied, as the integration of different diagrams is not supported, and anomalies in the semantic transparency exist. It is important to note that AStar has two anomalies with regard to semiotic clarity. However, as explained earlier, one anomaly (symbol deficit - the concept Schema does not have a corresponding symbol) does not impair cognitive effectiveness and the other anomaly (redundancy of symbols - the concept Collection can be represented using a package or qualified name) was introduced to comply with the principle of cognitive fit. In other words, there is a conflict between the principles of semiotic clarity and cognitive fit, which is common in modeling languages (MOODY, 2009).

Table 4 – AStar compliance with the PoN principles.

Principle name	Complies
Semiotic clarity	Yes
Perceptual discriminability	Yes
Semantic transparency	No
Complexity management	Yes
Cognitive integration	No
Visual expressiveness	Yes
Dual coding	Yes
Graphic economy	Yes
Cognitive fit	Yes

**Source:** The Author

<sup>1</sup> [www.lucidchart.com/](http://www.lucidchart.com/)

<sup>2</sup> [www.visual-paradigm.com](http://www.visual-paradigm.com)

<sup>3</sup> [www.edrawmax.com](http://www.edrawmax.com)

## 5.2 ASTARCASE

ASarCASE<sup>4</sup> is a prototype of CASE tool that implements AStar. The main features of this prototype are: (i) notification of syntactical errors (e.g., an unnamed DocumentType), (ii) prevention of semantic errors (e.g., relating two DocumentTypes that belongs to different Collections with an Embed), (iii) suggestion of actions that contribute to good DGDW performance (e.g., that the user analyzes the selectivity of the geospatial fields), and (iv) generation of JSON Schemas from a diagram.

AStarCASE is implemented with *Eclipse Modeling Framework* (EMF), which provides mechanisms to build a *Graphical Modeling Project* (GMP) (BUDINSKY, 2004). For this, the AStar metamodel is implemented with Emfatic language<sup>5</sup> (cf. Appendix C), which consists of a textual syntax for EMF metamodels. The Eugenia<sup>6</sup> framework is used to simplify development, as it can generate the configuration code required by EMF. The well-formedness rules are implemented with Java in the GMP, as shown in the Java code snippet in Listing 5.1. Finally, EVL is used to validate schemas and alert syntactic errors, as well as advise the user to take certain actions as shown in Listing 5.2.

Listing 5.1 – Java code snippet to prevent the relationship between DocumentTypes that belong to different collections with an Embed.

```

1  public class EmbedCreateCommand extends EditElementCommand {
3      ...
5      public boolean canExecute() {
7          // check that the source and target are set
9          if (getSource() == null && getTarget() == null) {
11             return false;
13         }
15         if (getSource() instanceof DocumentType == false) {
17             return false;
19         }
21         if (getTarget() instanceof DocumentType == false) {
23             return false;
25         }
27     }

```

<sup>4</sup> Available under a free license at <https://github.com/mrcferro/AStarCase-page>.

<sup>5</sup> <https://www.eclipse.org/emfatic>.

<sup>6</sup> <https://www.eclipse.org/epsilon/doc/eugenia/>.

```

19      // check that the DocumentType source and DocumentType target
      // belong to the same collection
21
22      if (getTarget().eContainer() != getSource().eContainer()) {
23          return false;
24      }
25
26      // can create the Embed...
27
28      return SchemaBaseItemSemanticEditPolicy.getLinkConstraints().
          canCreateEmbed_4002(getContainer(), getSource(), getTarget());
29  }
30
31  ...
32  }

```

Listing 5.2 – EVL code to advise the user to analyze the selectivity of a geospatial field.

```

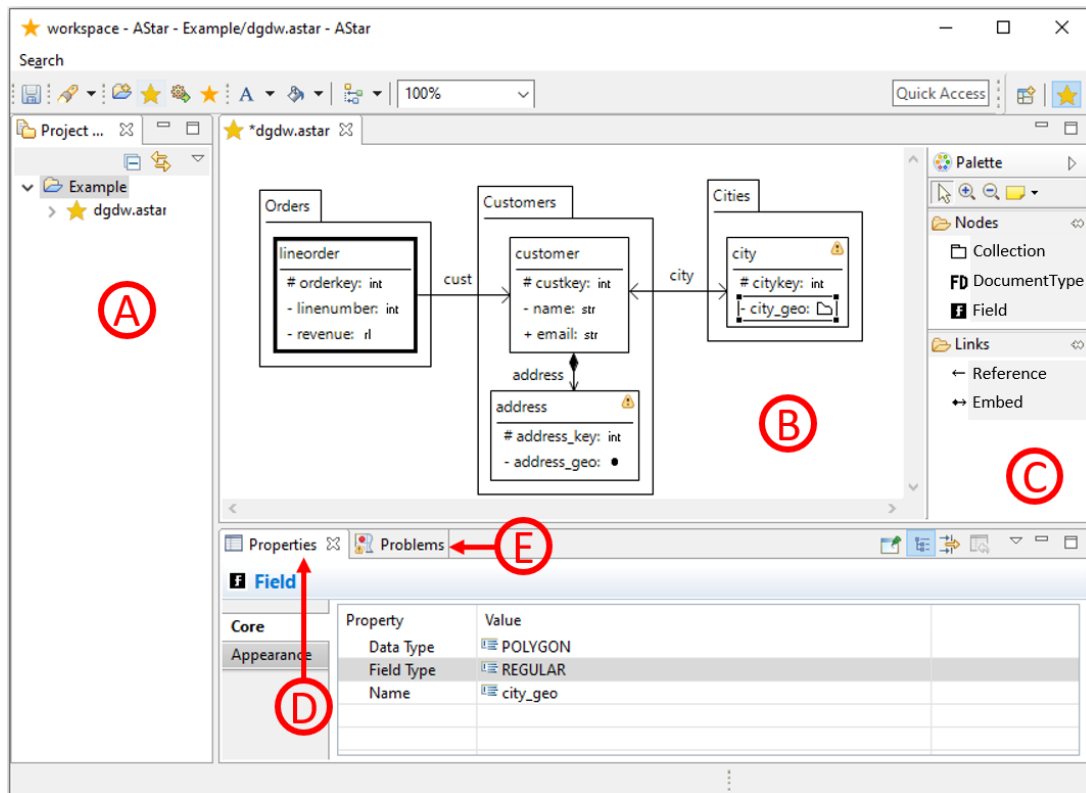
context DocumentType {
2    critique SelectivityGeoField {
        check {
4            for(f in self.fields) {
                if( f.dataType = DataType#POINT or f.dataType = DataType#LINE or f.
                    dataType = DataType#POLYGON or f.dataType = DataType#MULTIPOINT
6                    or f.dataType = DataType#MULTILINE or f.dataType = DataType#
                        MULTIPOLYGON or f.dataType = DataType#COLLECTION){
                            return false;
8                        }
                    }
10            return true;
        }
12        message: '(C001) Analyze the selectivity of the geospatial field.\n It is
            recommended that geospatial fields with high selectivity be denormalized
            , while geospatial fields with low selectivity be referenced.'
14    }
}

```

Figure 39 shows AStarCASE and highlights some important features. Area A presents the AStar projects with their DGDW logical schemas. Area B is the drawing area, where the designer creates, positions, and forms relationships between the symbols of a logical schema. Area C contains a palette of symbols with nodes and links. Note that this palette does not make distinctions between the different Fields (identifier, unique, and regular) or References (1:1, 1:N, and M:N), as these symbols are customized in the properties

bar, highlighted as area D. To change, for example, the cardinality of a Reference, it is unnecessary to remove the existing symbol and create another. This can be done simply by changing the symbol characteristics in the properties bar. Area E contains the problem bar, which displays error messages or warnings produced during the validation of the logical schema from the drawing area, performed when the user saves the diagram. Note in Figure 40 that pictograms are used for the symbols in the drawing area to facilitate identification of the one that produced the error or warning displayed in the problems bar.

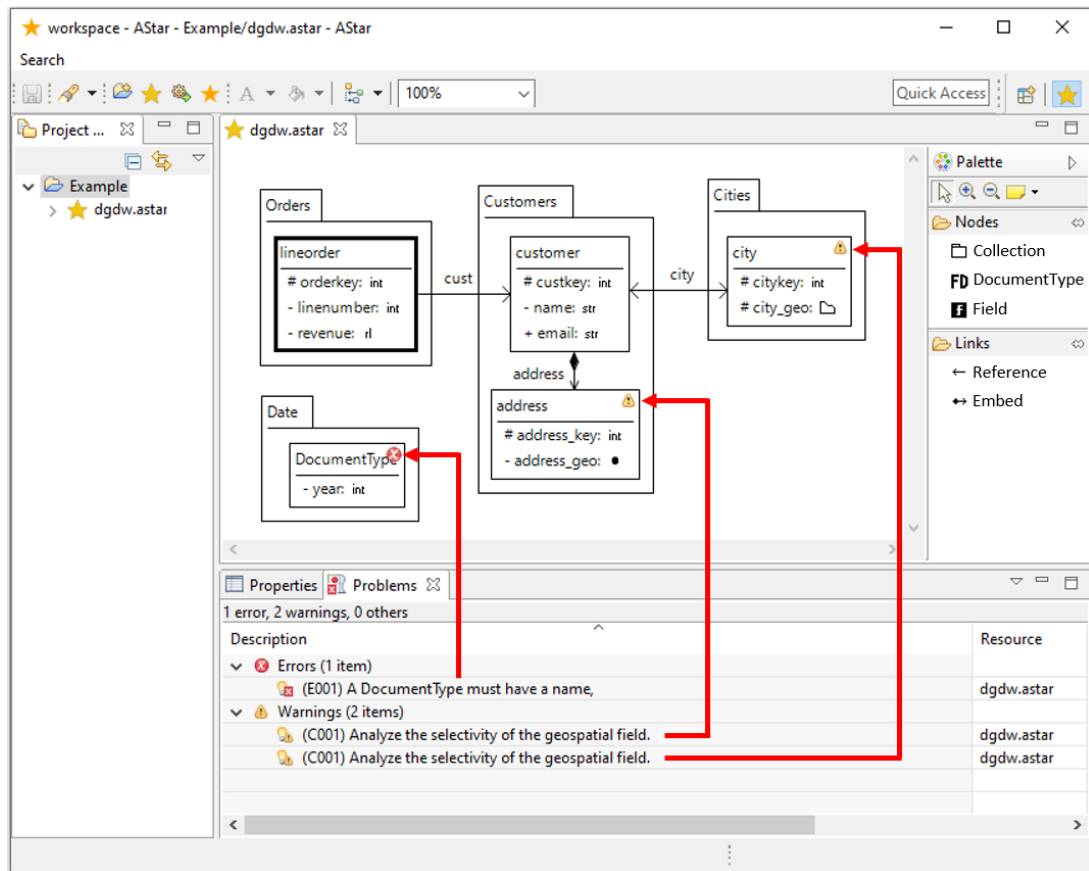
Figure 39 – AStarCASE overview.



Source: The Author

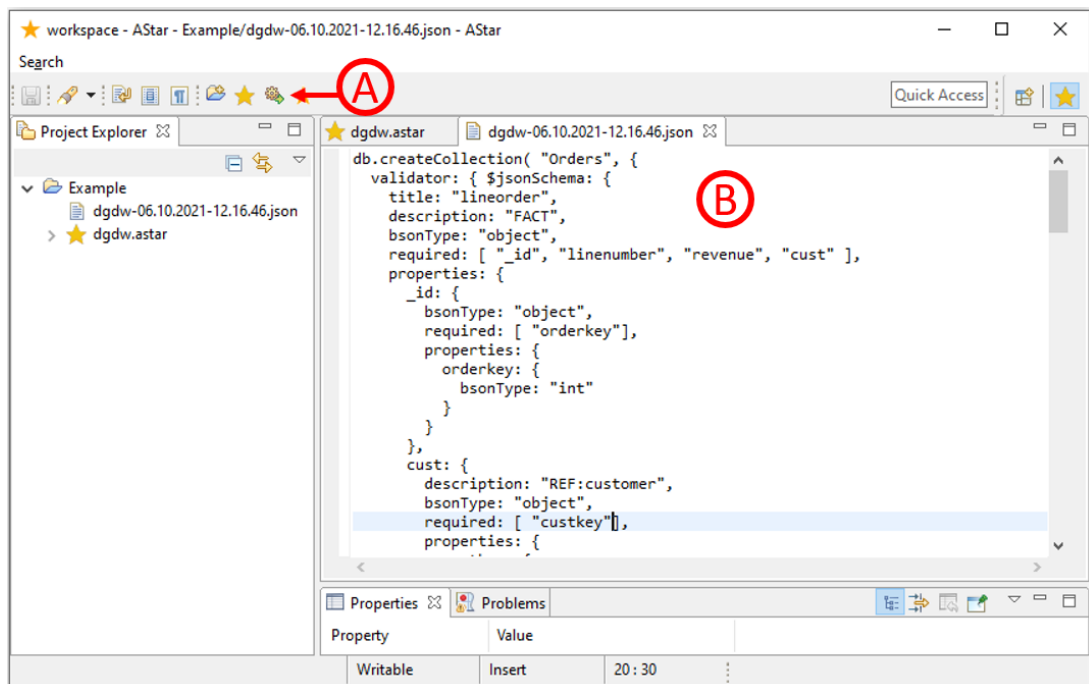
Figure 41 displays the *Model-to-Text* (M2T) transformation implemented in AStarCASE, which consists of the generation of the JSON Schema from a diagram. To run this feature, the user should click on the button shown as A, with the code generated appearing in area B. The JSON Schema of this figure can be seen in Appendix D.

Figure 40 – Errors and warnings issued by AStarCASE.



Source: The Author

Figure 41 – Generation of JSON Schema using AStarCASE.



Source: The Author

### 5.3 CHAPTER FINAL CONSIDERATIONS

This chapter presented the evaluation of the AStar graphical notation based on PoN principles. This evaluation showed that AStar is in accordance with seven of the nine PoN principles. Furthermore, this chapter also presented a prototype of CASE tool as a proof of concept for AStar, which implements AStar's metamodel and well-formedness rules. This tool, called AStarCASE, can be used to design DGDW schemas and generate the corresponding code in order to create the schema in MongoDB.

## 6 GUIDELINES TO DESIGN LOGICAL SCHEMAS WITH ASTAR

This chapter presents a guideline to help the design of DGDW logical schemas with AStar. Section 6.1 defines some models to depict facts and dimensions related as referenced or embedded documents, partitioned into one or more collections. Based on the experimental evaluation addressed in section 6.2, some practices to achieve low data volume and low query runtime in the DGDW are pointed out in section 6.3. The chapter final considerations are presented in section 6.4.

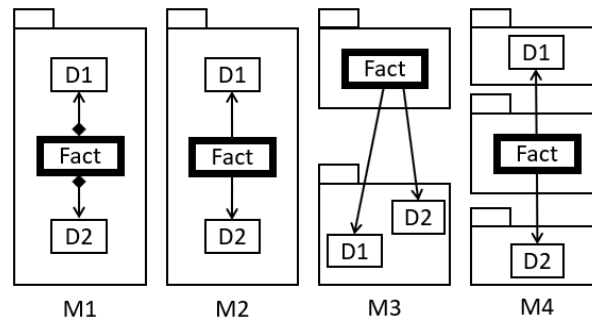
### 6.1 MODELING FACTS AND DIMENSIONS

This section presents some models to depict facts and dimensions in a schema with AStar. Because these models are design abstractions, they omit the collection name and fields. Furthermore, these models use a Reference N:1 to represent referenced documents.

Figure 42 introduces four models for partitioning DocumentTypes among collections, either as referenced or embedded documents. In model M1, there is a single collection of documents. All dimensions (D1 and D2 represent Dimension DocumentTypes) are embedded in a Fact DocumentType, producing data redundancy. Model M2 also has a single collection of documents. However, in this collection, facts and dimensions are normalized into distinct documents. Model M3 also normalizes facts and dimensions, but this model partitions these documents into two collections: one to store the facts, and the other to store the dimensions. Model M4 differs from the M3 only in the number of collections, because there is one collection for each dimension.

Figure 43 shows three ways to model the relationships between dimensions that contain conventional or geospatial data, represented as a Conventional Dimension DocumentType (CD) and a Geospatial Dimension DocumentType (GD), respectively. Model A represents a GD embedded/denormalized into a CD, that is, if a GD needs to be used by more than one CD, it must be replicated for each CD. Model B shows a referenced relationship between CD and GD. In this model, if a GD needs to be used by more than one CD, there will be no redundancy. Model C is a hybrid representation of the two previous

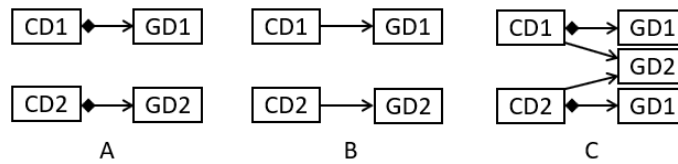
Figure 42 – Relationships between facts and dimensions.



Source: The Author

models. In this model, a geospatial field with low selectivity (e.g., *'nation':aNation*) is mapped to a GD and normalized/referenced by a CD, whereas a geospatial field with high selectivity (*'address':anAddress*) is mapped to a GD and denormalized/embedded into a CD.

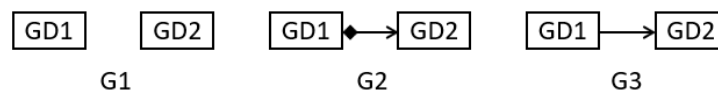
Figure 43 – Relationships between CD and GD.



Source: The Author

Figure 44 presents three ways of representing the relationships between GDs and themselves. In model G1, there is no relationship between GDs, because they are related only to the CD. In model G2, the GD is embedded/denormalized into another GD, so that the GD with higher selectivity embeds/contains the GD with lower selectivity (e.g., *'city': aCity*  $\supset$  *'nation': aNation*  $\supset$  *'region': aRegion*). Model G3 normalizes the GD into separate and referenced documents.

Figure 44 – Relationships between GDs.



Source: The Author

## 6.2 EXPERIMENTAL EVALUATION

This experimental evaluation aims to apply the models presented in the previous section in practice, and analyze the performance impact on the DGDW when modeling facts and dimensions that are related as embedded or referenced documents, and are partitioned into one or more collections. To this, the models presented in section 6.1 were combined to transform one RGDW in 36 DGDWs. The data volume and query runtime on these DGDWs are compared, in order to highlight the constructions that provide low storage cost and good query runtime.

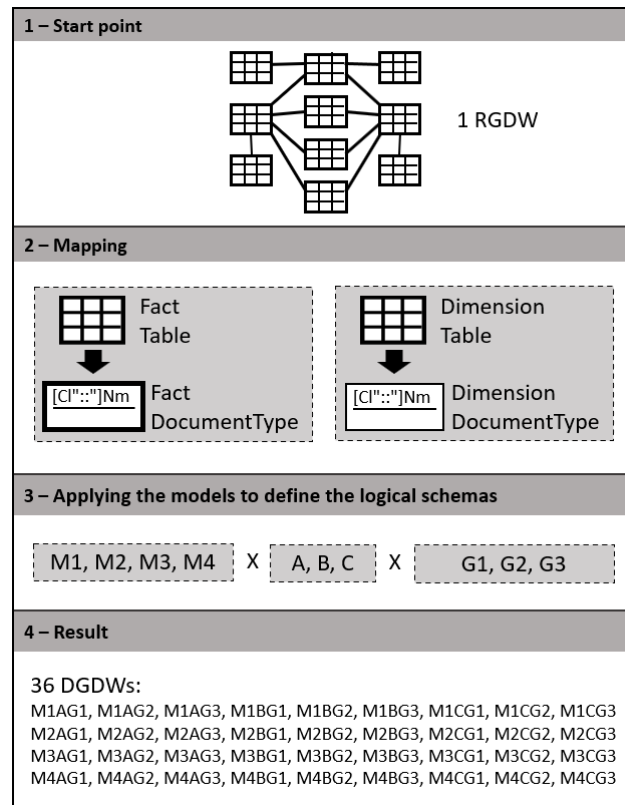
Figure 45 illustrates the steps used to generate the 36 DGDW logical schemas. In step 1, the RGDW of Figure 46 (SIQUEIRA et al., 2009) was adopted as a starting point. This RGDW is composed of one fact table (*lineorder*), four dimensions tables containing conventional data (*part*, *supplier*, *date*, and *customer*), and five dimensions tables containing geospatial data (*s\_address*, *c\_address*, *city*, *nation*, and *region*). In step 2, the fact and dimension tables were transformed into Fact and Dimension DocumentTypes (i.e., CDs and GDs). In step 3, collections were created to support the partitioning of documents as defined in the models shown in Figure 42, and the DocumentTypes were related as defined in Figure 43 and Figure 44. The combination of the models in step 3 results in 36 DGDW logical schemas, which is the result shown in step 4.

Figure 47 presents the 36 logical schemas, whose names are defined by the concatenation of the model names from Figure 42, Figure 43, and Figure 44 (e.g., M1+A+G1 = M1AG1). Due to the large number of schemas, Figure 47 omits the collection names, fields, and the dimensions *part* and *date*. This figure focuses on (i) the partitioning of DocumentTypes among collections, (ii) the use of referenced or embedded documents to relate the DocumentTypes, and (iii) the relationships between facts, CDs, and GDs<sup>1</sup>.

Figure 48 illustrates the process of building the 36 DGDWs. First, the data for the RGDW in Figure 46 were generated using scale factor 1 (sf=1). The data were stored on a single machine in a PostgreSQL 10.3 database with PostGIS 2.4.3 extension. Table 5 shows the number of records, number of columns, and table size, which corresponds

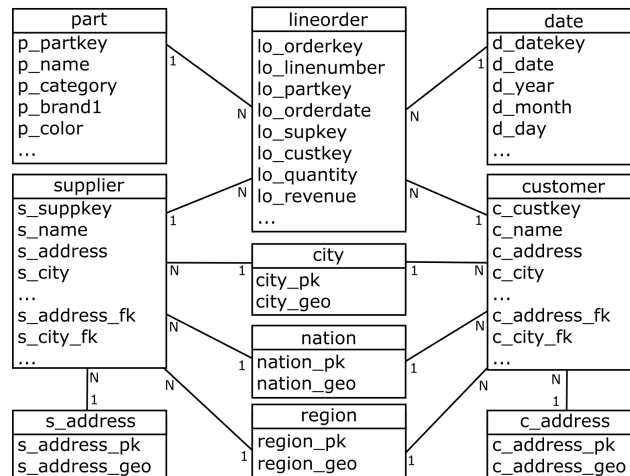
<sup>1</sup> In order to shrink the size of the Figure 47, the dimensions *s\_address* and *c\_address* were called A, *city* C, *nation* N, and *region* R.

Figure 45 – Steps performed to generate the DGDW logical schemas.



Source: The Author

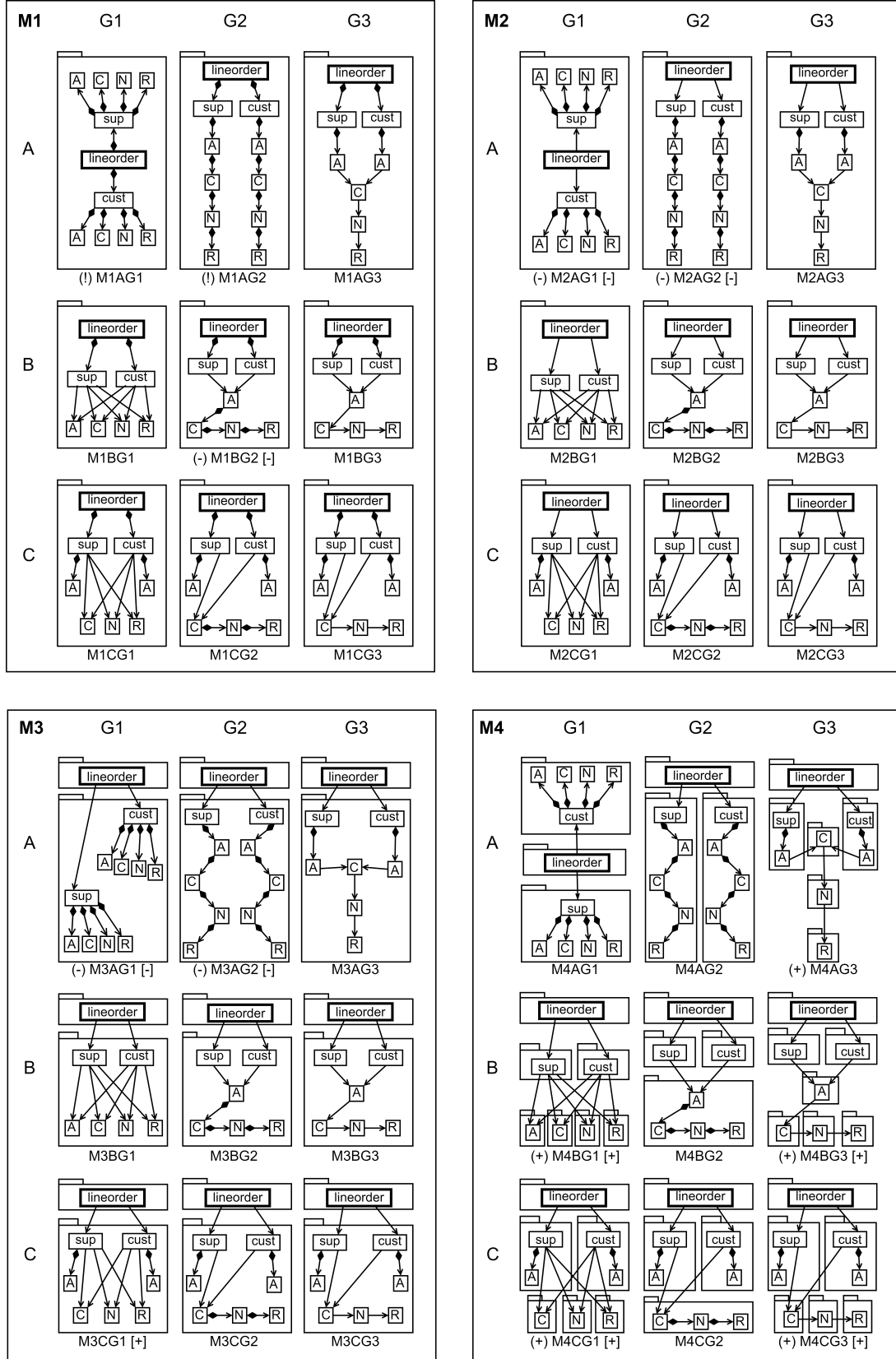
Figure 46 – RGDW schema used to generate the DGDW schemas.



Source: The Author

to a total of 1.03 GB. Using Pentaho Data Integration 8.0, the data were transformed as defined in each DGDW logical schema from Figure 47. They were stored in a cluster composed of a central node (Intel Xeon with 1 TB HD, 8 GB RAM, and a 1Gbit/s network) and three other nodes (Core i3 processors, each with 500 GB HD, 4 GB RAM, and a 1Gbit/s network each). The computers used the CentOS 7.1 operating system

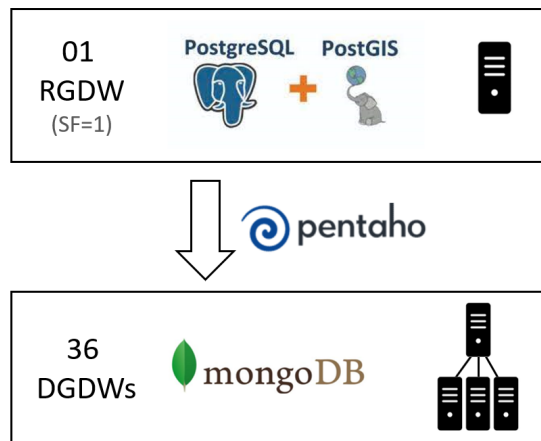
Figure 47 – DGDW logical schemas depicted with AStar. The symbols (!), (+), (-), [+], and [-] are addressed in the experimental results.



Source: The Author

and the MongoDB database, version 3.6.4. MongoDB was configured in sharding mode, which provides better performance for read and write operations, as it distributes the data between the nodes and increases the overall processing and storing capacity (MONGODB, 2021a). In other words, this cluster favors horizontally scalability, the reason why document-oriented databases have been used as an alternative to relational databases. It is important to highlight that, due to a storage space limitation in the experimental environment, it was not possible to use data generated with scale factors greater than 1 (e.g.,  $sf=10$  or  $sf=100$ ). This is because DGDWs containing denormalized geospatial data would require a larger data volume than the cluster could support. Furthermore, indexes were created for only the fields used to reference documents, making the performance comparison between referenced and embedded documents fairer.

Figure 48 – Process to build the 36 DGDWs from one RGDW.



Source: The Author

Given a lack of benchmarks for DGDW performance analysis, used in both industry and academics, six queries<sup>2</sup> (i.e., Q1, Q2, Q3, Q4, Q5, and Q6) were defined to evaluate the 36 DGDWs. These queries simulate frequently used SOLAP operations (drill down, roll up, slice, and dice) and aggregate data from a set of customers whose addresses lie within an area (circle or polygon) determined by geospatial coordinates. Queries Q1 and Q2 evaluate geospatial selection and conventional grouping, with Q2 extending this evaluation because it includes a conventional selection. Queries Q3, Q4, and Q5 evaluate two levels of conventional groupings and explore geospatial selection from incrementally larger areas. Finally, Q6 differs from Q5 because it performs an *intersect* operation instead

<sup>2</sup> As each query must be adapted to the schemas, they would require many pages to be presented in this document. Therefore, these queries, written in *MongoDB Query Language* (MQL), are available at GitHub (<https://github.com/mrcferro/gdw>).

Table 5 – RGDW used in numbers.

Table	Records	Columns	Size in MB
lineorder	6,001,171	17	981.0
customer	30,000	12	27.0
date	2,556	17	0.3
part	200,000	9	27.0
supplier	2,000	11	1.4
c_address	30,000	2	2.8
s_address	2,000	2	0.2
city	250	2	7.7
nation	25	2	5.7
region	5	2	2.3

**Source:** The Author

of a *within* operation. Q6, therefore, selects a greater quantity of facts from the DGDW, being more complex than the previous queries. Although the schemas shown in Figure 47 have two conventional dimensions (i.e., CD *sup* and CD *cust*), it was decided to perform all queries on CD *cust*, because it contains more documents than CD *sup* ( $|cust| = 30,000 > |sup| = 2,000$ ), making the analysis more rigorous. The query statements are as follows:

- Q1** How many customers made purchases and have an address within 10 miles of a given geospatial point<sup>3</sup>, grouped by year?
- Q2** How many customers bought a product from a specific category and have an address within a 10-mile radius of a given geospatial point, grouped by year?
- Q3** What is the sum of revenues from customers whose address lies within a polygon corresponding to a given city, grouped by year and brand of products?
- Q4** What is the sum of revenues from customers whose address lies within a geospatial polygon corresponding to a given nation, grouped by year and brand of products?
- Q5** What is the sum of revenues from customers whose address lies within a polygon corresponding to a given region, grouped by year and brand of products?

<sup>3</sup> Random geospatial data were chosen in the queries. Q1 and Q2 used a point whose coordinates are  $[-87.42, 41.24]$ , Q3 used the polygon from the city with primary key 20, Q4 used the polygon from the nation with primary key 2, Q5 used the polygon from the region with primary key 2, and Q6 used the polygon from the region with primary key 4.

**Q6** What is the sum of revenues from customers whose address intersects a polygon corresponding to a given region, grouped by year and brand of products?

The experimental evaluation investigated each schema shown in Figure 47, analyzing the data volume and the arithmetic mean of five execution times for each query. We chose to run each query only five times because some schemas required much time (e.g., more than two hours) to run some queries. Furthermore, we took special care to control the experiments. Each computer node was used exclusively for the proposed assessment and no other background processes were allowed to run while executing the queries. We rebooted all machines before executing the queries for each schema, cleaning any cached data used by the previous query.

### 6.2.1 Experimental Results

During the transformation from the RGDW to the 36 DGDWs, we noted that the M1AG1 and M1AG2 schemas would each need approximately 16 TB of storage. These schemas were therefore discarded from the experimental evaluation, because their data volume was much larger than that supported by the test environment ( $\approx 2.5$  TB). This high amount of storage was a consequence of considerable data redundancy, which follows the successive nesting of documents (i.e.,  $Fact \supset CD \supset GD$ ).

Table 6 presents the results of the experimental evaluation, showing the following columns: *Schema*, the name of the schema; *Size*, the disk space required to store each schema (in GB); *Q1* to *Q6*, the average of the five execution times for each query in seconds, followed by its respective standard deviation;  $Avg(Q_{1-6})$ , the average of the execution times for the six queries, also in seconds. The table is sorted in ascending order on column  $Avg(Q_{1-6})$ .

The schemas were evaluated based on data volume (*Size* column), and the average runtime of the 6 queries ( $Avg(Q_{1-6})$  column). Figure 49 depicts the volume vs. runtime relationship, showing each schema by its identifier (M1, M2, M3, and M4). Among the 34 schemas in Figure 49, 18 have very similar results, with volume and runtime below 3 GB and 400 s, respectively. Figure 50 shows these 18 results at a larger scale.

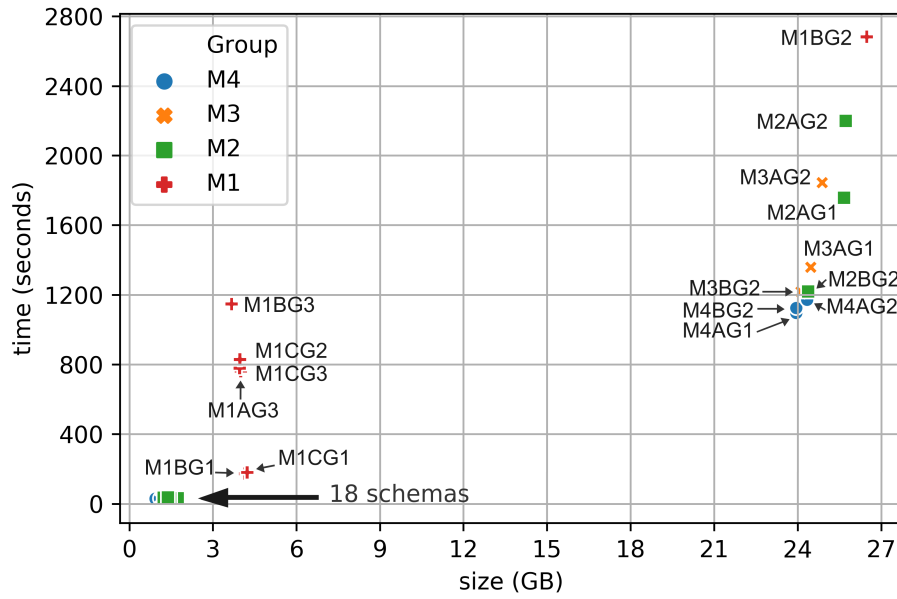
Table 6 – Results of the experimental evaluation. Size in GB and query execution time in seconds. Symbols used to highlight the DGDW schemas: (+) Best results in volume size; (-) Worst results in volume size; [+] Best results in query performance; [-] Worst results in query performance; (!) Schemas discarded from experimental evaluation.

Schema	Size	Q1	Q2	Q3	Q4	Q5	Q6	$Avg(Q_{1-6})$
M4CG1	0.95(+)	0.59(0.00)	0.59(0.00)	1.33(0.00)	12.87(0.02)	61.12(0.07)	61.00(0.08)	22.92[+]
M4CG3	0.94(+)	0.57(0.00)	0.59(0.00)	1.33(0.00)	12.94(0.03)	61.76(0.08)	61.84(0.12)	23.17[+]
M4BG1	0.95(+)	0.58(0.01)	0.60(0.02)	1.36(0.02)	13.06(0.05)	62.26(0.16)	62.19(0.15)	23.34[+]
M4BG3	0.96(+)	0.59(0.01)	0.60(0.01)	1.36(0.02)	13.18(0.03)	62.49(0.16)	62.64(0.19)	23.48[+]
M3CG1	1.28	0.68(0.00)	0.69(0.00)	1.48(0.00)	13.67(0.03)	64.85(0.20)	65.01(0.14)	24.40[+]
M3CG3	1.15	0.67(0.01)	0.69(0.00)	1.47(0.00)	13.68(0.05)	65.19(0.14)	65.19(0.10)	24.48
M3BG3	1.20	0.69(0.00)	0.71(0.01)	1.51(0.03)	13.87(0.04)	65.28(0.12)	65.50(0.07)	24.59
M3BG1	1.14	0.70(0.00)	0.73(0.03)	1.53(0.02)	14.03(0.01)	65.57(0.21)	65.81(0.13)	24.73
M4AG3	0.94(+)	0.70(0.01)	0.65(0.03)	1.59(0.04)	15.64(0.03)	75.26(0.16)	75.51(0.08)	28.22
M3AG3	1.19	0.77(0.01)	0.72(0.03)	1.66(0.01)	15.71(0.07)	75.34(0.23)	75.77(0.43)	28.33
M2CG1	1.49	5.32(0.11)	5.33(0.21)	6.20(0.10)	18.92(0.09)	71.80(0.29)	71.53(0.13)	29.85
M2CG3	1.74	5.48(0.13)	5.59(0.12)	6.43(0.06)	19.22(0.33)	72.62(0.37)	70.41(0.37)	29.96
M4CG2	1.10	0.59(0.00)	0.61(0.02)	1.38(0.02)	13.72(0.01)	73.76(0.12)	90.78(0.14)	30.14
M3CG2	1.18	0.98(0.01)	0.95(0.04)	1.70(0.07)	13.65(0.04)	73.45(0.20)	90.56(0.11)	30.22
M2BG3	1.32	5.01(0.25)	5.24(0.20)	6.02(0.22)	19.04(0.40)	73.70(0.87)	73.18(1.10)	30.37
M2BG1	1.46	5.30(0.11)	5.44(0.05)	6.20(0.11)	19.14(0.14)	74.10(0.27)	73.95(0.19)	30.69
M2AG3	1.23	5.33(0.08)	5.51(0.22)	6.29(0.19)	20.91(0.35)	82.91(0.22)	83.24(0.25)	34.03
M2CG2	1.38	5.38(0.11)	5.72(0.13)	6.51(0.17)	18.73(0.06)	80.87(0.93)	98.46(0.26)	35.94
M1BG1	4.15	74.63(0.83)	56.99(0.95)	95.72(3.24)	391.71(6.75)	311.12(1.94)	102.51(1.64)	172.11
M1CG1	4.23	60.55(1.03)	59.72(0.93)	59.04(1.63)	417.80(10.98)	356.54(2.66)	119.12(1.52)	178.80
M1AG3	4.00	54.71(0.58)	53.89(0.47)	53.99(1.36)	410.78(11.61)	1978.58(33.86)	2016.48(34.39)	761.40
M1CG3	3.96	54.55(1.37)	56.11(2.42)	56.22(1.42)	413.32(6.86)	2039.98(19.72)	2050.72(18.22)	778.48
M1CG2	3.97	57.24(1.21)	56.03(1.84)	57.25(1.55)	454.27(5.40)	2159.45(17.65)	2188.14(20.80)	828.73
M4AG1	23.95	277.31(1.62)	282.22(2.92)	291.47(3.70)	439.22(8.45)	1654.81(10.47)	3631.29(5.74)	1096.05
M4BG2	23.95	296.27(0.50)	297.90(0.44)	312.31(1.14)	485.87(3.53)	1702.12(8.47)	3652.17(2.71)	1124.44
M1BG3	3.67	66.50(1.99)	52.48(2.34)	92.89(2.83)	565.32(11.80)	3013.13(49.15)	3096.65(56.61)	1147.83
M4AG2	24.34	353.25(0.88)	354.44(0.87)	364.86(3.16)	527.89(2.59)	1725.18(7.17)	3704.93(9.00)	1171.76
M3BG2	24.12	297.31(7.38)	299.04(6.18)	314.91(6.62)	674.98(11.82)	1907.40(58.06)	3814.66(12.06)	1218.05
M2BG2	24.37	360.39(10.07)	373.56(8.05)	397.82(6.87)	589.98(24.62)	1823.89(43.51)	3772.91(24.40)	1219.76
M3AG1	24.47(-)	299.21(7.13)	293.12(2.03)	329.84(7.88)	691.96(31.36)	1847.49(57.95)	4682.02(192.67)	1357.27[-]
M2AG1	25.66(-)	508.84(6.84)	508.41(9.92)	533.87(7.33)	901.93(7.50)	3023.39(40.32)	5061.43(36.19)	1756.31[-]
M3AG2	24.88(-)	309.38(3.61)	306.38(2.68)	347.55(4.29)	832.18(8.78)	3664.03(150.74)	5604.45(43.73)	1843.99[-]
M2AG2	25.72(-)	509.16(1.35)	504.56(4.62)	556.20(3.47)	1177.88(58.13)	4196.18(107.57)	6247.27(127.26)	2198.54[-]
M1BG2	26.48(-)	455.58(1.49)	441.85(3.47)	502.59(3.79)	1325.18(25.19)	5635.84(66.01)	7725.80(105.64)	2681.14[-]
M1AG1	≈16000(!)	-	-	-	-	-	-	-
M1AG2	≈16000(!)	-	-	-	-	-	-	-

Source: The Author

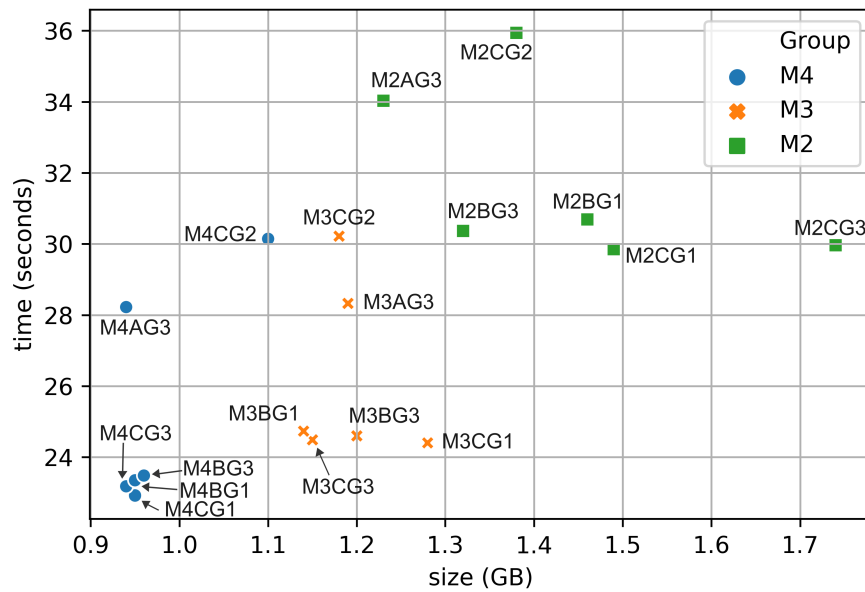
Due to the great number of schemas, we will discuss the features of the top 5 (best) schemas and the bottom 5 in order to have some insight regarding the two ends of the results spectrum. Section 6.2.2 discusses the results based on data volume while Section 6.2.3 takes the average query execution time into account.

Figure 49 – Volume vs. arithmetic mean of query execution times for the 34 evaluated schemas.



Source: The Author

Figure 50 – Volume vs. arithmetic mean of query execution times for the 18 schemas having similar results.



Source: The Author

### 6.2.2 Data volume analysis

The top 5 and bottom 5 schemas are indicated in Figure 47 by the symbols (+) and (-) at the left side of its name, respectively. In Table 6, these symbols are also found in the *Size* column.

The five schemas with the lowest storage cost are M4CG3 (0.94 GB), M4AG3 (0.94 GB), M4BG1 (0.95 GB), M4CG1 (0.95 GB), and M4BG3 (0.96 GB). In Figure 50, these schemas used a data volume of less than 1 GB.

The schemas with low storage costs tend to normalize their GDs. For example, all GDs of the M4BG1 and M4BG3 schemas were normalized. Except for the GD *address*, all other GDs of M4CG3, M4AG3, and M4CG1 schemas are also normalized. It is easy to see that the normalization of GDs with low selectivity geospatial fields (i.e., *city*, *nation*, and *region*) strongly contributes to reducing the DGDW storage cost.

Analyzing the five schemas with regard to document partitioning shows that partitioning documents into specific and homogeneous collections also contributes to reduced data volume. This happens because the storage volume of the index structure is smaller in schemas holding the same document type in the collections. Thus, all schemas in the M4 group (which have a collection for each dimension, i.e., homogeneous collections) have a lower volume than their corresponding schemas (e.g.,  $M4CG1 < M3CG1 < M2CG1$ ) (cf. Table 6).

The five schemas with the highest storage costs are M1BG2 (26.48 GB), M2AG2 (25.72 GB), M2AG1 (25.66 GB), M3AG2 (24.88 GB), and M3AG1 (24.47 GB). These schemas are identified in Figure 49 as having volumes greater than 24 GB.

The results show that geospatial document redundancy increases data volume. The combination of the model A with G1 and G2, and the model B with G2 (i.e., schemas that have names ending in AG1, AG2, and BG2) results in high geospatial data redundancy when GDs that contain geospatial fields of low-selectivity (i.e., *city*, *nation*, and *region*) are denormalized into CDs. As shown in Table 6, the denormalization of geospatial fields with low-selectivity contributes strongly to raising the storage cost of the DGDWs.

Furthermore, the M1 group schemas have the highest data volumes. Indeed, the M1AG1 and M1AG2 schemas were removed from the experiment because they would have a size of approximately 16 TB. The M1 group also has higher volumes than those of its corresponding schemas (e.g.,  $M1BG2 > M2BG2 > M3BG2 > M4BG2$ ). Although the M1 group has schemas with homogeneous collections (cf. M1AG1 and M1AG2 in Figure 47) which contribute to reduced data volume, the denormalization of CDs (and their

GDs) into fact documents strongly increases the data volume of a DGDW, which makes it impractical and not feasible.

### 6.2.3 Query runtime analysis

The top 5 and bottom 5 schemas are depicted in Figure 47 by the symbols  $[+]$  and  $[-]$ , respectively, to the right of their names. In Table 6, these symbols are also found in the  $Avg(Q_{1-6})$  column.

The five schemas with the lowest average query runtimes are M4CG1 (22.92 s), M4CG3 (23.17 s), M4BG1 (23.34 s), M4BG3 (23.48 s), and M3CG1 (24.40 s). These schemas are depicted in Figure 50 with query runtimes close to 24 s.

The top 5 schemas have their low-selectivity geospatial fields normalized into GDs (i.e., *city*, *nation*, and *region*). The computational cost of joins to perform the queries is less than the cost of high data redundancy in the document-oriented database. However, the denormalization of the GD that contains a high-selectivity geospatial field (i.e., *address*) into CDs (i.e., M4CG1, M4CG3, and M3CG1 schemas) positively influences the query runtimes, by reducing the number of query joins (i.e., they do not need to perform joins to obtain *address*). Both the normalization of low-selectivity geospatial fields and the denormalization of high-selectivity geospatial fields contribute positively to the performance of the DGDWs.

Regarding schema partitioning, except the M3CG1 schema, the best schemas all have one collection for each dimension (i.e., these schemas partition the fact documents, CDs, and GDs into specific and homogeneous collections). The M4 schemas also have better performance than their corresponding M3 and M2 schemas (e.g.,  $M4CG3 < M3CG3$  and  $M4CG3 < M2CG3$ ). The partitioning of documents into homogeneous collections therefore contributes to improved DGDW performance.

The five schemas with the highest average query runtimes are also the schemas with highest storage cost: M1BG2 (2681.14 s), M2AG2 (2198.54 s), M3AG2 (1843.99 s), M2AG1 (1756.31 s), and M3AG1 (1357.27 s). These schemas are depicted in Figure 49, having query runtimes greater than 1300 s.

These schemas have high data redundancy, because their GDs that contain low-selectivity geospatial fields (i.e., *city*, *nation*, and *region*) are denormalized into CDs. When performing queries, selection operations and joins among dimensions require more processing power to manipulate the large volumes of data, greatly increasing the query runtimes.

Regarding the partitioning of these schemas, we noticed that they have heterogeneous collections (i.e., more than one dimension stored in the same collection). There is a loss of query performance in collections that contain different document types, and the more heterogeneous the collection, the worse the performance. This finding can be shown by comparing, in Table 6, the results of corresponding schemas that have different partitioning models (e.g., M2BG2 vs. M3BG2).

### 6.3 BEST PRACTICES

Based on the results of the experimental evaluation, we point out some constructions that contribute to improve performance on the DGDW:

- Model homogeneous collections. This practice reduces the data volume of index structures and contributes to query performance because the documents of the collection have the same field structure.
- Normalize geospatial fields with low selectivity (e.g., *region*). This implies the use of referenced documents and, consequently, perform joins in queries. However, the computational cost to perform joins in referenced documents is less than the cost to manipulate embedded documents with high geospatial data volume.
- Denormalize geospatial fields with high selectivity (e.g., *address*). This practice avoids performing joins in queries and does not substantially increase the data redundancy because the geospatial field is not replicated in many embedded documents.

## 6.4 CHAPTER FINAL CONSIDERATIONS

This chapter presented guidelines for the use AStar. Some models are defined to represent facts and dimensions that are related as embedded or referenced documents and partitioned into one or more collections. Using these models, an experimental evaluation was carried out to analyze the impact produced by the constructions addressed in the models on the data volume and query performance. Based on the results of this experimental evaluation, some constructions that contribute to improved DGDW performance were pointed out.

## 7 CONCLUSIONS

This chapter presents the conclusions of this study. Section 7.1 summarizes the final considerations, section 7.2 lists the contributions and articles published or under review, and section 7.3 presents the limitations and some suggestions for future studies.

### 7.1 FINAL CONSIDERATIONS

The logical schema of a DGDW establishes relationships between facts and dimensions as either referenced or embedded documents, which can be partitioned into one or more collections. However, to the best available knowledge, these features are not covered by any modeling languages. In order to resolve this shortcoming, AStar, a DSML for the design of DGDW logical schemas, was defined.

AStar is composed of a graphical notation (concrete syntax), its own and custom-tailored metamodel (abstract syntax) for the design of DGDW schemas, and well-formedness rules (static semantics). Translational semantics are used to map the concepts defined in the syntax to their corresponding semantics. AStar does not mix SOLAP concepts into its modeling, because a DGDW may need to be used by applications other than SOLAP (e.g., reporting and data mining tools). It is important to highlight that the AStar graphical notation is inspired by the UML class diagram notation, but the syntax and semantics of its symbols are defined by the AStar metamodel and well-formedness rules. This reduces the complexity of implementing a CASE tool because, unlike a UML profile, it does not require stereotypes or constraints (e.g., OCL rules) to adapt its symbols to a specific context.

AStar addresses the particularities of the document-oriented data model and the logical design of a GDW. Regarding the document-oriented data model, AStar provides support for the design of homogeneous or heterogeneous collections; the specification of the document field structures, which are composed of conventional or geospatial fields that can be set as identifier, unique, or regular; and the establishment of relationships using embedded documents with 1:1 cardinality, or referenced documents with 1:1, 1:N,

or M:N cardinality. It is important to note that AStar does not allow relationships using embedded documents to be depicted with 1:N or M:N cardinality. These relationship cardinalities would be implemented as arrays of embedded documents, producing documents with a very high volume of data, which can impair DGDW performance. Furthermore, embedded documents with M:N cardinality can produce cyclic dependence. AStar also has resources to forbid the design of incorrect constructions, such as documents from different collections being embedded and the definition of identifier fields in embedded documents. Concerning the logical design of a GDW, AStar provides support to model facts and dimensions, distinguish between them, and prevent the modeling of disconnected facts or dimensions.

In the evaluation, AStar is shown to be in accordance with seven of the nine PoN principles, an adequate level of cognitive effectiveness. AStar was then implemented as a prototype CASE tool called AStarCASE. In its current version, AStarCASE can be used to design DGDW logical schemas and generate code from the diagrams, consisting of JSON schemas that define the database schema in MongoDB. Because the AStar concrete syntax is based on symbols from the UML class diagram, UML tools can be used to model DGDW logical schemas according to AStar. However, only AStar implementations (e.g., AStarCASE) can prevent syntactically invalid constructions or flag semantically contradictory constructions.

A guideline was presented to show how to design logical schemas of DGDW with AStar. This guideline defines some models for the design of logical schemas that use different types of relationships between facts and dimensions, producing different levels of data redundancy and different ways to partition documents among collections. An experimental evaluation was also performed with 36 DGDW logical schemas, depicted with AStar. The evidence obtained from the experimental analysis showed that the denormalization of geospatial fields with high selectivity (i.e., using embedded documents), the normalization of geospatial fields with low selectivity (i.e., using referenced documents), and the partitioning of documents having facts and dimensions into homogeneous collections (i.e., more than one collection) contribute to achieving low data volume and low query execution time.

## 7.2 CONTRIBUTIONS

This study presents the following contributions:

- A DSML for DGDW logical schemas, composed of:
  - a graphical notation (concrete syntax) inspired by the UML class diagram, which enables any UML tool to model DGDW logical schemas;
  - a metamodel (abstract syntax) that defines the concepts of the modeling language, addressing features of the document-oriented data model and GDW logical design;
  - a set of well-formedness rules (static semantics) that is used to prohibit invalid constructions in a logical schema;
  - translational semantics, which map the concepts defined in the graphical notation to the concepts defined in the metamodel, and to the JSON schema definition.
- A guideline showing how to use AStar. This guideline presents some models that can assist in the design of logical schemas, as well as indications to achieve good DGDW performance.
- A prototype CASE tool, which can be used to design DGDW logical schemas and generate JSON schema code.

Some of these contributions have already been published or are under review. All of these publications are listed below:

- **Ferro, M.;** Fragoso, R.; Fidalgo, R. (2019). *Document-oriented geospatial data warehouse: An experimental evaluation of SOLAP queries*. Conference on Business Informatics (CBI) - This paper presents the design of 9 DGDW logical schemas, using AStar. These schemas have facts and dimensions related as embedded or referenced documents, producing different levels of geospatial data redundancy. The schemas were evaluated, in order to analyze the data volume and query runtime.

- **Ferro, M.;** Lima, R.; Fidalgo, R. (2019). *Evaluating redundancy and partitioning of geospatial data in document-oriented data warehouses*. In Big Data Analytics and Knowledge Discovery (DaWaK) - This paper presents the design of 36 DGDW logical schemas, using AStar. These schemas have facts and dimensions related as referenced or embedded documents, and have different approaches of partitioning documents among collections. The schemas were evaluated and their data volume and query runtimes were determined.
- **Ferro, M.;** Silva, E.; Fidalgo, R. *AStar: A Modeling Language for Document-Oriented Geospatial Data Warehouses* - Submitted to Data & Knowledge Engineering on December 2021. This paper shows the complete AStar graphical notation, whereas the previous papers only used part of this notation. Moreover, this paper defines the metamodel and well-formedness rules of AStar. An AStar evaluation based on PoN is also shown.

### 7.3 THREAT TO VALIDITY, LIMITATIONS, AND FUTURE STUDIES

Despite the relevant results, this thesis has some threats to validity and limitations that can be addressed in future studies.

Although the search for anomalies was carried out carefully in the PoN assessment presented in section 5.1, it cannot be ignored that the evaluation has only been performed by the authors. This can be considered a potential threat to validity, as it can affect the ability to draw proper conclusions about the validation of the PoN's principles. Therefore, a user evaluation should be performed as a future study, in order to compare the results obtained with those presented in this thesis.

Due to hardware limitations, the experimental evaluation presented in section 6.2 used only one data load, with a scale factor of 1. Furthermore, given the lack of benchmarks for DGDW performance analysis, used in both the industry and academics, the authors defined a concise set of queries to be used in this experiment. This can be considered a potential threat to validity, as the behavior of the query execution when used with different data loads was not analyzed; and the used queries may have produced results

that overlooked important aspects in the analysis, such as the performance of different geospatial functions and data aggregation from many dimensions. Therefore, in order to present a more in-depth experimental analysis, an evaluation using higher data loads and a set of systematic queries should be performed as a future study.

AStar is a DSML whose syntax and semantics are custom-tailored to the DGDW domain. Therefore, features that can be used in a transactional database (e.g., arrays) are prohibited in AStar. In order to enable AStar to model logical schemas for transactional databases, an extension will be addressed in a future study.

AStarCASE is a prototype that can be evolved with a set of features to assist designers in modeling and maintaining DGDWs. Therefore, features such as the generation of physical project metadata for different document-oriented databases (e.g., MongoDB, CoachDB, and DynamoDB), the generation of document examples from diagrams, and reverse engineering from an existing DGDW will be covered in future studies.

## REFERENCES

- ABDELHEDI, F.; BRAHIM, A. A.; ATIGUI, F.; ZURFLUH, G. Big data and knowledge management: How to implement conceptual models in nosql systems'. In: . [S.l.]: SciTePress, 2016. v. 3, p. 235–240. ISBN 9789897582035.
- ABDELHEDI, F.; BRAHIM, A. A.; ATIGUI, F.; ZURFLUH, G. Logical unified modeling for nosql databases. In: . [S.l.]: SciTePress, 2017. v. 1, p. 249–256. ISBN 9789897582479.
- ABDELHEDI, F.; BRAHIM, A. A.; ATIGUI, F.; ZURFLUH, G. Mda-based approach for nosql databases modelling. In: BELLATRECHE, L.; CHAKRAVARTHY, S. (Ed.). *Big Data Analytics and Knowledge Discovery*. Cham: Springer International Publishing, 2017. p. 88–102. ISBN 978-3-319-64283-3.
- AGUILA, P. S. R. D.; FIDALGO, R. d. N.; MOTA, A. Towards a more straightforward and more expressive metamodel for sdw modeling. In: *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP*. New York, NY, USA: ACM, 2011. (DOLAP '11), p. 31–36. ISBN 978-1-4503-0963-9. Available at: <<http://doi.acm.org/10.1145/2064676.2064682>>.
- ANDOR, C.-F.; VARGA, V.; SăCĂREA, C. A graph based knowledge and reasoning representation approach for modeling mongodb data structure and query. In: *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. [S.l.: s.n.], 2019. p. 1–6.
- AZURE, M. *Data modeling in Azure Cosmos DB*. 2020. Accessed July 31, 2020. Available at: <<https://docs.microsoft.com/en-us/azure/cosmos-db/modeling-data>>.
- BANERJEE, S.; SHAW, R.; SARKAR, A.; DEBNATH, N. C. Towards logical level design of big data. In: *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*. [S.l.: s.n.], 2015. p. 1665–1671.
- BENSALLOUA, C. A.; BENAMEUR, A. Towards nosql-based data warehouse solution integrating ecdis for maritime navigation decision support system. *Informatica*, v. 45, n. 3, 2021.
- BERSON, A.; SMITH, S. J. *Data warehousing, data mining, and OLAP*. [S.l.]: McGraw-Hill, Inc., 1997.
- BERTIN, J. *Semiology of graphics; diagrams networks maps*. [S.l.], 1983.
- BOULIL, K.; BIMONTE, S.; PINET, F. Conceptual model for spatial data cubes: A uml profile and its automatic implementation. *Computer Standards & Interfaces*, v. 38, p. 113 – 132, 2015. ISSN 0920-5489. Available at: <<http://www.sciencedirect.com/science/article/pii/S0920548914000774>>.
- BRAHIM, A. A.; FERHAT, R. T.; ZURFLUH, G. Extraction process of conceptual model from a document-oriented nosql database. In: *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*. [S.l.: s.n.], 2019. p. 1–5.

- BRAMBILLA, M.; CABOT, J.; WIMMER, M. Model-Driven Software Engineering in Practice: Second Edition. *Synthesis Lectures on Software Engineering*, v. 3, n. 1, p. 1–207, 2017. ISSN 2328-3319.
- BRAY, T. *The JavaScript Object Notation (JSON) Data Interchange Format*. [S.l.], 2017. (Request for Comments, 8259). Available at: <<https://rfc-editor.org/rfc/rfc8259.txt>>.
- BREAULT, J. L.; GOODALL, C. R.; FOS, P. J. Data mining a diabetic data warehouse. *Artificial Intelligence in Medicine*, v. 26, n. 1, p. 37–54, 2002. ISSN 0933-3657. Medical Data Mining and Knowledge Discovery. Available at: <<https://www.sciencedirect.com/science/article/pii/S0933365702000519>>.
- BRUCE, T. A. *Designing quality databases with IDEF1X information models*. [S.l.]: Dorset House Publishing Co., Inc., 1992.
- BRUCK, J.; HUSSEY, K. *Customizing UML: Which Technique is Right for You?* 2007. <[https://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing\\_UML2\\_Which\\_Technique\\_is\\_Right\\_For\\_You/article.html](https://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html)>. Accessed: 2022-03-03.
- BRYANT, B. R.; GRAY, J.; MERNIK, M.; CLARKE, P. J.; FRANCE, R. B.; KARSAI, G. Challenges and directions in formalizing the semantics of modeling languages. 2011.
- BUDINSKY, F. *Eclipse Modeling Framework: A Developer's Guide*. [S.l.]: Addison-Wesley, 2004. (The eclipse series). ISBN 9780131425422.
- BUGIOTTI, F.; CABIBBO, L.; ATZENI, P.; TORLONE, R. Database design for nosql systems. In: SPRINGER. *International Conference on Conceptual Modeling*. [S.l.], 2014. p. 223–231.
- BUTLER, H.; DALY, M.; DOYLE, A.; GILLIES, S.; SCHAUB, T.; SCHAUB, T. *The GeoJSON Format*. [S.l.], 2016. (Request for Comments, 7946). Available at: <<https://rfc-editor.org/rfc/rfc7946.txt>>.
- CANDEL, C. J. F.; RUIZ, D. S.; GARCÍA-MOLINA, J. J. A unified metamodel for nosql and relational databases. *Information Systems*, v. 104, p. 101898, 2022. ISSN 0306-4379. Available at: <<https://www.sciencedirect.com/science/article/pii/S0306437921001149>>.
- CATARCI, T.; COSTABILE, M. F.; LEVIALDI, S.; BATINI, C. Visual query systems for databases: A survey. *Journal of Visual Languages & Computing*, Elsevier, v. 8, n. 2, p. 215–260, 1997.
- CHAUDHURI, S.; DAYAL, U. An overview of data warehousing and olap technology. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 26, n. 1, p. 65–74, Mar. 1997. ISSN 0163-5808. Available at: <<http://doi.acm.org/10.1145/248603.248616>>.
- CHEVALIER, M.; El Malki, M.; KOPLIKU, A.; TESTE, O.; TOURNIER, R. Implementing multidimensional data warehouses into NoSQL. *ICEIS 2015 - 17th International Conference on Enterprise Information Systems, Proceedings*, v. 1, p. 172–183, 2015.
- CHEVALIER, M.; MALKI, M.; KOPLIKU, A.; TESTE, O.; TOURNIER, R. Document-oriented models for data warehouses. In: *Proceedings of the 18th International Conference on Enterprise Information Systems (ICEIS 2016)*. [S.l.: s.n.], 2016. v. 1, p. 142–149. ISSN 978-989-758-187-8.

- CHEVALIER, M.; MALKI, M. E.; KOPLIKU, A.; TESTE, O.; TOURNIER, R. Implementation of multidimensional databases with document-oriented nosql. In: MADRIA, S.; HARA, T. (Ed.). *Big Data Analytics and Knowledge Discovery*. Cham: Springer International Publishing, 2015. p. 379–390. ISBN 978-3-319-22729-0.
- CHEVALIER, M.; MALKI, M. E.; KOPLIKU, A.; TESTE, O.; TOURNIER, R. Document-oriented data warehouses: Complex hierarchies and summarizability. In: EL-AZOUZI, R.; MENASCHE, D. S.; SABIR, E.; PELLEGRINI, F. D.; BENJILLALI, M. (Ed.). *Advances in Ubiquitous Networking 2*. Singapore: Springer Singapore, 2017. p. 671–683. ISBN 978-981-10-1627-1.
- COACHDB, A. *Document Design Considerations*. 2020. Accessed July 31, 2020. Available at: <<https://docs.couchdb.org/en/latest/best-practices/documents.html>>.
- COPELAND, R. *MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database*. [S.l.]: "O'Reilly Media, Inc.", 2013.
- CUZZOCREA, A.; FIDALGO, R. do N. Sdwm: An enhanced spatial data warehouse metamodel. In: CITESEER. *CAiSE Forum*. [S.l.], 2012. p. 32–39.
- DAVOUDIAN, A.; CHEN, L.; LIU, M. A survey on nosql stores. *ACM Computing Surveys (CSUR)*, ACM, v. 51, n. 2, p. 40, 2018.
- DB-ENGINES. *DB-Engines Ranking*. 2020. Accessed August 27, 2020. Available at: <<https://db-engines.com/en/ranking>>.
- DE LA VEGA, A.; GARCÍA-SAIZ, D.; BLANCO, C.; ZORRILLA, M.; SÁNCHEZ, P. Mortadelo: Automatic generation of nosql stores from platform-independent data models. *Future Generation Computer Systems*, v. 105, p. 455–474, 2020. ISSN 0167-739X. Available at: <<https://www.sciencedirect.com/science/article/pii/S0167739X19312063>>.
- DEMARCO, T. Structure analysis and system specification. In: *Pioneers and Their Contributions to Software Engineering*. [S.l.]: Springer, 1979. p. 255–288.
- DRUGS, U. N. O. on; CRIME. *International Classification of Crime for Statistical Purposes (ICCS)—Version 1.0*. [S.l.]: Author Vienna, Austria, 2015.
- DYNAMODB, A. *Create Table*. 2021. Accessed Janary 27, 2021. Available at: <<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithTables.Basics.html#WorkingWithTables.Basics.CreateTable>>.
- ELMASRI, R.; NAVATHE, S. *Fundamentals of database systems*. [S.l.]: Addison-Wesley Publishing Company, 2010.
- EVANS, E. *Domain-driven design: tackling complexity in the heart of software*. [S.l.]: Addison-Wesley Professional, 2004.
- FERRAHI, I.; BIMONTE, S.; KANG, M.-A.; BOUKHALFA, K. Design and implementation of falling star-a non-redudant spatio-multidimensional logical model for document stores. In: SCITEPRESS. *International Conference on Enterprise Information Systems*. [S.l.], 2017. v. 2, p. 343–350.

- FIDALGO, R. D. N.; SOUZA, E. M. D.; ESPAÑA, S.; CASTRO, J. B. D.; PASTOR, O. Eermm: a metamodel for the enhanced entity-relationship model. In: SPRINGER. *International Conference on Conceptual Modeling*. [S.l.], 2012. p. 515–524.
- FIDALGO, R. N.; ALVES, E.; ESPAÑA, S.; CASTRO, J.; PASTOR, O. Metamodeling the enhanced entity-relationship model. *Journal of Information and Data Management*, v. 4, n. 3, p. 406, 2013.
- FIDALGO, R. N.; TIMES, V. C.; SILVA, J.; SOUZA, F. F. Geodwframe: A framework for guiding the design of geographical dimensional schemas. In: SPRINGER. *International Conference on Data Warehousing and Knowledge Discovery*. [S.l.], 2004. p. 26–37.
- FILHO, W. B.; OLIVERA, H. V.; HOLANDA, M.; FAVACHO, A. A. Geographic data modeling for nosql document-oriented databases. *GEOProcessing*, v. 72, p. 2015, 2015.
- FLECK, M.; TROYA, J.; WIMMER, M. Search-based model transformations with momot. In: SPRINGER. *International Conference on Theory and Practice of Model Transformations*. [S.l.], 2016. p. 79–87.
- FONTOURA, M.; PREE, W.; RUMPE, B. *The UML profile for framework architectures*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2000.
- GESSERT, F.; WINGERATH, W.; FRIEDRICH, S.; RITTER, N. Nosql database systems: a survey and decision guidance. *Computer Science-Research and Development*, Springer, v. 32, n. 3-4, p. 353–365, 2017.
- GLORIO, O.; TRUJILLO, J. An mda approach for the development of spatial data warehouses. In: SONG, I.-Y.; EDER, J.; NGUYEN, T. M. (Ed.). *Data Warehousing and Knowledge Discovery*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 23–32. ISBN 978-3-540-85836-2.
- GROUP, O. M. *Meta-Object Facility*. 2020. Accessed March 13, 2020. Available at: <<https://www.omg.org/mof/>>.
- GUY, C.; COMBEMALE, B.; DERRIEN, S.; STEEL, J. R.; JÉZÉQUEL, J.-M. On model subtyping. In: SPRINGER. *European Conference on Modelling Foundations and Applications*. [S.l.], 2012. p. 400–415.
- GÓMEZ, P.; RONCANCIO, C.; CASALLAS, R. Analysis and evaluation of document-oriented structures. *Data & Knowledge Engineering*, v. 134, p. 101893, 2021. ISSN 0169-023X. Available at: <<https://www.sciencedirect.com/science/article/pii/S0169023X21000203>>.
- HAI, R.; GEISLER, S.; QUIX, C. Constance: An intelligent data lake system. In: *Proceedings of the 2016 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2016. (SIGMOD '16), p. 2097–2100. ISBN 9781450335317. Available at: <<https://doi.org/10.1145/2882903.2899389>>.
- HAN, J.; HAIHONG, E.; LE, G.; DU, J. Survey on nosql database. In: IEEE. *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. [S.l.], 2011. p. 363–366.

- HAREL, D. On visual formalisms. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 31, n. 5, p. 514–530, May 1988. ISSN 0001-0782. Available at: <<https://doi.org/10.1145/42411.42414>>.
- INDEXES, U. *Unique Indexes*. 2021. Accessed Janary 27, 2021. Available at: <<https://docs.mongodb.com/manual/core/index-unique/>>.
- INMON, W. H. *Building the Data Warehouse*. [S.l.]: Wiley Publishing Inc., 2005.
- JOVANOVIĆ, V.; BENSON, S. Aggregate data modeling style. In: *Proceedings of the Southern Association for Information Systems Conference*. [S.l.: s.n.], 2013. p. 70–75.
- KANADE, A.; GOPAL, A.; KANADE, S. A study of normalization and embedding in mongodb. In: IEEE. *Advance Computing Conference (IACC), 2014 IEEE International*. [S.l.], 2014. p. 416–421.
- KAUR, K.; RANI, R. Modeling and querying data in nosql databases. In: *2013 IEEE International Conference on Big Data*. [S.l.: s.n.], 2013. p. 1–7.
- KELLY, S.; TOLVANEN, J.-P. *Domain-specific modeling: enabling full code generation*. [S.l.]: John Wiley & Sons, 2008.
- KIMBALL, R.; ROSS, M. *The data warehouse toolkit: The definitive guide to dimensional modeling*. [S.l.]: John Wiley & Sons, 2013.
- KLEPPE, A. A language description is more than a metamodel. In: MEGAPLANET. ORG. *Fourth international workshop on software language engineering*. [S.l.], 2007. v. 1, p. 1–4.
- KOLOVOS, D.; ROSE, L.; GARCIA-DOMINGUEZ, A. et al. The epsilon validation language', in. *The Epsilon Book*, p. 57–76, 2017.
- LARKIN, J. H.; SIMON, H. A. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, Wiley Online Library, v. 11, n. 1, p. 65–100, 1987.
- LEE, J.-G.; MINSEO, K. Geospatial big data: Challenges and opportunities. *Big Data Research*, v. 2, 02 2015.
- LI, D. Research on Data Sharing of Water Conservancy Informatization Based on Data Mining and Cloud Computing. *Journal of Physics: Conference Series*, v. 1982, n. 1, p. 012130, jul 2021. ISSN 1742-6588. Available at: <<https://iopscience.iop.org/article/10.1088/1742-6596/1982/1/012130>>.
- LIMA, C. de; MELLO, R. dos S. A workload-driven logical design approach for nosql document databases. In: *Proceedings of the 17th International Conference on Information Integration and Web-Based Applications & Services*. New York, NY, USA: Association for Computing Machinery, 2015. (iiWAS '15). ISBN 9781450334914. Available at: <<https://doi.org/10.1145/2837185.2837218>>.
- LINDEN, D. van der; HADAR, I. A systematic literature review of applications of the physics of notations. *IEEE Transactions on Software Engineering*, v. 45, n. 8, p. 736–759, 2019.

LOPES, F.; SANTOS, M.; FIDALGO, R.; FERNANDES, S. A software engineering perspective on sdn programmability. *IEEE Communications Surveys Tutorials*, v. 18, n. 2, p. 1255–1272, Secondquarter 2016.

MALINOWSKI, E.; ZIMÁNYI, E. Representing spatiality in a conceptual multidimensional model. In: *Proceedings of the 12th Annual ACM International Workshop on Geographic Information Systems*. New York, NY, USA: ACM, 2004. (GIS '04), p. 12–22. ISBN 1-58113-979-9. Available at: <<http://doi.acm.org/10.1145/1032222.1032226>>.

MATEUS, R.; SIQUEIRA, T.; TIMES, V.; CIFERRI, R.; CIFERRI, C. How Does the Spatial Data Redundancy Affect Query Performance in Geographic Data Warehouses? *Journal of Information and Data Management*, v. 1, n. 3, p. 519, 2010. ISSN 2178-7107.

MATEUS, R. C.; SIQUEIRA, T. L. L.; TIMES, V. C.; CIFERRI, R. R.; CIFERRI, C. D. de A. Spatial data warehouses and spatial olap come towards the cloud: design and performance. *Distributed and Parallel Databases*, v. 34, n. 3, p. 425–461, Sep 2016. ISSN 1573-7578. Available at: <<https://doi.org/10.1007/s10619-015-7176-z>>.

MILOSLAVSKAYA, N.; TOLSTOY, A. Big data, fast data and data lake concepts. *Procedia Computer Science*, Elsevier, v. 88, p. 300–305, 2016.

MOHAGHEGHI, P.; DEHLEN, V.; NEPLE, T. Definitions and approaches to model quality in model-based software development – a review of literature. *Information and Software Technology*, v. 51, n. 12, p. 1646 – 1669, 2009. ISSN 0950-5849. Quality of UML Models. Available at: <<http://www.sciencedirect.com/science/article/pii/S0950584909000457>>.

MONGODB. *Introduction to MongoDB*. 2020. <<https://docs.mongodb.com/manual/introduction/>>. Accessed: 2021-01-03.

MONGODB. *How to Set Up a MongoDB Cluster*. 2021. Accessed April 02, 2021. Available at: <<https://www.mongodb.com/basics/clusters/mongodb-cluster-setup>>.

MONGODB. *Schema Validation*. 2021. Accessed January 27, 2021. Available at: <<https://docs.mongodb.com/manual/core/schema-validation/>>.

MONGODB. *Model One-to-Many Relationships with Embedded Documents*. 2022. Accessed April 02, 2022. Available at: <<https://www.mongodb.com/docs/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/>>.

MOODY, D. The “physics” of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, v. 35, n. 6, p. 756–779, 2009.

NICKERSON, J. V. Visual programming: Limits of graphic representation. In: IEEE. *Proceedings of 1994 IEEE Symposium on Visual Languages*. [S.l.], 1994. p. 178–179.

OGC. *OGC Standards*. 1999. Online; accessed 11 February 2020. Available at: <<https://www.ogc.org/docs/is>>.

OMG. *Object Constraint Language, v2.4*. 2014. [Http://www.omg.org/spec/OCL/2.4/PDF](http://www.omg.org/spec/OCL/2.4/PDF). Accessed: 2016-05-05.

- O'NEIL, P. E.; O'NEIL, E. J.; CHEN, X. The star schema benchmark (ssb). *Pat*, v. 200, n. 0, p. 50, 2007.
- SARKAR, A.; ROY, S. S. Graph semantic based conceptual model of semi-structured data : An object oriented approach. In: . [S.l.: s.n.], 2011.
- SHIN, K.; HWANG, C.; JUNG, H. Nosql database design using uml conceptual data model based on peter chen's framework. *International Journal of Applied Engineering Research*, v. 12, n. 5, p. 632–636, 2017.
- SILVA, A. R. D. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, Elsevier, v. 43, p. 139–155, 2015.
- SIQUEIRA, T. L. L.; CIFERRI, R. R.; TIMES, V. C.; CIFERRI, C. D. de A. A spatial bitmap-based index for geographical data warehouses. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. [S.l.: s.n.], 2009. p. 1336–1342.
- SIQUEIRA, T. L. L.; CIFERRI, R. R.; TIMES, V. C.; CIFERRI, C. D. de A. Benchmarking spatial data warehouses. In: SPRINGER. *International Conference on Data Warehousing and Knowledge Discovery*. [S.l.], 2010. p. 40–51.
- SRAI, A.; GUEROUATE, F.; BERBICHE, N.; DRISSI, H. An mda approach for the development of data warehouses from relational databases using atl transformation language. *International Journal of Applied Engineering Research*, v. 12, n. 12, p. 3532–3538, 2017.
- TEIXEIRA, M. d. G. da S.; QUIRINO, G. K.; GAILLY, F.; FALBO, R. de A.; GUIZZARDI, G.; BARCELLOS, M. P. Pon-s: a systematic approach for applying the physics of notation (pon). In: *Enterprise, Business-Process and Information Systems Modeling*. [S.l.]: Springer, 2016. p. 432–447.
- THORNTHWAITE, W. *Dealing With Nulls In The Dimensional Model*. 2003. Online; accessed 30 March 2020. Available at: <<https://www.kimballgroup.com/2003/02/design-tip-43-dealing-with-nulls-in-the-dimensional-model/>>.
- TING, I.-H. Complex data warehousing and knowledge discovery for advanced retrieval development: Innovative methods and applications. *Online Information Review*, Emerald Group Publishing Limited, 2010.
- TSOIS, A.; KARAYANNIDIS, N.; SELLIS, T. K. Mac: Conceptual data modeling for olap. In: *DMDW*. [S.l.: s.n.], 2001. v. 39, p. 5.
- VERA, H.; BOAVENTURA, W.; HOLANDA, M.; GUIMARAES, V.; HONDO, F. Data modeling for nosql document-oriented databases. In: *CEUR Workshop Proceedings*. [S.l.: s.n.], 2015. v. 1478, p. 129–135.
- WRIGHT, A.; ANDREWS, H.; HUTTON, B.; DENNIS, G. *JSON Schema: A Media Type for Describing JSON Documents*. [S.l.], 2020. Work in Progress. Available at: <<https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00>>.
- YANGUI, R.; NABLI, A.; GARGOURI, F. Automatic transformation of data warehouse schema to nosql data base: comparative study. *Procedia Computer Science*, Elsevier, v. 96, p. 255–264, 2016.

ZEČEVIĆ, I.; BJELJAC, P.; PERIŠIĆ, B.; STANKOVSKI, S.; VENUS, D.; OSTOJIC, G. Model driven development of hybrid databases using lightweight metamodel extensions. *Enterprise Information Systems*, Taylor & Francis, v. 12, n. 8-9, p. 1221–1238, 2018. Available at: <<https://doi.org/10.1080/17517575.2018.1445295>>.

ZYP, K.; COURT, G. *A JSON Media Type for Describing the Structure and Meaning of JSON Documents*. [S.l.], 2010. Work in Progress. Available at: <<https://datatracker.ietf.org/doc/html/draft-zyp-json-schema-03>>.

## APPENDIX A – JSON SCHEMAS FOR GEOSPATIAL DATA TYPES

The following JSON Schemas are adapted from:

**Point** - <https://geojson.org/schema/Point.json>

**Line** - <https://geojson.org/schema/LineString.json>

**Polygon** - <https://geojson.org/schema/Polygon.json>

**MultiPoint** - <https://geojson.org/schema/MultiPoint.json>

**MultiLine** - <https://geojson.org/schema/MultiLineString.json>

**MultiPolygon** - <https://geojson.org/schema/MultiPolygon.json>

**Collection** - <https://geojson.org/schema/GeometryCollection.json>

Listing A.1 – Point

```
{
2  "bsonType": "object",
   "required": [
4    "bsonType",
    "coordinates"
6  ],
   "properties": {
8     "bsonType": {
        "bsonType": "string",
10       "enum": [
          "Point"
12     ]
    },
14   "coordinates": {
        "bsonType": "array",
16       "minItems": 2,
        "items": {
18         "bsonType": "number"
        }
20     }
   }
22 }
```

Listing A.2 – Line

```
{
2  "bsonType": "object",
  "required": [
4    "bsonType",
    "coordinates"
6  ],
  "properties": {
8    "bsonType": {
      "bsonType": "string",
10     "enum": [
        "LineString"
12     ]
    },
14    "coordinates": {
      "bsonType": "array",
16     "minItems": 2,
      "items": {
18       "bsonType": "array",
        "minItems": 2,
20       "items": {
        "bsonType": "number"
22       }
      }
24    }
  }
26 }
```

Listing A.3 – Polygon

```
{
2  "bsonType": "object",
  "required": [
4    "bsonType",
    "coordinates"
6  ],
  "properties": {
8    "bsonType": {
      "bsonType": "string",
10     "enum": [
        "Polygon"
12     ]
    },
14    "coordinates": {
      "bsonType": "array",
16     "items": {
        "bsonType": "array",
```

```
18     "minItems": 4,  
    "items": {  
20     "bsonType": "array",  
     "minItems": 2,  
22     "items": {  
       "bsonType": "number"  
24     }  
    }  
26  }  
    }  
28  }  
}
```

Listing A.4 – MultiPoint

```
1 {  
  "bsonType": "object",  
3  "required": [  
    "bsonType",  
5    "coordinates"  
  ],  
7  "properties": {  
    "bsonType": {  
9      "bsonType": "string",  
      "enum": [  
11       "MultiPoint"  
      ]  
13    },  
    "coordinates": {  
15      "bsonType": "array",  
      "items": {  
17        "bsonType": "array",  
        "minItems": 2,  
19        "items": {  
          "bsonType": "number"  
21        }  
      }  
23    }  
  }  
25 }
```

Listing A.5 – MultiLine

```
1 {  
  "bsonType": "object",  
3  "required": [  
    "bsonType",  
5    "coordinates"
```

```

],
7  "properties": {
    "bsonType": {
9      "bsonType": "string",
      "enum": [
11         "MultiLineString"
      ]
13    },
    "coordinates": {
15      "bsonType": "array",
      "items": {
17        "bsonType": "array",
        "minItems": 2,
19        "items": {
          "bsonType": "array",
21          "minItems": 2,
          "items": {
23            "bsonType": "number"
          }
25        }
      }
27    }
  }
29 }

```

Listing A.6 – MultiPolygon

```

1  {
    "bsonType": "object",
3  "required": [
    "bsonType",
5    "coordinates"
  ],
7  "properties": {
    "bsonType": {
9      "bsonType": "string",
      "enum": [
11         "MultiPolygon"
      ]
13    },
    "coordinates": {
15      "bsonType": "array",
      "items": {
17        "bsonType": "array",
        "items": {
19          "bsonType": "array",
          "minItems": 4,

```

```

21         "items": {
22             "bsonType": "array",
23             "minItems": 2,
24             "items": {
25                 "bsonType": "number"
26             }
27         }
28     }
29 }
30 }
31 }
32 }

```

Listing A.7 – Collection

```

{
2   "bsonType": "object",
3   "required": [
4       "bsonType",
5       "geometries"
6   ],
7   "properties": {
8       "bsonType": {
9           "bsonType": "string",
10          "enum": [
11              "GeometryCollection"
12          ]
13      },
14      "geometries": {
15          "bsonType": "array",
16          "items": {
17              "oneOf": [
18                  {
19                      "bsonType": "object",
20                      "required": [
21                          "bsonType",
22                          "coordinates"
23                      ],
24                      "properties": {
25                          "bsonType": {
26                              "bsonType": "string",
27                              "enum": [
28                                  "Point"
29                              ]
30                          },
31                          "coordinates": {
32                              "bsonType": "array",

```

```

        "minItems": 2,
34         "items": {
            "bsonType": "number"
36         }
    }
38 }
},
40 {
    "bsonType": "object",
42     "required": [
        "bsonType",
44         "coordinates"
    ],
46     "properties": {
        "bsonType": {
48             "bsonType": "string",
            "enum": [
50                 "LineString"
            ]
52         },
        "coordinates": {
54             "bsonType": "array",
            "minItems": 2,
56             "items": {
                "bsonType": "array",
58                 "minItems": 2,
                "items": {
60                     "bsonType": "number"
                }
62             }
        }
64     }
},
66 {
    "bsonType": "object",
68     "required": [
        "bsonType",
70         "coordinates"
    ],
72     "properties": {
        "bsonType": {
74             "bsonType": "string",
            "enum": [
76                 "Polygon"
            ]
78         },
        "coordinates": {

```

```

80         "bsonType": "array",
81         "items": {
82             "bsonType": "array",
83             "minItems": 4,
84             "items": {
85                 "bsonType": "array",
86                 "minItems": 2,
87                 "items": {
88                     "bsonType": "number"
89                 }
90             }
91         }
92     }
93 }
94 },
95 {
96     "bsonType": "object",
97     "required": [
98         "bsonType",
99         "coordinates"
100 ],
101     "properties": {
102         "bsonType": {
103             "bsonType": "string",
104             "enum": [
105                 "MultiPoint"
106             ]
107         },
108         "coordinates": {
109             "bsonType": "array",
110             "items": {
111                 "bsonType": "array",
112                 "minItems": 2,
113                 "items": {
114                     "bsonType": "number"
115                 }
116             }
117         }
118     }
119 },
120 {
121     "bsonType": "object",
122     "required": [
123         "bsonType",
124         "coordinates"
125 ],
126     "properties": {

```

```

    "bsonType": {
128       "bsonType": "string",
        "enum": [
130         "MultiLineString"
        ]
132     },
    "coordinates": {
134       "bsonType": "array",
        "items": {
136         "bsonType": "array",
            "minItems": 2,
138         "items": {
            "bsonType": "array",
140             "minItems": 2,
            "items": {
142                 "bsonType": "number"
            }
144         }
        }
146     }
    }
148 },
{
150     "bsonType": "object",
    "required": [
152         "bsonType",
        "coordinates"
154     ],
    "properties": {
156         "bsonType": {
            "bsonType": "string",
158             "enum": [
                "MultiPolygon"
160             ]
        },
162         "coordinates": {
            "bsonType": "array",
164             "items": {
                "bsonType": "array",
166                 "items": {
                    "bsonType": "array",
168                     "minItems": 4,
                    "items": {
170                         "bsonType": "array",
                            "minItems": 2,
172                         "items": {
                            "bsonType": "number"
                        }
                    }
                }
            }
        }
    }
}

```

174 }  
176 }  
178 }  
180 }  
182 ]  
182 }  
184 }  
}

## APPENDIX B – EXAMPLES OF JSON SCHEMAS

Listing B.1 – JSON Schema for the example of Figure 24

```

1 db.createCollection( "Dates", {
  validator: { $jsonSchema: {
3     title: "date",
    description: "dimension",
5     bsonType: "object",
    required: [ "_id", "date", "dayofweek" ],
7     properties: {
      _id: {
9         bsonType: "object",
        required: [ "datekey" ],
11        properties: {
          datekey: {
13            bsonType: "int"
          }
15        }
      },
17      date: {
        bsonType: "string",
19        pattern: "^[1-9][0-9][0-9][0-9]-[0-1][0-9]-[0-3][0-9]\\$",
      },
21      dayofweek: {
        bsonType: "int"
23      }
    }
  } },
25 );
27 db.Dates.createIndex({ "date":1},{unique:1});

```

Listing B.2 – JSON Schema for the example of Figure 25

```

1 db.createCollection( "Orders", {
  validator: { $jsonSchema: {
3     title: "lineorder",
    description: "fact",
5     bsonType: "object",
    required: [ "_id", "revenue", "cust" ],
7     properties: {
      _id: {
9         bsonType: "object",
        required: [ "orderkey", "linenumber" ],
11        properties: {
          orderkey: {

```

```

13         bsonType: "int"
14     },
15     linenumber: {
16         bsonType: "int",
17     }
18 }
19 },
20 revenue: {
21     bsonType: "number",
22 },
23 cust:{
24     title: "customer",
25     description: "dimension",
26     bsonType: "object",
27     required: [ "name", "address_geo"],
28     properties: {
29         name: {
30             bsonType: "string",
31         },
32         address_geo: {
33             type: "object",
34             required: [ "type", "coordinates"],
35             properties: {
36                 type: {
37                     type: "string",
38                     enum: ["Point"]
39                 },
40                 coordinates: {
41                     type: "array",
42                     minItems: 2,
43                     items: {
44                         "type": "number"
45                     }
46                 }
47             }
48         }
49     }
50 },
51 },
52 },
53 });

```

Listing B.3 – JSON Schema for the example of Figure 26

```

1 db.createCollection( "Orders", {
3     validator: { $jsonSchema: {

```

```

anyOf: [
5   {
      title: "lineorder",
7      description: "fact",
      bsonType: "object",
9      required: [ "_id", "revenue", "cust"],
      properties: {
11         _id: {
            bsonType: "object",
13            required: [ "orderkey", "linenumber"],
            properties: {
15                orderkey: {
                    bsonType: "int"
17                },
                linenumber: {
19                    bsonType: "int",
21                }
            }
        },
23        revenue: {
            bsonType: "number",
25        },
        cust: {
27            description: "REF_1N:customer",
            bsonType: "object",
29            required: ["custkey"],
            properties: {
31                custkey: {
                    bsonType: "int"
33                }
            }
        }
35    }
37 }
,
39 {
      title: "customer",
41      description: "dimension",
      bsonType: "object",
43      required: [ "_id", "name", "address_geo"],
      properties: {
45         _id: {
            bsonType: "object",
47            required: ["custkey"],
            properties: {
49                custkey: {
                    bsonType: "int"

```

```

51         }
52     },
53 },
54 name: {
55     bsonType: "string",
56 },
57 address_geo: {
58     type: "object",
59     required: [ "type", "coordinates"],
60     properties: {
61         type: {
62             type: "string",
63             enum: ["Point"]
64         },
65         coordinates: {
66             type: "array",
67             minItems: 2,
68             items: {
69                 "type": "number"
70             }
71         }
72     }
73 },
74 },
75 ],
76 },
77 },
78 });

```

Listing B.4 – JSON Schema for the example of Figure 27

```

1 db.createCollection( "Orders", {
2     validator: { $jsonSchema: {
3         title: "lineorder",
4         description: "fact",
5         bsonType: "object",
6         required: [ "_id", "cust", "revenue"],
7         properties: {
8             _id: {
9                 bsonType: "object",
10                required: [ "orderkey", "linenumber"],
11                properties: {
12                    orderkey: {
13                        bsonType: "int"
14                    },
15                    linenumber: {

```

```

17         bsonType: "int",
18     }
19 }
20 },
21 cust: {
22     description: "REF_1N:customer",
23     bsonType: "object",
24     required: ["custkey"],
25     properties: {
26         custkey: {
27             bsonType: "int"
28         }
29     }
30 },
31 revenue: {
32     bsonType: "number",
33 }
34 }
35 } },
36 ));
37
38 db.createCollection( "Customers", {
39     validator: { $jsonSchema: {
40         title: "customer",
41         description: "dimension",
42         bsonType: "object",
43         required: ["_id", "name", "address_geo"],
44         properties: {
45             _id: {
46                 bsonType: "object",
47                 required: ["custkey"],
48                 properties: {
49                     custkey: {
50                         bsonType: "int"
51                     }
52                 }
53             },
54             name: {
55                 bsonType: "string",
56             },
57             address_geo: {
58                 type: "object",
59                 required: [ "type", "coordinates"],
60                 properties: {
61                     type: {
62                         type: "string",
63                         enum: ["Point"]

```

```

        },
        coordinates: {
            type: "array",
            minItems: 2,
            items: {
                "type": "number"
            }
        }
    }
}
} },
});

```

Listing B.5 – JSON Schema for the example of Figure 28

```

1 db.createCollection( "Orders", {
2   validator: { $jsonSchema: {
3     anyOf: [
4       {
5         title: "lineorder",
6         description: "fact",
7         bsonType: "object",
8         required: [ "_id", "revenue", "order", "commit"],
9         properties: {
10          _id: {
11            bsonType: "object",
12            required: [ "orderkey", "linenumber"],
13            properties: {
14              orderkey: {
15                bsonType: "int"
16              },
17              linenumber: {
18                bsonType: "int",
19              }
20            }
21          },
22          revenue: {
23            bsonType: "number",
24          },
25          order: {
26            description: "REF_1N:date",
27            bsonType: "object",
28            required: ["datekey"],
29            properties: {
30              custkey: {

```

```

33         bsonType: "int"
34     }
35 },
36     commit: {
37         description: "REF_1N:date",
38         bsonType: "object",
39         required: ["datekey"],
40         properties: {
41             custkey: {
42                 bsonType: "int"
43             }
44         }
45     }
46 }
47 ,
48 {
49     title: "date",
50     description: "dimension",
51     bsonType: "object",
52     required: [ "_id", "date", "dayofweek" ],
53     properties: {
54         _id: {
55             bsonType: "object",
56             required: ["datekey"],
57             properties: {
58                 datekey: {
59                     bsonType: "int"
60                 }
61             }
62         },
63         date: {
64             bsonType: "string",
65             pattern: "^[0-9][0-9][0-9][0-9]-[0-1][0-9]-[0-3][0-9]$",
66         },
67         dayofweek: {
68             bsonType: "int"
69         }
70     }
71 }
72 }
73 ]
74 } },
75 });

```

Listing B.6 – JSON Schema for the example of Figure 29

```

1 db.createCollection( "Homicides", {
3   validator: { $jsonSchema: {
4     anyOf: [
5       {
6         title: "homicide",
7         description: "fact",
8         bsonType: "object",
9         required: [ "_id", "vi", "occurrenceNum", "local", "hom_perp"],
10        properties: {
11          _id: {
12            bsonType: "object",
13            required: ["homicideId"],
14            properties:{
15              homicideId: {
16                bsonType: "int"
17              }
18            }
19          },
20          vi: {
21            bsonType: "object",
22            required: ["victimSSN"],
23            properties: {
24              victimSSN: {
25                bsonType: "int"
26              }
27            }
28          },
29          occurrenceNum: {
30            bsonType: "int",
31          },
32          local: {
33            title: "GeoJSON Point",
34            type: "object",
35            required: [ "type", "coordinates"],
36            properties: {
37              type: {
38                type: "string",
39                enum: ["Point"]
40              },
41              coordinates: {
42                type: "array",
43                minItems: 2,
44                items: {
45                  type: "number"
46                }
47            }
48          }
49        }
50      }
51    ]
52  }
53 }

```

```

    }
49  },
    hom_perp: {
51      description: "REF:perpetrator",
        type: "object",
53      required: [ "perpetratorSSN"],
        properties: {
55          perpetratorSSN: {
            type: "array",
57            minItems: 0,
            items: {
59                type: "number"
            }
61          }
        }
63      }
    }
65  },
  ,
67  {
    title: "victim",
69    description: "dimension",
    bsonType: "object",
71    required: [ "_id", "sex"],
    properties: {
73        _id: {
            bsonType: "object",
75            required: ["victimSSN"],
            properties: {
77                victimSSN: {
                    bsonType: "int"
79                }
            }
81        },
        sex: {
83            bsonType: "int"
        }
85    }
  }
87  ,
  {
89      title: "perpetrator",
        description: "dimension",
91        bsonType: "object",
        required: [ "_id", "sex", "recidivistStatus", "hom_perp"],
93        properties: {
            _id: {

```

```
95         bsonType: "object",
96         required: ["perpetratorSSN"],
97         properties: {
98             perpetratorSSN: {
99                 bsonType: "int"
100             }
101         },
102     },
103     sex: {
104         bsonType: "int"
105     },
106     recidivistStatus: {
107         bsonType: "int"
108     },
109     hom_perp: {
110         type: "object",
111         description: "REF:homicide",
112         required: ["homicideId"],
113         properties: {
114             homicideId: {
115                 type: "array",
116                 minItems: 0,
117                 items: {
118                     type: "number"
119                 }
120             }
121         }
122     }
123 }
124 ]
125 } },
126 ));
```

## APPENDIX C – IMPLEMENTATION OF ASTAR METAMODEL

Listing C.1 – Emfatic code - metamodel implementation

```

@namespace(uri="schema", prefix="astar")
2 @gmf
  package schema;
4
  @gmf.diagram(onefile="true", diagram.extension="astar")
6  class Schema {
    val Link[+] links;
8    val Box[+] boxes;
    attr String name;
10 }

12 @gmf.node(label="name", resizable="false")
  abstract class Node {
14    attr String name;
  }
16
  @gmf.node()
18  abstract class Box extends Node{
  }
20
  @gmf.node(border.width="1", border.color="0,0,0", label.icon="false")
22  class Collection extends Box {
    @gmf.compartment
24    val DocumentType[+] contains;
  }
26
  @gmf.node(border.style="solid", border.color="0,0,0", label.icon="false")
28  class DocumentType extends Box {
    attr DocumentTypeRole role;
30    @gmf.compartment(layout="list")
    val Field[+] fields;
32 }

34
  @gmf.node(label.icon="false", border.color="255,255,255")
36  class Field extends Node{
    attr DataType dataType;
38    attr FieldType fieldType;
  }
40
  @gmf.link(source="source", target="target", incoming="true", style="solid", width="1
    ", color="0,0,0")

```

```
42 abstract class Link {
    attr String name;
44     ref DocumentType[1] source;
    ref DocumentType[1] target;
46 }

48 @gmf.link(target.decoration="arrow", label="name")
    class Reference extends Link {
50     attr Cardinality cardinality;
    }
52
    @gmf.link(source.decoration="filledrhomb",target.decoration="arrow", label="name")
54 class Embed extends Link {

56 }

58 enum DataType {
    INT = 1;
60     STRING = 2;
    DATE = 3;
62     REAL = 4;
    BOOLEAN = 5;
64     POINT = 6;
    LINE = 7;
66     POLYGON = 8;
    MULTIPOINT = 9;
68     MULTILINE = 10;
    MULTIPOLYGON = 11;
70     COLLECTION = 12;
    }
72
    enum Cardinality {
74     ONE_TO_ONE = 1;
    ONE_TO_MANY = 2;
76     MANY_TO_MANY = 3;
    }
78
    enum DocumentTypeRole {
80     FACT = 1;
    DIMENSION = 2;
82 }

84 enum FieldType {
    IDENTIFIER = 1;
86     UNIQUE = 2;
    REGULAR = 3;
88 }
```

## APPENDIX D – ASTARCASE TRANSFORMATION M2T

Listing D.1 – JSON Schema generated by AStarCASE

```

1  db.createCollection( "Orders", {
3    validator: { $jsonSchema: {
4      title: "lineorder",
5      description: "FACT",
6      bsonType: "object",
7      required: [ "_id", "linenumber", "revenue", "cust" ],
8      properties: {
9        _id: {
10         bsonType: "object",
11         required: [ "orderkey" ],
12         properties: {
13           orderkey: {
14             bsonType: "int"
15           }
16         }
17       },
18       cust: {
19         description: "REF:customer",
20         bsonType: "object",
21         required: [ "custkey" ],
22         properties: {
23           custkey: {
24             bsonType: "int"
25           }
26         }
27       },
28       linenumber: {
29         bsonType: "int"
30       },
31       revenue: {
32         bsonType: "number"
33       }
34     }
35   }},
36 );
37 db.createCollection( "Customers", {
38   validator: { $jsonSchema: {
39     title: "customer",
40     description: "DIMENSION",
41     bsonType: "object",
42     required: [ "_id", "name", "email", "address" ],

```

```
43   properties: {
44     _id: {
45       bsonType: "object",
46       required: [ "custkey" ],
47       properties: {
48         custkey: {
49           bsonType: "int"
50         }
51       }
52     },
53     name: {
54       bsonType: "string"
55     },
56     email: {
57       bsonType: "string"
58     }
59   },
60   address: {
61     title: "address",
62     description: "DIMENSION",
63     bsonType: "object",
64     required: [ "address_geo" ],
65     properties: {
66       address_geo: {
67         title: "GeoJSON Point",
68         type: "object",
69         required: [ "type", "coordinates" ],
70         properties: {
71           type: {
72             type: "string",
73             enum: [ "Point" ]
74           },
75           coordinates: {
76             type: "array",
77             minItems: 2,
78             items: {
79               type: "number"
80             }
81           },
82           bbox: {
83             type: "array",
84             minItems: 4,
85             items: {
86               type: "number"
87             }
88           }
89         }
90       }
91     }
92   }
93 }
```

```
        } }  
91     }  
        }},  
93 });  
  
95 db.Customers.createIndex( "email", { unique: true } );
```