

Compact and Efficient Representations of Graphs

Autora: Sandra Álvarez García

Tesis doctoral UDC / 2014

Directores:
Nieves Rodríguez Brisaboa
Mauricio Marín Caihuan

Departamento de Computación



PhD thesis supervised by
Tesis doctoral dirigida por

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
brisaboa@udc.es

Mauricio Marín Caihuan
Departamento de Ingeniería Informática
Universidad de Santiago de Chile
Av Libertador Bernardo O'Higgins 3363 Santiago (Chile)
Tel: +56 2 2718 0000
mauricio.marin@usach.cl

Nieves Rodríguez Brisaboa y Mauricio Marín Caihuan, como directores, acreditamos que esta tesis cumple los requisitos para optar al título de doctor internacional y autorizamos su depósito y defensa por parte de Sandra Álvarez García cuya firma también se incluye.

A mi familia

Agradecimientos

Quiero agradecer en primer lugar a mis directores por guiarme a lo largo de estos años. A Nieves por haberme dado la oportunidad de realizar esta tesis y de poder formar parte del LBD. Gracias por tu confianza y paciencia, y por tantas horas invertidas en darme ideas, consejos y orientaciones. Y a Mauricio gracias por toda tu ayuda en la distancia, pero también durante mi estancia en Santiago de Chile.

A lo largo de este camino he conocido a muchas personas con las que he tenido la oportunidad de trabajar. Gracias a Gonzalo Navarro por tu generosidad y tus ideas. Gracias a Miguel A. Martínez y Javier Fernández por todo lo que he aprendido con vosotros.

Gracias a todas las personas que componen el LBD y Enxenio, porque es una suerte trabajar en un ambiente tan bueno. Me gustaría dar un agradecimiento especial a Susana, por los viajes que hemos compartido, por escucharme tantas veces y por todos los ánimos y apoyo. A Fani por haber podido trabajar en TweetGalego contigo. A Guillermo por todas las ideas aportadas. A Diego, por ser mi consejero en la distancia. Y con un cariño especial, a Leti y a Javi, porque es un placer trabajar con amigos así, gracias por tantos buenos momentos compartidos y por no haberme dejado sola en ninguno de los malos. Nada de esto sería lo mismo sin vosotros.

Gracias a los amigos de la carrera, a los que siguen en Vigo y a los nuevos que han ido apareciendo. Y quiero agradecer especialmente a toda mi familia por su comprensión, apoyo y confianza. A mis abuelos, a mis primos, a mis tíos y a mi ahijada por estar siempre ahí. Gracias a mis padres y a mi hermano por escucharme, entenderme y apoyarme siempre. No sería quien soy sin todo lo que he aprendido y vivido con vosotros. Gracias por entender estos años de sacrificio y de estar lejos. Gracias por haberlos compensado de tantas formas. Me siento muy afortunada de tener la familia que tengo. También quiero dar las gracias a mi nueva familia de Coruña, por hacerme sentir como en casa.

Por último, quiero dar las gracias a Sergio, porque no cabe en este párrafo todo lo que me gustaría agradecerte. Nadie mejor que tú sabe cuánto ha costado llegar

hasta aquí, y sé que no podría haber encontrado mejor compañero de viaje. Gracias por creer que era posible, por darme las fuerzas cuando no quedaban y sobre todo, por creer en mí.

Abstract

In this thesis we study the problem of creating compact and efficient representations of graphs. We propose new data structures to store and query graph data from diverse domains, paying special attention to the design of efficient solutions for attributed and RDF graphs.

We have designed a new tool to generate graphs from arbitrary data through a rule definition system. It is a general-purpose solution that, to the best of our knowledge, is the first with these characteristics. Another contribution of this work is a very compact representation for attributed graphs, providing efficient access to the properties and links of the graph. We also study the problem of graph distribution on a parallel environment using compact structures, proposing nine different alternatives that are experimentally compared. We also propose a novel RDF indexing technique that supports efficient SPARQL solution in compressed space. Finally, we present a new compact structure to store ternary relationships whose design is focused on the efficient representation of RDF data.

All of these proposals were experimentally evaluated with widely accepted datasets, obtaining competitive results when they are compared against other alternatives of the State of the Art.

Resumen

En esta tesis estudiamos el problema de la creación de representaciones compactas y eficientes de grafos. Proponemos nuevas estructuras para persistir y consultar grafos de diferentes dominios, prestando especial atención al diseño de soluciones eficientes para grafos generales y grafos RDF.

Hemos diseñado una nueva herramienta para generar grafos a partir de fuentes de datos heterogéneas mediante un sistema de definición de reglas. Es una herramienta de propósito general y, hasta nuestro conocimiento, no existe otra herramienta de estas características en el Estado del Arte. Otra contribución de este trabajo es una representación compacta de grafos generales, que soporta el acceso eficiente a los atributos y aristas del grafo. Así mismo, hemos estudiado el problema de la distribución de grafos en un entorno paralelo, almacenados sobre estructuras compactas, y hemos propuesto nueve alternativas diferentes que han sido evaluadas experimentalmente. También hemos propuesto un nuevo índice para RDF que soporta la resolución básica de SPARQL de forma comprimida. Por último, presentamos una nueva estructura compacta para almacenar relaciones ternarias cuyo diseño se enfoca a la representación eficiente de datos RDF.

Todas estas propuestas han sido experimentalmente validadas con conjuntos de datos ampliamente aceptados, obteniéndose resultados competitivos comparadas con otras alternativas del Estado del Arte.

Resumo

Na presente tese estudiamos o problema da creación de representacións compactas e eficientes de grafos. Para isto propoñemos novas estruturas para persistir e consultar grafos de diferentes dominios, facendo especial fincapé no deseño de solucións eficientes nos casos de grafos xerais e grafos RDF.

Deseñamos unha nova ferramenta para a xeración de grafos a partires de fontes de datos heteroxéneas mediante un sistema de definición de regras. Trátase dunha ferramenta de propósito xeral e, até onde chega o noso coñecemento, non existe outra ferramenta semellante no Estado do Arte. Outra das contribucións do traballo é unha representación compacta de grafos xerais, con soporte para o acceso eficiente aos atributos e aristas do grafo. Así mesmo, estudiamos o problema da distribución de grafos nun contorno paralelo, almacenados sobre estruturas compactas, e propoñemos nove alternativas diferentes que foron avaliadas de xeito experimental. Propoñemos tamén un novo índice para RDF que soporta a resolución básica de SPARQL de xeito comprimido. Para rematar, presentamos unha nova estrutura compacta para almacenar relacións ternarias, cun deseño especialmente enfocado á representación eficiente de datos RDF.

Todas estas propostas foron validadas experimentalmente con conxuntos de datos amplamente aceptados, obténdose resultados competitivos comparadas con outras alternativas do Estado do Arte.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions	2
1.3	Structure	6
2	Previous concepts	9
2.1	Bitmap representations	10
2.1.1	Rank and Select operations	10
2.1.2	First approaches	10
2.1.3	Compressed bitmap representations	12
2.2	Directly Addressable Variable-Length Codes	12
2.2.1	Encoding schema	13
2.2.2	Building DAC	13
2.2.3	Accessing DAC	16
2.2.4	Analysis	16
2.3	K^2 -tree	17
2.3.1	Conceptual representation	17
2.3.1.1	Building the K^2 -tree	17
2.3.1.2	Querying the K^2 -tree	18
2.3.2	Data structure	19

2.3.3	Analysis	20
2.3.4	Optimizations of the K^2 -tree	21
2.3.4.1	Hybrid approach	21
2.3.4.2	Leaves compression with DAC	21
2.4	Basic concepts of Graph Theory	22
2.4.1	Graph definitions	22
2.5	Summary	26
I	General graphs	27
3	GraphGen: a tool for graph generation	29
3.1	Motivation and related Work	30
3.2	GraphGen goals	31
3.3	Graph Generation Model	32
3.3.1	GraphGen Model	32
3.3.1.1	Nodes	32
3.3.1.2	Edges	34
3.3.2	Derivation or Decomposition Rules	34
3.3.3	Relation rules	36
3.3.4	Graph Model vs Instance Graph	39
3.4	The GraphGen tool	39
3.4.1	Load Rules	41
3.4.2	Predefined extraction functions	41
3.4.3	Filtering nodes and edges	42
3.5	Example of use cases	42
3.5.1	Bibliographic dataset DBLP	43
3.5.1.1	Dataset	43
3.5.1.2	Graph definition	43
3.5.1.2.1	Derivation rules	44

3.5.1.3	Graph generation and filtering	47
3.5.2	Query Log Analysis	48
3.5.2.1	Dataset	48
3.5.2.2	Graph definition	48
3.5.2.2.1	Load rule	48
3.5.2.2.2	Derivation rule	48
3.5.2.2.3	Relation rules	48
3.5.2.3	Description of the generated graph	49
3.5.3	Social Network Analysis	49
3.5.3.1	Dataset	51
3.5.3.2	Graph definition	51
3.5.3.3	Description of the generated graph	52
3.5.4	Spatial and temporal efficiency	52
3.6	Implementation details of GraphGen	53
3.6.1	Data Model	53
3.6.2	Architecture and Design	57
3.6.3	Graph generation	60
3.6.4	Memory management	63
3.6.4.1	XML assistance	63
3.6.4.2	Temporal data in main memory	64
3.7	Summary	64
4	Representing Graphs with attributes using K^2-trees	65
4.1	Systems for Attributed Graphs	65
4.1.1	DEX	66
4.1.2	Neo4J	68
4.1.3	HyperGraph	69
4.1.4	Other systems	70
4.1.5	Blueprints Graph API	70

4.2	Our proposal: Att K^2 -tree	70
4.2.1	Graph model supported by Att K^2 -tree	71
4.2.1.1	Formal definition	72
4.2.2	Data structure	75
4.2.2.1	Schema	76
4.2.2.2	Data	76
4.2.2.3	Relations	79
4.3	Navigation and operations	81
4.3.1	Operations over the Schema	81
4.3.2	Operations over the Data	82
4.3.3	Operations over the Relationships	83
4.4	Experimental evaluation	84
4.4.1	Experimental Framework	85
4.4.1.1	Tools	85
4.4.1.2	Queries	85
4.4.1.3	Datasets	86
4.4.2	Analysis	91
4.5	Summary	92
 II Graph distribution		93
 5 State of the Art		95
5.1	The Graph Partitioning problem	96
5.1.1	A brief definition	96
5.1.2	Measuring the quality of the partition	96
5.2	Graph partitioning strategies	97
5.2.1	Algorithms based on geometric information	97
5.2.1.1	Coordinate bisection	97
5.2.1.2	Inertial bisection	98

5.2.1.3	Geometric bisection	98
5.2.2	Structural algorithms	99
5.2.2.1	Refinement algorithms	99
5.2.2.2	Greedy algorithms	100
5.2.2.3	Spectral algorithms	101
5.2.2.4	Multilevel algorithms	102
5.3	Matrix partitioning strategies	104
5.3.1	1D partitioning	105
5.3.2	Rectilinear partitioning	107
5.3.3	Jagged partitioning	108
5.3.4	Disk allocation methods	110
5.4	Summary	112
6	Our proposal	113
6.1	Graph partitioning with the K^2 -tree structure	113
6.2	Proposals based on the Adjacency Matrix Partitioning	115
6.2.1	Block distribution	115
6.2.1.1	Adaptive block distribution	118
6.2.2	Cyclic distribution	119
6.2.2.1	Adaptive cyclic distribution	120
6.2.3	Basic grid distribution	122
6.2.4	Multilevel grid distribution	126
6.2.4.1	Adaptive multilevel grid distribution	128
6.3	Proposals based on the K^2 -tree structure	130
6.3.1	Edge-Balanced distribution	130
6.3.2	Perfect Spatial Balanced distribution	132
6.3.3	Querying the Spatial Balanced partitions	133
6.4	Latin-Square partition	135
6.4.1	Latin Squares	136

6.4.2	Algorithm of distribution	136
6.5	Execution Cycle	139
6.6	Summary	141
7	Experimental evaluation	143
7.1	Experimental Setup	144
7.1.1	Execution environment	144
7.2	Spatial evaluation	145
7.2.1	Total spatial cost	145
7.2.2	Spatial efficiency	148
7.3	Temporal evaluation	151
7.3.1	Temporal efficiency	151
7.3.2	Speed-up	156
7.3.3	Analysis	156
7.4	Summary	162
III	Representation of RDF Graphs	165
8	Representing RDF graphs with the K^2-tree: K^2-Triples	167
8.1	RDF graphs	168
8.1.1	The Web of Data	168
8.1.2	RDF as a data representation language	170
8.1.3	SPARQL as a query language	172
8.2	RDF stores	175
8.2.1	Relational Solutions	175
8.2.1.1	Single three-column table	175
8.2.1.1.1	Virtuoso	176
8.2.1.2	Property tables	176
8.2.1.3	Vertical partitioning	177

8.2.2	Native Solutions	180
8.3	K^2 -triples structure	185
8.3.1	Dictionary encoding	185
8.3.2	Vertical partitioning	187
8.3.3	Indexing predicates	189
8.4	Triple pattern resolution in K^2 -triples	192
8.5	Join Resolution in K^2 -triples	195
8.5.1	Join classification	196
8.5.2	Join algorithms	197
8.5.3	Join implementation	200
8.5.3.0.1	Joins with no variable predicates	200
8.5.3.0.2	Joins with one variable predicate	201
8.5.3.0.3	Joins with two variable predicates	202
8.6	Experimentation	202
8.6.1	Experimental Setup	203
8.6.1.0.4	Datasets	203
8.6.1.0.5	RDF Stores	205
8.6.1.0.6	Queries	206
8.6.2	Compression Results	206
8.6.3	Query Performance	208
8.6.3.0.7	Triple patterns	208
8.6.3.0.8	Joins	211
8.7	Summary	216
9	Interleaved K^2-tree	223
9.1	Our proposal	224
9.1.1	Data structure	225
9.1.1.1	Leaves compression with DAC	228
9.1.2	Navigation	228

9.1.2.1	Query patterns with bounded partitioning variable .	229
9.1.2.2	Query patterns with unbounded partitioning variable	230
9.1.3	Analysis	231
9.2	Experimental evaluation	232
9.3	A lazy evaluation	233
9.4	Summary	237
10	Conclusions and Future work	241
10.1	Summary of contributions	241
10.2	Future work lines	242
A	Publications and other research results	245
B	Descripción del Trabajo Realizado	249
B.1	Modelado de grafos con GraphGen	250
B.2	Grafos con atributos	251
B.3	Representación de grafos distribuidos	253
B.4	Representación de grafos RDF: K^2 -triples	254
B.5	Representación de relaciones ternarias	255
	Bibliography	257

List of Figures

2.1	The two-level directory proposed by Jacobson	11
2.2	Encoding of a sequence of integers using Direct Addressable Codes	14
2.3	Example of the K^2 -tree structure	17
2.4	An example of different kind of graphs	25
3.1	Example of a graph generated from a collection of documents	35
3.2	Example of a word extraction through a tree rule (left) and a graph rule (right)	37
3.3	Example of a model graph (top) and a instance graph created with this model graph (bottom)	40
3.4	Example of subgraph generated in GraphGen from DBLP data	44
3.5	Load rule definition for DBLP dataset	45
3.6	Derivation rule to extract the authors of the articles in DBLP	45
3.7	Graph model in GraphGen	46
3.8	Graph model in GraphGen for the Query Log dataset	49
3.9	Space analysis in GraphGen	54
3.10	Temporal results for GraphGen	54
3.11	Data Model of GraphGen	55
3.12	GraphGen architecture	59
3.13	Dependencies between the services and Data Access Objects of GraphGen	61

4.1	DEX internal representation (bottom) for a labelled attributed graph (top)	67
4.2	An example of hypergraph model including n -ary edges	69
4.3	Example of labelled, directed attributed multigraph	73
4.4	Internal representation of Schema and Data subsystems in $AttK^2$ -tree	77
4.5	Multi-subsystems in $AttK^2$ -tree	80
4.6	Attributed graph representing Movielens 100K dataset	87
4.7	Attributed graph representing Movielens 10M dataset	87
4.8	Temporal results obtained for operations over the schema in Movielens dataset	89
4.9	Temporal results obtained for operations over the data in Movielens dataset	90
4.10	Temporal results obtained for operations over the links in Movielens dataset	91
5.1	Recursive coordinate bisection in 4 regions	98
5.2	First iteration of the Kernighan and Lin Algorithm for an unweighted graph	100
5.3	First coarsening (right) of a given graph (left)	104
5.4	Uniform Block Distribution (top) and row-wise block stripping (bottom)	106
5.5	Three first steps of the iterative refinement process for a rectilinear partitioning	107
5.6	Three different jagged partitioning strategies	109
5.7	Latin Square (top left), linear (top right), lattice (bottom left) and Coordinate Modulo Distribution (bottom right) allocation	111
6.1	An example of horizontal block partitioning for 4 processors in a graph of 16 nodes, including the maps between the global and the local adjacency matrices	116
6.2	An example of adaptive block partitioning for 4 processors in a graph of 16 nodes	118

6.3	An example of horizontal cyclic partitioning for 4 processors in a graph of 16 nodes, including the maps between the global and the local adjacency matrices	121
6.4	An example of basic grid partitioning (top) partitioning for 4 processors in a graph of 16 nodes and a multiple grid partitioning with L=2 (bottom)	127
6.5	An example of adaptive grid partitioning for 4 processors in a graph of 16 nodes for a edge cut 6	129
6.6	An example of an edge-balanced partition (bottom) for an adjacency matrix of 16 nodes (top), where z-ordering for first cells is also shown	131
6.7	An example of a perfect spatial balanced distribution (bottom) for an adjacency matrix of 16 nodes (top), where relative cost C' for each edge is shown	134
6.8	An example of a Latin Square partition showing the final assignation grid (bottom-left) for an adjacency matrix of 16 nodes (top-left), using a Latin Square template (top-right)	138
6.9	An example of three steps in the execution cycle for two processors and the external communication involved between the two processors needed to complete the queries received in the step i)	140
7.1	Epinions and Livejournal total space	146
7.2	EU and UK total space	147
7.3	Epinions and Livejournal spatial efficiency	149
7.4	EU and UK spatial efficiency	150
7.5	Epinions and Livejournal temporal efficiency for direct neighbors	152
7.6	EU and UK temporal efficiency for direct neighbors	153
7.7	Epinions and Livejournal temporal efficiency for reverse neighbors	154
7.8	EU and UK spatial temporal efficiency for reverse neighbors	155
7.9	Epinions and Livejournal speed-up for direct neighbors	157
7.10	EU and UK speed-up for direct neighbors	158
7.11	Epinions and Livejournal speed-up for reverse neighbors	159
7.12	EU and UK spatial speed-up for reverse neighbors	160

8.1	Web of Data in September of 2011	169
8.2	Web of Data in September of 2011	171
8.3	Vertical partitioning proposed by SW-store engine (top) and its alternative implementation in single column (bottom)	179
8.4	Internal representation of the six indices in Hexastore. List of objects shared by SPO and PSO indexes are shown	181
8.5	3D cube model proposed in BitMat (top) and an extract of its physical representation with run-lengths (right-bottom)	184
8.6	Example of the simple dictionary encoding for a RDF dataset	187
8.7	Internal representation of K^2 -triples system using K^2 -trees	188
8.8	SP and OP indexes for K^2 -triples	191
8.9	Simple pattern resolution	193
8.10	196
8.11	Classification of the different pair joins	198
8.12	Solution time (in milliseconds) for triple patterns in jamendo and dbpedia (warm scenario).	210
8.13	Solution time (in milliseconds) for triple patterns in jamendo.	212
8.14	Solution time (in milliseconds) for triple patterns in dblp.	212
8.15	Solution time (in milliseconds) for triple patterns in geonames.	212
8.16	Solution time (in milliseconds) for triple patterns in dbpedia.	212
8.17	Solution time (in milliseconds) for joins A-E2 in dbpedia (warm scenario).	213
8.18	Solution time (in milliseconds) for joins F-H in dbpedia (warm scenario).	214
8.19	Solution time (in milliseconds) for joins A-E2 in jamendo (warm scenario).	217
8.20	Solution time (in milliseconds) for joins F-H in jamendo (warm scenario).	218
8.21	Solution time (in milliseconds) for joins A-E2 in dblp (warm scenario).	219
8.22	Solution time (in milliseconds) for joins in dblp (warm scenario).	220

8.23	Solution time (in milliseconds) for joins A-E2 in geonames (warm scenario).	221
8.24	Solution time (in milliseconds) for joins F-H in geonames (warm scenario).	222
9.1	Vertical partitioning over a labelled multigraph	225
9.2	A ternary relation represented with the Interleaved K^2 -tree	226
9.3	First step of the top-down traversal of the lazy evaluation	237
9.4	Reaching the leaf level of the lazy evaluation	238
9.5	Starting the bottom-up traversal of the lazy evaluation	238
9.6	Reaching the root in lazy evaluation	239

List of Tables

3.1	Statistical information of the generated coauthorship graph	47
3.2	Statistical information of the generated query graph	50
3.3	Statistical information of the generated social graph	52
4.1	Spatial results obtained for Movielens dataset	88
7.1	Web and social graphs used in this experimentation	145
8.1	Summary of joins resolution in k^2 -triples (* means that removing duplicates is required for join resolution)	203
8.2	Statistical dataset description	204
8.3	Space requirements (all sizes are expressed in MB)	207
8.4	Solution time (in milliseconds) for the patten $(?,P,?)$ on dbpedia (warm scenario)	209
9.1	Space comparison for different RDF datasets (in MB)	233
9.2	Time evaluation of simple patterns for RDF, in μs per result	234
9.3	Time, in μs per result, of basic and lazy evaluation in patterns with unbounded predicate on DBpedia	237

List of Algorithms

6.1	Adaptive Block Limits Algorithm	123
6.2	Adaptive Cyclic Limits Algorithm	124
6.3	Adaptive Cyclic Mapping Algorithm	125
6.4	Perfect Spatial Balanced Algorithm	133
6.5	Latin-Square distribution	137
6.6	Data distribution for the Latin-Square approach	137

Chapter 1

Introduction

1.1 Motivation

Graphs are a natural way for modelling data in such domains where the most relevant information relies on the relationships between the entities. Some representative examples are the Web graphs or the Social networks. Last years, many research lines have emerged focusing on the analysis of data showing this graph nature. Furthermore, in the Big Data Era, huge volumes of data are generated every day. This information needs to be stored and processed efficiently in terms of space and time. In this context, the design of new compact graph representations that can be accessed efficiently has become an important research field. The work presented in this thesis deals with this new problem, proposing indexing techniques to efficiently represent graphs from different domains and characteristics, in a single processor but also in distributed environments.

This thesis presents the results of the research developed in different areas related to the representation of graphs in an efficient way. A research line has emerged due to the need of graph generation in many domains that, even though the information fits very well with a graph structure, the data is available in other common formats (for instance, in XML). In this context, an automatic tool to generate graphs from arbitrary data that follows the model specified by the user could improve this common but currently tedious task.

In many cases, the graph models used to represent the relevant information of a domain are simple directed graphs. Many compact and efficient proposals have appeared to represent this kind of graphs. However, in many cases, this model is not enough, because nodes and edges contain complex information. Last years, Graph

Databases emerged to represent and efficiently query graphs with attributes. This is an important research line that we explore in this thesis, focused on the compact representation of attributed graphs.

Many of the domains that can be represented as a graph include information generated from the activity of the users on the Internet (for instance, graphs representing the information of a query log), so the number of nodes and edges of the resulting graphs can be huge. In this context, distributed environments have been applied in this area to deal with these special requirements. This thesis studies the problem of the graph distribution using compact structures.

This work also analyzes the problem of the representation of a specific kind of graphs, RDF, which has become the standard of the Open Linked Data. The design of solutions for a specific domain requires the implementation of efficient algorithms to answer the most frequently required queries in this domain.

1.2 Contributions

In this thesis we propose new structures and algorithms to deal with the representation of different kind of graphs, focusing on the compression of the data while providing efficient access to the information. Next, we describe each of the five main contributions of this thesis.

Graph modelling: GraphGen

In real applications, even though the nature of the data fits well with a graph, this information usually is represented using standard approaches such as XML or even relational databases. For instance, query logs are usually provided as a list of structured records, where each record represents an action (a click) of a user in the search engine. In many cases, the transformation of this data into a graph is not trivial. Notice that it is usually possible to create very different graphs from the same source of information. Just to mention some examples, query logs can be modelled as click graphs, url cover graphs, link graphs, etc. Therefore, users must define the most useful graph model to represent their data. This very common task carried out to exploit a new source of information, usually called Graph Modelling, consists in transforming the data source according to a given graph model. Some solutions were proposed in the State of the Art to extract a graph from datasources of a specific domain.

To the best of our knowledge, no general graph generation tools are available in the State of the Art. Most of these solutions are ad-hoc designed to solve the transformation of data from a specific domain and format. In this thesis we propose

a general purpose tool that aims to fill this gap in the State of the Art. Our tool *GraphGen* allows users to generate graphs from arbitrary semistructured input data.

GraphGen was designed as a general purpose application, applicable to any kind of data. The purpose of *GraphGen* is to provide a standard mechanism to generate new information structured like a graph from arbitrary data. We have designed a rule generation model that gives theoretical support to the practical application we developed. *GraphGen* generates labelled directed graphs from any kind of input datasource (supporting formats like XML or CSV). The way in which the final graph is generated is defined by the user through a rule definition system, which allows the user to progressively decompose the original data into simpler contents which will be represented as nodes. Additional relationships can be defined to relate different nodes according to the structural properties of the graph that is being generated.

The versatility of *GraphGen* was proved with its application in three different use cases: the generation of co-authorship graphs from bibliographic databases, the transformation of query logs into query graphs and, finally, the creation of social graphs. The result of this work was published in an international workshop [ÁGBYB⁺12] and finally in the Journal of Systems and Software indexed in JCR [ÁGBYB⁺14].

Property graphs: Att K^2 -tree

We have already noted that many data from real contexts can be modelled as a graph. A simple directed graph is used to model many domains, like Social Networks or Web graphs. However, this kind of graphs is not enough when additional data has to be associated to the nodes (and even to the edges) of the graph. These domains where nodes and edges include a set of attributes (key/value) define a new model of graphs. They are graphs with attributes, which are usually called attributed graphs or property graphs.

Last years, Graph Database Models emerged to give a theoretical support to the Attributed Graphs. These new models are characterized by representing the schema, the data, the queries and the results as a graph. Built over those theoretical models, many practical Graph Databases Engines have been proposed. DEX or Neo4J are two relevant examples.

Given that the amount of information that these graph database engines have to manage, the design of efficient and compact structures to represent property graphs improves their management and querying. We propose a compact structure to store attributed graphs, whose internal representation is based on the K^2 -tree, a static structure designed to represent simple directed graphs (binary relationships) in main memory. Our goal was to study the possibility of extending this compact structure

to obtain a very compact representation of attributed graphs which supports efficient access to the attributes of the nodes and edges of the graph.

The result of this study is $\text{Att}K^2$ -tree, a compact structure to store attributed graphs based on the representation of binary relations in a very compact way using the K^2 -tree structure. We evaluate the performance of this structure and we study some fields of application for this static but compact attributed graph representation. We compare our proposal against another attributed graph representations in the State of the Art, obtaining the best compression results and competitive temporal results for basic query operations. The results of this work were published in a national conference [ÁGBLP10a] and an international workshop [ÁGBLP10b].

Graph distribution using K^2 -trees

Last years, distributed environments have appeared as a solution to manage the huge amounts of data that are produced every day, where the requirements of processing exceed the capacity of the mono-processor environments. Parallelism seems to be a good strategy to deal with these scalability and efficiency issues.

The application of parallel techniques in graph mining opens a new research area, whose main goal is to obtain a good graph partitioning scheme in terms of efficiency and load balance. Many algorithms were proposed in the State of the Art aiming to obtain a good partitioning of the data.

The third contribution of this thesis is the design and implementation of several partitioning algorithms to distribute a simple graph in multiple processors, supporting direct and reverse neighbor operations. We store the different partitions using the K^2 -tree data structure (and some additional information), in order to obtain a distributed but compact and efficient graph representation. We initially represent the graph in an adjacency matrix. Then, we propose different ways of partitioning this matrix, allocating a set of cells of this matrix for each processor. Therefore, we do not follow a classic graph partitioning strategy, where nodes of the graph are distributed. Instead, we propose several edge distributions.

Some of our graph partitioning algorithms are based on classical approaches originally designed to parallelize the matrix multiplication. We also propose some adaptive distributions designed to adapt the graph partitioning depending on the distribution of the edges along the adjacency matrix. A different kind of algorithms that we propose consists in the partitioning of the K^2 -tree using vertical divisions (from the root to the leaves). They are focused on obtaining a good spatial balance taking into account the structural properties of the K^2 -tree. Finally, partitioning algorithms based on the use of the Latin Square mathematical concept were studied. We analyze and compare the temporal and spatial efficiency of the

different partitioning algorithms proposed. The results of this work were published in the *20th International Symposium (SPIRE 2013)* [ÁGBGPM13].

Representing RDF graphs: K^2 -triples

The Web of Data emerged to deal with the management of the huge amount of information which is daily generated on the Internet. It supports the principles of the Semantic Web and it proposes a new data publishing format, available for machine processing and also supporting the connectivity between heterogeneous data sources. In this context, the Resource Description Framework (RDF) provides a common language to describe facts of the world in a structured and linked way. It provides a description framework that structures and links data as a set of triples (*subject, predicate, object*). A RDF dataset can be modelled as a labelled graph, where the subject and the object are the nodes and the predicate is the labelled edge that connects them.

Given the increasing importance of RDF, many specific RDF querying engines appeared in the State of the Art. Hexastore, RDF-3X, Virtuoso or Jena are representative examples of complete systems to store RDF graphs supporting SPARQL, the native query language designed specifically to access RDF data.

An important contribution of this thesis consists in a new technique to store RDF datasets in a very compact way in the main memory, providing at the same time efficient query algorithms. It follows the vertical partitioning approach, a very common strategy in the State of the Art about RDF stores. K^2 -triples represents an RDF dataset as $|P|$ binary relationships. Each binary relationship represents the relations between subjects and objects for a different predicate. Those binary relations are represented using the compact K^2 -tree data structure. Additionally to those multiple K^2 -trees, K^2 -Triples includes some additional indexes that reduce the main weakness of the vertical partitioning approaches: a poor efficiency in queries with unbounded predicate (that is, queries which do not specify a particular predicate). A basic triple pattern resolution is provided and different join resolution strategies are implemented. The set of basic query patterns that K^2 -triples implements efficiently is expected to be the basis for more complex queries, since an efficient SPARQL resolution strongly depends on the efficiency of the basic triple patterns. This structure was evaluated against other representative approaches in the State of the Art, obtaining very competitive results. We published the results of this investigation in the *Americas Conference on Information Systems (AMCIS 2011)* [ÁGBFMP11] and in *Knowledge and Information Systems indexed in JCR* [ÁGBF⁺14]

Representing ternary relationships: Interleaved K^2 -tree

An RDF graph can be modelled as a labelled graph, but it can also be considered as a ternary relationship. Many other domains are ternary relationships. For instance, raster data stores a value k for each cell (i, j) . Several approaches were followed to represent ternary relations in the State of the Art. Most of them are focused on solving the characteristics of a specific domain.

In this thesis we propose the Interleaved K^2 -tree, a compact structure to represent ternary relations in a very compact and efficient way. This new structure is an evolution of the K^2 -tree and it provides indexing capabilities over the three dimensions. However, it is specially designed for data where the three dimensions are not equally sized, with one dimension smaller (with a smaller number of different values) than the others. We design and implement this new structure and some querying algorithms. We also propose, for such relationships where the third dimension has a large number of different values, a lazy evaluation which improves the results in some kind of queries. This work was published in *Proceedings of the 2014 Data Compression Conference (DCC 2014)* [ÁGBdBN14].

1.3 Structure

This thesis is organized as follows. Chapter 2 describes the previous concepts that are used along our work. First, we present some bitmap representations. Then, two compact structures of special interest for our work are explained in detail: the Directly Addressable Variable-Length Codes and the K^2 -tree. This chapter ends with a review of the basic concepts in Graph Theory.

The structures and algorithms we propose in this thesis are organized in three main parts.

Part one describes two different proposals for general graphs. Chapter 3 proposes a tool to generate basic multi-partite graphs obtained from associations found in arbitrary data. This chapter presents the theoretical graph model underlying this tool. It also describes the tool we have developed, providing implementation details and showing its behavior with three examples of use cases.

Chapter 4 proposes a compact data structure to store graphs with attributes (*attributed graphs*) based on the K^2 -tree. This chapter includes a review of the State of the Art in Graph Databases. Then, we propose the structure and a set of operations, which are evaluated with graph datasets. We compare our results against other proposals in the State of the Art.

Part two studies the graph distribution problem using the K^2 -tree structure. Chapter 5 describes the graph partitioning problem, reviewing classical partitioning algorithms in the State of the Art and then, matrix partitioning techniques, which set the basis of our work.

Chapter 6 proposes several partitioning algorithms to distribute a simple graph in multiple processors, supporting direct and reverse neighbor operations. Each partition is stored using a K^2 -tree data structure.

Chapter 7 studies and compares the graph partitioning proposals in terms of temporal and spatial results. The overall performance is evaluated, and the spatial and temporal balance is also studied.

Part three is focused on the representation of RDF graphs. Chapter 8 proposes K^2 -triples, a new technique to store RDF datasets in a very compact way in the main memory, providing at the same time efficient query algorithms. It follows the vertical partitioning approach, a very common strategy in the State of the Art about RDF stores. This chapter reviews the State of the Art in RDF stores, explains the structure we propose and compares it against other relevant proposals.

Chapter 9 proposes a compact structure to store and query ternary relationships. This new structure is an evolution of the K^2 -tree and it provides indexing capabilities over the three dimensions. The performance of this structure is evaluated in different RDF graphs.

Chapter 2

Previous concepts

The design of new succinct data structures to support the storage and exploitation of different kinds of data has been a prolific research line, aiming to provide efficient access using the least space. In this chapter we review some of these succinct structures of the State of the Art that are specially significant for our work. They are the basis of many of the structures proposed.

Section 2.1 introduces the most relevant bitmap representations, analyzing their temporal and spatial cost. The two basic operations over a bitmap, *Rank* and *Select*, are defined and their implementation in the different structures is described. Details about the practical implementation of bitmaps with rank and select access used along this thesis are also given.

Section 2.2 describes Directly Addressable Codes, a variable-length encoding scheme for sequences of integers, which supports direct access to the elements in the encoded sequence. Some indexes designed to improve the efficiency of our K^2 -triples in Chapter 8 are encoded with DAC.

In Section 2.3 we explain the K^2 -tree structure, a compact representation of binary relationships, which is the basis of the structures K^2 -triples and $\text{Att}K^2$ -tree presented in this thesis. The distribution of Web and Social graphs using the K^2 -tree structure is also a contribution of this thesis. Therefore, given the importance of this structure for our work, we analyze its implementation details in depth.

Finally, we review some basic concepts of Graph Theory in Section 2.4.

2.1 Bitmap representations

Bitmaps are one of the first structures that appeared in the State of the Art, as they are the basis of many other succinct structures. Many approaches have been proposed to achieve their efficient access in a compact space, improving the overall efficiency of the structures built over them. This section reviews the most relevant bitmap representations. First, the operations that a basic bitmap has to support are described. Then, the first solutions appeared in the State of the Art are presented. After that, approaches which are mainly focused on the compression of the data (with efficient access support) are detailed. Finally, the practical implementation used in many of the structures proposed in this thesis is given.

2.1.1 Rank and Select operations

A bitmap B is a sequence of n bits. We can define three basic operations over this structure:

- $rank_b(B, i)$ counts the number of times that a bit b (1 or 0) appears in B between the positions 0 and i (both included). Consider the bitmap $B = 0111001110$. Then, $rank_0(B, 4)$ counts the number of *zeros* up to the position 4, having as result $rank_0(B, 4) = 2$. On the other hand, $rank_1(B, 4) = 3$, since three *ones* appear in the binary subsequence 01110. By default, if no b is specified, we consider that $rank$ operation counts the number of *ones*.
- $select_b$, which is the complementary operation to $rank$, returns the position in which the i -th bit of kind b is located. For instance, $select_0(B, 2) = 4$ and $select_1(B, 3) = 3$. Again, if no bit value is specified, the operation $select$ searches for the i -th 1 by default.
- $access$ checks the value of a given position in the bitmap, returning a 0 or a 1. Following with the example, $access(B, 5) = 0$ and $access(B, 2) = 1$.

Many structures have been designed in order to represent a bitmap in a very compact way implementing $rank$, $select$ and $access$ operations efficiently.

2.1.2 First approaches

$Rank$ and $select$ operations were defined by Jacobson [Jac88], who also defined the first bitmap representation supporting the $rank$ operation in constant time. It is composed by a two-level directory. The first level stores the result of the computation of $rank_1(B, i - 1)$, for each i multiple of $s = \lfloor \log n \rfloor \lfloor \log \frac{n}{2} \rfloor$. Figure 2.1 shows an

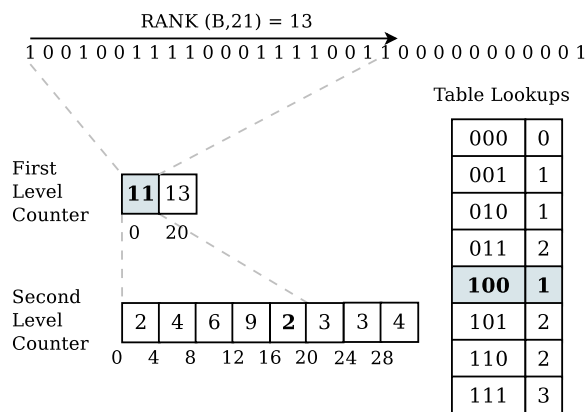


Figure 2.1: The two-level directory proposed by Jacobson

example, where the first level counter stores $rank_1(B, 19)$ and $rank_1(B, 31)$, where $s = 20$. A second level stores, for each block of the first level, several local rank results inside its corresponding block. Specifically, it computes $rank_1(S_i, j - 1)$, where S_i is the sequence of bits between two counters of the first level and j is a multiple of $b = \lfloor \log \frac{n}{2} \rfloor$. Figure 2.1 shows this second level directory. For instance, the first value of this sequence is 2, because $rank_1(S_1, 3) = 2$, where $S_1 = 1001001111000111$. Using this directory, we can solve the operation $rank_1(B, i)$. First, we compute the number of bits appearing from the beginning of the sequence to the position of the nearest multiple of s less than i (that we called p), by using the first level of the directory. Then, we use the second level to compute the number of ones from the position $p + 1$ to the nearest multiple of b less than i (which we called p') adding this count to the previous result. Finally, we use *table lookups* to compute the number of ones from position $p' + 1$ to i . Those bits are used as an index to access to one table that stores the number of ones for every combination of bits, which is shown on the right of the Figure.

The additional space needed to support *rank* operations in this solution is given by the directory and the lookup table. In the first level of the directory, the partial results for $\frac{n}{s}$ superblocks are stored. The cost of storing a counter for each of them is $\log n$ bits, so the total space of the first level is $O(\frac{n}{\log n})$ bits. In the second level, $\frac{n}{b}$ blocks are stored, spending $\log s$ bits for each one, so the total space of this second level is $O(n \frac{\log \log n}{\log n})$. Finally, the lookup table stores the *rank* values for every combination of b bits, costing $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log \log n)$ bits. Considering that partial results, the total space to support *rank* and *select* operations with this approach is $o(n)$.

In this thesis we use the practical implementation of Rodrigo González [GGMN05],

which is based on this classical solution proposed by Jacobson. Specifically, we use a solution with one level directory where the extra space can be parametrized (usually we use the 5% of the bitmap size), providing in this way an interesting space/time tradeoff.

The initial results obtained by Jacobson were improved by Clark and Munro [Cla98, Mun96]. They proposed a structure that supports *rank* and *select* operations in constant time in $n + o(n)$ bits where n bits are used to store the bitmap, while an additional structure costing $o(n)$ gives *rank* and *select* support.

2.1.3 Compressed bitmap representations

Previous approaches built additional structures to support *rank* and *select* operations. In this section, we describe new structures that store the original bitmap in a very compact way, while they provide efficient implementations of *rank*, *select* and *access* operations.

Pagh [Pag99] divides the bitmap in blocks of the same size. Each block is represented by the number of bits with value 1 they contain. It uses a compression schema that clusters adjacent blocks into intervals of varying length.

Raman *et al.* [RRR02] proposed a bitmap compression based on a numbering scheme. As in the previous approach, the sequence is divided into a set of blocks of the same size. Each block will have two values associated: c_i , which is the number of *ones* that this block contains, and o_i , which identifies the block in a vocabulary composed by the different combinations of bits. This vocabulary is sorted in the way that blocks with few *ones* or few *zeros* have shorter identifiers. Considering that the bitmap is divided into blocks with length u , storing each c_i costs $\lceil \log(u + 1) \rceil$ bits and each o_i costs $\lceil \log \binom{u}{c_i} \rceil$ bits.

Many other implementations are included in the current State of the Art. For instance, Okanohara and Sadakane [OS07] proposed a very compressed solution specially designed to manage sparse bitmaps. Another implementation specially convenient for sparse bitmaps is the *gap encoding* strategy, which encodes the gaps between the sequences of consecutive bits with value 1 [Sad03, MN07, GHSV06].

2.2 Directly Addressable Variable-Length Codes

Directly Addressable Codes (*DAC*) is a technique proposed by S. Ladra *et al.* [Lad11] [BLN13] to encode sequences of integers by using a variable-length encoding schema. Its main strength is that it provides direct access to any codeword included in the sequence, and no sampling techniques are required.

2.2.1 Encoding schema

DAC technique is based on the VByte codification [WZ99]. The encoding of a given integer n_i with VByte starts from dividing its binary representation in chunks of b consecutive bits. For instance, the integer 1571, whose binary representation is 11000100011, could be split in three chunks of $b = 4$ bits, that is, 0110, 0010, 0011 where the most significant chunk is completed with zeros (if it is necessary, in order to have 4 bits). In addition to that, the VByte representation adds an extra bit to each chunk. This bit will have value 0 for the most significant block and 1 for the remaining blocks. For instance, the final code for $n_i = 1571$ will be 001101001010011. Regarding to an optimal encoding, VByte loses one bit per chunk and, additionally, at most b bits to complete the last chunk. In exchange of that loss of compression, the decoding with VByte is very fast, in particular for chunks of 8 bits (which are called *byte codes*), taking advantage of the byte alignment.

For the sake of simplicity, the process of construction for DAC codes is given by using this VByte encoding. However, the practical implementation of DAC uses a dense coding scheme, variant of the VByte encoding, which was designed for the text compressor ETDC [BFNP07], and makes use of all the combinations of chunks, achieving better spatial results.

2.2.2 Building DAC

Based on the encoding technique described in Section 2.2.1, a sequence of integers $X = \{x_1, x_2, \dots, x_n\}$ can be encoded providing direct access to any $x_i \in X$. The construction of this index is carried out through the next three steps.

Chunk division (C) First, the binary representation of each integer x_i is divided into chunks of size $(b + 1)$ bits. Figure 2.2 shows an example of a sequence X composed by 5 integers given $b = 3$. The vector C' shows the binary representation of these integers, divided into chunks of three bits. Note that the last chunk of each integer is filled with *zeros* if necessary. For instance, the second integer of the sequence in Figure 48, is represented by three chunks $\{001, 001, 000\}$.

However, as we described in Section 2.2.1, VByte uses $b + 1$ bits to represent each chunk of b bits, where the extra bit is 0 for the most significant chunk, and 1 for the remaining chunks. Vector C in Figure 2.2 shows the chunks of $b + 1$ bits. For instance, the integer 48 is represented by $\{0001, 1001, 1000\}$, where the extra bit is 0 only for the most significant chunk.

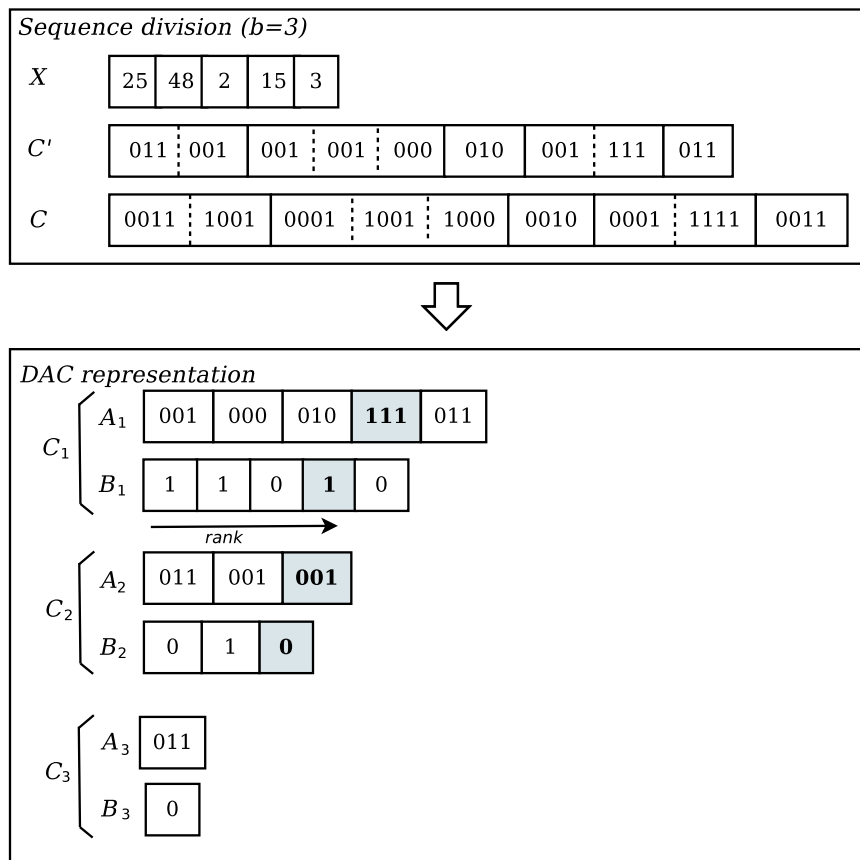


Figure 2.2: Encoding of a sequence of integers using Direct Addressable Codes

Stream representation (A) The sequence of bits of the vector C built in the previous step is structured to provide direct access. The chunks of bits are represented in streams structured by levels. In that way, a vector A_1 stores the least significant chunk for each integer in the sequence. Therefore, the i -th chunk in A_1 contains the b least significant bits of the integer x_i . Next vector, A_2 , contains the second least significant chunk of b bits, but only for that integers containing a number of chunks greater or equal to 2 in A_2 . The chunks are also ordered by the position of its corresponding integer in the sequence. The process continues until all the chunks are included in some stream A_j . Note that the number of different streams A_j is given by the maximum integer represented in the sequence.

Figure 2.2 shows the three streams $\{A_1, A_2, A_3\}$ needed to represent the sequence from the example. For instance, the three chunks of $x_2 = 48$ are represented in the elements $A_1[2], A_2[2], A_3[1]$. However, the element $x_3 = 2$ is composed by a single chunk, so it is only represented in the first stream A_1 , located in the third position ($A_1[3]$). Note that the stream A_1 always contains $|X|$ elements. However, the numbers of elements in the remaining streams is progressively decreased, in the way that $A_1 \leq A_2 \leq \dots A_n$. This property is due to the fact that if an integer has a chunk represented in A_i , it also contains a chunk in every $A_j, j \leq i$.

Chunk significance representation (B) The extra bit used in VByte to mark the most significant chunk is represented in a separated bitmap in DAC. We build as many bitmaps B_i as streams A_i , where the j -th element of bitmap B_i is represented by a 0 if the j -th element of A_i is the most significant chunk of the integer it belongs, and 1 otherwise. In other words, the value of this bitmap shows if the integer represented in that position continues to be represented in the next stream or not.

For instance, the first integer of the sequence, $x_1 = 25$, is represented in A_1 and A_2 , since its VByte code is 0011 – 1001. The first chunk, $A_1[1]$, has associated a 1 in B_1 , since it is not the last chunk of x_1 . However, $B_2[1] = 1$, because it is the most significant chunk of that integer.

In this way, the VByte encoding of the sequence of integers is finally structured with DAC through a set of streams A_i representing the chunks and a set of bitmaps B_i storing the extra bit of each chunk. An additional data structure to provide *rank* access is set up on the B_i bitmaps. These rank structures will answer a *rank* operation in constant time with $O(\frac{n_i \log \log N}{\log N})$ bits, being N the number of bits of the encoded sequence and n_i the number of bits the bitmap B_i .

2.2.3 Accessing DAC

Given the DAC structure whose building was described in Section 2.2.2, the process of extraction of the i -th integer of the sequence is as follows. First, the least significant chunk of x_i is extracted, which is always located in the position $i_1 = i$ of the first stream. Therefore, it is located in $A_1[i_1] = A_1[i]$. Then, the element $B_1[i_1]$ is checked. If it is a *zero*, the decoding of x_i is $A_1[i]$ and the process ends. Otherwise, if it is a 1, the next chunk has to be extracted and the process is repeated.

In order to obtain the position of the next chunk (in A_2), a *rank* operation is performed over B_1 , to count the number of *ones* from the position 0 to the position i . This number gives the integers that will be represented in A_2 before x_i . Therefore, we set $i_2 = \text{rank}(B_1, i_1)$, which is the location of the next chunk of x_i in A_2 . The second chunk of x_i is $A_2[i_2]$ and $B_2[i_2]$ shows if the integer has additional chunks in lower levels. If so, we set $i_3 = \text{rank}(B_2, i_2)$ and the same step is repeated. The process continues until a 1 is located in the $B_j[i_j]$ position.

Figure 2.2 highlights the elements explored to decode the fourth element of the sequence. The least significant chunk of that integer is located in $A_1[4] = 111$. Then, $B_1[4]$ has to be checked. In this case, it stores a 1, which means that the fourth element is composed by more than one chunk. So, we compute $\text{rank}(B_1, 4) = 3$, meaning the next chunk of x_4 corresponds to the third element of A_2 . We have $A_2[3] = 001$, which is the next chunk of x_i . Checking the bitmap $B_2[3] = 0$ we note that this is the last chunk of the element. So, $x_4 = 001111_2 = 15$.

2.2.4 Analysis

In the worst case, the number of accesses needed to extract an integer in the sequence is given by $\lceil \frac{\log M}{b} \rceil$, where M is the maximum integer of the sequence and b is the size of the chunks. However, when n consecutive integers are extracted, the process can be optimized. For each stream, instead of performing n rank operations, only one rank operation in each level is needed (for the first element of the searched sequence), while the remaining $n - 1$ elements are checked by a sequential exploration.

The total size of a DAC representation is $\sum_{k=1}^{L-1} n_k \cdot (b + 1 + X) + n_L \cdot b$, where L is the number of levels, b the number of bits of each chunk, n_k the number of chunks in the level k and X a parameter representing the number of extra bits per bit in B_i used to represent the rank structure, depending on the chosen implementation. The parameters b and X can be modified carrying out different space-time tradeoffs.

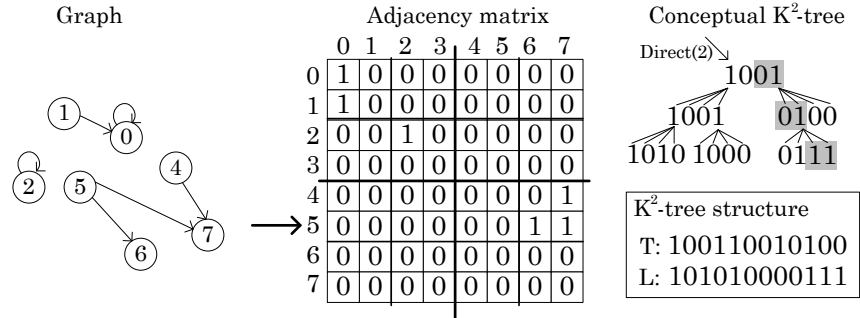


Figure 2.3: Example of the K^2 -tree structure

2.3 K^2 -tree

In this section we describe a compact structure of special importance for the work developed along this thesis, as it sets the basis for some of the structures we propose. It was designed by S. Ladra *et al.* [Lad11, BLN09]. The K^2 -tree was originally designed to store and query the links included in a Web graph in a very efficient way. It supports the most common operations in this domain, like recovering the pages pointed by a given page (direct neighbors) or recovering the pages pointing to a given page (reverse neighbors).

This structure is based on a conceptual representation of a K^2 -ary tree, whose features are explained in Section 2.3.1. The structure is finally stored in a very compact way through two bitmap structures supporting *rank* access, as it is described in Section 2.3.2. Some variations and optimizations were also proposed by their authors, described in Section 2.3.4.

2.3.1 Conceptual representation

2.3.1.1 Building the K^2 -tree

K^2 -tree structure is conceptually a K^2 -ary tree, whose construction starts from the adjacency matrix of the graph. As it is well known, a binary adjacency matrix of a simple graph (understanding simple graph as unweighted and directed) of n nodes is a matrix of size $n \times n$ where the cell (i, j) contains a 1 if and only if an edge exists in the graph starting from the node i and pointing to the node j . Otherwise, this cell will contain a 0. Figure 2.3 shows a graph of 8 nodes (left) and its corresponding adjacency matrix (middle). For instance, cell $(5, 6)$ of the adjacency matrix contains a 1 because an edge starts from 5 and points to 6 in the original graph.

The process of construction of the conceptual K^2 -tree starts by dividing the matrix into K^2 squared submatrices of size $\frac{n^2}{K^2} \times \frac{n^2}{K^2}$, following an MX-Quadtree strategy [Sam06]. The root node of the tree will represent each one of these matrices with a bit, following a left to right and top to bottom ordering. In this way, a submatrix will be represented with a 1 in the tree if it contains at least a 1 in one of its cells. Otherwise, if it is an empty matrix, it will be represented as a 0 in its corresponding position of the tree. This root of K^2 bits composes the first level of the K^2 -tree. Figure 2.3 shows the first subdivision of the adjacency matrix in 4 submatrices (for $K = 2$) and its corresponding root node (1001). First 1 of the root means that the top-left submatrix contains at least a 1. On the other hand, the second bit is a 0, meaning all the cells of the top-right submatrix (rows 0 – 4 and columns 4 – 7) are zeros.

Each not empty submatrix (represented with a 1 in the tree) will be divided again in K^2 submatrices, following the same process. Therefore, each 1 in the i -th level of the tree will have K^2 children in the $i + 1$ -th level, representing the K^2 submatrices in which this element is divided. Again, each bit denotes if its corresponding matrix is an empty matrix or not. For instance, first bit of the root in Figure 2.3 will have 4 children in the second level. The first child is represented by a 1, meaning the submatrix between the rows 0 – 1 and the columns 0 – 1 contains at least an edge. However, the second bit, which is a 0, means that the submatrix between the rows 0 – 1 and the columns 2 – 3 is empty. The process continues recursively, by subdividing each not empty matrix and representing with one bit in next level each child, until the leaf level is reached, where each bit corresponds with a cell of the tree.

Note that in order to make possible the application of this method, the number of nodes n should be restricted to a power of K . For such adjacency matrices where n is not a power of K , the matrix will be conceptually filled with rows and columns of *zeros* until the next power of K is reached. Since the empty submatrices are represented with very few bits in the tree (because they are not subdivided and represented in lower levels) this assumption will have almost no impact in the overall performance of the solution.

The height of the K^2 -tree for a graph with n nodes will have $h = \lceil \log_K n \rceil$. Consequently, lower values of K produce higher trees.

2.3.1.2 Querying the K^2 -tree

K^2 -tree was designed to represent and query Web graphs, so in this section we focus on describing how the basic operations (direct and reverse neighbors) are implemented over the conceptual K^2 -tree explained in the previous section.

Obtaining the direct neighbors of a node r is equivalent to recovering the cells

with a 1 value in its corresponding row r of the adjacency matrix. The operation consists in a top-down traversal over the tree, starting at the root and moving down the tree until reaching the leaves (or until no *ones* are found in the explored branches). For each node, at most K bits have to be checked, depending on the row we are exploring.

As an example, we describe how the direct neighbors of the node 5 are recovered for the graph of the Figure 2.3. In other words, we want to check which cells of the row 5 are set to 1. We start at the root of the tree (1001), checking the third and fourth bits, since they correspond to the bottom submatrices of size 4 (which include the row 5). The third bit is a 0, so it will be not explored in the next levels. However, the fourth bit is a 1, so we go down the tree to check its children, 0100. For that level, the row 5 is included in the top submatrices (rows 4 and 5), so we check the two first bits. The second bit is a 1, so we continue exploring the tree, reaching the leaf level (0111). The third and the fourth bits represent the cells (5, 6) and (5, 7), respectively, and they have value 1, so the final result of that operation will be $\{(5, 6), (5, 7)\}$, meaning node 6 and node 7 are direct neighbors of the node 5.

Reverse neighbor operation can be implemented in a symmetric way, as a top-down traversal over the tree in which we check K bits, at most for each explored node.

2.3.2 Data structure

The conceptual K^2 -tree explained in the previous section is represented in a very compact way by performing a level-wise traversal over the bits of the tree. Specifically, the full tree is stored using two bitmaps:

- $T(\text{tree})$ stores the bits of the root and the intermediate levels of the tree, excluding the leaf level. In order to build this bitmap, the bits of the tree are traversed by levels, from the root to the next to last level, and from left to right.
- $L(\text{leaves})$ stores the last level of the tree, where each bit corresponds with a cell of the matrix.

Figure 2.3 shows the bitmaps T and L for the same example. In this case, T stores in the first $K^2 = 4$ bits the root level, and the remaining bits correspond with the second level of the tree, from left to right. L represents the 12 bits of the leaf level. An auxiliary structure is created over the bitmap T in order to enable *rank* operations over it. As it will be shown next, query algorithms do not need to perform *rank* operations over L , so this is the main reason to store the last level separately to the rest of the levels.

Navigation over T and L

Section 2.3.1.2 explained how the conceptual K^2 -tree is performed by a top-down traversal over the tree. In this section we show how the same traversal can be executed over the bitmaps T and L and how the location of the K^2 children of an element can be directly accessed through a *rank* operation.

In order to obtain the direct neighbors of a node of the graph, the root node of the conceptual K^2 -tree has to be explored. It is located in the first K^2 bits of T . In the case of our example (recovering the direct neighbors for the node 5), the third and fourth bits of the root node are checked. They are $T[2] = 0$ and $T[3] = 1$. The next step consists in obtaining the position of the children of $T[3]$, since it is a *one*. The i -th children of any bit x in T can be computed according to the formula $T : L[\text{rank}_1(T, x)K^2 + i]$, where $T : L$ is the concatenation of the two bitmaps. Therefore, the first child of the bit 3 is in the position $\text{rank}_1(T, 3)K^2 = 8$, and, since all the children of an element are in consecutive position of the bitmap, the children of the bit 3 are in $T : L[8 \dots 11]$. The first and second children correspond with the row 5, so we check $T : L[8] = 0$ and $T : L[9] = 1$. Next step explores the children of the bit 9, through $\text{rank}_1(T, 9)K^2 = 20$, where the third ($T : L[22] = 1$) and fourth ($T : L[23]$) children are explored.

We already explained how, given a bit, the positions of its children are computed. However, another calculus has to be specified, consisting in which children of an element correspond to the row we are checking. Noting that the size of a submatrix in the level l is K^{h-l} (where h is the height of the tree), a row r_l at a matrix of level l belongs to the submatrices at row $\lfloor \frac{r}{K^{h-l-1}} \rfloor$. The relative row inside their corresponding submatrices in the level $l + 1$ is $r_l \bmod K^{h-l-1}$, where r_l is the relative row in the previous level and $r_0 = r$. Symmetrical formulas are defined for the columns of the matrix. Using these two formulas, the children that correspond to the explored row can be obtained.

2.3.3 Analysis

In the worst case (for graphs with their edges very isolated), the total space in bits of the described structure is $K^2 m (\log_{K^2} \frac{n^2}{m} + O(1))$, where n is the number of nodes of the graph and m the number of links. The extra space needed to store the cells to fill the matrix until the next power of K is given by $O(K^2 n)$. However, for real Web graphs, the space is much better than the worst case. Web graphs are not uniformly distributed, usually presenting clustering of ones and zeros that improves the worst case space by far.

The navigation time in the worst case is $O(n)$. However, in practice, it is much better. Supposing that the m links are uniformly distributed, the navigation is

$O(\sqrt{m})$ in the worst case. Furthermore, if the matrix is clustered, the average performance is even better.

Other interesting operations can be implemented with similar algorithms. For instance, we can retrieve the cells included inside a rectilinear window over the adjacency matrix. This operation is specially helpful for Web graphs. If nodes are located in the matrix following a lexicographical order, it can be used to find links between domains. The algorithm for this operation is quite similar to a direct neighbor operation, excepting by the fact that up to K^2 children per element could be explored, depending on the bounds of the given window.

2.3.4 Optimizations of the K^2 -tree

2.3.4.1 Hybrid approach

The structure was described for a fixed value of K . However, a hybrid approach to the previous structure was also proposed, which uses a larger K (for instance, $K = 4$) for the first levels of the tree and a small K ($K = 2$) for the lowest levels. Using a big K in the top levels helps to reduce the height of the tree, improving the time results of the traversals. On the other hand, using small values for K in the last levels avoids to store too many bits for each isolated 1 of the adjacency matrix (which is frequent in the lowest levels of the tree).

2.3.4.2 Leaves compression with DAC

In Section 2.3.1, we observe how the construction of the K^2 -tree divides each submatrix recursively until reaching the leaves of the tree, where each bit represents an individual cell of the adjacency matrix.

An improvement for the spatial cost of this structure consists in stopping this subdivision in previous levels, in the way that each leaf of the tree represents a submatrix of size $s \times s$ (instead a cell). Each element of the leaf level with value 1 will be a non empty matrix $s \times s$. The sequence of $s \times s$ matrices located in this level (from left to right) will be encoded with a statistical and variable-length encoding technique. Therefore, a vocabulary with the different existing $s \times s$ submatrices is created, in the way that each one will have a different code. The most frequent matrices will have the shortest codes in this matrix vocabulary. The codes of the matrix sequence corresponding to the ones in the last level of the tree, will be stored using Direct Addressable Codes (explained in Section 2.2). In this way, the i -th 1 of the leaf level will be located in the i -th position of the DAC structure that stores the code of the corresponding $s \times s$ matrix in the matrix vocabulary.

2.4 Basic concepts of Graph Theory

Graphs are a mathematical construct that have been widely studied [BM76]. Recently, other scientific areas, like the relational databases or the information systems in computer science, have paid special attention to the graph theory. This interest has appeared because graphs are a natural mechanism to represent this kind of information, emphasizing the relations between the elements of the domain.

For instance, a social network can be represented as a graph in a very natural way, where the nodes of the graph are the users of the network and the edges are the relationships between the users. These relationships are the most interesting information of the network. Therefore, modelling the information as a graph highlights in a natural way the relationships of the nodes, giving them more visibility and clarity, and making these relationships more easily to study. The same information could be represented in more conventional storages, like a relational database. For instance, the relationships of the users can be stored in a table with two columns (representing the users which are related). This database would be representing the same information than the previous graph. However, typical queries over a social network can be asked more easily using graph mining techniques than querying a traditional relational database.

Among the contributions from the computer science to the graph theory we can mention the graph databases, that appeared to represent and query graphs from many domains [NEO14]. An important effort has been also done to design a standard format to exchange graphs between different applications, like GraphML [GRA09] and visualization graph tools [GEP12] that were proposed to represent graphs provided in that standard formats in a graphical way.

In the same line, many graph mining algorithms have been developed that try to extract information from the graphs.

2.4.1 Graph definitions

Basic definitions A Graph $G(N, E)$ is a pair of sets where N is the set of vertices or nodes of the graph and E is the set of edges [BM76]. An edge is an unordered pair $(n_1, n_2) \in N \times N$ that relates two nodes of the graph. This graph definition corresponds to an **undirected graph**. Figure 2.4 (top-left) shows an undirected graph composed by 4 nodes and 4 edges.

A **directed graph** is also a graph $G(N, E)$ where each edge of the graph is an ordered pair (n_1, n_2) , meaning n_1 is the origin node and n_2 the target node of the edge. Edges of a directed graph are also called arrows or arcs. Figure 2.4 (top-right) shows a directed graph composed by 4 nodes and 5 edges.

A **labelled graph** is a graph $G(N, E)$ that includes a mapping $\alpha : N \rightarrow A$, where A is the label set of the nodes. Similarly, a mapping for the edges are defined like $\beta : E \rightarrow B$, where B is the label set of the edges. When $A \in \mathbb{Z}$ and $B \in \mathbb{Z}$, we called it a **weighted graph**. Figure 2.4 (center-left) shows an example of this kind of graphs.

A directed or a labelled graph is also a **multigraph** when multiple edges can connect a pair of nodes. Otherwise, if no more than one edge is allowed between a pair of nodes, it is called **simple graph**. Figure 2.4 (center-right) shows an example of multigraph, where two different edges starts from n_3 and point to n_2 .

Nodes and edges can contain more complex data. A **property graph** is a graph $G(N, E)$, where each node n_i and each edge e_i is composed by a set of pairs key/value, also called attributes. Figure 2.4 (bottom) shows an attributed graph for a restaurant recommendation site, where nodes n_1 and n_2 are described with the keys *name* and *address*, since they represent users of the application. Nodes n_3 and n_4 are restaurants described by their name and the kind of food they offer. Users qualify the restaurants, including information of their last check in. Users can follow other users of the application, which is represented also by an edge, while each restaurant identifies the most frequent customers.

Graph representations A simple directed graph can be represented in different ways:

- **Adjacency list** The graph is represented as a collection of lists, one list for each different node of the graph. In this way, for a node n_1 , its adjacency list L_1 is composed by all the nodes that are target of any edge which starts in n_1 . For instance, the directed graph shown in the Figure 2.4 (top-right) could be represented by: $\{L_1 = [2, 4], L_2 = [3], L_3 = [1, 2], L_4 = []\}$.
- **Adjacency matrix** The graph is represented as a matrix, where the cell (i, j) is a *one* if an edge exists starting from n_i and pointing to n_j [MM14]. For instance, the adjacency matrix of the same graph can be represented with the next adjacency matrix:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Notice that if the simple directed graph is sparse, an adjacency list spends less space than an adjacency matrix in representing all the graph.

Properties of a graph Depending on the structural characteristics of a graph, many properties can be accomplished.

A graph is **connected** if, for every pair of nodes of the graph (n_i, n_j) , a path exists starting from n_i and ending in n_j . This path is defined as a sequence of nodes with the form n_i, n_{i+1}, \dots, n_j , which starts from n_i and traverses a set of nodes through existing edges until reaching n_j .

A cycle is a path starting and ending in the same node n_i . A **tree** is a connected graph with no cycles, while a **forest** is a no connected graph where each independent connected graph has tree properties.

A graph is **complete** if, for every pair of nodes $(n_i, n_j), i \neq j$, an edge exists between them. A **clique** is a complete subgraph of G .

A graph is **regular** if all the nodes of the graph have the same number of neighbors. A complete graph is also a regular graph.

A node n_i of the graph is a **cut node** if $G - n_i$ has a greater number of connected components than G . A subset of edges $E' \subset E$ is a cut set if $G(N, E \setminus E')$ is no connected and $G(N, E \setminus E'')$ is connected for every set $E'' \subset E'$.

A graph is **bipartite** if the set of nodes N can be divided in two sets N_1 and N_2 , having $N_1 \cap N_2 = \emptyset$ and $N_1 \cup N_2 = N$ and every edge (n_1, n_2) connects a node in N_1 with a node in N_2 . This definition can be generalized to multiple sets, in the way that a k -partite graph (called in general, a **multipartite graph**) is a graph that can be divided in k independent sets, where every edge connects nodes from different sets.

Application domains Graphs are used to model data in many different domains. We give some illustrative graph modelling examples:

- **Web graph:** it represents the links between the different web pages on the Internet. In this way, an edge (n_i, n_j) means that the page represented with node n_i includes a link pointing to n_j .
- **Social network:** it represents the relationships between the different users of the network.
- **Query log:** many different graphs can be generated from a query log record, which stores the queries that users perform over a query search engine and the web pages they click after each query:
 - **Click graph:** it models the behavior of the users in a search site. Queries performed by the users and the internet pages are the nodes of the graph [JLNL13]. Edges start from a query node and connect with an

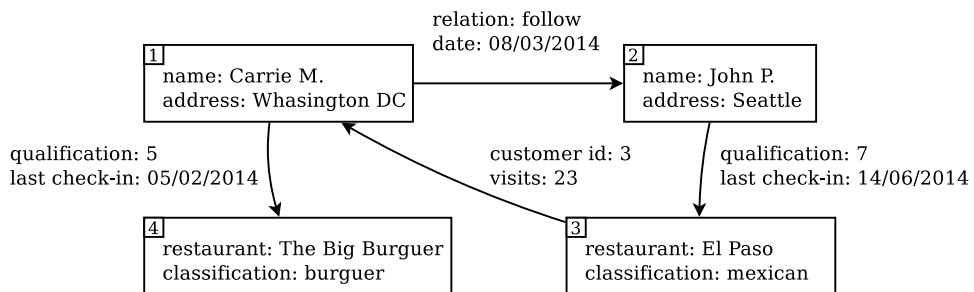
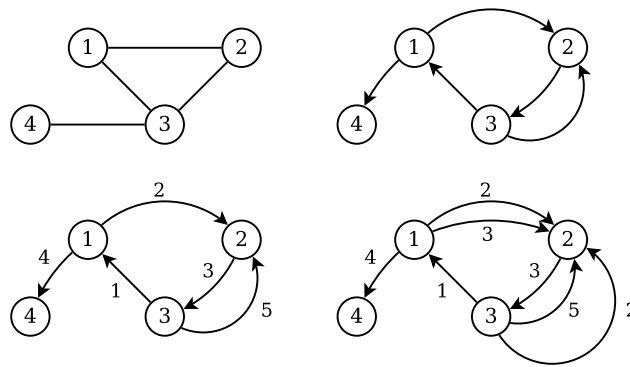


Figure 2.4: An example of different kind of graphs

internet page, representing that a user clicked that page after performing the corresponding query. It usually is a weighted graph, where the weights of the edges are the number of times that the page was clicked after that query. In other words, the weight gives a measure of the relevance of this page for the query.

- **Query graph:** it also represents the information of a query log but in a different way. Nodes of a query graph are queries, and edges relate similar queries following different criteria: nodes that share some words, nodes having the same user session, queries after which users click the same urls (called url cover graph), etc [BBC⁺08].
- **Molecular graph:** this model, used in the chemical domain, represents atoms like nodes, while the chemical bonds are the edges of the graph.

2.5 Summary

This chapter described three basic structures relevant for the works proposed in this thesis. We reviewed the State of the Art in bitmap representations, specially focused on the compressed alternatives. Then, we explained the basic behaviour of the variable-length encoding Scheme DAC, which will be used to design the SP and OP indexes of the K^2 -triple structure, the compressed store for RDF we present in Chapter 8. We studied the implementation details of the K^2 -tree structure, which will be the basis of many of our contributions. Finally, we reviewed some basic definitions of Graph Theory.

Part I

General graphs

Chapter 3

GraphGen: a tool for graph generation

In this chapter we present GraphGen, a tool to generate graphs describing relationships among a collection of complex objects provided as input. GraphGen is designed to easily process that collection of complex objects when they are represented in a table, XML or plain format. In this way, GraphGen can be applied over a wide set of domains.

In Section 3.1, we explain the motivation of this work. To the best of our knowledge, no other tool for graph generation from any domain exists in the State of the Art. We review some tools appeared to generate graphs of specific domains. Section 3.2 outlines the goals we pursue with the design and implementation of GraphGen.

Section 3.3 describes the model of GraphGen. We explain the kind of graphs that GraphGen generates, and we define the rules to generate these graphs from the input data sources.

In Section 3.4 we present GraphGen, providing details about the functions included in the tool to generate nodes and edges from the collection of complex objects and other features it provides.

In Section 3.5 the performance of GraphGen is analyzed through its evaluation in three different use cases. A bibliographic database, a query log and a social network were transformed into a graph by defining in GraphGen specific load, derivation and relation rules. Some measures of the temporal and spatial results are provided as a proof of the scalability of this application.

Finally, Section 3.6 presents some implementation details of GraphGen, describing the internal architecture of the system. We also provide some details of the graph generation algorithm and some memory management optimizations.

3.1 Motivation and related Work

We have already explained in Section 2.4 that graph theory has been widely studied in mathematics. However, in recent years, computer science has also paid attention to the graph theory, taking advantage of the explicit representation of the relationships between the nodes, which is one of the main characteristics of the graphs. In computer science, many efforts have been done to facilitate the work with graphs. For instance, some standard formats for graph representation (like GraphML) were proposed, and graph visualization tools were developed [GEP12]. In the same way, many algorithms for graph mining have been proposed.

However, most of the data collections (extracted from sensors, web actions, digital libraries, ...) that could be represented as graphs, are usually provided in other formats (like relational databases or XML). Therefore, in order to use the power of graph visualization, analysis and mining tools, it is necessary first to represent the data collection into a graph. First, that process requires to define the graph model suitable to represent this data collection. Then, a set of rules can be specified to transform the input data into a graph belonging to that model. Generally this task, which can be called graph modelling, is done with ad-hoc graph modelling tools.

In general, in order to be able to transform a dataset into a graph, we need to have a predefined graph model beforehand. For example, to model the collection of web pages in order to study whether a page points to another, we can decide to use a simple directed graph model.

Network attacks are other domain usually modelled as graphs. An automated method was proposed [SHJ⁺02] to generate and exploit network attack graphs through an algorithm based on model checking. The main problem of this solution relies on the efficiency issues for medium-size networks. An alternative solution, based on logic programming, was proposed by [OBM06]. It achieves more efficient and scalable results on generating this kind of graphs.

Bibliographic databases can also take advantage of their representation as a graph, in order to represent common relationships (like citation or coauthorship) between papers and authors, which could be the nodes of the graph. The Computer Science bibliographic database DBLP [Ley09] is one of the most analysed bibliographic datasets. Currently, DBLP provides its data in XML format and has parsers available to generate the corresponding coauthorship graph. BIBEX [GVSMGV⁺08], built over the graph database DEX (which will be described in section 4.1.1) is an example

of a tool that models bibliographic data as a graph. BIBEX provides not only graph representation from datasets in XML format but also a full querying system that supports specific bibliographic queries.

The information registered in query logs can also obtain benefit for its representation as a graph. A query log registers the behaviour of search engine users, including the queries and pages they click among the suggested by the search engine [BY04] [BY07]. Each click performed by the user is usually logged as an individual record. The information registered in the query logs can be transformed into a graph, having different model alternatives depending on their future exploitation. For instance, *the click graph model* includes two different kind of nodes: queries and pages. Each edge of this graph relates a query with a page that the users click after searching this query. The query log can also be modelled as a *query-flow graph* [BBC⁺08], where nodes are queries and a link between two queries exists if they are part of the same search session. Some works study the combination of different graph models from query logs, to detect query transitions or spam queries [BDBY10].

Similar graph modelling tools appeared in other domains, like source code analysis, chemistry or biology [AW10]. All of them transform data from a specific domain into a graph through ad-hoc parsing algorithms.

3.2 GraphGen goals

In this chapter we present GraphGen, a tool we have implemented to transform arbitrary data collections into graphs. It is currently complete and available to be used¹.

GraphGen was designed to improve the procedure of modelling and creating graphs to represent collections of complex objects. Usually, objects belonging to those collections present a similar structure and, therefore, they can be decomposed in a similar way. The purpose of GraphGen is to avoid the tedious ad-hoc procedure of graph creation that is used nowadays every time we want to represent data collection as graphs.

As explained, GraphGen considers that the input datasets are collections of complex objects which can be recursively divided in simpler elements of information. For instance, a query log is composed by a collection of records. Each record is composed by a query, a user identifier and a set of clicked pages. Another example could be a bibliographic dataset, which is a collection of articles, each of them composed by a title, a set of authors, the conference or journal where it was published, etc.

¹<http://lbd.udc.es/research/graphgen/>

In order to model as a graph a collection of complex data, GraphGen allows the user to define the graph model he or she wants to use to represent the information. That is, the user has to specify the node types, the edge types relating the different node types, and the procedure for the weight computation of the nodes and edges if they are required. GraphGen also provides a set of extraction functions, which are used to decompose the input complex objects in their smaller components, generating the nodes of the graph and the edges (relationships between them).

3.3 Graph Generation Model

The purpose of GraphGen is to implement a general mechanism to generate graphs that represent the collection of complex objects provided as input. We consider input data as a collection of complex objects composed by heterogeneous elements. The GraphGen interface allows the user to specify the kind of nodes that are relevant for each specific domain. That is, nodes are obtained by the progressively decomposition of the complex objects, applying simple decomposition rules. Edges are created as links between a node and the nodes obtained by its decomposition.

3.3.1 GraphGen Model

GraphGen generates labelled multigraphs, composed by a set of nodes and edges. Nodes of the graph are generated from the decomposition of more complex nodes. The *derivation and decomposition rules* are the mechanism to define how the nodes are progressively decomposed in simpler nodes. An edge relates each complex node with each simpler node produced by its decomposition. The kind of nodes and the kind of edges that GraphGen generates are described next.

3.3.1.1 Nodes

Nodes of the graph are units of information. A node $n \in N$ in GraphGen has a *content* and a *type* $\in TN$, where TN is the set of types existing in G . The type of a node gives semantic information about the meaning of its content in the context of the graph. Besides, this type will play an important role in the generation of the graph, since the available derivation rules that can be applied over a node, in order to generate new nodes of the graph, are determined by the type of the node. The graphs generated by GraphGen include two kinds of nodes: the *input nodes*, which are the most complex nodes of the graph; and the *derived nodes*, resulting of the decomposition of complex nodes (input or other derived nodes).

Input nodes The purpose of GraphGen is to generate a graph that represents the information contained in semi-structured input files, like files in XML, CSV format or even a relational database. Input nodes are the most complex objects of this graph, which are progressively decomposed in simpler nodes. Each input node has associated a *type*, which gives information about the nature of its content. A graph can have different *types* of input nodes.

Let consider a context where the input data is a collection of documents $\{D_i\}$. Then, each document D_i can be an input node of the graph. Its type is *document* and its content is the text that composes that document. That complex information will be progressively decomposed during the graph generation process.

Another example we can consider is a relational database with the information of the members of a University. Each row of each table is an input node of the graph. The type of the input node is given by the table it belongs to. For instance, a database of the members of a university could have input nodes of different types: $TN = \{student, professor, subject, \dots\}$.

Note that input nodes have complex content, which will be decomposed through the process of graph generation until obtaining simpler nodes. For instance, each *document* described in previous example can be divided in several *paragraphs*, and paragraphs can be divided in *words*. In the example of the University, input nodes of type *student* can be decomposed attending to the columns of the table student, in the way that each column will produce a new node in the graph. Note that in the case of the paragraphs of a document, the order of the elements matters. Consequently, a document is divided in a **list** of paragraphs. However, the different attributes describing a student in the database are unrelated, so a student is decomposed in a **set** of nodes.

Derived nodes A derived node is the result of the decomposition of complex nodes (input nodes or other derived nodes) in one or several simpler nodes. The mechanism to define this decomposition is given by the derivation rules which will be explained in Section 3.3.2. Going back to the previous examples, paragraphs are derived nodes created from an input node *document* (D_i), with their own associated type *paragraph*. In that way, from each input node D_i , it is possible to derive an ordered list of paragraphs $[P_j]$. At the same time, each paragraph P_j is decomposed in simpler nodes of type *word*: $[w_k]$. In the case of the node *student* for the example of the database of a University, the attributes *name* or *telephone* of each student will be nodes derived from each input node of type *student*.

In this way, the nodes of the graph are progressively decomposed in simpler nodes until reaching such nodes which are not decomposed in other nodes. The number of steps until these atomic nodes are reached depends on the domain and the set of rules defined by the user.

3.3.1.2 Edges

Edges of the graph represent the relation between the nodes they connect. In GraphGen, edges are directed (that is, they distinguish between the origin and the target node). Just as the nodes, they have an associated *type* $\in TE$, where TE is the set of edge types in the graph. In addition to this, depending on the kind of the rule that produces them, they could have a numeric value (or weight) with different meanings.

Edges relating a complex node with simpler nodes derived from it could use the weight to represent the order of the derivation. Let consider again the collection of documents, which are decomposed in paragraphs. For a document with n paragraphs, n edges will be generated, starting from the input node D_i and targeting to the n paragraph nodes P_j . Each of the n edges will have a different value from $1 \dots n$, representing its position in the full document.

Figure 3.1 shows an example of graph generated in GraphGen. It contains two input nodes of type *document*, from which two derived nodes of type *paragraph* are derived. Each paragraph is decomposed in several words. Each node of the graph is related to the complex node that generate it.

3.3.2 Derivation or Decomposition Rules

Decomposition rules are the mechanism that GraphGen provides to define new nodes and edges in the graph. Each rule specifies a type, named *source type*, which restricts the kind of nodes that are decomposed by this derivation rule. All the nodes generated by applying a rule will belong to the same type, which we called *target type*. The set of possible node types TN in a graph model is inferred from the definition of the decomposition rules.

In addition to the creation of new nodes, derivation rules also create new edges between nodes of the graph. For this reason, the rules also need to specify an *edge type*, so all the new edges created with a decomposition rule have the same edge type. The set of edge types (TE) is inferred from the derivation rules definition.

Together with the types of the nodes and edges, rules contain the extraction function that encapsulates the logic of the rule. Its definition strongly depends on the format in which the source node presents the information. GraphGen implementation proposes a set of predefined extraction functions for some of the most common formats to represent data (like XML or CSV). However, GraphGen does not impose any restriction about the extraction functions, so it provides a flexible mechanism that can be adapted to the needs of every domain.

A derivation rule could be formulated as:

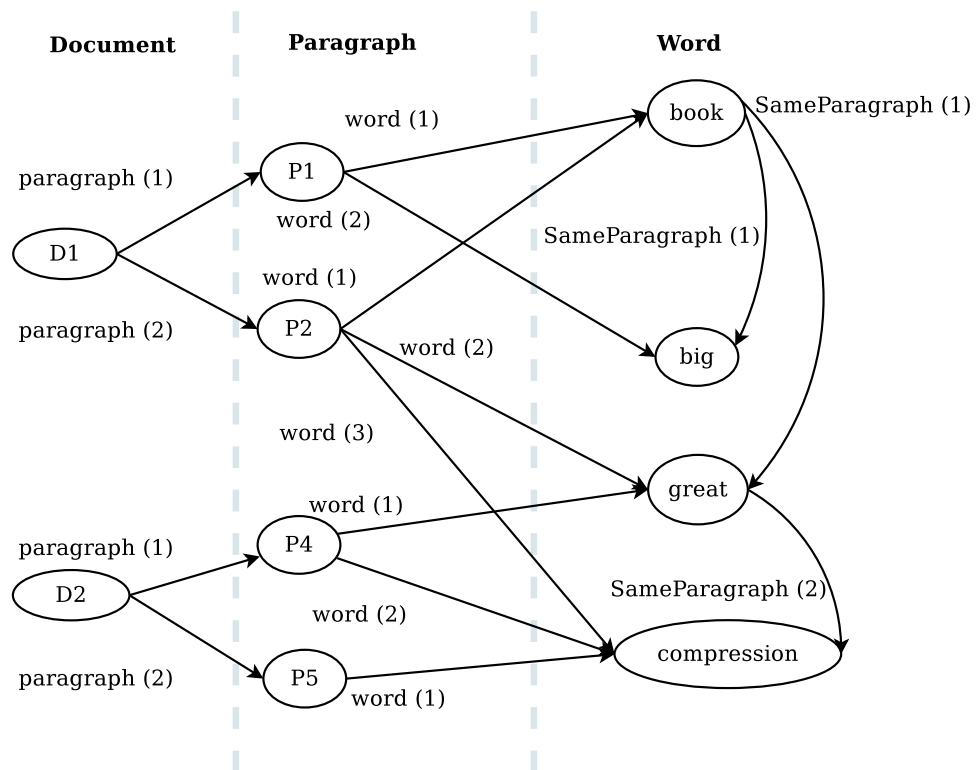


Figure 3.1: Example of a graph generated from a collection of documents

$$N_{targetType} \leftarrow N_{sourceType} \circ DR_{name,extractionFunction}$$

Two additional parameters can be considered in the definition of the derivation rules:

- **Tree/Graph** it determines the structure of the graph. When a rule is defined as a tree rule, the application of the rule over a complex node will produce a set of new nodes of the graph and a new edge relating the source node with each target node will be generated. However, when the rule is a *graph derivation rule*, a new target node will be created only if no nodes with the same content and target type exist in the graph. Otherwise, the corresponding edge created with the rule starts from the source node and targets to the already existing node with the appropriate content.

Considering only tree derivation rules, the generated graph follows a tree structure, since each target node has only one edge targeting to it. However, for graph rules, many edges can exist pointing to the same target node, creating a graph structure. Figure 3.2 shows a graph generated through a tree rule (left) in opposition to the same graph generated through a graph rule (right).

- **Ordering derivation rule:** When a derivation rule is defined with ordering type, the nodes that are produced from the same source node will be pointed by edges weighted with correlative integers, representing the order in which these nodes were produced. Figure 3.1 shows an example of ordering rule. Edges relating the documents and the paragraphs they contain are weighted to show the relative position of the paragraph into the document.

3.3.3 Relation rules

An initial version of the graph model previously described was presented in [BYBLP10]. This model only defined one kind of rules: the derivation rules, which decompose complex objects to a set of simpler components. In this way, the graphs that are generated with this model are multipartite graphs, where each partite set is given by a different node type. Edges of this graph relate each complex object with the nodes derived from it. However, in some contexts, a different kind of edges can be needed to link nodes that are not necessary related through a decomposition process. Therefore, we define the relation rules that, although they can break the multipartite nature of the graph, are very useful to represent relations in the graph.

Relation rules provide the possibility of defining customized edges. This kind of rules allow the user to define how new edges will be created among already existing nodes. Note that this kind of rules do not generate new nodes in the graph. Instead

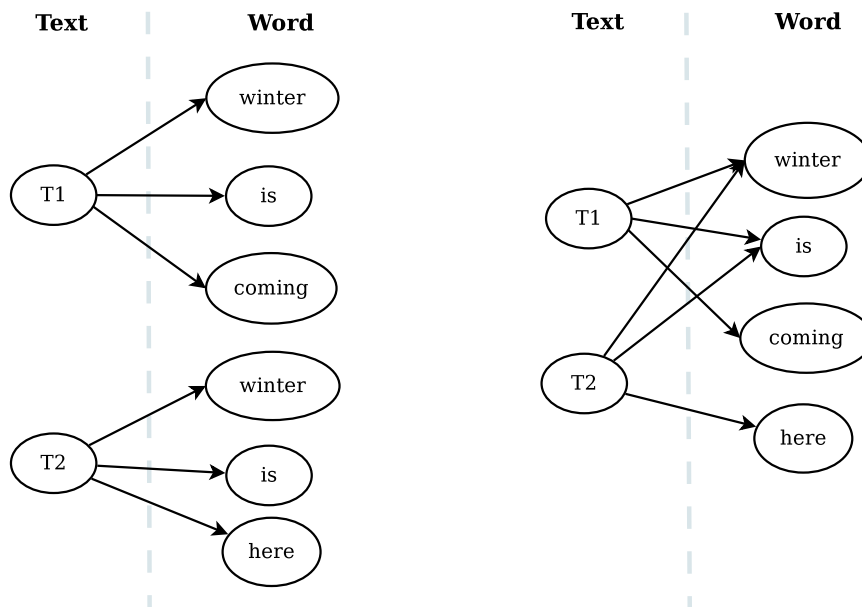


Figure 3.2: Example of a word extraction through a tree rule (left) and a graph rule (right)

of this, they only define new edges representing relationships between the existing nodes of the graph.

A relation rule specifies two types of nodes $TN_1 \in TN$ and $TN_2 \in TN$. The new edges created with the rule will point from a node n_1 with type TN_1 to a node n_2 with type TN_2 . All the edges created with this rule will be of the type $TE_1 \in TE$. Finally, the rule specifies a *condition* that the pair of nodes n_1 and n_2 has to accomplish to create an edge E_{12} with type TE_1 . A relation rule is formulated as:

$$E_{type} \leftarrow N_{sourceType1}, N_{sourceType2} \circ RR_{name,condition}$$

The condition that the nodes have to fulfill can vary depending on the domain. It can be defined in terms of the content of the nodes. For instance, a relation rule can specify that two nodes of type *word* will be related if those words have a Levenshtein distance less than a given boundary.

On the other hand, the condition can be specified on the basis of the topology of the graph. The most representative function is *Common Ancestor*, which establishes that two nodes n_1 and n_2 will be related through an edge E_{12} with type TE_1 if they have at least one common ancestor.

In the case of the relation rules, weights of the edges can be used to represent other information. For instance, if the condition of the rule is being *coauthor*, the edges between different authors will be weighted with the number of papers they wrote together, that is, the number of nodes of type paper they have as common ancestor.

Figure 3.1 shows one example of relation rule. The Figure includes the relation rule *SameParagraph* which relates nodes of type word. Two words will be related if they are extracted from the same paragraph (that is, they have a common ancestor of type *Paragraph*). The weight in this case represents the number of paragraphs where both words appear. For instance, *sameParagraph* between word *great* and word *compression* has weight 2 because both of them appear in the paragraph 2 and the paragraph 4. However, *book* and *great* only have a paragraph in common (paragraph 2) so they are related through an edge with weight 1.

Figure 3.3 (bottom) shows other example of relation rule. Relation rule $RR_{coauthor}$ produces several edges relating the coauthors of the paper. For instance, *John F.* and *Peter M.* are linked through an edge with weight 2 because both of them are authors in *Paper 1* and *Paper 2*.

3.3.4 Graph Model vs Instance Graph

The set of rules that a user defines to generate a graph from some input data can be represented as a graph that we called *model graph*. Nodes of that model graph represent the node types, whose creation will be produced by the derivation rules. Each edge of this graph model represents a different rule or edge type. The derivation rules are the structural edges which connect the different node types with their original source types. The remaining edges represent relation rules.

Figure 3.3 (top) shows an example of a model graph. That model graph represents a system of rules to extract a graph from a bibliographic database. Nodes of the graph model with in-degree 0 are the input nodes of the system. In this graph, the input nodes are of type *paper*. The model graph also contains three derivation rules, all of them applied over nodes of type *paper*, to extract the title, the different authors of the paper and the keywords. The three derivation rules are represented as edges in the model graph, starting from the source node type *paper*, which is decomposed in the three different kinds of target nodes: *title*, *author* and *keyword*. An additional rule *coauthor* is created, that relates nodes of type TN_{author} .

At the bottom of the figure, an example of the instance graph generated by that graph model is shown. It is a result of applying the derivation rule mechanism that the model graph on the top defines. It contains two *paper* nodes. The first paper is written by three different authors, while the second paper is written by two of them. We can see in the graph the rule DR_{author} , which is a graph derivation rule. Therefore, the authors of the second paper do not produce new nodes of type *author* because they were already produced by the first paper. Each paper will also generate a node of type TN_{title} and some nodes of type $TN_{keyword}$. This graph generated following the rules of a model graph is called *Instance graph*.

3.4 The GraphGen tool

In this section we present our tool, *GraphGen*, specifically designed to generate graphs from heterogeneous data sources through a rule definition system. It implements the data model that was detailed in the previous section. GraphGen includes a graphical user interface that allows the user to define the rules for the creation of nodes and edges, to check the generated data and to export a standard graph format.

In order to define the derivation rules of the custom graph, we implemented a collection of extraction functions to manipulate the most frequent formats used to structure data of real environments (*XML*, *CSV*, *HTML*, ...). However, the rule engine is designed to support the integration of new generic extraction functions as well as domain-specific functions.

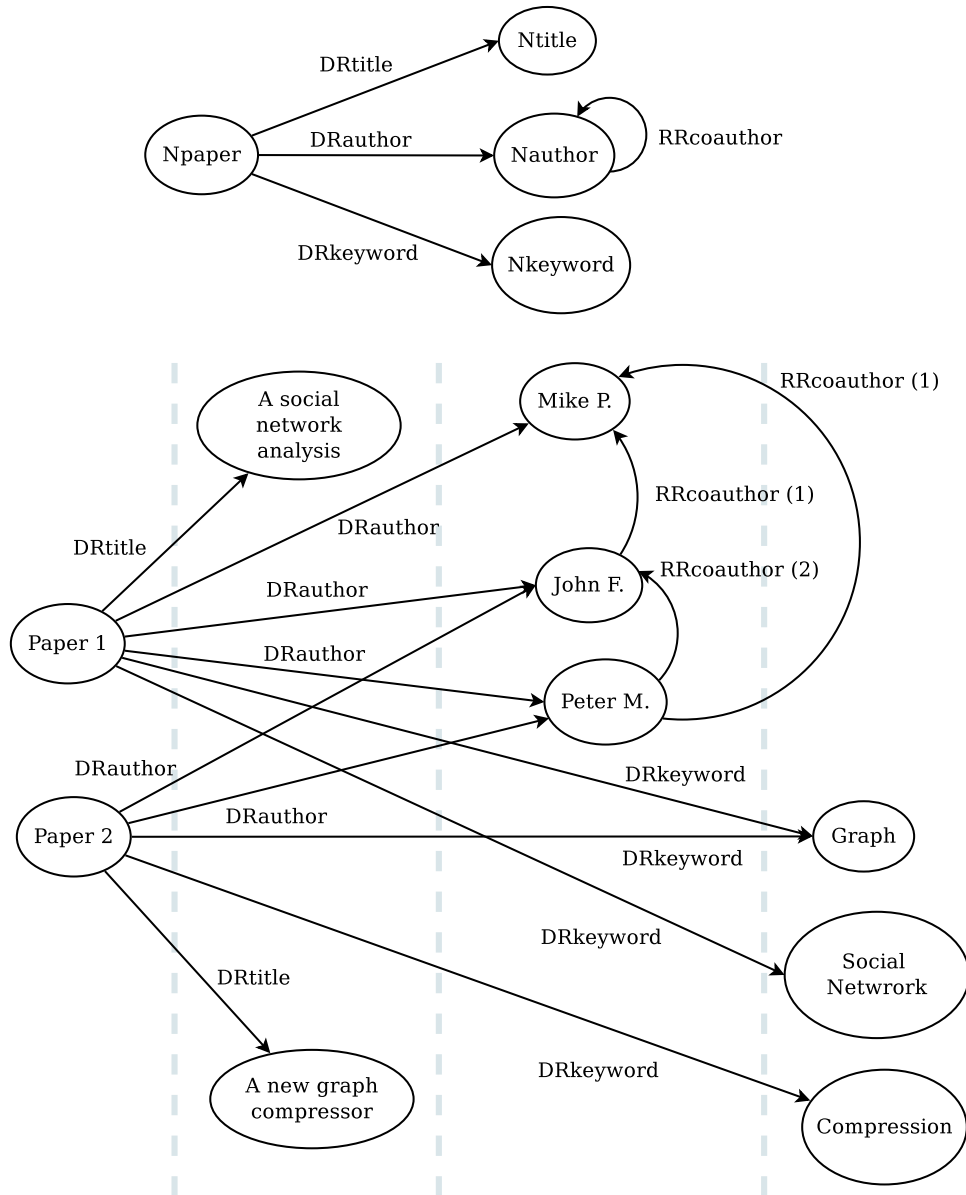


Figure 3.3: Example of a model graph (top) and an instance graph created with this model graph (bottom)

The graphs that are generated with GraphGen are internally stored in a relational database. Therefore, they can be accessed through SQL queries by connecting with the database server. Data can be also checked through the graphical user interface of the application. *GraphGen* incorporates the possibility of exporting the graph data in the standard language GraphML [GRA09]. In that way, the generated graph can be easily imported in a graph database (such as Neo4j [NEO14]) or in a graph visualization tool like Gephi [GEP12].

3.4.1 Load Rules

Load rules are the mechanism that GraphGen provides to generate inputs that represent the complex objects that compose the data collection. They are the first rules to be applied and they extract the information from the source files. In this way, the possible extraction functions for a load rule are quite similar to the extraction functions described for the derivation rules. However, instead of specifying a type of node as the *source* type of the rule, the external file must be provided.

A load rule will be applied over the input file and it will produce several input nodes as a result. Note that, given that a load rule is not executed over existing nodes of the graph, no edges are created pointing to the new nodes. Necessarily, input nodes created by a load rule will not have *source* edges. That is, they will have in-degree zero. The remaining nodes of the graph will contain at least one edge pointing to them.

3.4.2 Predefined extraction functions

GraphGen includes some predefined *extraction* functions to decompose nodes in simpler elements, and an additional function, named *checkrelation*, which is used to relate existing nodes with new edges. We design these functions to manage data in some of the most common structured formats, like XML, CSV or plain text. However, the tool was designed to be easily extended to other input formats and other extraction functions over the formats currently supported can be implemented.

Next, the pre-defined existing functions are described:

- *XMLTag*: this function extracts as many derived nodes as occurrences of a *tag* appear in the XML which is the content of a node. The tag can be included in the resulting node or not. For instance, we can apply the *XMLTag* over the tag `book` by excluding the tag in the next XML: `<library><book>Philadelphia</book> <book>The book thief</book> </library>`. This function will generate two elements: *Philadelphia* and *The book thief*. Otherwise, if the function *XMLTag* is applied including the tag over the

same example, the result will be `<book>Philadelphia</book>` and `<book>The book thief</book>`.

- *XMLAttribute*: given a *tag* and an *attribute*, it produces one element for each time that this tag presents this attribute, and the value of the new node will be the value of the attribute. For instance, the function *XMLAttribute* for the tag *book* and the attribute *title* for the XML `<book title="Philadelphia"/>` will produce one node with value *Philadelphia*.
- *Label*: it is mainly used in such not structured datasets, where some values are delimited for special characters or words. *initLabel* and *endLabel* defines those limits. For instance, the function *Label* for the separators `..1` and `..2` for the text `The film was directed by ..1Coppola..2` will extract an element with value *Coppola*.
- *Table*: this function is specifically designed for such datasets in *csv* format. It allows to extract a complete row of the input data. On the other hand, it can also be used to extract a specific column of a row.

GraphGen currently supports one function to create new edges between the existing nodes of the graph:

- *CommonAncestor*: this function checks if two nodes have an ancestor of a given type in common, in order to establish a new edge between them.

3.4.3 Filtering nodes and edges

Some of the rules generate new nodes by decomposition of other nodes of the graph. These rules also generate an edge between the original node and the new nodes which are generated. Some of these nodes are usually created as temporal nodes, used as transitional nodes that are progressively decomposed until reaching the desired nodes and edges of the final graph. GraphGen allows the user to filter the nodes and edges of the final graph after the graph generation, which will be exported to GraphML. This filtering is performed by type. That is, all the nodes or edges of the specified types will be removed from the final graph in order to obtain the desired graph.

3.5 Example of use cases

In this section we show how GraphGen is used to create graphs from heterogeneous input data through three complete examples with real datasets. The three next

sections describe, for each dataset, the nature of the data and the structure of the graph we can obtain from this data. After that, the process of rule definition to obtain this graph is described. Then, some statistically results of the generated graph are provided. Finally, last section shows a brief analysis of the temporal and spatial efficiency measured for the three study cases.

3.5.1 Bibliographic dataset DBLP

Bibliographic databases are a common source of data, specially used for research purposes. There are many application fields for the bibliographic information. Recommendation systems, community detection or plagiarism detection are some illustrative examples. For many of these applications, structuring the information as a graph allows researchers to extract new knowledge by applying mining graph techniques.

3.5.1.1 Dataset

In this section we propose the transformation of the computer science bibliographic dataset DBLP [Ley09]², which gathers information of computer science publications, including articles in journals, papers in conferences, workshop proceedings, or even thesis. We use an XML extracted from <http://dblp.uni-trier.de/xml/> that contains a sample of 100,000 journal articles with a size of 43 Mb. Each article includes information about its authorship, title and information relative to the publication, like the journal, conference and year of publication.

The purpose of this example is to obtain, from this XML data source, a graph which connects each author with its coauthors, in the way that the weight of the edge would represent the number of papers in common (providing a good measure of the level of collaboration between the two authors). In addition to this co-authorship, each author will be related with the journals where he/she published, and the weight of the edges in this case would represent the number of publications in each journal for each author, suggesting favourite publication sites.

Figure 3.5 shows an example of the kind of graph we aim to obtain from the XML of DBLP through the definition of load, derivation and relation rules.

3.5.1.2 Graph definition

We detail the rules defined in GraphGen to obtain the graph described in Section 3.5.1.1.

²<http://www.informatik.uni-trier.de/~ley/db/>

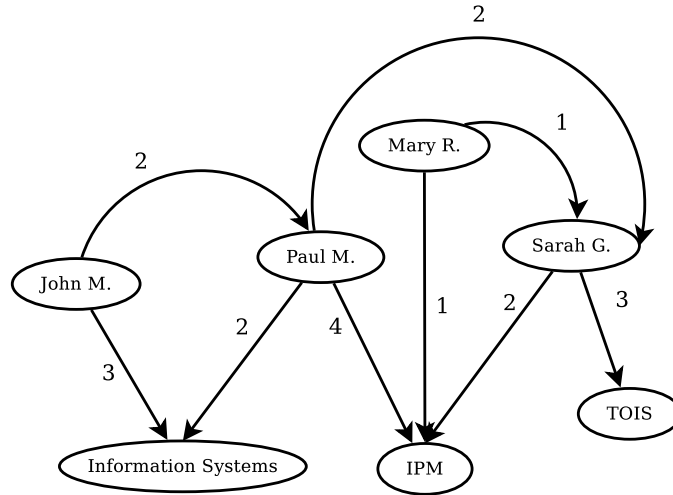
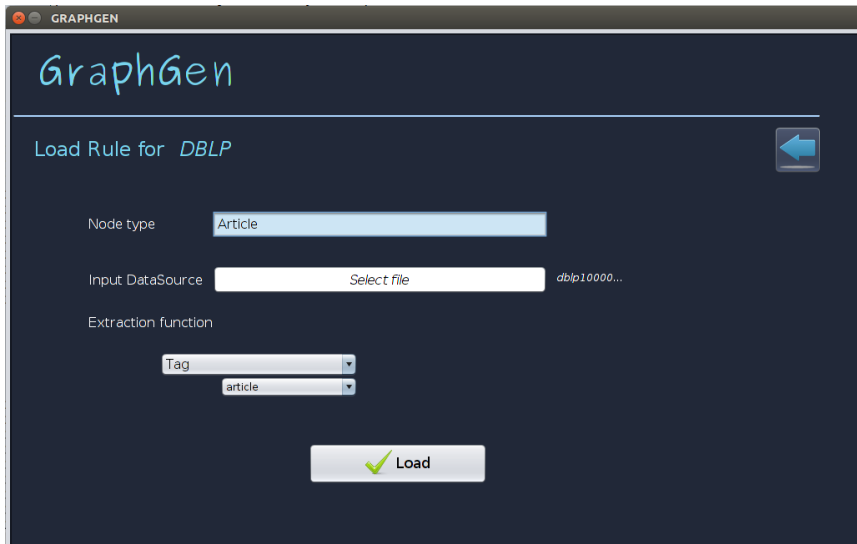


Figure 3.4: Example of subgraph generated in GraphGen from DBLP data

Load rules The first step extracts from the input XML the information about the different articles included in the dataset, which are represented by the tag `< article >`. Therefore, we define a load rule whose target node type is *Article* and an extraction function of type *XMLTag* for the tag *article*. Figure 3.5 shows a screenshot of the form filled in GraphGen to create this load rule. This rule will produce one node for each tag *article* included in the input data. The value of the nodes will be the XML included between the opening and ending tag *article*, including all the information about the given article.

3.5.1.2.1 Derivation rules Previous load rule will produce one node for each different *paper* included in the XML dataset. We want to extract the authors and the journals of each article. Consequently, two derivation rules having TN_{paper} as source type have to be defined:

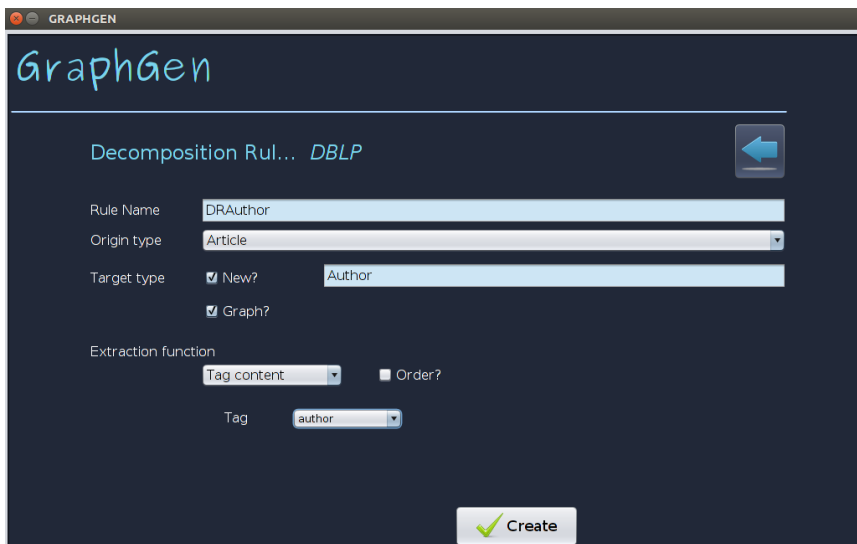
- A derivation rule to extract the authors for each article. The authors are represented in the XML dataset by the tag *author*, descendant of the corresponding *paper*. Therefore, this derivation rule will have an extraction function of type *TagXML*. The purpose of the final graph is to represent each author by an individual node, so this derivation rule will be a graph rule, in order to all the tags *author* with the same value be clustered in a single node. Figure 3.6 shows the process of definition of this rule in GraphGen.
- Analogously, a derivation rule is defined to extract the journals where the



The screenshot shows the GraphGen application window with the title 'GraphGen'. The main heading is 'Load Rule for DBLP'. Below this, there are several input fields and a button:

- Node type:** A text input field containing 'Article'.
- Input DataSource:** A text input field with a 'Select file' button and a truncated path 'dblp10000...'. To the right is a blue arrow button pointing left.
- Extraction function:** A dropdown menu labeled 'Tag' with 'article' selected below it.
- Load button:** A button with a green checkmark and the text 'Load'.

Figure 3.5: Load rule definition for DBLP dataset



The screenshot shows the GraphGen application window with the title 'GraphGen'. The main heading is 'Decomposition Rule... DBLP'. Below this, there are several input fields and a button:

- Rule Name:** A text input field containing 'DRAuthor'.
- Origin type:** A dropdown menu containing 'Article'.
- Target type:** A dropdown menu containing 'Author'. To its left are two checked checkboxes: 'New?' and 'Graph?'.
- Extraction function:** A dropdown menu labeled 'Tag content' with 'author' selected below it. To its right is an unchecked checkbox labeled 'Order?'.
- Tag:** A dropdown menu containing 'author'.
- Create button:** A button with a green checkmark and the text 'Create'.

Figure 3.6: Derivation rule to extract the authors of the articles in DBLP

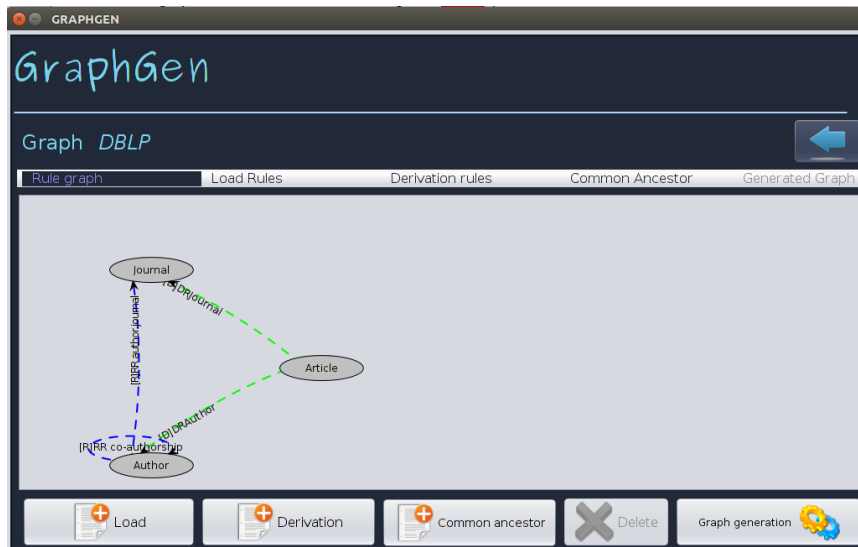


Figure 3.7: Graph model in GraphGen

articles have been published. The final graph will contain only one node for each different journal in the dataset, so as in the previous rule, we define this rule as a graph rule with an extraction function of type *XMLTag*.

Relation rules With one load rule and two derivation rules we have defined all the nodes of the final graph. The co-author relationship and the number of publications of an author in the same journal are defined through two relation rules:

- A relation rule is defined to relate authors which collaborate in the same paper. It is defined through a check relation function based on sharing at least a common ancestor, and the value of the edge between two co-authors will have as value the number of articles where they collaborate (common ancestors).
- A relation rule is defined to connect the authors with the journals where they usually publish. Similarly to the co-authorship relation rule, it consists in a common ancestor rule where the common ancestors is of type article. The value of the edge represents the number of publications of an author in a given journal.

Figure 3.7 shows a graphical representation of the graph model defined in GraphGen.

Type		Number of elements
Nodes	Article	100,000
	Author	141,203
	Journal	176
Total Nodes		241,379
Derivation edges	R_{author}	258,375
	$R_{journal}$	99,982
Relation edges	$R_{journal-author}$	175,645
	$R_{coauthor}$	585,528
Total Edges		1,119,530

Table 3.1: Statistical information of the generated coauthorship graph

3.5.1.3 Graph generation and filtering

After the definition of the rules that extract information from the input data sources, the environment in GraphGen is ready to generate the instance graph: that is, to create the set of edges and nodes according with the different rules.

Table 3.1 gives statistical results of the final graph generated for a XML of DBLP including 100,000 articles. In the final graph, nodes of type *Article* are not needed, so it will be filtered using the filter method by node type provided by GraphGen. As a consequence of this removal, edges starting from article (DR_{author} and $DR_{journal}$) will be also deleted. Therefore, the final graph, which can be exported to GraphML, contains 141,379 nodes and 761,173 edges.

While the process of graph definitions is lightweight, the generation of the graph requires many temporal and spatial resources. Current implementation produces around 30 elements of the graph per second, mainly due to the communication with the relational database. However, we already mentioned that obtaining an efficient tool is not the purpose of this work. Instead of this, we aim to implement a practical tool supporting the theoretical model, proving its completeness and flexibility to create graphs for multiple purposes.

3.5.2 Query Log Analysis

Query logs gather relevant data about the interaction of the users with a search engine, including information like the query submitted by the user and the pages the user click among the results provided by the search engine for that query [BY04,BY07]. The analysis of this user behaviour can help to understand the user requirements and to improve the search engines, applying in new researcher fields like query expansion or query recommendation, but can also help to discover interesting relationships between the different URL pages clicked by the users. A graph obtained from a query log will allow the application of graph mining algorithms.

3.5.2.1 Dataset

In this section we use an anonymized query log to generate a graph that relates the queries of the users with the pages they clicked for those queries. In that way, the value of the edge gives a measure of the level of accuracy of the page for a given query. We also represent users as nodes of the graph, and we relate them with queries and clicked URLs. The input data source used in this study case contains 100,000 records, each of them composed by a user identifier, a query submitted by this user and the first URL the user clicked.

3.5.2.2 Graph definition

3.5.2.2.1 Load rule The query log is provided in CSV format, so in order to extract information from the collection of records, we will extract each individual record through a load rule using the extraction function *Table*, parametrized with *isRow = true* to create one node per row in the dataset. The target type is named *Record*.

3.5.2.2.2 Derivation rule For each node of type TN_{record} generated through the load rule, the user, the query and the URL will be extracted, in a similar way by the corresponding derivation rules, all of them using the extraction function *Table* and specifying their corresponding position in the record, which is organised in columns. Therefore, the three derivation rules we need in this case are a rule to extract the users in the first column, a rule to extract the query in the second column and a rule to extract the URL in the third column. These rules will generate the three different kind of nodes in the final graph: users, queries and pages.

3.5.2.2.3 Relation rules Relation rules will create the relationships between the three different nodes (users, queries, pages), based on their belonging to the

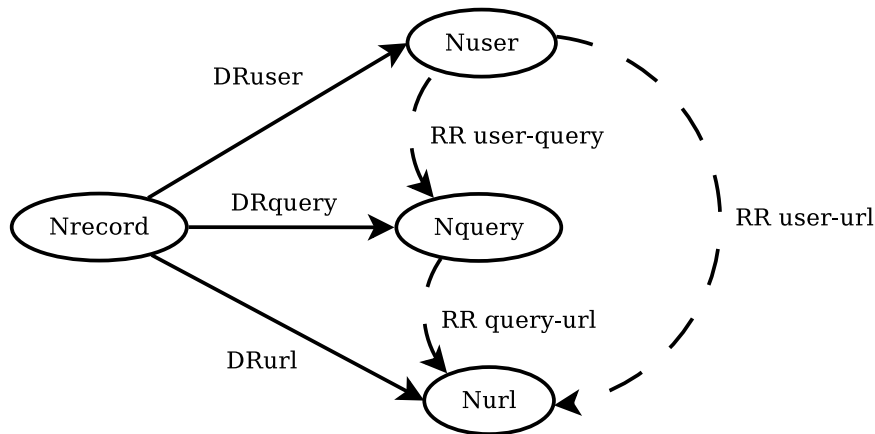


Figure 3.8: Graph model in GraphGen for the Query Log dataset

same record, through common ancestor functions having as an ancestor a node of type record. We define three relation rules, relating users with queries created from the same record, users with pages and queries with pages.

3.5.2.3 Description of the generated graph

Table 3.2 shows the number of nodes and relations created from the query log with 100,000 records.

3.5.3 Social Network Analysis

Last years, social networks have experienced an impressive evolution, involving millions of people which use them for communication purposes but also to share contents and experiences. This kind of networks can be represented as a graph in a natural way, since its relevant information relies mainly on the user relationships. Many interesting properties about the social behaviour can be extracted from the analysis of the social networks. Community analysis or recommendation systems are some of the application fields. Therefore, the transformation of the information generated in social networks into graphs is needed to apply mining graph techniques and to infer properties of the network.

Type		Number of elements
Nodes	Record	100,000
	User	1,004
	Query	36,186
	URL	41,209
Total Nodes		178,399
Derivation edges	R_{user}	100,000
	R_{query}	100,000
	R_{URL}	100,000
Relation edges	$R_{user-query}$	39,250
	$R_{user-url}$	63,214
	$R_{query-url}$	71,455
Total Edges		473,919

Table 3.2: Statistical information of the generated query graph

3.5.3.1 Dataset

As a representative example of a social graph generation using GraphGen we use a dataset extracted from an online social network in the popular photo-sharing site Flickr³. We use CoPhIR [BEF⁺09], which contains data from 106 million images, including their author, tags, comments and MPEG7 visual descriptors of the image. We use a sample of this collection, containing the information of 100,000 images that comprise 260 Mb, where each object is represented by an XML.

We use this dataset as input to generate a graph relating each user with the tags used in his submitted photos, each user with the location of his photos and finally each location with the tags more commonly used in photos in that location.

3.5.3.2 Graph definition

Load rules The input dataset is composed by a set of different XML nodes, one per image (identified by the tag *SapirMMObject*). Therefore, we define a load rule to extract each image with an XMLTag extraction function that creates a node containing the XML between the tags $\langle SapirMMObject \rangle \dots \langle SapirMMObject/ \rangle$.

Derivation rules We define different rules to decompose the nodes representing the objects in the previous step into the users, tags and locations of each photo. First of all, we extract the element photo from the MMObject and then, we extract from the photo the users, tags and locations. User and location are extracted through an XMLAttribute extraction function, while the tags are extracted through two XMLTag extraction functions (because they are descendants of the tag *photo* which is descendant of the tag *SapirMMObject*). Therefore, we define four derivation rules to model this behaviour.

Relation rules Once the structural model of the graph is defined, we add some relation rules to create relations between users, tags, and locations produced by derivation rules. As in the previous examples, the rules will be defined by check relation functions based on their common ancestors. In this way, each relation will be weighted by the number of common ancestors they have (photos in this case). We define these rules to relate users and tags, users and locations and tags with locations.

³<https://www.flickr.com>

Type		Number of elements
Nodes	SapirObject	100,000
	Photo	100,000
	User	5,525
	Location	387
	TagSet	62,669
	Tag	32,299
Total Nodes		300,880
Derivation edges	R_{photo}	100,000
	R_{user}	100,000
	$R_{location}$	2,608
	R_{tagset}	100,000
	R_{tag}	277,883
Relation edges	$R_{user-tag}$	69,643
	$R_{location-tag}$	465
	$R_{user-location}$	4,282
Total Edges		654,881

Table 3.3: Statistical information of the generated social graph

3.5.3.3 Description of the generated graph

We test the graph generation processing for the set of 100,000 images according to the rules defined in Section 3.5.3.2. Table 3.3 shows the number of different elements obtained.

3.5.4 Spatial and temporal efficiency

In this section we perform an analysis of the spatial requirements and the temporal efficiency of the graph generation process for the use cases previously presented: the bibliographic database DBLP, the query log and the social network of Flickr (COPHIR). GraphGen was designed to prove the Graph Model described in Section

3.3.1 and obtaining a solution efficient in terms of space and time is not the purpose of this work. However, we give some temporal and spatial results in order to show that our tool can generate graphs for real cases containing millions of nodes. We measure the execution time of the graph generation, since it is the unique process in the tool that requires long time to be executed.

The experiments were run on a computer with an Intel(R) Core(TM) i5-3337U 1.80 GHz processor, 6GB DDR-3 main memory, and a 256 GB SSD hard drive. It ran a windows 7 operative system, Java JDK 1.7.051 and MySQL Community Server version 5.6.15.

In the experiments, we generate the graphs for the three different use cases by applying the rules defined by the graph model, varying the size of the input from 10,000 to 100,000 input objects. Figure 3.9 shows the number of elements created from different input sizes in each use case. Figure 3.10 shows the execution times of each graph generation. These results show that both the size of the graph and the time needed to generate the graph grow about linearly with the number of objects.

Results also show that the format of the input data clearly affects to the temporal results. When the input dataset is represented in XML (like in the DBLP and COPHIR datasets) the relative time per element is more costly, mainly due to the extra work needed to apply extraction functions based in XML parsing compared with extraction functions for texts structured in rows and columns. Nevertheless, we can conclude that, with a mid-range computer, tens of millions of nodes and edges can be generated per hour with our tool.

3.6 Implementation details of GraphGen

In this section we describe the most relevant implementation details of GraphGen, explaining the data model and the architecture of the application. We also specify the extraction functions included in the current version of the tool. Finally, we show some improvements incorporated to optimize the spatial and temporal efficiency of the generation process, specially necessary for processing big datasets.

3.6.1 Data Model

The data model of GraphGen is designed to support all the features of the theoretical model described in Section 3.3. The overall set of entities is composed by objects that support different generation stages: some of them belong to the rule definition system, while others are used to store the final graph. Figure 3.11 shows all the objects involved, which will be described next:

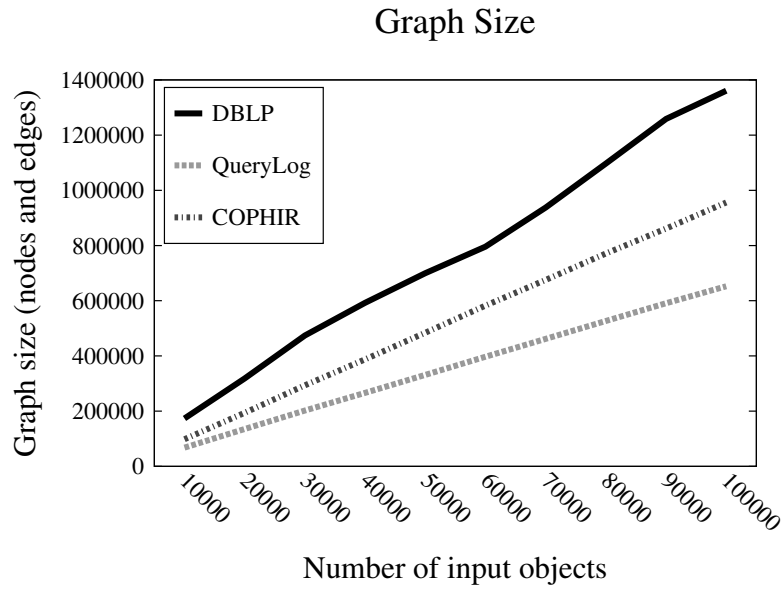


Figure 3.9: Space analysis in GraphGen

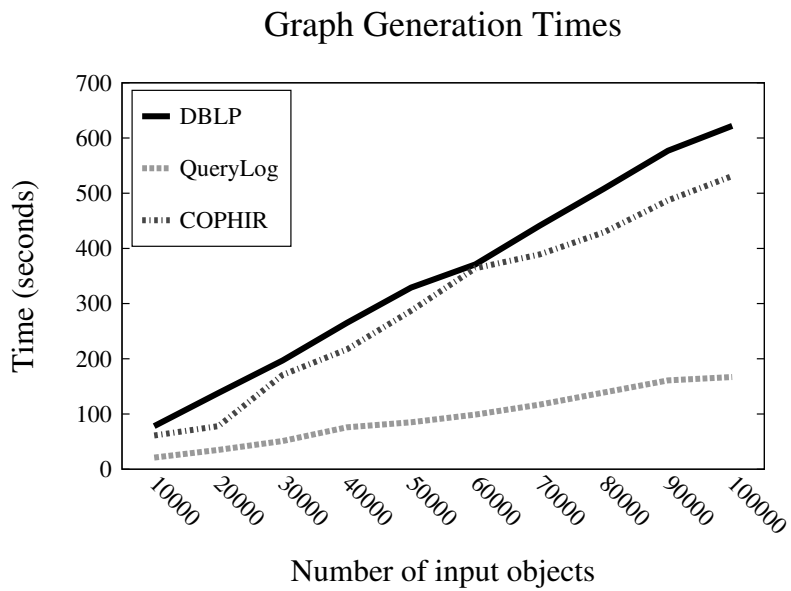


Figure 3.10: Temporal results for GraphGen

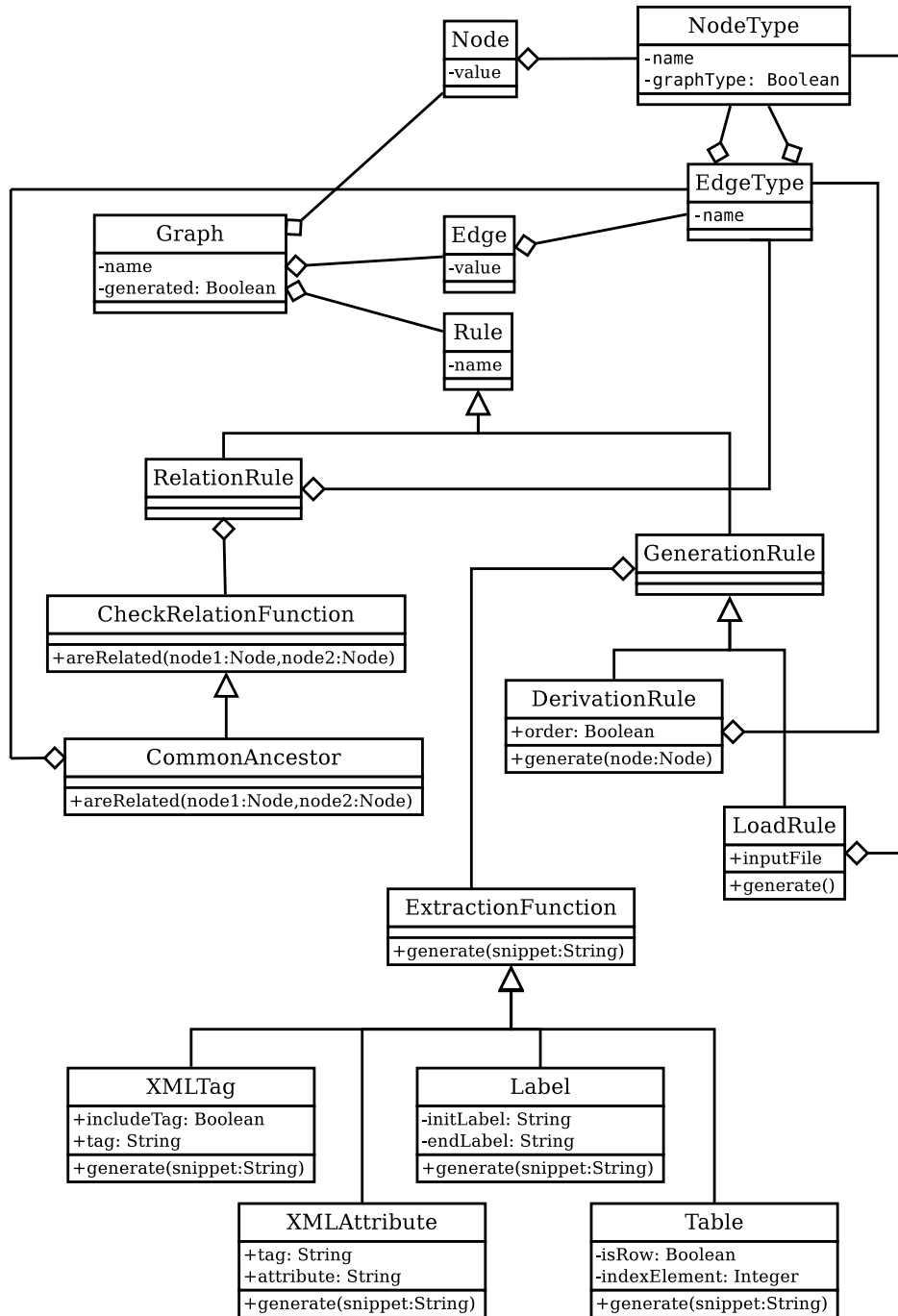


Figure 3.11: Data Model of GraphGen

- **Graph:** it is the entity that encompasses the remaining elements of the data model, giving them a work environment. That is, GraphGen allows the user to define as many graphs as are needed, but each element that is created with the tool (such as rules, nodes or edges) belongs to only one graph. The graph is described by a *name* provided by the user and by a status *generated*, which shows if the rules were already applied and the set of nodes and edges were created. This status determines the set of operations allowed. For instance, no more rules can be defined when a graph is already generated.
- **Node:** nodes of the graph will be generated by applying the Generation Rules over the input data sources. Each node contains a value and belongs to one node type, depending on which rule generated it.
- **Node Type:** nodes of the graph are generated through the different generation rules of the graph. Those rules, as will be described later, have associated a node type, identified by a name. All the nodes generated through a generation rule will have the same node type. Two different kinds of node types are contemplated in GraphGen. If it is defined as a *graph*, the set of nodes belonging to it will have unique values. Otherwise, the node type is composed by a list of nodes, where two different nodes can contain the same value.
- **Edge:** the edges of the graph are generated through the application of the rules defined for that graph. Both relation rules and generation rules can produce edges that relate two nodes of the graph. Note that relation rules establish connections between existing nodes, while generation rules create new nodes, relating the new derived nodes with their origin.
- **Edge Type:** edges of the graph are clustered in types, each of them is implicitly created when a rule is defined. An edge type is defined by a *name* and the origin and target node types. That is, all the edges belonging to an edge type start from the node type and point to the node type specified by the edge type.
- **Rule:** rules are the mechanism of *GraphGen* to define the graph generation. All the rules contain a name, and they can be classified in two big groups: relation rules and generation rules.
- **Generation Rule:** it defines how new nodes are generated in the graph. It can be a derivation rule, where the content of the new nodes are extracted from the content of an existing node, or a load rule, where the content is directly extracted from an input data source. For both kind of rules, an extraction function has to be defined, which will be applied over the input content to produce one or several new nodes (the value of the node or the content of the input data source).

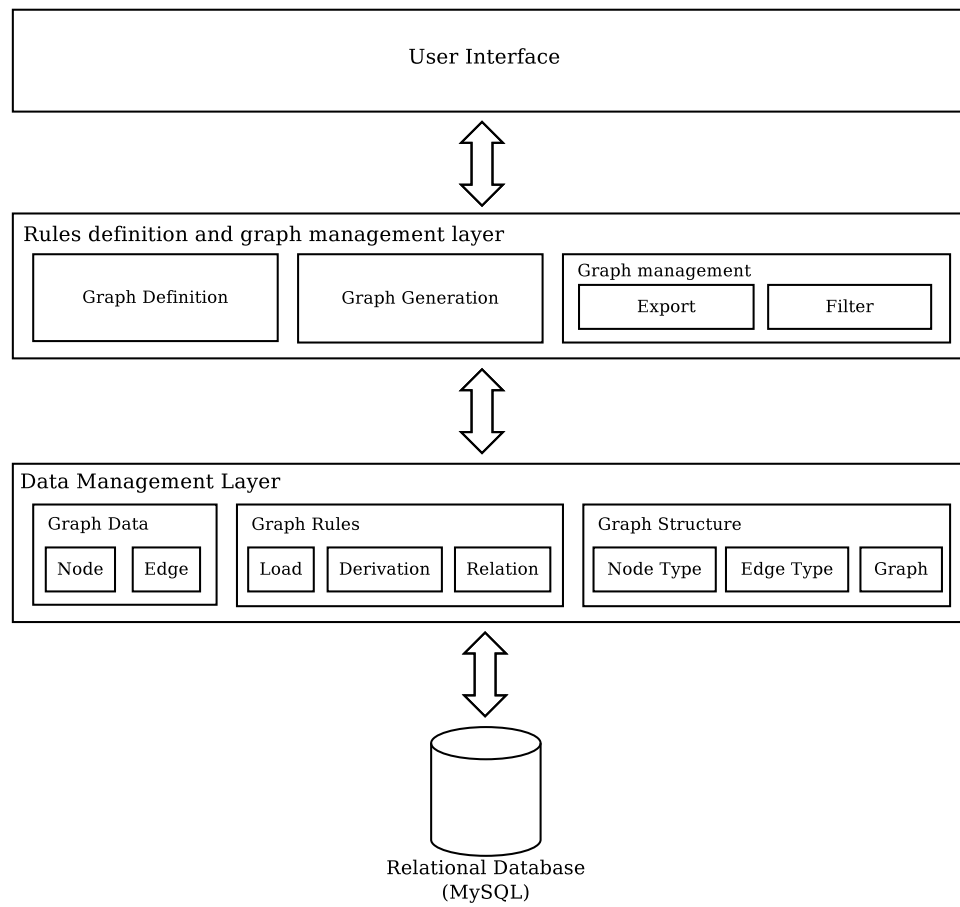
- **Derivation rule:** a derivation rule produces new nodes from existing nodes. It is composed by an edge type, determining the nodes over which this rule is applied (origin of the edge type) and the node type of the generated nodes (target of the edge type). It defines with a method how the new nodes are derived. This method, which is called *generate*, will delegate the processing of the new contents in the extraction function. The edges created from the origin node to the new nodes can be labelled by the order in which they were extracted from the content of the origin node: this behaviour is controlled through the parameter *order*.
- **Load rule:** a load rule produces the input nodes of the graph, whose content is extracted from an external data source. As in the derivation rule case, it delegates the processing of the content to an extraction function. Since content of the nodes is extracted from a data file, they will be source nodes, so no new edges are created with a load rule.
- **Extraction function:** extraction function defines a method *generate*, that receives a *snippet* (text) as a parameter and, by applying one extraction method, outputs a list of values. GraphGen provides some predefined extraction functions and more functions can be added in a simple way by extending the *Extraction Function* abstract class. Section 3.4.2 described those predefined functions, which are implemented as classes that inherit of the abstract class *ExtractionFunction*.
- **Relation Rule:** both of previous rules (*derivation* and *load*) produce new nodes by decomposition of existing nodes. However, the edges they create are only hierarchical, limited to relate each new node with the parent which produced it. Relation rules provide a more powerful mechanism to create edges between nodes without derivation relationships. A relation rule is defined by the edge type which all the new edges will belong to. The behaviour of a relation rule is implemented through a *check relation* function.
- **CheckRelation function:** it determines, for a pair of nodes, if a new edge have to be created between them. For such cases where the relation is created, it also defines the value of that new edge. Many different functions can be defined. Section 3.4.2 described the function *CommonAncestor*, currently supported in this tool.

3.6.2 Architecture and Design

In this section we describe the overall architecture of GraphGen. Our tool is implemented in Java, using Swing for the graphical user interface. All the relevant information is persisted in the relational database *MySQL*. This architecture is

presented in Figure 3.12. It is organized in four layers, from the persistence to the user interface. Next, each layer is detailed:

- **Data persistence:** all the non-temporary information of the application is persisted through this layer in a relational database. The database stores the graph structural information, defined by the user through the rule definition mechanism, but also the graph data when this graph is generated by the tool (including the values of the nodes and edges). Although we use a relational database to persist all the information, any other alternative could be used instead of modifying the data management layer which will be explained next. For instance, the graph instance (nodes and edges and its values) is a kind of data that fits with a key/value store database so they could be managed in stores like Redis or CoachDB. Although a more efficient implementation could maintain the full information in main memory, we decide to use a secondary storage in order to give priority to the scalability of the system, since the size of the graph, depending on the context, could overflow the capacity of a standard-size main memory.
- **Data management layer:** this layer manages the communication between the application and the database, providing methods to recover, create, modify and delete the elements of the data model (defined in Section 3.6.1). The objects of this data model can be classified, according to their role in the application, in three groups: **graph data**, including the nodes and edges of the graph and their values; **graph rules**, containing the rules defined by the user for each graph; and finally the **graph structure**, which includes the remaining elements needed to define a work environment in the application, like the types of the elements and the general information of each graph. The current version of the application implements this layer using a JDBC driver for MySQL.
- **Rules definition and graph management layer:** this layer provides implementations, structured in services, for all the possible actions that the user can perform in the application. Figure 3.13 shows the internal structure of this layer and the methods of the Data management layer they invoke. All the actions are structured in three services. *GraphDefinitionService* includes methods to define the input data sources, the rules to transform this data in the final graph and how nodes and edges are typified in this graph. The *GraphGenerationService* implements the process of, given the rules defined by the user applying them to the input data sources. The algorithm for this graph generation is detailed in next section. *GraphManagementService* includes the actions that work over a generated graph in order to obtain a sub-graph containing the desired data to be exported. Methods to prune the nodes and the edges of the generated graph can be done. In this way, such

**Figure 3.12:** GraphGen architecture

node (or edge) types that only represented intermediate states of processing can be deleted, because they only were needed in the generation process, but they do not belong to the final graph. After this filtering, the final graph can be exported through the method *export*, which transforms the internal representation of the graph in the relational database to the standard format *GraphML*. In order to optimize the implementation of these methods, specially such of them involving many I/O operations, some *cache* mechanisms in the main memory were implemented, which will be detailed in Section 3.6.4.

- **User interface:** All the interaction with GraphGen is carried out through a graphical user interface implemented with the Java GUI widget toolkit *Swing*. During the process of rule definition, the graph model is dynamically visualized (implemented with the library *Jung*, or Java Universal Network/Graph Framework).

3.6.3 Graph generation

In this section, we give more details about the algorithm we implement to generate the graph from the input datasets into a graph according to the set of rules defined by the user. The main challenge is to find an order of execution of the rules that is complete (all the elements that can be obtained by the application of the rules will be actually generated) and finite (the process always ends, for every set of rules). For that purposes, we start by applying all the load rules, which generate the input nodes of the graph. Then, the derivation rules are applied over every node generated in the graph, until all the nodes are completely explored and no more new nodes are generated. Finally, over the created nodes, new edges are created through the relation rules, in basis on all the nodes and the decomposition edges.

Load Rules are the first rules to be executed. They create the input nodes from the data collection. If this data collection is composed by complex objects of the same type, only one load rule will exist. Otherwise, a different load rule will be defined for each different input node type. The execution of a load rule is independent from the remaining load rules, so each rule is executed only one time, following the extraction function defined by the user in the rule. No edges are created in this step. Therefore, for each load rule, several input nodes can be extracted, whose values are given by the extraction function.

Derivation Rules Each application of a derivation rule will produce new nodes in the graph. Therefore, the derivation rule is applied over all the nodes belonging to the source node type of the rule. Taking into account that different derivation rules

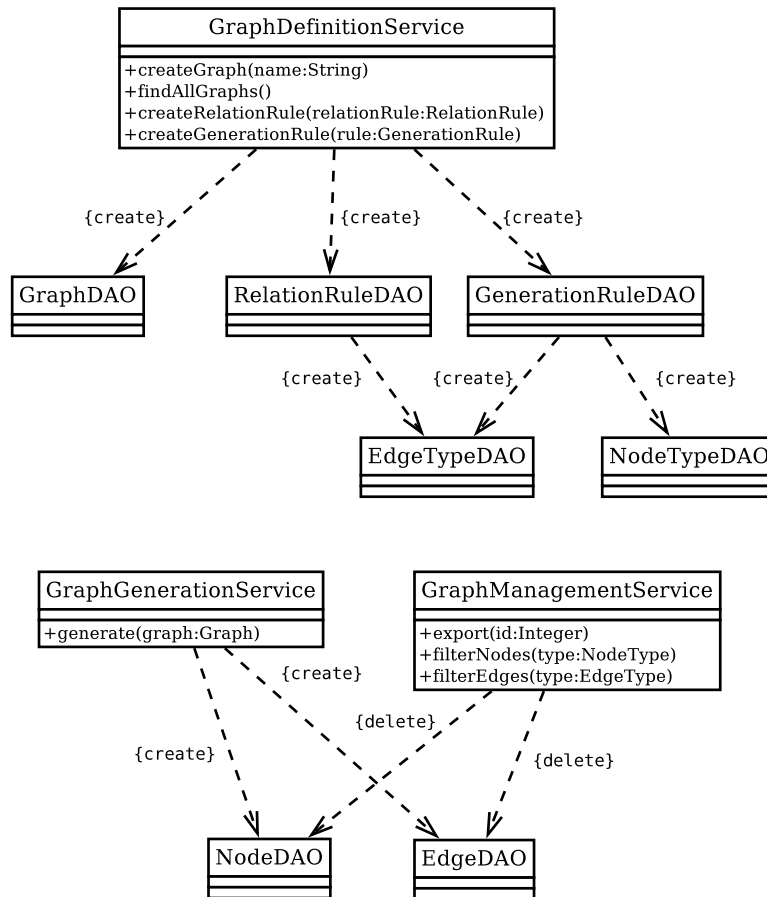


Figure 3.13: Dependencies between the services and Data Access Objects of GraphGen

can produce nodes of the same type, the sequential application of the derivation rules could miss the production of some nodes. One possible solution consists in, for each new node added to the graph, applying all the applicable derivation rules, that is, all the rules whose source type is the type of that node. This solution is complete and finite, since each rule is applied over a node once (and only once). However, it can be inefficient, because for every node we create, we have to check what derivation rules exist for that node type.

To summarize, the application of each derivation rule to the current set of existing source nodes may not produce the complete graph. On the other hand, the application of every rule over each node is a complete solution. However, it produces a lot of checks of the rules that are applicable to each node. Instead of this, we follow a hybrid (and complete) strategy. Our approach consists in applying the rules over the set of nodes of the same type to reduce the number of checkings for the applicability of rules. We start by several sets of input nodes.

Consider each load rule of the graph LR_i that produces n_i nodes belonging to the same type. We start the derivation rule application by creating a queue with l blocks of nodes, one for all the created nodes for each applied load rule. Therefore, all the elements of the block are of the same type. The next steps are repeated until the queue of blocks is empty:

- Extracting the next block composed by n_i nodes of the same type from the queue. Then, for their node type (which is the same for all the nodes of the block), the list of possible derivation rules that can be applied is obtained.
- Each derivation rule is applied over the n_i nodes, producing n'_i nodes that belong to the same target type. If the rule is a graph rule, then for each candidate node that the extraction function produces is only transformed in a real node if no node exists with the same value and type in the graph. Additionally, a new edge is created between the new node and the source node which produced it. Again, if the rule is a graph rule and the candidate node already existed in the graph, the new edge will relate the current source node with the existing node.
- Each set of nodes created as a result of the application of a derivation rule over the current block of nodes composes a new block of unexplored nodes which is added to the queue.

In this way, we implemented a complete generation algorithm, since all the nodes are explored one time (and only one) and all the possible rules are applied over each node. Furthermore, exploring nodes by blocks allows to optimize the process of computing the rules applied over each node.

Relation Rules After the application of the derivation rules, all the possible nodes are already created. Then, the relation rules can be applied. As in the case of the load rules, the result of a relation rule does not affect to the remaining rules, since each rule produces a set of independent edges. Then, each relation rule is applied over the current graph following the same steps:

- The set of existing nodes of type TN_1 of the relation rule is obtained.
- In the same way, the set of existing nodes of type TN_2 of the relation rule is obtained.
- For each pair $(n, m) \in TN_1 \times TN_2$ the check relation function is computed. If the function returns *true* as result, then a new edge can be created. If an edge of the same type already existed between the two nodes, then the weight of the edge is increased, instead of creating a new edge.

3.6.4 Memory management

As a consequence of the in secondary memory storage, some operations involve a lot of I/O communication affecting to the performance and the scalability of GraphGen. In order to optimize the most critical operations, some cache structures were designed. In this section, we describe two optimizations we implemented. First, we explain how the descendants of an XML tag are efficiently maintained to work with XML files in GraphGen. After that, the *caching* of the current working nodes used to optimize the generation process is described.

3.6.4.1 XML assistance

GraphGen provides some extraction functions specially designed for XML data. When the input data is an XML file, the elements are divided progressively by extracting descendant XML tags of the file. The derivation rules specify the name of the tag delimiting the content of the XML in the same node to be extracted as content of the target node, which will be obtained from a source node. In order to facilitate the definition of these rules, we provide to the user the descendant tags for a given tag in the XML dataset. When no schema (DTD or XSD) is provided, these suggestions require the full exploration of the text. We optimize this process by maintaining the descendants of each tag in main memory.

In addition to that, when a huge input XML dataset is progressively divided, the contents of the complex nodes that are extracted from the input nodes can be very large. Consider an XML register of a libraries network, where each book information (tagged with `< book >`) is a descendant of the *library* where it is located (tagged with

$\langle library \rangle$). A first decomposition rule could create nodes $\langle library \rangle$, whose value would be the XML text with all the books of this library. If all of this values were explicitly stored, the needs of space would be very high. Therefore, depending on the size of the value of a node, we contemplate two alternative approaches: for small values the value is explicitly stored. However, for huge values, only a pointer to its initial and final position in the input data source is stored, saving space but also allowing faster access when this value is loaded to continue its division. This mechanism was implemented using the facilities that a SAX parser provides.

3.6.4.2 Temporal data in main memory

During the application of a rule, we maintain the set of current nodes whose type is the source or the target of that rule in main memory. In this way, for the graph rules, checking the existence of a node with the same type and value is faster. Furthermore, the storage in the relational database is performed by blocks of nodes. That is, when a derivation rule is applied over a block of nodes, all the new nodes created as a result are saved in the database as a block, reducing the I/O communication.

3.7 Summary

In this chapter we designed a model to generate graphs from arbitrary data. This graph transformation is defined through different kind of rules (load rules, derivation rules and finally the relation rules), which allows the user to specify how the input datasources are decomposed in elements (nodes of the final graph) and how they are connected (edges of the graph). We proposed a tool implementing this model: GraphGen. GraphGen is a general purpose application, since rule definition is generic and it does not include any domain-specific transformation. It can be used to transform data from any domain into a graph through rules that are defined by the user through a graphical interface.

We have experimentally validated GraphGen by transforming three datasets from different domains and formats into a standard graph. We create a graph of reviewer compatibility from a bibliographic database, a graph relating the query search terms with the most related URLs from a query log and a graph with tagged photos from a social network.

Although the design of this application does not pursue achieving a good spatial and temporal efficiency, the results obtained with GraphGen in the use cases show that GraphGen can be used to generate graphs with tens of millions of elements in less than an hour.

Chapter 4

Representing Graphs with attributes using K^2 -trees

In this chapter we propose a compact data structure to store graphs with attributes (*attributed graphs*) based on the K^2 -tree, a very compact data structure designed to represent a simple directed graph, which can be also seen as a binary relationship. The idea we propose can be seen as an extension of the K^2 -tree to support attributed graph models. We also provide an implementation of a basic set of operations, which can be combined to form complex queries over these graphs with attributes. We evaluate the performance of this system and we study some fields of application for this static but compact attributed graph representation.

This chapter is structured as follows. Section 4.1 describes the most representative tools existing in the State of the Art to manage attributed graphs. In Section 4.2 we present our compact data structure to store such graphs with attributes. This section focuses on the physical storage. Section 4.3 presents the operations implemented in $AttK^2$ -tree. Finally, Section 4.4 provides an experimental evaluation of the system using representative cases of study where we compare our proposal against another attributed graph representations in the State of the Art.

4.1 Systems for Attributed Graphs

Last years, Graph Database Models have been proposed to represent attributed graphs. They specify the data, the queries, the results and, in many cases, even the schema as a graph [AG08]. Many theoretical models and their corresponding query languages were proposed to represent and navigate graphs. Some examples

are the *Hypernode Model* [LP90], which main feature is that nodes can be a graphs by themselves; and GOOD (Graph Oriented Database Model) [GPVdBVG94], where data manipulation operations (insertions and deletions of nodes and edges and clustering of nodes depending on some properties) are specified as graph transformations.

Built over those theoretical models, many practical Graph Databases Engines have been proposed. In this section we describe some of the most relevant works in this area, providing their internal data structures.

4.1.1 DEX

DEX [MBÁLMM⁺12, MBMMGV⁺07] is a graph database that efficiently stores and queries labelled and directed attributed multi-graphs. It keeps the graphs in secondary memory using different bitmaps. The graph model of DEX defines labelled nodes and directed edges where extra information is associated to each node and edge, represented as a list of attributes. Therefore a Graph in DEX is defined as $G = (V, E, L, T, H, \{A_1 \dots A_n\})$ where:

- V denotes the set of node keys.
- E defines the set of edge keys.
- L is a key-value list that includes, for each key node (or key edge) its label.
- T and H represent, for each key edge, the keys of its corresponding origin (tail) and target (head).
- Each $\{A_1 \dots A_n\}$ represents a different attribute. Nodes and edges of the graph can take values for some of these attributes.

DEX represents this graph model through a set of maps. Figure 4.1 shows an example of a graph internal representation in DEX. The figure at the top shows a graph for a social network where two members (*Carrie* and *Saul*) are joined in different groups (a *reading group* and a *football team*). Users can be related through a *follow* relationship and users can participate in the groups through two different relationships (*join* for a *reading group* and *play* for a *football team*) described by different attributes. Elements of the graph (nodes and edges) have an object identifier associated. For instance, the object identifier of the member *Carrie* is $O1$.

The Figure at the bottom shows the bitmaps used in DEX to represent this graph. Labels storage is located on the left. A map (implemented using a $B+$ -tree) points from each object identifier (node or edge) to its corresponding label value. For instance, the object identifier 3 (o_3) is related to the label *group* (since it represents

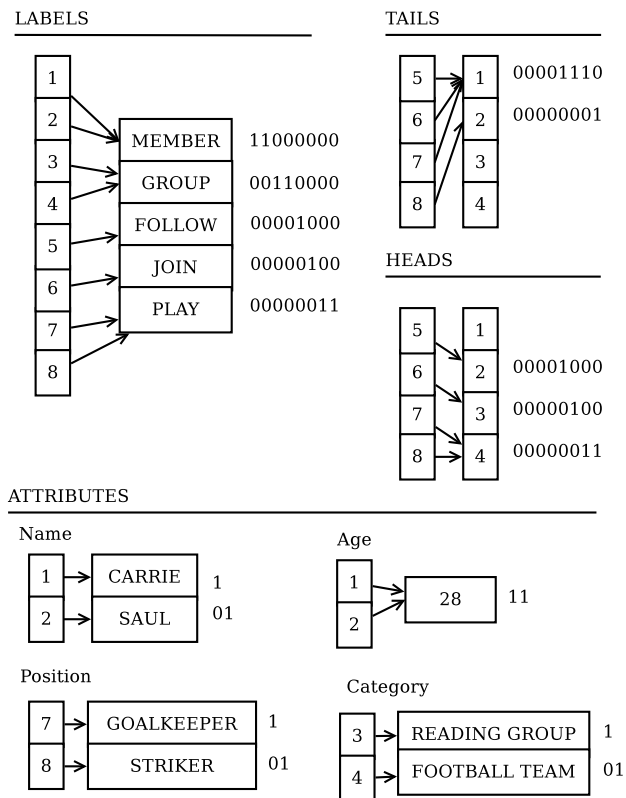
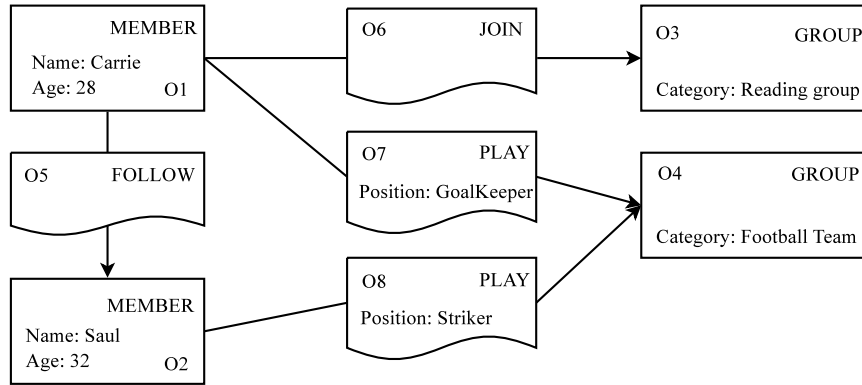


Figure 4.1: DEX internal representation (bottom) for a labelled attributed graph (top)

the *Reading group*). Each label has associated a compressed bitmap that conceptually contains as many *bits* as objects the graph has. For a label value, a *one* in the position i means that o_i is labelled with that value. However, in practice, bitmaps are only stored until the position of the last *one*. In this way, for the non-stored positions a *zero* value is inferred for this label. The $B+$ -tree and these compressed bitmaps compose a double mapping. The purpose of storing this double mapping is to provide a bidirectional navigation. The first map is used to obtain the label of each object identifier. On the other hand, the bitmap answers the opposite query: given a label value (for instance, *member*), obtaining the objects with that label is performed by retrieving the *ones* in the corresponding bitmap. The bitmap for *member* is [11000000], meaning that o_1 and o_2 are objects with the *member* label. The remainder information for the graph is managed in a similar way: a double map is used to represent the tails of the edges, another one represents the heads and finally one map per attribute is stored.

The main purpose of this internal structure is to provide bidirectional access. That is, the data for a given node can be recovered using the maps indexed by its identifier. On the other hand, given a label or an attribute value, finding the nodes or edges with this label or value is performed by checking its corresponding bitmap and recovering the positions with a value *one*. Direct and reverse neighbor nodes are recovered by using the bitmaps *head* and *tail*.

DEX query engine is built over this internal representation. It efficiently implements a small set of primitives. More complex queries are built on the top of this core engine.

4.1.2 Neo4J

Neo4j is an open-source Graph Database which supports the storage and query of labelled directed attributed graphs. A graph in Neo4J can be defined as $G = (N, E)$, where N is a set of nodes and E is the set of edges. Each node is a pair $n_i = (L_i, P_i)$. L_i is the set of labels and P_i is the set of properties (or attributes) of the node. Labels of a node can be seen as tags. They can be used to define constraints over a group of nodes, to represent temporary states of nodes or, in general, to define a target group of nodes over which an operation will be performed. A property $p_j \in P_i$ is a key-value pair, where the value can be a primitive (the typical primitive types of any programming language like Boolean, Integer and String are supported) or a list of elements from one of these primitive types. An edge of a graph in Neo4j is defined as $e_i = (n_j, n_k, t_i, P_i)$, where n_j and n_k are the nodes related through this edge, t_i is the label of the edge and P_i is the set of properties the edge contains (equivalent to the properties of the nodes). Neo4j defines its own query and update language, Cypher, which is a declarative language, where data is obtained by pattern matching.

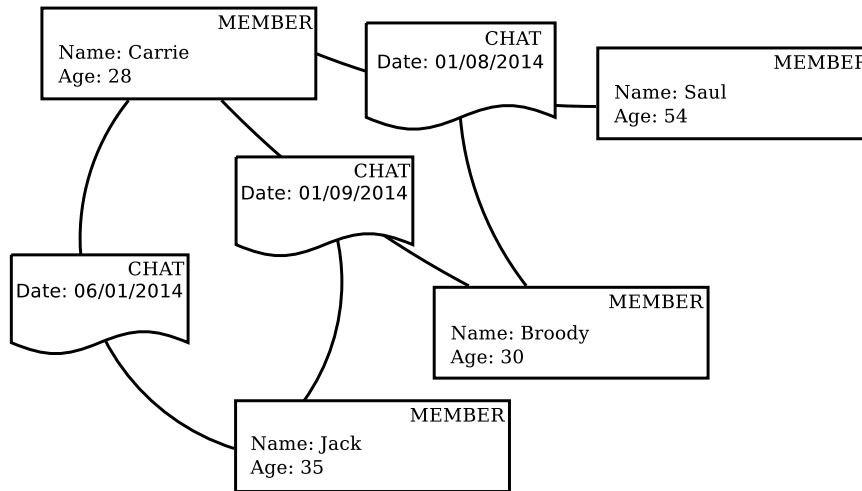


Figure 4.2: An example of hypergraph model including n -ary edges

4.1.3 HyperGraph

Hypergraph DB [Ior10] is a graph database based on the Hypergraph model. It supports a hypergraph $HG = (N, E)$, where N is the set of nodes and E is the set of edges. Each edge $e_i \in E$ relates two or more nodes, that is, $e_i = \{n_j \in N\}, |\{n_j\}| \geq 2$. The hypergraph defines a more expressive structure that can be useful to model domains where more than two entities are usually related. For instance, each conversation of people in an online chat program could be modelled using this hypergraph model as an edge that relates the participants in the conversation. Figure 4.2 shows an example of conversations along the time. For instance, the chat conversation in 01/08/2014 is represented as an edge that involves 3 members.

Hypergraph uses an atom as the basic unit of representation. It contains a *typed value* and a *target set* composed by a set of atoms. When the number of atoms in the target set is 0, atoms are called nodes. On the other hand, when at least one atom is associated with it, the atom is a link.

The storage of Hypergraph DB is basically distributed in two layers. The primitive storage layer includes the information of the links (for each edge identifier the set of related identifiers is stored) and the data (the RAW value corresponding to each identifier). Over this primitive layer, the model layer manages the type system, the querying engine and some optimization facilities like caching and indexing. Hypergraph DB is physically stored using a key-value store.

The Hypergraph DB query engine provides two different ways of specifying the

query: an API to define standard graph traversals and a SQL-style language where a set of constraints over the required atoms are set.

4.1.4 Other systems

Last years, many other graph database systems have emerged. They are focused on managing large amounts of data in a very efficient way. OrientDB¹ is a good example, which is Document and Graph oriented, implemented in Java and uses SQL as query language. Finally, many proposals were designed to work in distributed environments. Titan², Giraph³ or Pregel [MAB⁺10] are just some examples.

4.1.5 Blueprints Graph API

As we reviewed in this section, many different Graph Databases solutions were proposed in the last decades. Each one defines its own graph model. Although, in general, all of those approaches are designed to support attributed graphs, their models present small differences regarding to the type and label system and other features. Furthermore, different graph database systems support different query languages. Some of them are based on languages from other applications like SQL or SPARQL. Others define ad-hoc APIs composed by specific primitives. Some of them define a complete new query language, such as Neo4j.

In this situation, when a new graph application is developed, it has to be implemented over an specific Graph Database, being platform-dependent. The Property Graph Model interface named Blueprints⁴ was proposed to overcome that problem, providing a common graph API to facilitate the integration between the applications and the graph stores. In this way, graph databases that implement the Blueprints API can be used by any Blueprints application. Likewise, a Blueprints application can use any graph database system that implements the Blueprints API interface. Nowadays, the most relevant graph databases like Neo4j, OrientDB, DEX and Titan support this interface.

4.2 Our proposal: Att K^2 -tree

In this section, we propose our system we called **Attributed K^2 -tree (Att K^2 -tree)** to efficiently store and manage attributed graphs. The internal representation

¹<http://www.orienttechnologies.com/orientdb/>

²<http://thinkaurelius.github.io/titan/>

³<http://giraph.apache.org/>

⁴<http://blueprints.tinkerpop.com/>

of the graphs is based on the K^2 -tree, a static data structure designed to work in main memory. Therefore, we propose an in-main-memory compact attributed graph representation designed to be used in contexts where big amounts of static data need to be intensively queried.

4.2.1 Graph model supported by $AttK^2$ -tree

We described several attributed graph stores in Section 4.1. All of them are based on specific attributed graph models, presenting small differences between them. Therefore, before describing the internal representation of our structure, we first consider the features of the attributed graph model that $AttK^2$ -tree supports. Figure 4.3 shows an example of graph supported by $AttK^2$ -tree. It represents a paper authorship and review network, where several information about papers and their corresponding authors is included. Authors and papers are modelled as nodes in the graph, and the different collaborations between them are reflected as edges (like thesis direction or collaboration in a research project). Researchers are related to the papers that they co-authored through edges. Researchers can also be related with a paper through a *review* relation. This graph is an example of the data that could provide support to an application searching for conflicts of interest to assign the paper reviewers.

The graph model of $AttK^2$ -tree presents the following properties:

- **Directed graph:** Edges of the graph will be directed, meaning they distinguish between origin and target node. Figure 4.3 shows how edges from the graph explicitly identify its origin and target nodes. For instance, edge e_1 represents the authorship of a paper, having researcher n_3 as origin and the paper n_1 as target of the edge. As usual, an edge of an undirected graph could be also represented in this model by using two directed edges (in opposed directions) between the two nodes it relates.
- **Attributed graph:** Attributes or properties are the most meaningful characteristic of general graphs. Many approaches can be followed to define the attributes of an element, including complex data types, range domains and another constraints over each attribute. However, we define a more simplistic conception of attributes. Each node and each edge of the graph is described through a set of attribute-value pairs. Values are not restricted to a domain or a data type. They can take any value which will be managed as plain text. In Figure 4.3, we can observe that the edge e_6 takes the value *Medium* for the attribute *Expertise*.
- **Labeled graph:** Several definitions can be considered for a labeled graph. As it was described in Section 4.1, DEX considers that each component of the

graph (nodes and edges) contains a unique label (or main value) that identifies the kind of element it belongs to. On the other hand, labels in Neo4J are considered as tags, supporting the definition of multiple tags for each element. We consider, in line with DEX, that each element (node or edge) of the graph has just a label, which we call type. This type determinates the attributes that an element of that type can contain. In that sense, the label and the list of valid attributes for each label composes a *schema* which can be very helpful to work with domains with structured data. Figure 4.3 shows the label of each node and edge. In this example, two different labels are contemplated for the nodes. That is, that graph has only two node types: *researcher* and *paper*. The label determines the attributes that describe each node. Researchers are described through the attributes *name*, *university* (where they work) and *position* in that university. On the other hand, papers are described through the *title* and the main *topic* of the paper. Edges of the graph can be labelled with the labels *author*, *PhDDirector*, *reviewer* and *colleague*. *Colleague* relates two researches which have collaborated in some research projects. To summarize, labels are used to identify the type of a node or edge.

- Multigraph: Att K^2 -tree does not constrain the number of edges which connect two nodes, so many edges with the same origin and the same target can be defined. This characteristic is useful to represent in a natural way contexts where several kind of relationships can be established between two nodes. This multi-edge nature, combined with the labelled and attributed properties, makes this model very expressive. Therefore Att K^2 -tree can fit with the structural properties of many real graphs. Figure 4.3 shows an example of multigraph, where nodes n_4 and n_5 are related through two edges (e_4 and e_5) representing relationships with different nature (*PhDDirector* and *colleague*) between those nodes

4.2.1.1 Formal definition

A formal definition of a labeled, directed, attributed and multi (*LDAM*) graph is represented as a 10-tuple $G = (L_N, L_E, N, E, R, L_A, S_N, S_E, A_N, A_E)$ where

- L_N is the set of possible labels that the nodes of the graph can take. For the graph in the Figure 4.3, $L_N = \{Paper, Researcher\}$. They are the node types.
- L_E is the set of possible labels that the edges of the graph can take. Regarding to the same example, $L_E = \{Author, Colleague, PhDDirector, Reviewer\}$.
- $N = \{n_i, l_j\}$ is the set of nodes, being $n_i \in 1 \dots |N|$ a numeric identifier of the node and $l_j \in L_N$ the label of the node. In the example, the set of nodes

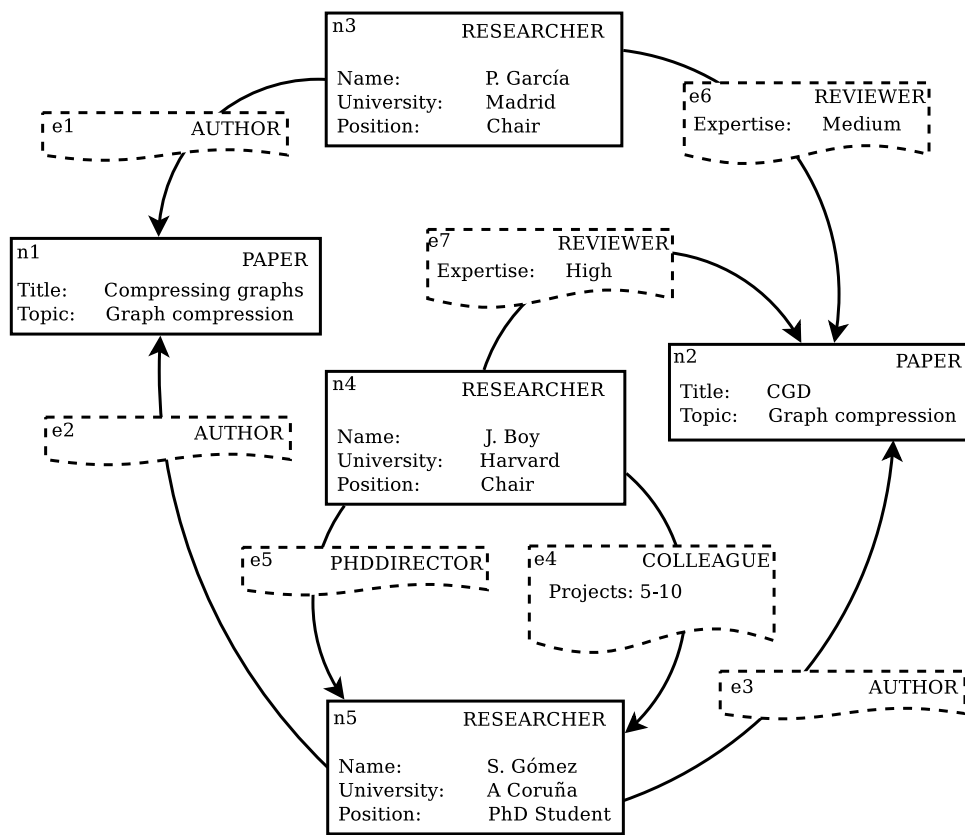


Figure 4.3: Example of labelled, directed attributed multigraph

is composed by $N = \{(1, Paper), (2, Paper), (3, Researcher), (4, Researcher), (5, Researcher)\}$.

- $E = \{e_i, l_j\}$ defines the set of edges where $e_i \in 1 \dots |E|$ is the identifier of the edge and $l_j \in L_E$ is its label. The set of edges of the graph in Figure 4.3 is $E = \{(1, Author), (2, Author), (3, Author), (4, Colleague), (5, PhDDirector), (6, Reviewer), (7, Reviewer)\}$.
- R contains the relations between the nodes, that is, the origin and target of the edges. Each element of R is a triple (e_i, o_i, t_i) , where e_i is the edge identifier, o_i the node identifier which is the origin of the edge and t_i the node identifier playing the role of target in the edge. It is easy to note that, by definition, $|R| = |E|$. In the example:

$$R = \{(1, 3, 1), (2, 5, 1), (3, 5, 2), (4, 4, 5), (5, 4, 5), (6, 3, 2), (7, 4, 2)\}$$

- $L_A = \{a_i\}$ is the set of the different attribute labels of the graph. In other words, L_A is the union of all different properties that describe the nodes and the edges of the graph. In the example,

$$L_A = \{Title, Topic, Name, University, Position, Expertise\}$$

- $S_N = \{sn_i\}$ is the set of schemas for the types of the nodes. Each element of S_N defines the set of attributes for a node type. Each element sn_i of the schema is represented as a pair $(l_i, \{a_j\})$, where the node label $l_i \in L_N$ has associated a set of attributes $a_j \in L_A$ which define that node type. Note that an attribute is not exclusive of a node type. In other words, several node types can be defined through the same attribute. Next, the schema of the nodes for the graph in Figure 4.3 is shown:

Label (l_i)	Attributes($\{a_j\}$)
Paper	$\{Title, Topic\}$
Researcher	$\{Name, University, Position\}$

- $S_E = \{se_i\}$ is the set of schemas for the types of the edges, in a completely analogous way to S_N . Each element of S_E defines a valid schema for an edge type. Each $se_i = (l_i, \{a_j\})$ is a pair where $l_i \in L_E$ is the corresponding label and $\{a_j\}$ is the set of valid attributes for each edge type. Next table details the schema of the edges for the same example:

Label (l_i)	Attributes ($\{a_j\}$)
Author	{ }
Colleague	{ <i>Projects</i> }
PhDDirector	{ }
Reviewer	{ <i>Expertise</i> }

- $N_A = \{(n_i, a_j, v_k)\}$ defines the properties of the nodes. It is a set of triples, where each triple defines the value v_k that the node $n_i \in 1 \dots |N|$ takes for the attribute $a_j \in L_A$. Note that a triple (n_i, a_j, v_k) is valid in a data source if $\exists l_m \mid (n_i, l_m) \in N \wedge (l_m, \{\dots a_j \dots\}) \in S_N$. That is, a node can only take a value for an attribute included in this schema, given by this node type. For instance, the set of triples describing the properties of the node n_3 in Figure 4.3 are:

Node Identifier (n_i)	Attribute (a_j)	Value (v_k)
1	Name	P. Garcia
1	University	Madrid
1	Position	Lecturer

- $E_A = \{(e_i, a_j, v_k)\}$ describes the properties of the edges (analogously to N_A). As an example, the triple describing the edge e_6 is provided:

Edge Identifier (e_i)	Attribute (a_j)	Value (v_k)
6	Expertise	Medium

Next, we detail the internal representation of AttK²-tree designed to support the graph model presented in this section.

4.2.2 Data structure

AttK²-tree stores a directed, attributed and labelled multi-graph by storing binary relationships with K²-tree structures. It is a compressed solution composed by a set of K²-trees and some additional structures. The graph is represented by three sub-systems: the schema of the data, the data included in the nodes and the edges and, finally, the relations between the elements of the graph topology. Next, we present the three sub-systems.

4.2.2.1 Schema

The schema manages the labels (*types* in the model) and the attributes describing each type. This schema layer works as a simple index to the other subsystems in $\text{Att}K^2$ -tree. The elements of the graph model L_N, L_E, N, E, S_N, S_E are stored in this schema layer. Figure 4.4 on the left shows the schema storage for the graph of the example. It is composed by:

- **Nodes Schema:** it is stored by a table where each row r_j represents the information of a label $l_i \in L_N$. Rows are ordered lexicographically by the label. The nodes of the graph will have identifiers according to the order of the labels in this Schema. That is, the m_1 nodes with the first type in the Schema, will have identifiers from $1 \dots m_1$. The m_2 nodes from the second node type will have a range of identifiers $m_1 + 1 \dots m_1 + m_2$ and so on. Each entry of the nodes schema will store the highest node identifier with this label. Figure 4.4 shows the nodes schema for the graph of the Figure 4.3. It has two entries: *paper* (having 2 as the highest identifier) and *researcher* (with limit 5). That means that the nodes with identifiers in the range $1 \dots 2$ are papers, while the nodes with identifiers from 3 to 5 are researchers. Each label also points to its valid attributes in the data subsystem.
- **Edges schema:** a table describing the characteristics of the different edge types is implemented in the same way that the Nodes Schema. Therefore, the edge identifiers will also be ordered by type. In that way, given an edge identifier, its corresponding type can be computed by performing a binary search over the entries in the Schema. For instance, to recover the type of the edge 6 in the Figure 4.4, a binary search over the upper limits is performed, until reaching the range $6 \dots 7$ that includes it, concluding that the node type of the edge 6 is *Reviewer*.

The schema layer is the starting point of the internal representation of the graph in $\text{Att}K^2$ -tree, providing indexed access to the other two layers. It is used to retrieve the ranges of identifiers for a label, and to recover the label for a given identifier. It also stores references to the valid properties for a given label.

4.2.2.2 Data

The *Data* layer describes the nodes and edges of the graph through their properties. It stores the values that each element of the graph (node or edge) takes for each valid attribute according to each type. Each different attribute can be represented in two different ways depending on the frequency distribution of its value. The first kind are the *Dense Attributes*, where many nodes or edges of the graph share the same

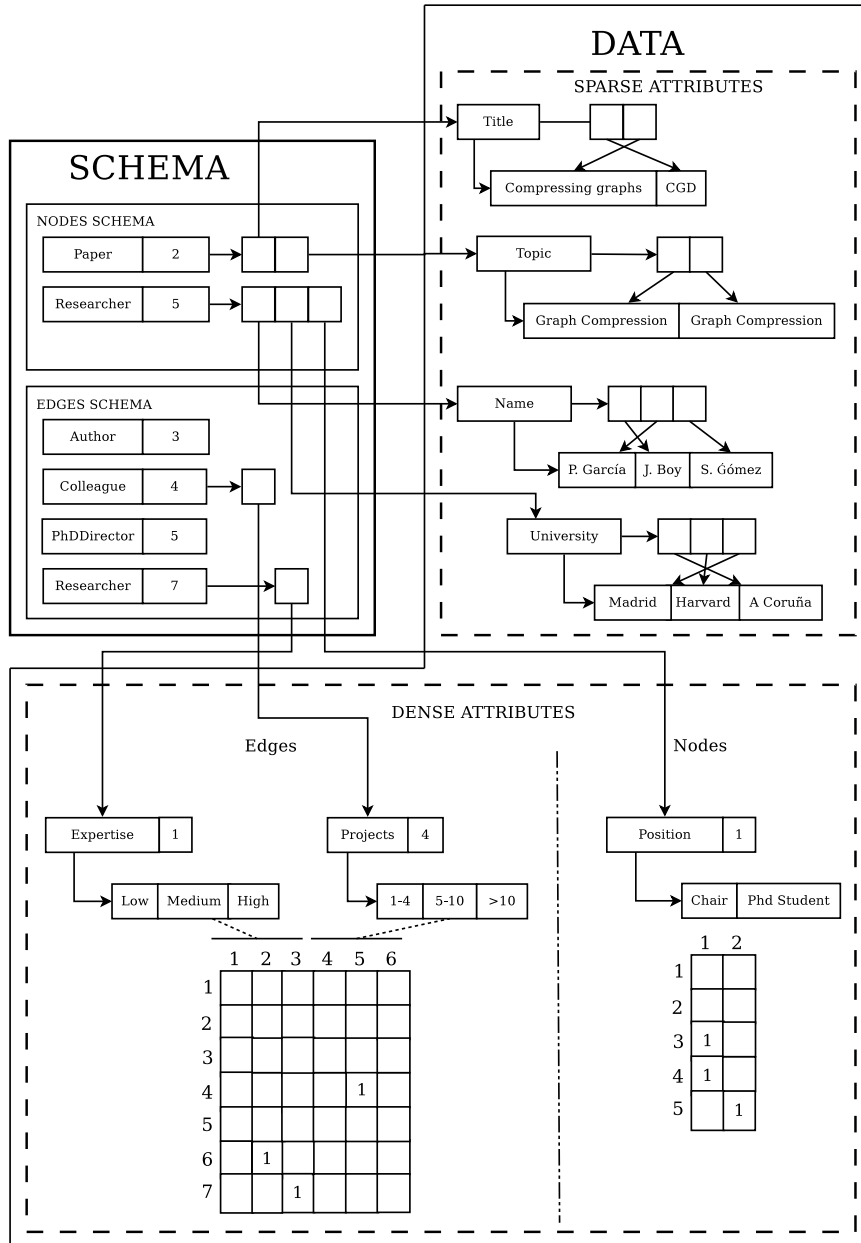


Figure 4.4: Internal representation of Schema and Data subsystems in AttK²-tree

value for that attribute. In opposition to the dense attributes, in *Sparse Attributes* nodes or edges usually take different values for that attribute. Titles, URLs or identifiers are common examples of sparse attributes whether age or nationality are examples of dense attributes. This two kind of attributes will have a different internal representation in $\text{Att}K^2$ -tree:

- **Sparse attributes:** attributes where elements usually take different values will be stored as a list indexed by element identifier. This list is double-indexed: in addition to the implicit index by element identifier, there is an additional index to maintain the entries in lexicographical ordering. This additional index is used to recover the elements taking a specific value by a binary search. Figure 4.4 (top-right) shows four Sparse Attributes: *Title*, *Topic*, *Name*, *University*. For instance, *Name* is an attribute valid for the type *Researcher*. The values in the list are sorted by node identifier. The first element of this list (*P. Garcia*) is the value that the first researcher (n_3) takes for the attribute *name*. Given a node (n_i, l_j) , its value for a sparse value will be in the position $i - \text{limit} + 1$, where *limit* represents the lowest node identifier of the type l_j . The additional index provides support to perform a binary search over the attribute values. We can see in the example that the first element of this additional index in the attribute *Name* points to *J. Boy*, the first element of the list in a lexicographical order.
- **Dense attributes:** All the dense attributes of the graph are stored in just two K^2 -trees: a K^2 -tree for the dense attributes of the nodes and another K^2 -tree for the dense attributes of the edges. Next, the construction of the K^2 -tree for edge attributes is shown (the K^2 -tree for the nodes is built in the same way). Each dense attribute A_i can be seen as a binary relationship between the $|E|$ edges and the set of different values that the edges take for that attribute. These relationships can be represented in consecutive columns of the adjacency matrix. Rows of the adjacency matrix represent the edges of the graph, ordered by their identifiers. Columns will represent the possible different values of each attribute. Each group of consecutive columns represents the different values for an attribute. A 1 in a cell (i, j) of this adjacency matrix means that the edge with identifier i takes the value j for the attribute located in this range of columns. This adjacency matrix will be represented by a K^2 -tree. An additional structure stores, for each attribute, the block of columns which correspond to this attribute, and the specific values that represent each column. Figure 4.4 (bottom-right) shows the representation of the Dense Attributes. The adjacency matrix for the nodes is on the right, where the 5 rows represent the 5 nodes of the graph. The adjacency matrix on the left contains a row for each one of the 7 edges. On the top of this adjacency matrix, the meaning of each column is specified by several lists. The attribute *Expertise* contains three possible values (*Low, Medium, High*). This attribute includes the index

one, locating its three columns from the column 1 of the global adjacency matrix. Then, the cell (6, 2) which contains a *one*, means that the edge e_6 values *Medium* for the attribute *Expertise*. On the other hand, the attribute *Projects*, which is specified in the Schema as a valid attribute for the edge type *Colleague*, contains three possible values which starts from the column 4 of the global adjacency matrix. In that way, the 1 in cell (4, 5) means that e_4 takes the value 5-10 for the attribute *Projects*. Note that in some regions of this matrix, due to the schema constraints, no ones can appear. For instance, the matrix between the rows 1 ... 3 and the columns 1 ... 3 is empty because the label *Author* does not have the attribute *Expertise*, according to the schema.

Note that for some attributes, the choice of representing them as a sparse or dense attribute could be not obvious. A possible criteria could be based on the number of different values regarding to the number of elements taking a value for that attribute. Experimental evaluation in Section 4.4 includes an analysis of behavior of these two kinds of storage.

4.2.2.3 Relations

The third subsystem of the AttK²-tree stores the *Relations*, that is, the different edges that connect the nodes of the graph. We store these relations with a K²-tree, extended to store the edge identifiers corresponding to each connection.

K²-tree represents, in a very compact way, simple graphs that can be described through a binary adjacency matrix. A *one* in a cell (i, j) shows the existence of an edge from the node i pointing to the node j . However, additional information is needed to store the relations in AttK²-tree. First of all, each one of the matrix has to be related to its edge identifier, which is used as pointer to the data layer (for instance, to recover the attributes of that connection). On the other hand, AttK²-tree supports multi-graphs. That means that more than one edge can relate a pair of nodes. So, several edges can be represented in the same cell of the adjacency matrix. Figure 4.5 shows the relationships of the same example and the corresponding adjacency matrix containing those edge identifiers. For instance, the cell (4, 5) contains two edge identifiers because two different edges connect n_4 and n_5 in the original graph (e_4, e_5).

The relationships in AttK²-tree are stored with the original K²-tree and some additional structures to represent multi-edges and to trace their edge identifiers. Figure 4.5 shows the structure we call Multi-edge K²-tree, composed by the following elements:

- **K²-tree** A K²-tree is built to represent a binary relation among nodes in which two nodes are related if at least one edge connects them in the original

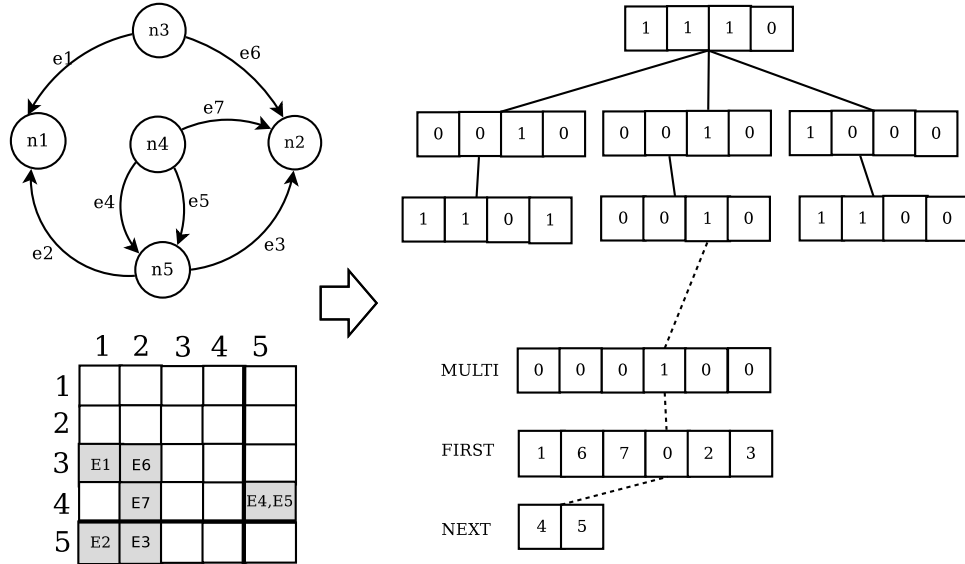


Figure 4.5: Multi-subsystems in $\text{Att}K^2$ -tree

graph. Figure 4.5 shows the K^2 -tree corresponding to the same example. It is a standard K^2 -tree except by the fact that in this case, the bitmap of the last level also needs an additional structure to perform rank and select operations over them. The relative position of the bits with value 1 in the last level of the tree will be used as an index to the *Multi* bitmap (which will be explained in the next paragraph). So, when a leaf is reached, a rank operation over the last level of the tree will provide its relative position in the *Multi* bitmap. For instance, last level of the tree in Figure 4.5 contains 6 *ones*. If we perform a rank operation over the bitmap until the 7-th position, we have that this is the fourth leaf of the K^2 -tree.

- **Multi** Each leaf with a *one* value in the K^2 -tree represents one or several edges. *Multi* is a bitmap which stores, for each *one* element of the leaf level, whether it is a multiple edge (or it is representing only one edge). Therefore, $\text{Multi}[i]$ will have value *one* if the i -th one of the K^2 -tree is clustering multiple edges. In the example, only the fourth position of the bitmap *Multi* contains a *one* value (clustering the edge e_4 and e_5). This information is used to read the next array (*First*).
- **First** This array stores the first edge identifier of each *one* of the K^2 -tree. For the i -th one of the leaf level, if it is a single edge (that is, if $\text{Multi}[i] = 0$) then $\text{First}[i]$ contains the identifier of that edge. Otherwise, when the i -th one is a

multiple edge, $First[i]$ represents the position in $Next$ array, where the first edge is located. In the example of the Figure 4.5, $Multi[1] = 0 \wedge First[1] = 1$, so the first edge (corresponding to the cell (3, 1)) has identifier 1. On the other hand, $Multi[4] = 1 \wedge First[4] = 0$ so the first edge of the cell (4, 5) will be in $Next[0]$.

- **Next** This array contains the identifiers of the multi-edges and it is indexed by the $First$ and $Multi$ arrays. Figure 4.5 shows the two identifiers for the only multi-edge of the example: e_4 and e_5 , corresponding with the cell (4, 5).

The three layers Schema, Data and Relationships compose the internal representation of $AttK^2$ -tree, used to store directed, attributed, labelled multi-graphs. These structures, based on the usage of K^2 -trees, were designed to provide a compressed representation of attributed graphs, which could be accessed to three basic queries. The next section presents the navigation over the internal representation of $AttK^2$ -tree.

4.3 Navigation and operations

We present the query API of $AttK^2$ -tree composed by a set of basic operations over attributed graphs. This API aims to provide a basis for the construction of more complex queries. Our API contains 11 operations, which can be classified according to the layer of $AttK^2$ -tree that they imply.

4.3.1 Operations over the Schema

Some of the basic operations in $AttK^2$ -tree work with the types (or labels) of the graph.

- **Retrieval of labels.** The operation $Get\{Node|Edge\}Types$ returns the different labels of the nodes (or the edges) of the graph. According to the internal representation of $AttK^2$ -tree, it is trivially implemented by recovering all entries of the Nodes Schema (or the Edges Schema). In the graph of the example, $GetNodeTypes$ returns the labels *Paper* and *Researcher*. On the other hand, $GetEdgeTypes$ returns the labels *Author*, *Reviewer*, *PhDDirector*, *Colleague*.
- **Filter by type.** $Scan\{Node|Edge\}(type)$ recovers the nodes or the edges of a given type. Taking into account that the identifiers were allocated according to the type of the elements, this operation becomes quite straightforward.

For instance, the operation *ScanNodes("Researchers")* is implemented by performing a binary search over the labels in the Nodes Schema, showed in the Figure 4.4. When the entry 2 is retrieved, the upper limit of the range of identifiers with label *Researcher* is obtained, with value 5. The lower limit of the range is retrieved from the previous entry (that is, the first entry) with value 2. Therefore, *Researcher* nodes range from 3...5.

- **Find by element identifier.** *Get{Node|Edge}Type(id)* gets the type corresponding with an identifier. *GetNodeType* starts by performing a binary search over the upper-limits of the Node Schema, until the correct range is found. Then the label can be returned. For instance, *GetNodeType(4)* starts from a binary search over the nodes schema, until the lowest upper-limit is found (in this case is the second entry with value 5) and the highest lower limit (the first entry with value 2). Consequently, the node 4 has type *Researcher*. The behaviour of *GetEdgeType(id)* is totally symmetric to *GetNodeType(id)* and it is implemented exactly in the same way.

4.3.2 Operations over the Data

Next two operations involve the Data subsystem. They work over the attribute values of the nodes and edges of the graph.

- **Attribute retrieval.** *Get{Node|Edge}Attribute(id,att)* is the basic operation that obtains the value that a node (or edge) with identifier *id* takes for the attribute with label *att*. The operation starts by obtaining the type of the given node, which is solved with the operation *GetNodeType(id)*. Then, the list of valid attributes of the node is checked looking for the label *att*. If the label *att* is not included in the list of valid attributes for that type, then the attribute is undefined for that node and no result is returned. For instance, *GetNodeAttribute(3,"Title")* in the example searches the list of attributes of the type *Researcher*, which are *Name*, *University* and *Position*, so *Title* is not a valid attribute and no result is returned. Otherwise, the attribute is checked. If it is a sparse attribute, the procedure is quite simple: the list of plain values is checked at the position $id - limit + 1$, where *limit* is the lowest identifier of the type *GetNodeType(id)*. The value of this cell is returned. For instance, *GetNodeAttribute(3,Name)* will return the value of the position 2 in the list of values for *Name*, that is, *J. Boy*. For dense attributes, a range operation has to be performed in the K^2 -tree. The range includes only one row (corresponding to *id*) and the columns representing the dense attribute that is being checked. For instance, for the operation *GetEdgeAttribute(Expertise,6)* the row 6 between the columns 1...3 is checked. A *one* appears in the second

column, so the second position in the *att* list, *Medium*, is finally returned as a value.

- **Filter by attribute value.** $Select\{Nodes|Edges\}(type,att,val)$ returns all nodes (or edges) belonging to *type* which takes the value *val* for the attribute *att*. It is a classical filtering by property and type. In the graph of the example, queries like *researchers from Coruña*, *papers with the topic Graph Compression or PhD Students* are examples of this select operation. The operation starts by recovering the entry corresponding to the specified *type* in the same way $Scan(type)$ does. It obtains the lower and upper limits for the identifiers of that type, which will be necessary later for this query (*inf,sup*). Then, the attribute *att* is searched in the attribute list of that entry. If it is a dense attribute, the value is searched in the list of labels of that attribute in order to compute the limits of the needed range search over the K^2 -tree. For instance, when $SelectNodes(Researcher,Position,Chair)$ is queried, a range query is performed between the rows 4...5 (since these are the lower and upper limits of the Researcher type) and the column 1 (corresponding to *Chair*). The rows taking a *one* in this range will be returned as a result (in this case, nodes 3 and 4). Since attributes are located in the K^2 -tree ordered by value, in addition to the equality, other patterns of comparison could be implemented efficiently. For sparse queries, the operation is similar (binary search over all the labels of this attribute list). When the valid values are reached, their positions determine the node identifiers which have to be returned.

4.3.3 Operations over the Relationships

Last kind of queries involves conditions over the relationships of the graph. Two basic queries can be the basis of the exploration of the relationships in the graph:

- **Find neighbors by node type.** $Neighbors(type,id)$ returns all nodes of the specified type which are neighbors of the node with the identifier *id*. The operation starts by retrieving the range of valid identifiers (according to the given type (*low,upper*)). Then the multi-edge K^2 -tree is explored in the row *id* and between the columns (*low,upper*). Consider the query $Neighbors(Researcher,4)$, asking for the neighbors of the node 4 (*the researcher J. Boy*) which have *Researcher* type. First of all, the limits of Researcher are computed (3,5). Therefore, a range query between the row 4 and the columns (3,5) is performed. A multi-edge in the cell (4,5) is recovered (containing edges e_4 and e_5) so the node 5 (the researcher S. Gomez) is the result of that query.
- **Find neighbors by edge type.** $Related(type,id)$ it returns all nodes related to the identifier *id* connected to them through an edge with the given *type*.

In this operation, the filtering is in the edge identifier, which is recovered after performing the query over the K^2 -tree. So, this filtering has to be processed after finishing the query. The query is executed as follows. First of all, the valid range of identifiers of the given edge type is computed. Therefore, the full row *id* is queried in the Multi-edge K^2 -tree. After that, all the results are processed sequentially, removing from the result the columns that do not contain any edge in the range of edge identifiers for the given edge type. The result will be the identifiers of the remaining columns. For instance, the query *Related(Author, 3)* starts by computing the valid identifiers for *Author*, which are 1...3. Then, the row 3 is queried in the multi-edge K^2 -tree, obtaining two cells with results: (3, 1) and (3, 2). The edge identifier of the cell (3, 1) (e_1) is included in the range of valid identifiers, so n_1 is a result of that query (the paper *Compressing graphs*). However, the edge identifier in the cell (3, 2) is not valid (e_6 has type *Reviewer*) so the node n_2 is not returned as a result.

The set of operations we implement in *AttK²-tree* aims to provide a basic but efficient querying to the attributed graphs. More complex queries can be implemented on the top of this basic operations as intersections, unions or chains of them. For instance, the query *Papers reviewed by P. Garcia and written by S.Gomez* could be implemented as an intersection of three different operations: *Scan(Researcher)*, *Related(SelectNodes(Reviewer, Researcher, Name, P. Garcia))* and *Related(SelectNodes(Author, Researcher, Name, S. Gomez))*. Experimental evaluation in Section 4.4 gives some experimental results of the spatial requirements and the temporal efficiency obtained with *AttK²-tree*. Furthermore, as a proof of concept, it is evaluated against other proposals in the State of the Art.

4.4 Experimental evaluation

In this section we analyze the spatial and temporal performance of our structure, which was designed to support basic operations over an attributed graph in a very compact way. We compare our structure against DEX and Neo4j, two of the most relevant graph databases in the State of the Art. However, it is important to note that the results are provided in order to prove that we propose a compact structure with some basic search capabilities which is competitive in terms of space and time, but we are not proposing an alternative to DEX and Neo4j, since the purposes of our structure are different. We designed a compact attributed graph representation with some queryable capabilities and we implemented some basic operations, but our structure is not a full graph database. *AttK²-tree* does not support the algorithms and operations characteristic in those kind of engines. Furthermore, it is a static structure designed to work in main memory. So this comparison has to be understood just as a proof of concept of the structure we proposed.

4.4.1 Experimental Framework

We run experiments on an Intel(R) Core™i5-3470 CPU @ 3.20GHz (4 cores), 8GB DDR3@1333MHz, running Ubuntu 14.04.

4.4.1.1 Tools

We compare our approach $AttK^2$ -tree with two of the most relevant graph databases engines in the State of the Art: Neo4J and Dex.

- $AttK^2$ -tree: is implemented in C, compiled it with the gcc compiler version 4.8.2.
- Neo4j [NEO14]: Neo4J is the commercial graph database described in Section 4.1.2. In order to execute the same operations implemented over $AttK^2$ -tree, the queries are implemented in the Cypher language. Those Cypher queries are called from a program implemented in Java, which uses the Neo4J Java driver.
- Dex [MBÁLMM⁺12]: Dex is a very compact graph database that we described in Section 4.1.1. We implemented the operations that $AttK^2$ -tree supports through a Java program which invokes the corresponding native functions of the Dex library.

4.4.1.2 Queries

We measure the performance of our structure through the execution of 10 different kinds of queries, whose implementation in our structure was described in Section 4.3. They include operations over the relations and the attributes of the graphs.

We design a synthetic query set of 500 queries per each kind of operation:

- **Query set 1:** *GetNodeTypes* recovers the labels of the different node types of the graph.
- **Query set 2:** *GetEdgeTypes* recovers the labels of the different edge types.
- **Query set 3:** *GetNodeType* obtains the type of a given node.
- **Query set 4:** *GetEdgeType* obtains the type of a given edge.
- **Query set 5:** *GetNodeAttribute* obtains the value that a given node takes in a specific attribute.

- **Query set 6:** *GetEdgeAttribute* obtains the value that a given edge takes in a specific attribute.
- **Query set 7:** *SelectNode* obtains the set of nodes that takes a given value for an attribute.
- **Query set 8:** *SelectEdge* obtains the set of edges that takes a given value for an attribute.
- **Query set 9:** *Neighbors* returns the nodes of a given type related to a node.
- **Query set 10:** *Related* returns the nodes related to a given type through a specific edge type.

Note that previously described operations *ScanNode* and *ScanEdge* are not included in this evaluation as they are relevant only for our proposed structure.

These query sets are analyzed in three categories: the operations over the schema (queries 1 to 4), which are the most simple queries. The times obtained for each alternative in this operations give a briefly idea of the minimal time of communication with the database. Queries from 5 to 8 represents operations over the data (properties and types) of nodes and edges. Finally, the query sets 9 and 10 establish conditions over the relationships in the graph.

4.4.1.3 Datasets

Movielens 100K The first use case we analyze is a dataset extracted from a movie recommendation website, Movielens ⁵, which contains ratings on movies from different users of the web, including statistical information of the users and tags of the movies. We use a subset of 100,000 ratings for 1,700 movies from 1,000 users [Gro14].

Figure 4.6 shows the attributed graph representing the small movielens dataset, that we will represent using DEX, Neo4j and our own structure. The graph model for this dataset has three kinds of entities: users, movies and genres. Movies and users contains attributes presenting different value distributions. Regarding to the representation in our structure, we use a K^2 -tree to represent the dense attributes *Age*, *Gender* or *Occupation*, while the remaining sparse attributes are directly stored through a indexed-plain list.

⁵<http://movielens.umn.edu/>

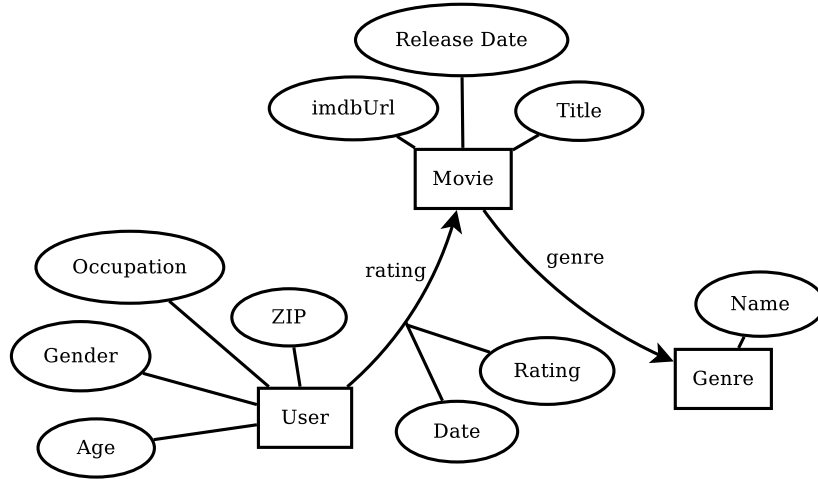


Figure 4.6: Attributed graph representing Movielens 100K dataset

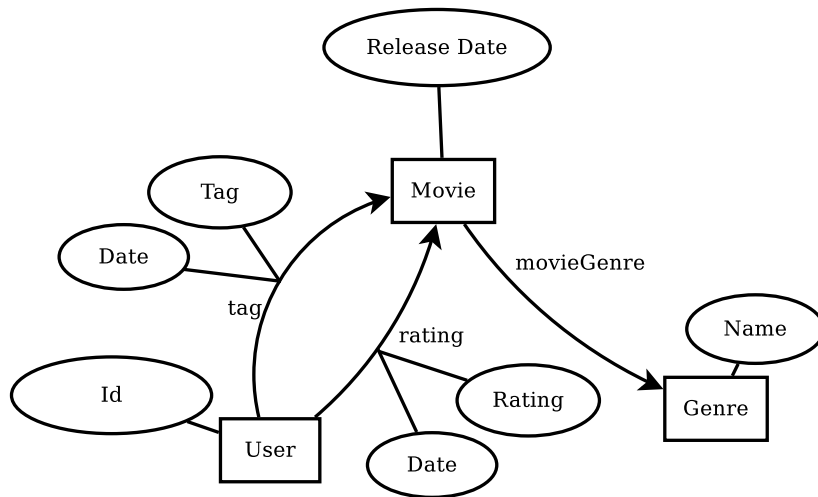


Figure 4.7: Attributed graph representing Movielens 10M dataset

Dataset	Spatial results (MB)		
	Att K^2 -tree	Neo4j	Dex
MovieLens 100k	6.05	27.00	17.50
MovieLens 10m	230.34	3280.41	1247.81

Table 4.1: Spatial results obtained for MovieLens dataset

MovieLens 10M We analyze the results of a different dataset which also represent movie recommendations from MovieLens ⁶. The model, which is shown in Figure 4.7, contains a more reduced set of properties. However, the number of entities it includes is larger than the previous use case. This dataset contains 10,000,000 ratings for 10,681 movies from 71,567 users [Gro14], where more than 20 million of properties need to be stored.

Results

We first show the spatial cost to represent both datasets on the three different approaches. Table 4.1 shows the cost in Megabytes. Note that in the case of Att K^2 -tree, the cost is in the main memory, while in Dex and Neo4j the results are measured as their cost in secondary memory with the graph engine system offline. Since we present a compact structure with some basic navigation capabilities, our structure achieves better spatial results, but note that we propose a compact graph representation, with some basic navigation capabilities, in opposition to the full graph database engines we analyze in this comparison, Neo4j and DEX. On the other hand, DEX is a compact graph database engine based on the massive usage of bitmap structures, so it obtains better results than the Neo4j solution.

Figure 4.8 shows some temporal results of simple and fast operations over the schema of the graph. The most relevant operations ask for the type of a node or an edge (queries *GetNodeType* and *GetEdgeType*). They are very lightweight operations in our system, since each type has a range of identifiers, so given a identifier, we only need to search over the list of node or edge types, which usually contains very few elements. It is important to mention that in the case of Neo4j, as we described in Section 4.4.1.1, every query performed in our experimental evaluation involves the parsing of the query, the connection with the database and other operations (like the formatting of the results). These factors, many of which also exist in DEX, explain the difference with the results obtained for our structure, which does not have any abstraction layer. The figure also compare the results

⁶<http://movielens.umn.edu/>

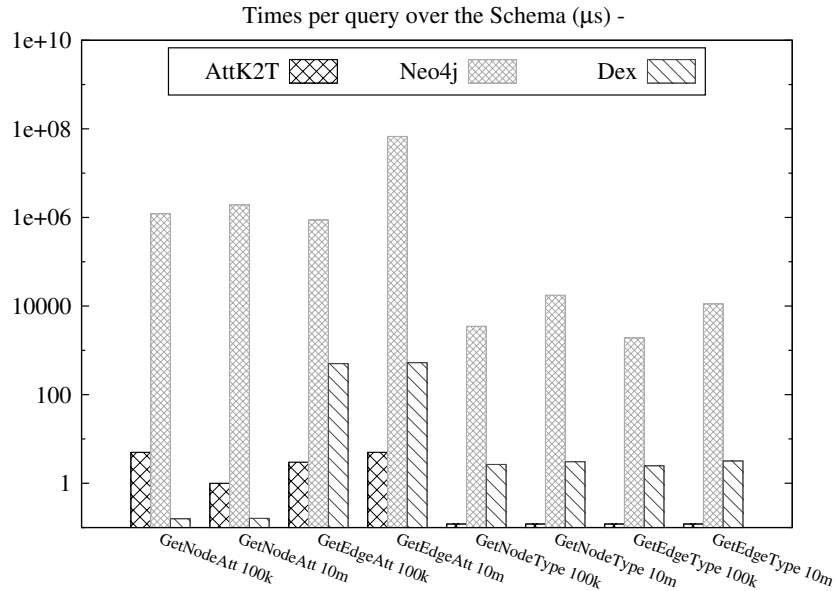


Figure 4.8: Temporal results obtained for operations over the schema in MovieLens dataset

obtained for both datasets in each system. We can observe that DEX and Att K^2 -tree obtain similar results for each operation independently on the size of the dataset. However, the temporal cost of Neo4j is increased when the number of nodes and edges grows.

Figure 4.9 shows the results obtained for the operations over the graph data. In Att K^2 -tree we can see how obtaining the property value for a given node is faster than obtaining the list of nodes which have a given value. In the case of DEX and Neo4, the select operation are also more costly than obtaining property values. The operation *GetNode* is faster in Att K^2 -tree. However, the *SelectNode* is executed faster in DEX graph engine for the MovieLens 100K dataset. Neo4J is slower in all the operations. In general, Att K^2 -tree shows a better scalability than Neo4j and DEX, in the sense of its increment of the temporal cost is smaller than the other two systems when the big dataset is analyzed.

Finally, Figure 4.10 shows operations over the relations over the nodes. The links in Att K^2 -tree are stored through a K^2 -tree (and additionally a structure of bitmaps), so these operations are very fast in our structure, since they do not involve operations over the attributes. The operation *Related* is slower than *Neighbors* in Att K^2 -tree because the later one implies an additional filtering of the final list of

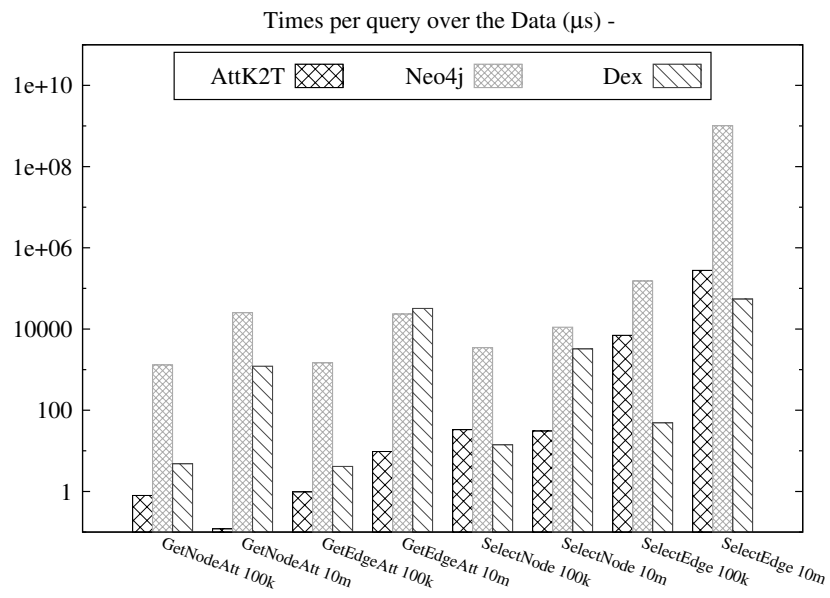


Figure 4.9: Temporal results obtained for operations over the data in Movielens dataset

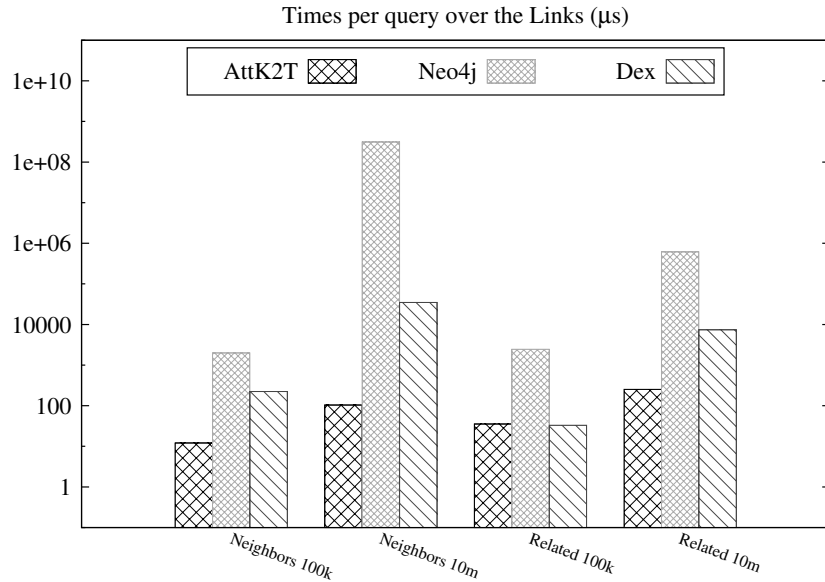


Figure 4.10: Temporal results obtained for operations over the links in MovieLens dataset

candidate edges, which is not necessary in the case of the *Neighbor* operation. DEX system improves the results obtained by AttK²-tree for the Related operation for the MovieLens 100K dataset. However, the temporal cost of DEX grows more than the temporal cost of AttK²-tree for the MovieLens 10M dataset regarding to the results of the small dataset. Again, Neo4J is slower in both operations, and it is more dependent on the size of the dataset.

4.4.2 Analysis

AttK²-tree is a very compressed representation of attributed graphs that supports efficient access to the properties and the relationships of the elements of the graph. The structure we propose is not a full graph database engine, we only support a set of basic queries. However, the spatial and temporal results obtained in the experimental evaluation shows that it is a very competitive approach to represent static graph data in a very compact way and to perform basic graph operations over the compressed structure.

4.5 Summary

This chapter presented a new compact representation of attributed graphs that support efficient access to the nodes, edges and their properties. It is designed as a static structure to work in main memory. We implemented an extension of the original K^2 -tree structure, named multi-edge K^2 -tree structure, to support multiple edges between the nodes of the graph. Regarding to the properties of the elements, in the case of dense attributes (presenting very few different values) were stored with a K^2 -tree. On the other hand, the values of the sparse attributes were stored as plain lists.

We implemented a basic set of operations to query the properties and the connections of the elements of the graph, which can be the basis of more complex operations.

We experimentally evaluated the spatial and temporal performance of our system *AttK²-tree* in three different datasets. We also store the same data in DEX and Neo4j in order to provide some spatial and temporal references of other systems. However, it is important to note that DEX and Neo4j are full attributed graph engines with many features and possible combinations, in addition to complete query APIs, so this comparison has to be understood only as a proof of concept of our structure.

Part II

Graph distribution

Chapter 5

State of the Art

Graphs are a natural and convenient way to represent many domains like Web graphs, Social networks or Geographic information. Nowadays, in the Big Data Era, huge volumes of data are generated every day. This information needs to be stored and processed efficiently in terms of space and time. In this context, efficient sequential algorithms may not be sufficient to meet large scale requirements where parallelism appears as a feasible approach to deal with these scalability and efficiency issues.

The parallel techniques were applied to process graphs in different domains. In this new research area, the challenge is to obtain a good graph partitioning scheme in terms of load balance. Finding the optimal partitioning scheme is an NP-hard problem [Cha98], but many heuristics have been proposed aiming to obtain a good partitioning. Some of these approaches are for general purpose problems, while other are designed for particular kind of graphs in specific domains.

This chapter is structured as follows. Section 5.1 defines the Graph Partitioning problem. Section 5.2 reviews the State of the Art in classical partitioning algorithms, describing the different families of algorithms and detailing some of the most relevant techniques to implement these approaches. Finally, Section 5.3 reviews partitioning techniques focused on distributing the cells of a matrix instead the nodes of a graph.

5.1 The Graph Partitioning problem

5.1.1 A brief definition

Consider a graph $G = (N, E)$, where $N = \{n_i | i = 1 \dots n\}$ and $E = \{e_{i,j} | n_i \rightarrow n_j\}$. The problem of P -partitioning consists in dividing the nodes in P sets, guaranteeing that each set contains the same number of nodes while the number of edges between the P sets is minimized [Els97]. The set of edges that connects nodes from different partitions is named *cut size*.

The problem of partitioning a graph can be generalized, reaching different levels of complexity depending on the kind of graph being partitioned. For instance, if we consider a weighted graph, the constraints for achieving a good node distribution change, because each partition would not aim to contain the same number of nodes but to balance the sum of their weights. In the same way, the edge weights are taken into account when the cut size is minimized.

5.1.2 Measuring the quality of the partition

The quality of the partitioning is theoretically given by the cut size that is being minimized, whose definition depends on the kind of the graph. In many cases, the equal-size partitions constraint is relaxed in order to reduce the cut size [Els97]. Therefore, the cut size gives us a measure of the quality of the graph partitioning. However, other features of the partition can be also relevant.

First of all, the partition affects the performance of the graph operations executed in parallel, so some partitioning algorithms could be more convenient for specific operations or specific graph topologies than others. In this way, some ad-hoc partitioning algorithms have been proposed to deal with a specific kind of graphs. In many cases, the more balanced is the work load of the processors, the more efficient the algorithms will be.

Balance of the size of the subgraphs is another measure of the quality of the partition. The size is usually given by the number of nodes that each partition includes (or the summarization of their weights in the case of weighted graphs).

In some kind of partitioning algorithms, the number of nodes are not a good measure of the size of each partition, since a node does not belong to only one partition. In the case of a graph represented as a connectivity matrix, the size of the partition can be determined by the number of cells or even the number of non-zero elements.

Obtaining a good graph partitioning is a costly operation, so complexity of the partitioning algorithm is also an important factor [FP10]. In many domains, an

online partition could be required, where partitioning is performed while the graph is processed. Streaming graph partitioning approaches [SK12] deal with these needs. Another new research line aims to parallelize [KK98b] the graph partitioning process. On the other hand, repartitioning graph algorithms have been also proposed to solve the partitioning of dynamic graphs. In the following we review the main techniques for graph partitioning that have been proposed in the literature.

5.2 Graph partitioning strategies

Graph partitioning algorithms have been classified in different ways [Els97, Fja98, Cha98] attending to the procedure that the algorithm follows and the kind of graph they divide.

The simplest problem that a partitioning algorithm can solve is the bisection of the graph, that divides the graph in two subsets. In order to solve the p -way partitioning, a bisection algorithm can be applied recursively, although proposals that directly divide the graph in p sets can obtain better cut sizes.

Next, we describe the most relevant algorithms in the State of the Art.

5.2.1 Algorithms based on geometric information

In some domains, nodes of the graph contain additional information about their location in a space (usually $2D$ or $3D$ space). Many times, the spatial proximity between two nodes is correlated with the shortest path between them. Geometric partitioning algorithms take advantage of this spatial information to compute the partition. They split the space in as many partitions as needed (P). In this way, nodes that are close in the space, which are expected to share many edges connecting them, will belong to the same partition.

5.2.1.1 Coordinate bisection

The recursive coordinate bisection is a geometric algorithm [BB87] that divides the space in P balanced rectangular regions by recursively cutting the regions with vertical and horizontal lines. It starts by dividing the space with a vertical line in such a way that the same number of nodes (or the same sum of weights for weighted graphs) are located on the left and right region that the line delimits. The algorithm continues recursively dividing, in each step, the biggest region in two subregions of the same size, alternating horizontal and vertical lines, until obtaining P regions. Figure 5.1 shows an example of 4-way partitioning using coordinate bisection. The

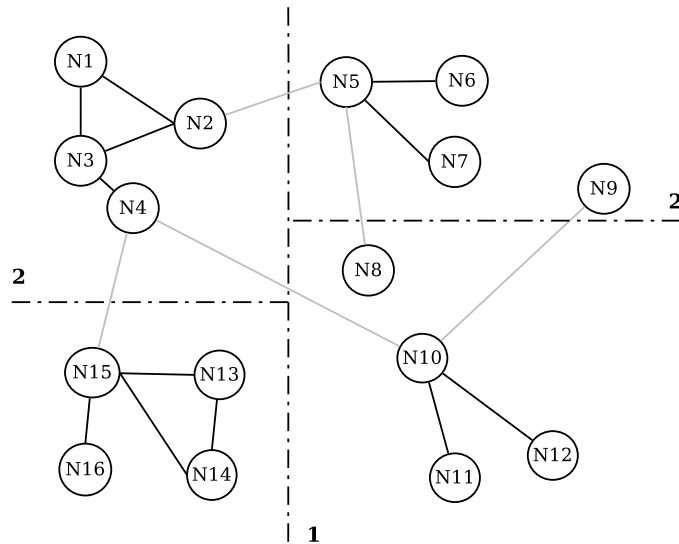


Figure 5.1: Recursive coordinate bisection in 4 regions

vertical line is computed in the first step. Each resulting region, is divided with an horizontal line in the next step.

The main advantage of this algorithm is its simplicity, both to compute the partition scheme and to store the final regions. Since they are rectangular blocks, only the coordinates of the square corners of each rectangle have to be stored to represent the final partitions.

5.2.1.2 Inertial bisection

The inertial bisection is an improvement of the coordinate bisection that does not restrict the division lines to horizontal and vertical lines. It divides the space in two regions through a hyperplane which is orthogonal to the axis of the minimal rotational inertia [Pot97]. This algorithm can be generalized to three dimensions and it can be applied recursively, in the same way that the coordinate bisection.

5.2.1.3 Geometric bisection

Coordinate and Inertial bisections only consider partitions of the space that are built using lines (or hyperplanes) of division. Geometric bisection proposed by Miller et

al. [MTTV98] works with circular and spherical separators, providing more flexible partitions.

The procedure starts by projecting the vertices of dimension R^d to the surface of the unit sphere in R^{d+1} by setting the $d + 1$ coordinate to 0 value. Next, the *centerpoint* of the projected points is computed. Note that a centerpoint in R^d means that every hyperplane that crosses them defines two subsets with a weight ratio less than $1 : d$.

Then, the points are moved in such a way that the centerpoint is in the coordinate origin. Finally, a great circle C is obtained from the intersection of one hyperplane with d dimensions that crosses the origin and the sphere in R^{d+1} . This circle is mapped to R^d by reverting the process, which will determine the partition.

This method usually obtains better partitions than the coordinate and the inertial bisection. Its main problem is the high cost to compute the centerpoints and to find the great circle. However, a simplified version improving the partitioning time was proposed, keeping a good cut size [GMT98].

5.2.2 Structural algorithms

In many contexts, the geometric information can be used to obtain good partitions of a graph. However, there are many other domains where nodes of the graph do not have an associated location. Next, we describe partitioning algorithms for that kind of graphs, based only on the information of the nodes and the connections between them. They are defined to work with a simple graph, but they can be generalized to weighted graphs (weighted nodes and edges).

5.2.2.1 Refinement algorithms

The family of the refinement algorithms starts from an existing partitioning, which will be progressively improved (by minimizing the cut size).

The Kernighan and Lin algorithm [KL70] was one of the first refinement algorithms that appeared in the State of the Art. Given an initial bisection, it iteratively exchanges a pair of nodes (one node of each partition) that minimizes the cut size.

Some previous definitions are needed to describe the KL algorithm. Suppose a given 2-way partition of a graph composed by two sets A and B . For a node $a \in A$, we define the reduction cost when it is moved to the set B as $D_a = E_a - I_a$, where $E_a = \sum_{x \in B} e_{ax}$ is the *external cost*, that is, the cost sum of the edges starting from a and pointing to nodes $b \in B$, and $I_a = \sum_{x \in A} e_{ax}$ is the internal cost, that

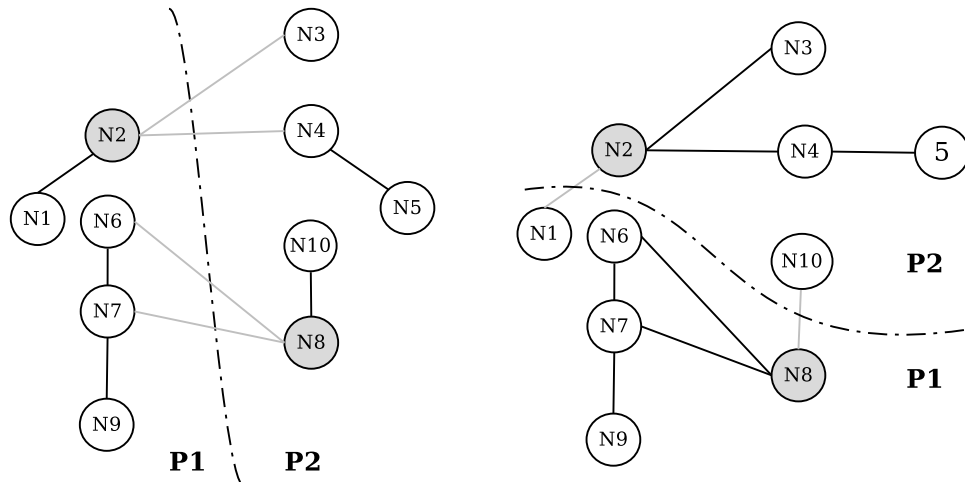


Figure 5.2: First iteration of the Kernighan and Lin Algorithm for an unweighted graph

is, the sum cost of the edges starting from a and pointing to other nodes $x \in A$. Therefore, for a pair of nodes $a \in A$ and $b \in B$, we can compute the reduction in cost of exchanging them like $D_a - D_b - 2e_{ab}$.

The algorithm starts with all the nodes set to an *unmarked* state. In each step, the pair of unmarked nodes ($a_i \in A, b_i \in B$) with a maximum *reduction cost* is exchanged, updating the remaining costs and marking a_i and b_i as already used. The algorithm stops when all nodes are marked or when there is no pair whose exchange reduces the cut size.

Figure 5.2 shows an example of iteration of this algorithm. The figure at the left shows the current state of the partitioning, where the cut size is 4 (composed by the light edges in the figure). Figure at the right shows the partitioning after a step of the algorithm, where the pair of nodes N_2 and N_8 were exchanged, producing a cut size with value 2.

Some variations of this method are focused on improving the execution time, like the implementation of Fiduccia and Mattheyses [FM82], that performs more efficiently the recalculation of the reduction costs when a pair of nodes is exchanged.

5.2.2.2 Greedy algorithms

Algorithms like Kernighan and Lin require to begin from an initial partition, which the algorithm will try to optimize. Greedy algorithms start from P empty partitions

and, through different heuristics, progressively choose nodes to be added to those partitions until all nodes of the graph have their corresponding partition assigned. When a node is assigned to its corresponding partition, this assignment is definitive (in contrast to refinement algorithms). Greedy algorithms usually start by assigning a node to each partition. Next nodes are usually chosen among the neighbors of the allocated nodes, by applying different heuristics [CJL94].

Some greedy partitioning algorithms that have been proposed are based on the Breadth-First Search algorithm. Consider a bisection (that is, a 2-way partitioning) computed by a greedy algorithm with Breadth-First Search. It would start by choosing a pair of nodes near to the maximum distance in the graph. Each partition set is initialized with one of those nodes. Then, among the nodes that are neighbors of the partition, the node less related to the unmarked nodes is added to the partition. This process is repeated until the partition contains half of the nodes. Then, the remaining nodes are assigned to the other partition [Els97].

Other greedy algorithms have been proposed, like the Farhat algorithm [FL93] or the Greedy Graph Growing algorithm of Karypis and Kumar [KK98a]. This last algorithm refines the Bread-First Search navigation by choosing, in each step, the neighbor that produces the biggest reduction of the *cut size*.

5.2.2.3 Spectral algorithms

Other approaches work over the global graph, attending to its general connectivity properties. They are opposed to greedy algorithms, that only work with local information (the neighbors of the current partitioning set) in each step. The Spectral Bisection [PSL90] is the most representative algorithm working with global heuristics.

Some previous concepts need to be defined before describing the Spectral Bisection. First of all, note that this algorithm works over a well-known mathematical representation of the graph, the *Laplacian Matrix*. Given an undirected graph $G = (N, E)$, its corresponding Laplacian Matrix $L(G)$ is a $|N| \times |N|$ matrix defined as follows:

$$L(G)[i][j] = \begin{cases} \text{deg}(n_i) & \text{if } i = j \\ -1 & \text{if } e_{i,j} \in E \\ 0 & \text{otherwise} \end{cases}$$

Note that $\text{deg}(n_i)$ is the number of connections (or edges) starting from n_i in the graph. Spectral bisection is based on the well-known mathematical eigenvector of a matrix. The eigenvector of the Laplacian Matrix $L(G)$ is any non-zero vector v that

accomplishes $Lv = \lambda v$, where λ is the corresponding eigenvalue for the eigenvector v [NDC11, New13].

The properties of the Laplacian eigenvalues were widely studied [MA91]. Consider the eigenvalues of the Laplacian Matrix in ascending order $\lambda_1, \lambda_2, \dots, \lambda_n$. Then, if and only if G is connected, $\lambda_1 = 0$ and $\lambda_2 \geq 0$. λ_2 presents some interesting properties that were studied by Fielder [Fie75]. The eigenvector that corresponds with λ_2 is commonly named *Fielder vector* (FV) and each element i of this eigenvector is a good indicator of the distance of n_i to other nodes in G . Therefore, given two nodes n_i and n_j , $FV[i] - FV[j]$ is inversely proportional to the distance between those nodes in the graph.

Taking into account the properties that the second eigenvector of the Laplacian Matrix presents, the Spectral Bisection is formulated as follows. Let m be the median value of the Fielder vector. This value determines the two sets in which the graph is partitioned: $P1 = \{n_i \in N | FV[i] < m\}$ and $P2 = \{n_i \in N | FV[i] > m\}$. The set of nodes $\{n_i \in N | FV[i] = m\}$ is located in the smallest partition in order to contribute to the balance.

The algorithm can be easily generalized to work with weighted graphs. It can also be applied recursively in order to obtain a P -partitioning where $P > 2$. The main challenge in order to implement this algorithm is the computation of the Fielder vector. Many algorithms in the State of the Art solve efficiently this computation. For instance, Lanczos Algorithm [Lan50] can be used to compute this second eigenvector.

5.2.2.4 Multilevel algorithms

Many of the previously described algorithms can result inefficient when the size of the graph is large. Besides, the quality of the partitions generated by those algorithms can also decrease when the number of nodes grows. Multilevel algorithms were designed to deal with scalability issues. They iteratively simplify the original graph, until obtaining a smaller graph that maintains the structural properties of the original graph. This summarization process is called the *coarsening* phase. At this point, the graph is small enough to apply a standard bisection algorithm (for instance, the Kernighan and Lin algorithm could be applied). Finally, the partitioning computed over the simplified graph is iteratively propagated to more complex graphs, reversing the process of the coarsening phase. This process is called the *Uncoarsening* phase. This general strategy was successfully combined with different algorithms from the State of the Art.

The **Multilevel Recursive Spectral Bisection** is a multilevel algorithm. It uses the Recursive Spectral bisection (described in Section 5.2.2.3) to partition the

simplified graph. Next, we describe the coarsening and refinement criteria of this algorithm.

In each step of the coarsening phase, the maximal independent set of the current graph is computed. A set $N' \subset N$ is an independent set of G if and only if $\forall n_i \in N', n_j \in N' \Rightarrow e_{i,j} \notin E$. An independent set is maximal when any node $n_k \in N \setminus N'$ makes the set not independent if it is added to N' .

Therefore, in each step of the coarsening phase, given the graph $G' = (N', E')$ obtained from the previous step, a new graph $G'' = (N'', E'')$ is created. N'' is the maximal independent set of G' , and E'' is computed as follows:

- A set of domains D is initialized with the nodes of the maximal independent set. That is, $D = \{\{n_i\} | n_i \in N''\}$.
- All the edges of the original graph G' are checked. For each edge $e_{i,j} \in E'$, if n_i and n_j belong to different domains, an edge is created between them. Otherwise, if n_i belongs to a domain but n_j does not belong to any domain yet, n_j is added to the domain of n_i and no edge is created. If none of them belong to any domain, the edge will be marked to be re-checked later.

Figure 5.3 shows an example of a step in the coarsening phase. Original graph G' is shown on the left, where the maximal independent set $\{n_2, n_3, n_5, n_6\}$ is highlighted. The graph on the right represents a simplified graph resulting of a coarsening step (G''). Nodes of G'' are the maximal independent set of G' ($\{n_2, n_3, n_5, n_6\}$). The coarsening step creates 4 domains: one per node of that set. Then, the edges E' are checked to add the remaining nodes to the corresponding domain. For instance, $\{n_6, n_7, n_8\}$ composes a final domain of that step. An edge is created in G'' when two nodes of different domains are related in G' . In the example, an edge between $\{n_6, n_7, n_8\}$ and $\{n_5\}$ is created due to the existence of a connection between n_5 and n_8 in G' .

After the repetition of the coarsening phase a given number of times, the resulting simplified graph is ready to be partitioned through a Spectral Bisection. In this point, The Fielder Vector of the coarsened graph is computed with the Lanczos Algorithm. Then, the Fielder vector is iteratively interpolated following a reverse process over the graphs computed in the coarsening phase. Suppose the Fielder vector (FV'') of the graph G'' . Then, the corresponding Fielder vector FV' of the graph G' is computed as follows. For each node $n_i \in G'$ belonging to the maximal independent set (and, as a consequence, also appearing in G'' , as a representant of a domain) its value for FV' is the same as in FV'' . For the remaining nodes, its value is interpolated by averaging the values of such neighbors that belong to the maximal independent set. In each step, the estimated eigenvector computed with the Lanczos Algorithm is improved using the Rayleigh quotient iteration.

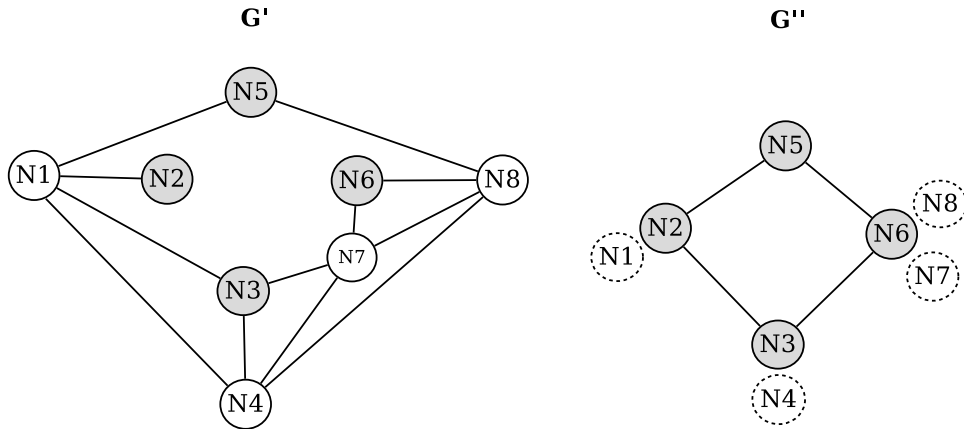


Figure 5.3: First coarsening (right) of a given graph (left)

5.3 Matrix partitioning strategies

Section 5.2 reviews the State of the Art in graph partitioning from a classical point of view. Different approaches of this problem were described in previous section. All of them have in common that they distribute the graph by allocating disjoint subsets of nodes to the different partitions. The quality of the partition in that kind of algorithms is usually measured by the cut size (that is, the weights of the edges between nodes in different partitions). Besides, if the total sum of the weights of the nodes is the same in all partitions, the partition is balanced. Note that those measures are based on the idea that the external edges (that is, edges between two different partitions) are a good measure of the communication needs between the processors in the distributed context where each partition is managed by a different processor.

However, when the problem is restricted to simple graphs (that is, unweighted nodes and edges) we could follow a different approach based on the adjacency matrix of the graphs. A simple graph can be represented by a binary adjacency matrix, where a *one* in a cell (i, j) means that an edge between the node i and the node j exists in the graph. Otherwise, the cell will have a value 0. According to that, a graph can be distributed by partitioning the adjacency matrix in such a way that the edges (instead the nodes) are allocated in the different processors.

With this new perspective, the problem of graph partitioning is reduced to the matrix partitioning, which has been widely studied in other areas where distributed environments are required. For example, areas such as matrix multiplication or raster processing developed matrix partitioning strategies to parallelize the processing

algorithms.

In matrix partitioning, the quality of the algorithms is not given by the cut size like in traditional approaches. They also aim to obtain a good balance of the distribution in terms of space, while the communication between the processors is minimized. However, that communication strongly depends on how the graph will be exploited.

As consequence, many measures for the partitioning quality can be defined, depending on the problem in which the partition is needed. In some cases, the amount of cells that each partition contains is a good metric of the balance of the distribution. In other words, the matrix representing the data could be distributed by allocating the same number of cells in each partition. For instance, in a multi-disk file system the goal is to distribute the data, allocating in each processor roughly the same number of data pages. However, in other cases, achieving a good balance is more related to the processing demands of the data. For instance, different approaches emerged from the matrix multiplication problem, that usually consider the number of non-zero elements, because they are the only cells that produce computation. In this context, the partitioning algorithm proposes to distribute the same number of non-zero elements as a measure of the balance. Given an adjacency matrix of a graph, balancing the non zero elements is equivalent to assigning the same number of edges to each processor.

To summarize, the communication between processors is strongly dependent on the requirements of the algorithm which will be executed over the matrix (and therefore the kind of data that the matrix represents). Many proposals emerged to solve the specific needs of different domains, aiming to find optimal partitioning matrices. Next, some of the most relevant matrix partitioning proposals are described. They try to obtain partitions with different characteristics, depending on the domain they were designed for.

5.3.1 1D partitioning

This family of partitioning algorithms divides the matrix in equally-sized sub-matrices using horizontal division lines. Suppose the matrix of dimensions $|M| \times |N|$ shown in the Figure 5.4 (top). **Uniform Block Distribution** [MS96] allocates $\lceil \frac{|M|}{|N|} \rceil$ rows for each processor in the way that each processor i contains a sub-matrix of size $\lceil \frac{|M|}{|N|} \rceil \times |N|$ including the full block from the row $(i - 1) * \lceil \frac{|M|}{|N|} \rceil + 1$ to the row $i * \lceil \frac{|M|}{|N|} \rceil$. Figure 5.4 (top) shows how the matrix is distributed among 4 processors following an Uniform Block Distribution.

Note that this strategy partitions the matrix independently of the data distribution, considering only the balance of the number of cells. For instance,

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	P1
3	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0		
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
5	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0		
6	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0		P2
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
9	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0		
10	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0		P3
11	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0		
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
14	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0		P4
15	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1		
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	P1
3	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0		
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
5	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0		
6	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0		P2
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
9	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0		
10	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0		P3
11	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0		
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
14	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0		P4
15	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1		
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		

Figure 5.4: Uniform Block Distribution (top) and row-wise block stripping (bottom)

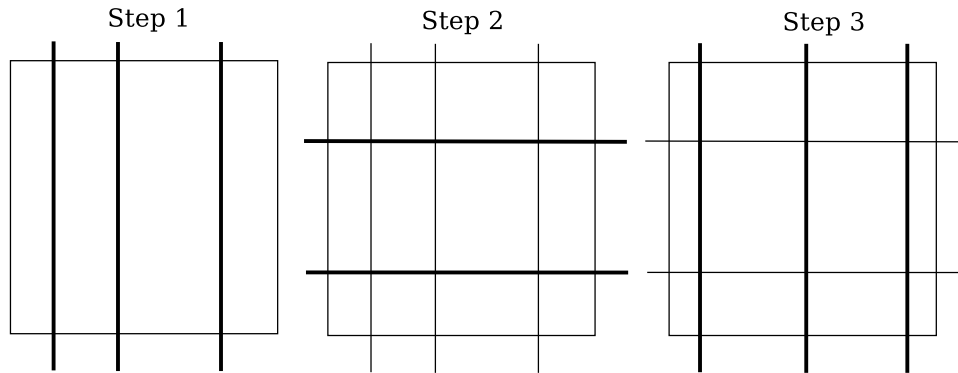


Figure 5.5: Three first steps of the iterative refinement process for a rectilinear partitioning

consider a matrix multiplication. In that operation, only the non-zero elements contribute to the processing times. Consequently, a more balanced 1D partitioning for matrix multiplication consists in dividing the matrix in $|P|$ blocks such as the number of non-zero elements (instead the number of cells) is the same in all partitions. This strategy is known as **Row-wise Block Stripping** and many implementations have been proposed [PA97]. An example is shown in the Figure 5.4 (bottom).

5.3.2 Rectilinear partitioning

The same philosophy followed in the 1D partitioning algorithms can be generalized to consider divisions along the two dimensions, using horizontal and vertical lines. For a mesh of $|P| \times |Q|$ processors we can define a rectilinear partitioning by dividing the matrix in $|P|$ horizontal blocks and $|Q|$ vertical blocks. In a similar way to the Row-wise Block Stripping, the horizontal and vertical lines can be located in order to balance the data according to the requirements of the application.

In order to obtain a well balanced rectilinear partitioning, an iterative refinement [Nic94,MS96] can be applied. It starts by obtaining the optimal row-wise partitioning (in the horizontal dimension) for $|P|$ blocks. Next, having fixed this row-wise partitioning, the optimal division in $|P| \times |Q|$ sub-matrices is calculated by moving the $|Q|$ vertical divisions. Then the vertical divisions are fixed and the optimal $|P|$ horizontal divisions are refined. The process continues iteratively by fixing one of the dimensions and finding the optimal partitioning modifying the other dimension, until no changes are obtained. Figure 5.5 shows an example of three iterations of a rectilinear partitioning for a mesh of $|P| \times |Q|$ processors. First step is shown in the left. Then, in the second step (middle of the figure), the horizontal lines are

allocated. Step 3 refines the vertical division from step 1 having fixed the horizontal lines.

5.3.3 Jagged partitioning

The previous strategy considers only partitions that result from full horizontal and vertical divisions. Jagged partitioning relaxes this constraint in order to obtain more fine grained distributions.

The binary recursive decomposition proposed as a geometrical graph partitioning [BB87] (described in Section 5.2.1.1), which partitions recursively a region of the space, can be also implemented to divide a matrix. In that way, in each step, each region is divided in two subregions. After n steps, 2^n subregions are obtained. Figure 5.6 (top) shows an example of this recursive decomposition. The main restriction of this partitioning is that it produces only partitions where the number of parts is a power of two.

However, this binary recursive decomposition can be generalized for a grid of $|P| \times |Q|$ processors. This method is called Multiple Recursive Distribution [RZ95]. First, we consider the prime factor decomposition of $|P| = p_1 * p_2 * \dots * p_n$ and $|Q| = q_1 * q_2 * \dots * q_m$. In a first step, the matrix is vertically partitioned recursively: first it is partitioned into p_1 blocks, then each vertical block is partitioned in p_2 partitions through independent vertical lines and so on. At the end, $|P|$ row-wise partitions are obtained. Note that each one of the resultant sub-matrices is partitioned first in q_1 row-wise partitions, then each one in q_2 partitions and so on. Figure 5.6 (center) shows an example for a mesh of 8×3 processors. First, we partition the matrix in 4 parts through the vertical lines marked with a 1 in the figure. After that, we divide each submatrix in two parts through another vertical lines (marked with a 2). Finally, each vertical block is partitioned in three parts through the horizontal lines marked with a 3. The result are 24 partitions, each of one is assigned to a different processor.

In [PA97] and [MS96] algorithms that efficiently compute a Jagged partitioning for a grid of $|P| \times |Q|$ processors were presented. They start by finding an optimal P row-wise partitioning. Then, for each row, the Q column-wise partitioning is computed. This approach obtains $|P| \times |Q|$ partitions (as the rectilinear partitioning of Section 5.3.2 does). Figure 5.6 (bottom) shows an example for a grid of 4×3 processors.

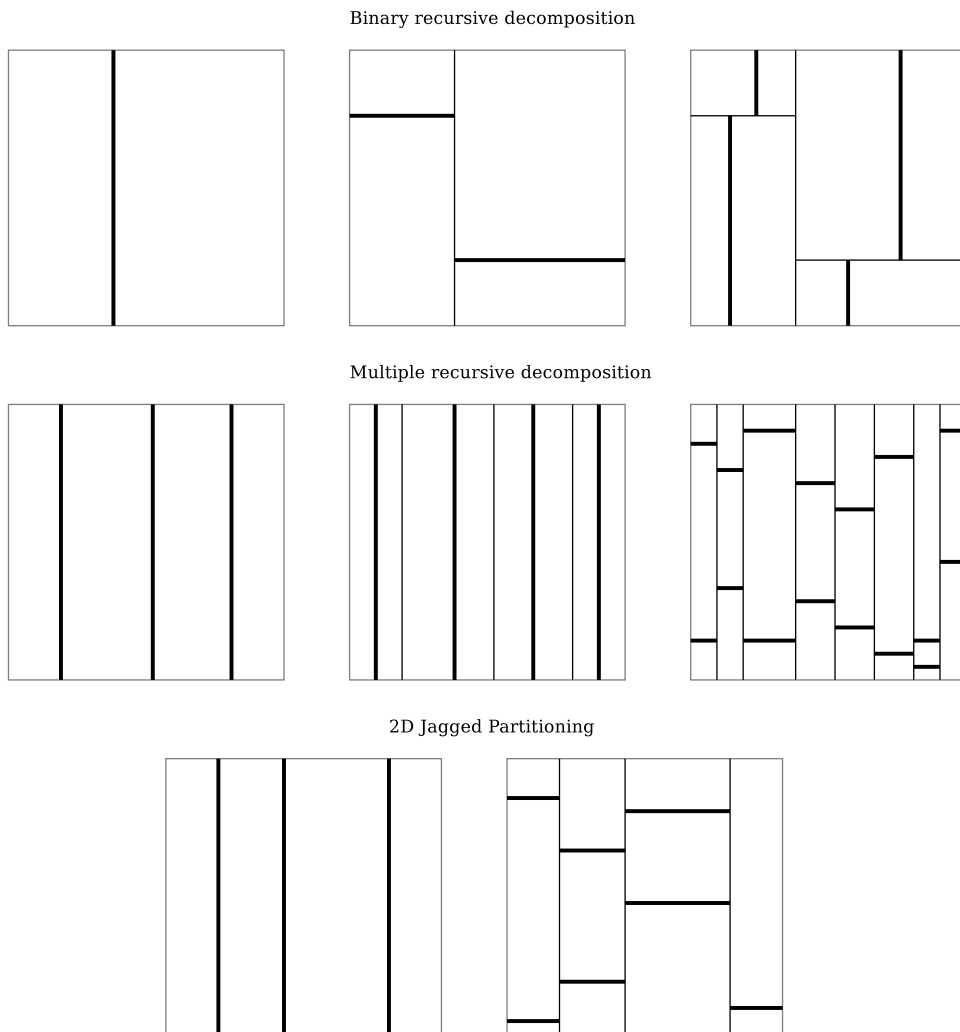


Figure 5.6: Three different jagged partitioning strategies

5.3.4 Disk allocation methods

Other alternatives emerged from the context in which grid data, like geographic databases, must be stored in a distributed disk. The purpose of this data partitioning is uniformly distributing the data pages among the different disk files while the number of disks needed to answer the queries (typically range queries) is minimized.

In [ZSC94] several approaches were proposed to solve that grid distribution problem. Considering a two dimensional data, we can represent as a matrix the $N \times N$ data pages storing the information. Then, the problem of distributing that data in P disks can be modelled as a matrix partitioning problem. In this context, the balance is measured in a slightly different way to the previously described distributions. When a parallel matrix multiplication problem is considered, the goal is to distribute the same number of non-zero elements to all processors. However, in disk allocation, the purpose is balancing the number of data pages (that is, elements of the matrix) while the rows and the columns of this two dimensional data are distributed along the different processors in order to balance the computation of a future row or column access to this data. Many distributions that accomplish this requirement have been proposed, and their behaviour were analysed depending on the type of the queries being performed. Four distributions proposed in this domain are:

Latin Square allocation method uses Latin Squares as a way of distributing the data. A *Latin Square* is an $n \times n$ matrix where each row and each column contains all the elements from $1 \dots n$. The matrix of data pages can be distributed by applying different Latin square matrices of size $|P| \times |P|$ across the two dimensions of the grid, where $|P|$ is the number of disks. Figure 5.7 (top-left) shows an example of this allocation method for six disks.

Linear allocation method proposes a round-robin allocation of the data pages for each row. Each row starts with a disk number which is defined as a shift of the first element in the previous row. Figure 5.7 (top-right) shows an example for six processors with a shift of 4.

Lattice allocation method can be viewed as a set of horizontal bricks, where each brick is homogeneously divided between all the processors. The bricks in consecutive rows are moved through a defined shift. Figure 5.7 (bottom-left) shows an example for 6 processors.

Coordinate Modulo Distribution [LSR92] proposes a partitioning scheme that is generalized to any number of dimensions. Suppose a data grid in two dimensions and a grid of $|P|$ processors. Defining a level of granularity l , the grid data is divided in $l * |P| \times l * |P|$ regions of equal size. Each region (x, y) is allocated following the formula $(x + y) \bmod P$. With this simple method, the space allocated is the same for all processors, while rows and columns are equally distributed among

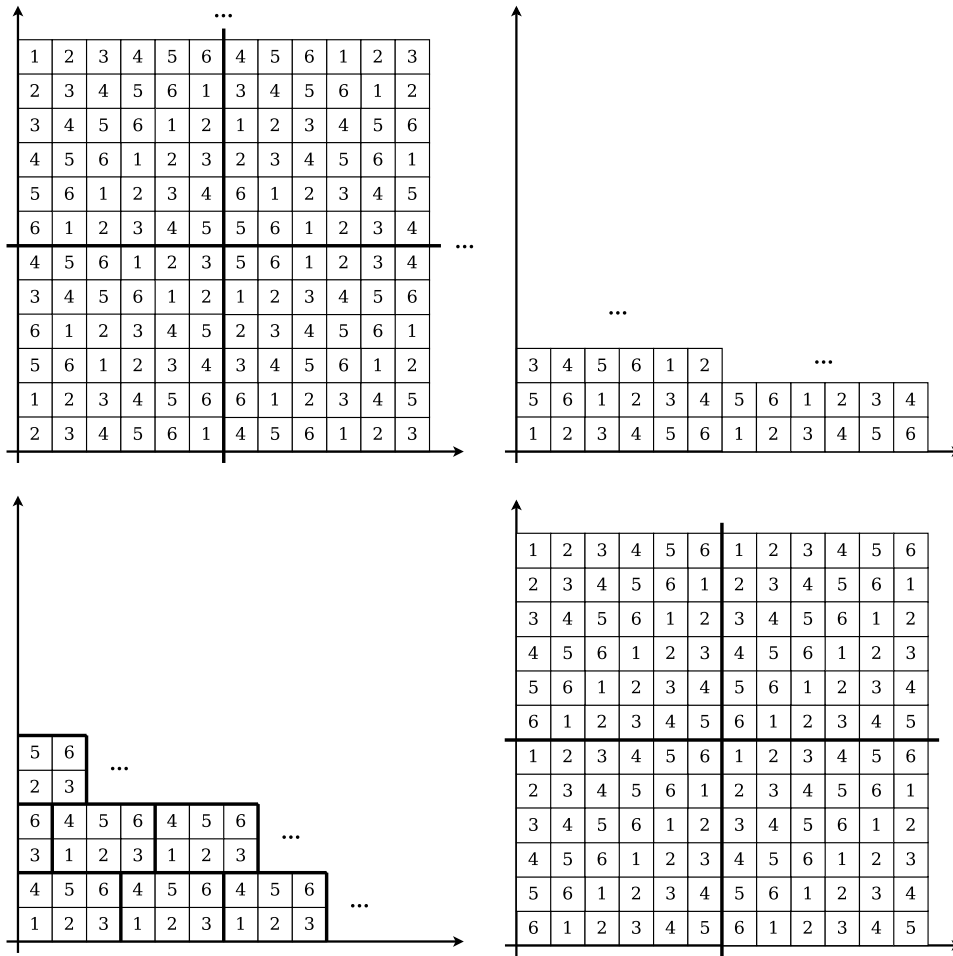


Figure 5.7: Latin Square (top left), linear (top right), lattice (bottom left) and Coordinate Modulo Distribution (bottom right) allocation

all the processors. It is easy to see how this method can be generalized in order to support the distribution of k -dimensional data. Figure 5.7 (bottom-right) shows an example of this allocation.

5.4 Summary

In this chapter we have reviewed some of the most important strategies in the State of the Art to partition a graph. First, we described the classical graph partitioning algorithms, which distribute the nodes of the graph into the different partitions trying to minimize the cut size (that is, the number of edges crossing different partitions). We provided some examples of partitioning strategies for graphs with spatial information. Then, we defined different kinds of structural algorithms, where it is noteworthy to mention the Kernighan and Lin Algorithm as it was one of the first algorithms proposed for this problem.

The second part of this revision focused on the matrix partitioning algorithms, which are specially interesting for the work we propose in the next chapter. Instead of distributing the nodes of the graph, these strategies distribute the cells of a matrix. Many of these strategies emerged from the parallel matrix multiplication problem, while other solved the problem of distributing pages in a distributed disk system. Although they emerged to solve different problems, all of them reduce the problem of the data distribution into a matrix partitioning. However, the measure of the quality of the partitioning is clearly dependent of the domain.

Based on these matrix partitioning strategies, we will propose different algorithms to distribute a graph by partitioning its adjacency matrix. In our case, the purpose is to achieve spatial and temporal balance to enable efficient performance so we will experimentally evaluate the behavior of the above strategies with respect to balance.

Chapter 6

Our proposal

6.1 Graph partitioning with the K^2 -tree structure

Nowadays, great amounts of information, which can be structured as graphs, are produced every day, and much valuable information can be inferred from those graphs by performing the convenient graph mining algorithms.

Consequently, efficient storing and mining of graphs have become a relevant issue. Many structures have been proposed not only for dealing with efficient query processing but also for trying to optimize its spatial cost, in order to handle huge graphs stored in main memory with one processor. However, even using very compact structures, a single processor could be insufficient when huge datasets have to be managed. Therefore, a new research line emerges to efficiently distribute graphs on a set of processors by using compact structures and processing queries in parallel.

K^2 -tree is a very compact data structure that can store simple graphs (that is, unweighted and directed graphs) or in general, any binary adjacency matrix. Algorithms to manage K^2 -trees efficiently implement basic graph operations like direct and reverse neighbors. As it was described in Section 2.3, K^2 -tree builds a binary tree from the representation of the graph through its adjacency matrix, subdividing recursively the matrix in smaller sub-matrices.

We propose several partitioning algorithms to distribute a simple graph in multiple processors, supporting direct and reverse neighbor operations. Each partition is stored using a K^2 -tree structure. In this way, we address the graph partitioning problem as an edge distribution, in contrast with the classic approaches based on distributing the nodes of the graph.

Therefore, we distribute the graph $G = (N, E)$ (where N is the set of nodes of the graph and E represents the edges connecting nodes) in a set of $P = \{P_i, i = 1, \dots, |P|\}$ independent processors, each one with its own local main memory. The resulting partitioned graph is represented as $G' = \{G_i, i = 1, \dots, |P|\}$, where each sub-graph $G_i = (N_i, E_i)$ is stored in the processor i . Taking into account that we propose edge distributions, we will have $\bigcup_{i=1}^{|P|} E_i = E$. Each $N_i \subset N$ is composed by the set of origin and target nodes of E_i . Since this is not a node distribution, it is not guaranteed that $\bigcap_{i=1}^{|P|} N_i = \emptyset$.

Several graph partitioning strategies are proposed in the following sections. All of them use K^2 -tree structures to represent the subgraphs, but they are used in different ways. Proposals described in Section 6.2 study the behaviour of classic partitioning of adjacency matrices, where each processor stores an adjacency sub-matrix using a K^2 -tree. They are based on the State of the Art strategies for parallel matrix multiplication. However, the spatial properties of the K^2 -tree are considered to design balanced partitioning algorithms based on those general strategies and new versions of these strategies adapted to our problem are proposed.

Next, in Section 6.3 more specific distributions designed to take advantage of the structural properties of the K^2 -tree are presented. They are focused on balancing the space consumed in the different processors, expecting that a good spatial balance can produce a good workload balance. Each processor builds a K^2 -tree representing its corresponding edges.

Finally, in Section 6.4, another kind of distributions following a different idea is proposed. They do not divide the graph in $|P|$ sub-graphs, one for each processor. Instead, m K^2 -trees ($m > |P|$) are built, reformulating the distribution problem as the distribution of these m K^2 -trees in the $|P|$ processors. This philosophy is K^2 -tree independent and it also could be used with different data structures.

We measure the quality of the different graph partitions using different metrics. First of all, since subgraphs are represented with a K^2 -tree in main memory, the spatial balance is an important factor. The bottle neck is given by the processor which has to manage the larger K^2 -tree. The more balanced the spatial cost is, larger datasets could be stored using a fixed number of processors. In addition to this, the workload balance is essential in order to optimize the overall querying time. Experimental evaluation in Chapter 7 shows the behaviour of the different distributions through those metrics.

6.2 Proposals based on the Adjacency Matrix Partitioning

In this section we propose several partitionings based on the division of the adjacency matrix of the graph into n sub-matrices, by using horizontal and vertical lines. Each submatrix is represented as a K^2 -tree in a different processor. They are inspired by the recursive coordinate bisection [BB87] described in Section 5.2.1.1 and by the rectilinear partitioning in Section 5.3.2.

All the proposals in this section define a function, $processorMap(i, j)$, which specifies in which processor $k \in 1 \dots |P|$ is stored the cell of the adjacency matrix (i, j) . Therefore, each processor k stores a K^2 -tree representing the set of cells $\{(i, j) \mid processorMap(i, j) = k\}$. Some of the proposed distributions store, by definition, the same number of cells in each processor. However, we also propose adaptive distributions that balance the number of edges allocated in each processor instead of the number of cells.

We also provide the different mapping functions designed to map a cell of the global adjacency matrix to the corresponding cell in a local adjacency matrix of one processor.

6.2.1 Block distribution

An adjacency matrix can be divided in $|P|$ equal-size regions by using $|P|$ parallel cut lines, following the 1D partitioning described in Section 5.3.1. Horizontal lines define a horizontal block distribution, while vertical lines would define a vertical block distribution. A horizontal block distribution will be considered in the rest of the section, taking into account that vertical distribution has a symmetrical behaviour. Each processor builds a K^2 -tree representing its corresponding adjacency matrix, that is, the horizontal block allocated in this processor.

Consider the global adjacency matrix GA representing the graph G , with dimensions $|N| \times |N|$. Given a horizontal block $|P|$ -partitioning, the size of the local adjacency matrix LM_i (which the processor i stores) has dimension $height \times |N|$, where $height = \lceil \frac{|N|}{|P|} \rceil$. That is, each processor (except the $|P|$ -th processor), contains $height$ full rows of the Global Adjacency Matrix. Figure 6.1 shows an example of an adjacency matrix representing a graph of 16 nodes connected by 32 edges, which is partitioned in 4 blocks with a block distribution strategy.

This partitioning can be defined through a map function that specifies, for each cell (i, j) , in which processor is stored. This map function is defined as $bProcessorMap(i, j) = \lfloor \frac{i}{height} \rfloor + 1$. That is, in a horizontal $|P|$ -partitioning, any

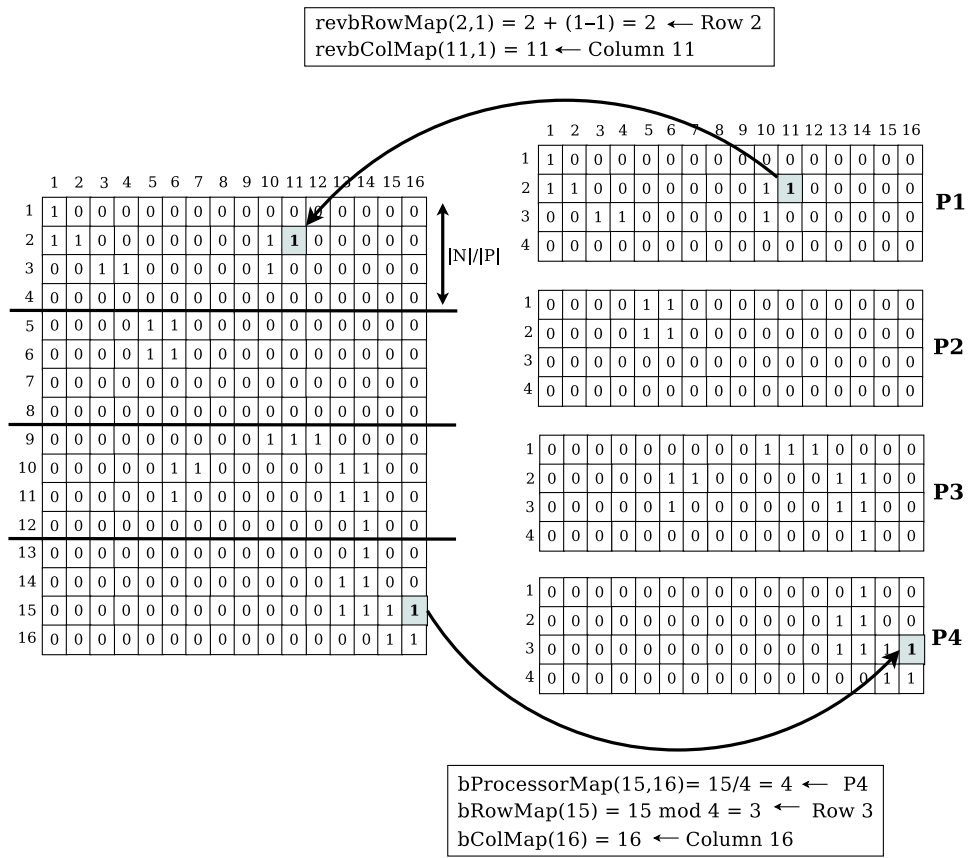


Figure 6.1: An example of horizontal block partitioning for 4 processors in a graph of 16 nodes, including the maps between the global and the local adjacency matrices

cell of the row i is stored in the processor $\lfloor \frac{i}{height} \rfloor + 1$. Since each processor contains an individual submatrix composed by its corresponding cells, we need additional map functions to compute the positions of each cell (i, j) to the corresponding cell (i', j') in the local adjacency matrix of the processor $processorMap(i, j)$.

- $bRowMap(i) = (i - 1) \bmod height + 1$
- $bColMap(j) = j$

With these three map functions, the partition is completely defined. Therefore, the complete map of a given cell in the global adjacency matrix $GM(i, j)$ to the distributed graph is given by the following function: $bMap(i, j) = LM_{bProcessorMap(i, j)}(bRowMap(i), bColMap(j))$. In this way, the distribution of the cells in $|P|$ local adjacency matrices is fully defined.

In a block distribution, a direct neighbor query over the node n (consisting in checking the 1 values of a row) is executed over a single adjacency matrix, because block distribution allocates all the cells of the same row in one local adjacency matrix. For a node n , the row $bRowMap(n)$ of the processor $k = bProcessorMap(n, 1)$ is checked.

On the other hand, in order to retrieve the reverse neighbors of the node n , the column $bColMap(n) = n$ has to be checked in all processors. In order to solve these operation, each processor k individually checks the corresponding row or column in LM_k , obtaining a set of cells as a result: $R_k = \{(i, j)\}$. The coordinates of the results are relative to the local adjacency matrix. To obtain the final results, a reverse map function translates the local cell coordinates to the absolute cells in the global adjacency matrix. This mapping is defined by the following formulas:

- $revbRowMap(i, k) = i + (k - 1) * height$
- $revbColMap(j, k) = j$

Therefore, a cell $LM_k(i, j)$, that is, a cell (i, j) from the local adjacency matrix of the processor k , is mapped to the global adjacency matrix GM cell: $revbMap(i, j, k) = GM(revbRowMap(i, k), revbColMap(j, k))$.

Note that this distribution has not a symmetric behavior of the basic operations: direct neighbor operation is only computed in one processor, while reverse neighbor operation has to be executed in all processors, since all processors contain the same number of cells of each column. The opposite behaviour appears for the vertical block partitioning.

Considering that each adjacency matrix is finally represented in each processor using a K^2 -tree, a first analysis of the convenience of this partitioning in the proposed system can be done, which will be extended in the Experimental Evaluation in Chapter 7.

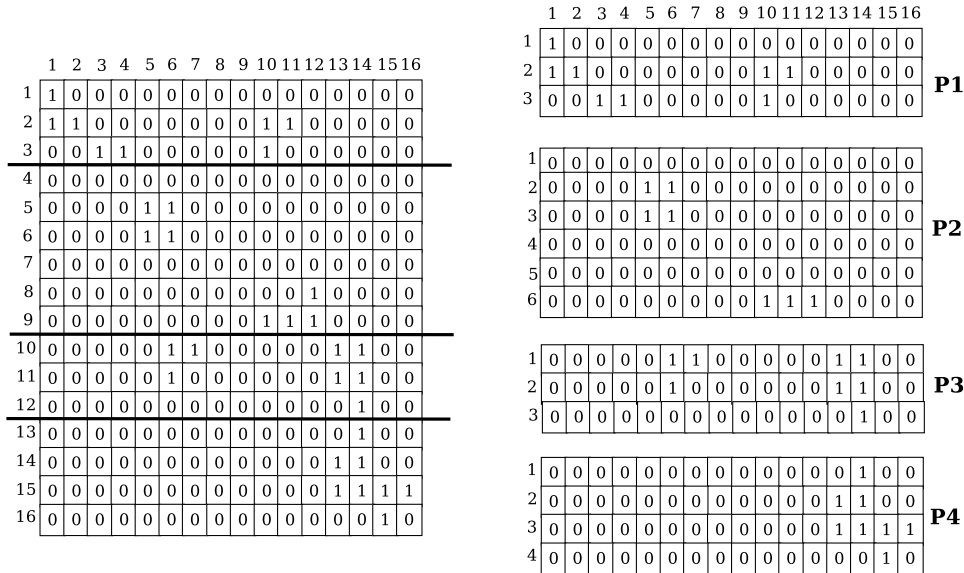


Figure 6.2: An example of adaptive block partitioning for 4 processors in a graph of 16 nodes

The main weakness of this method is its spatial balance, which strongly depends on the distribution of the edges among the adjacency matrix. For instance, an adjacency matrix with a higher density of *ones* in the first rows is expected to achieve a poor spatial balance, because the size of the K^2 -tree of the first processors will be greater than the others.

6.2.1.1 Adaptive block distribution

The previous section proposed a row-wise partitioning whose main problem is the possible spatial imbalance. The blocks in which the global adjacency matrix is divided allocate the same number of cells in each processor. However, the number of edges is a better estimator of the size than the number of cells in the final storage demanded by the K^2 -tree. An adaptive block distribution can be defined, trying to improve the spatial balance by balancing the number of edges of the final partitions.

The purpose of this distribution is partitioning the adjacency matrices in blocks with a similar number of edges. Therefore, we propose a more adaptive horizontal block distribution, which divides the adjacency matrix in $|P|$ matrices $LM_k(N_k, E_k)$, where $|E_k| = \frac{|E|}{|P|}$. Figure 6.2 shows the same adjacency matrix of the Figure 6.1 partitioned in 4 blocks by following an adaptive block distribution.

The map function described in Section 6.2.1 for the block partitioning is quite straightforward, since the limits of the blocks are directly computed. However, in the case of the maps for the adaptive block partitioning, some previous computing is necessary. Consider a list E' composed by the set of edges E ordered by the first coordinate (the row in the matrix), where $E'[i].x$ is the row and $E'[i].y$ is the column of the i -th cell. That is, $E'[i].x < E'[j].x \vee (E'[i].x = E'[j].x \wedge E'[i].y < E'[j].y) \Leftrightarrow i < j$. Algorithm 6.1 describes how lower and upper limits of each partition are computed by using the list E' . It receives the list of edges E' and the processors P as inputs. This algorithm outputs the boundaries (*lowLimit* and *uppLimit*) that define how the matrix is partitioned. *lowLimit*[i] is the first row of the global adjacency matrix that the processor i stores, while *uppLimit*[i] stores its corresponding last row. In adaptive block partitioning, all the cells of a row are stored in the same processor, just as in the original block partitioning.

In this way, the limits are adapted to the distribution of the global adjacency matrix, distributing a similar number of edges to the processors. These new limits change the definition of the function that maps a cell from the global adjacency matrix to the distributed system ¹.

- $bProcessorMap'(i, j) = k \mid lowLimit[k] \geq i \wedge uppLimit[k] \leq i$.
- $bRowMap'(i) = i - lowLimit[bProcMap'(i, 1)] + 1$
- $bColMap'(j) = j$

In a similar way, we can define the *reverse mapping* between a cell (i, j) of LM_k to a cell in GM :

- $revbRowMap'(i, k) = i + lowLimit[k] - 1$
- $revbColMap'(j, k) = j$

6.2.2 Cyclic distribution

One of the main disadvantages of the horizontal block partitioning is its dependency on the edges distribution over the global adjacency matrix. Adaptive block partitioning tries to balance the number of edges, but it unbalances the number of rows each processor contains. If the number of rows of one processor is high this

¹We use the notation $bProcessorMap'$ to specify the function *processorMap* in adaptive block distribution, where b means that is a block distribution and $'$ denotes that is the second strategy described following a block distribution. All the chapter uses a similar notation for the map functions.

processor would have to ask a high number of direct neighbor operations. This situation could produce the unbalancing of the work load.

An alternative partitioning which tries to balance the number of edges for each processor, but maintaining, at the same time, the same number of rows, is the cyclic distribution. It cyclically allocates the rows to the different processors in a round-robin fashion. In this way, the regions with a high density of edges will be distributed among all the processors, obtaining a good balance of the space.

As in the case of the block partitioning, we can define a vertical or horizontal cyclic distribution, but we consider the horizontal alternative for this description. Figure 6.3 shows an example of this partitioning. The functions that map a cell (i, j) of the matrix GM to the distributed system are defined as follows:

- $cProcessorMap(i, j) = ((i - 1) \bmod |P|) + 1$.
- $cRowMap(i, k) = \lfloor \frac{i}{|P|} \rfloor + 1$
- $cColMap(j, k) = j$

The *reverse map* from a cell (i, j) in LM_k to GM is given by the following definitions:

- $revcRowMap(i, k) = (i - 1) * |P| + k$
- $revcColMap(j, k) = j$

The main disadvantage of this distribution, considering that each adjacency matrix LM_k is stored with a K^2 -tree, is its low compressibility. This distribution breaks the natural clustering of ones that appears in many domains, and K^2 -tree compressibility strongly depends on this clustering to obtain good compression ratios. With a cyclic distribution, these clusters are distributed along the different processors, deteriorating the compression.

6.2.2.1 Adaptive cyclic distribution

Cyclic partitioning is proposed as an alternative to improve the spatial balance. However, the balance it guarantees is not perfect, since it allocates the rows of the matrix cyclically, but it does not take into account the number of the edges of each row. Therefore, in the same way that block distribution was designed to divide the matrix depending on the number of edges of the blocks, an adaptive cyclic partitioning is proposed.

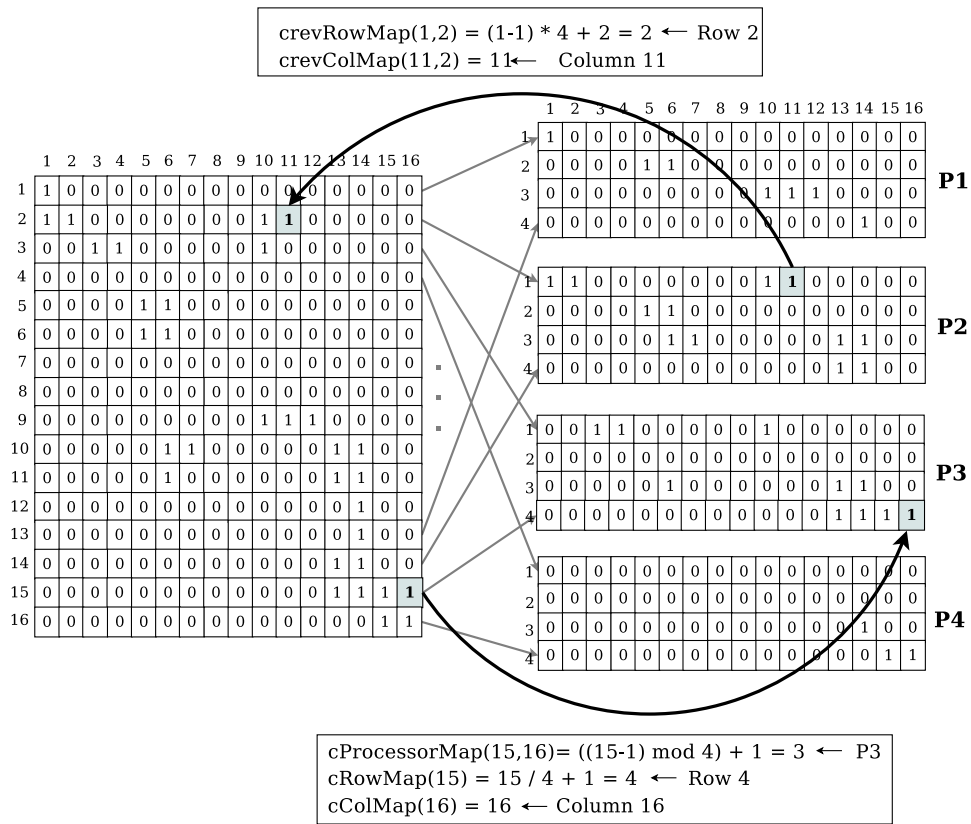


Figure 6.3: An example of horizontal cyclic partitioning for 4 processors in a graph of 16 nodes, including the maps between the global and the local adjacency matrices

Adaptive cyclic distribution still distributes rows cyclically into the processors. However, when a processor already includes a number of edges exceeding $\frac{|E|}{|P|}$, this processor is removed from the rotation, and no more rows are allocated to that processor.

The Algorithm 6.2 assigns each row to the corresponding processor. The inputs of the algorithm are the edges E , nodes N and the processors P . It returns as result the list of allocation $cProcessorMap'[i]$.

After the execution of this algorithm, the processor where the cell (i, j) is located is $cProcessorMap'(i, j) = cProcessorMap'[i]$. The column j of the cell (i, j) is also the column j in the local adjacency matrix ($cColMap'(j) = j$).

The map of the row r in GM to LM_p is computed with the function *ComputeRow* in Algorithm 6.3, receiving the number of rows each processor stores (computed with Algorithm 6.2) as parameter. The basic idea is that the process of mapping follows a round-robin strategy over the active processors, until one of the processors becomes inactive (when they exceed a given number of edges). Then, a new section of rows starts, which are cyclically distributed over a smaller number of active processors. *ComputeRow* computes the number of rows for each round-robin block located in P , using the auxiliary function *minimumIndex*, that returns the upper limit of the next block.

The map functions for the adaptive cyclic algorithm are defined as:

- $cProcessorMap'(i, j) = cProcessorMap'[i]$ (computed with Algorithm 6.2)
- $cRowMap'(i, k) = computeRow(countRows, P, i)$ (computed with Algorithm 6.3)
- $cColMap'(i, k) = j$

The reverse mapping (from a cell of LM_k to GM) is performed in a similar way, by using the vector *numberRows* in order to compute the different round-robin blocks.

6.2.3 Basic grid distribution

Previous algorithms, based on rows or columns allocations, have in common that direct and reverse neighbors are not executed symmetrically: in horizontal partitions direct neighbor queries are fully executed in a single processor, while reverse neighbor queries are homogeneously distributed among all the processors.

We propose an alternative approach that tries to achieve a symmetrical behaviour for direct and reverse queries. Each row and each column of the global adjacency

Algorithm 6.1 Adaptive Block Limits Algorithm

```

procedure COUNTBYROW( $N, E$ )
   $number \leftarrow 0$ 
  for  $i \leftarrow 1 \dots |N|$  do
     $edgesByRow[i] \leftarrow 0$ 
  end for
  for  $i \leftarrow 1 \dots |E|$  do
     $edgesByRow[E[i].x] ++$ 
  end for
  return  $number$ 
end procedure
procedure COMPUTEMAPPING( $N, E, P$ )
   $edgesByRow \leftarrow countbyRow(N, E)$ 
   $edgeBlock \leftarrow \lfloor \frac{|E|}{|P|} \rfloor$ 
  for  $i \leftarrow 1 \dots |P|$  do
     $numberRows[i] \leftarrow 0$ 
     $numberEdges[i] \leftarrow 0$ 
  end for
   $nextProcesor \leftarrow 0$ 
   $np \leftarrow 1$ 
  for  $i \leftarrow 1 \dots |N|$  do
     $numberRows[np] ++$ 
     $numberEdges[np] \leftarrow numberEdges[np] + edgesByRow[i]$ 
    if  $numberEdges[np] > edgeBlock$  then
       $np ++$ 
    end if
  end for
   $sumRows \leftarrow 0$ 
  for  $i \leftarrow 1 \dots |P|$  do
     $lowLimit[i] \leftarrow sumRows + 1$ 
     $uppLimit[i] \leftarrow sumRows + numberRows[i]$ 
     $sumRows \leftarrow sumRows + numberRows[i]$ 
  end for
end procedure

```

Algorithm 6.2 Adaptive Cyclic Limits Algorithm

```

procedure COMPUTE_MAPPING( $N, E, P$ )
   $edgesByRow \leftarrow countByRow(N, E)$ 
  for  $i \leftarrow 1 \dots |P|$  do
     $countRows[i] \leftarrow 0$ 
     $countEdges[i] \leftarrow 0$ 
     $active[i] \leftarrow true$ 
  end for
   $edgeBlock \leftarrow \lceil \frac{|E|}{|P|} \rceil$ 
   $np \leftarrow 1$ 
  for  $i \leftarrow 1 \dots N$  do
     $countEdges[np] \leftarrow countEdges[np] + edgesByRow[i]$ 
     $countRows[i] ++$ 
    if  $countEdges[np] > edgeBlock$  then
       $active[np] \leftarrow false$ 
    end if
     $cProcessorMap'[i] \leftarrow np$ 
    repeat
       $np \leftarrow np + 1$ 
      if  $np > |P|$  then
         $np \leftarrow 1$ 
      end if
    until  $active[np]$ 
  end for
  return  $cProcessorMap'$ 
end procedure

```

Algorithm 6.3 Adaptive Cyclic Mapping Algorithm

```

procedure MINIMUMINDEX(countRows, P)
  minimum  $\leftarrow \infty$ 
  for  $i = 1; i \leq |P|; i++$  do
    if countRows[ $i$ ] < minimum then
      minimum  $\leftarrow$  countRows[ $i$ ]
      pos  $\leftarrow i$ 
    end if
  end for
  return pos
end procedure

procedure COMPUTEROW(countRows, P, r)
  actP  $\leftarrow |P|$ 
  localRows  $\leftarrow 0$ 
  procRows  $\leftarrow 0$ 
  while procRows  $\leq r$  do
     $k \leftarrow$  minimumIndex(countRows, P)
    block  $\leftarrow$  (countRows[ $k$ ] - procRows) * actP
    if procRows + block  $\geq r$  then return  $\lfloor \frac{(r - \text{procRows})}{\text{actP}} \rfloor + \text{localRows} + 1$ 
    else
      localRows  $\leftarrow \frac{\text{block}}{\text{actP}} + \text{localRows}$ 
      procRows  $\leftarrow$  procRows + block
      actP  $\leftarrow$  actP - 1
    end if
    countRows[ $k$ ]  $\leftarrow \infty$ 
  end while
end procedure

```

matrix will be distributed in $\sqrt{|P|}$ processors. The global adjacency matrix is partitioned in $|P|$ squared sub-matrices by using $\sqrt{|P|} - 1$ equally-spaced horizontal lines and $\sqrt{|P|} - 1$ vertical lines.

Figure 6.4 (top) shows an example of basic grid distribution for 4 processors. Just as in the block and cyclic distributions, the mapping between the global adjacency matrix and the distributed system can be computed by simple formulas:

- $sizeBlock = \lceil \frac{|N|}{\sqrt{|P|}} \rceil$
- $gProcessorMap(i, j) = \lfloor \frac{i-1}{sizeBlock} \rfloor * \sqrt{|P|} + \lfloor \frac{j-1}{sizeBlock} \rfloor + 1$
- $gRowMap(i) = ((i - 1) \bmod sizeBlock) + 1$
- $gColMap(j) = ((j - 1) \bmod sizeBlock) + 1$

In a similar way, given a cell (i, j) in the local adjacency matrix of the processor k (LM_k), its corresponding cell in GM is computed as:

- $revgRowMap(i, k) = \lceil \frac{k}{\sqrt{|P|}} \rceil * sizeBlock + i$
- $revgColMap(j, k) = \lceil (k - 1) \bmod \sqrt{|P|} \rceil * sizeBlock + j$

With this distribution, each direct and reverse neighbor query is distributed among exactly \sqrt{P} processors. However, it presents an important restriction: it is designed for a grid where the number of processors is a power of two. Besides, depending on the data distribution, it can also result in unbalanced partitions in terms of space (just as the block partitioning), due to its division of the global adjacency matrix in big regions.

6.2.4 Multilevel grid distribution

Following the grid philosophy described in previous section, we try to avoid the strong dependency on the data distribution by dividing the global matrix in smaller blocks. The basic grid partitioning can be generalized by repeating the process of subdividing in $|P|$ squared regions recursively L times. That is, L is a parameter that is used to specify the number of times that the matrix is recursively subdivided. In this way, each direct and reverse query is also distributed among $\sqrt{|P|}$ processors, just as in the basic grid distribution, but the effect that the distribution of the global adjacency matrix has over the balance of the space is minimized when L grows. Figure 6.4 (bottom) shows an example of multilevel grid distribution for $L = 2$. The

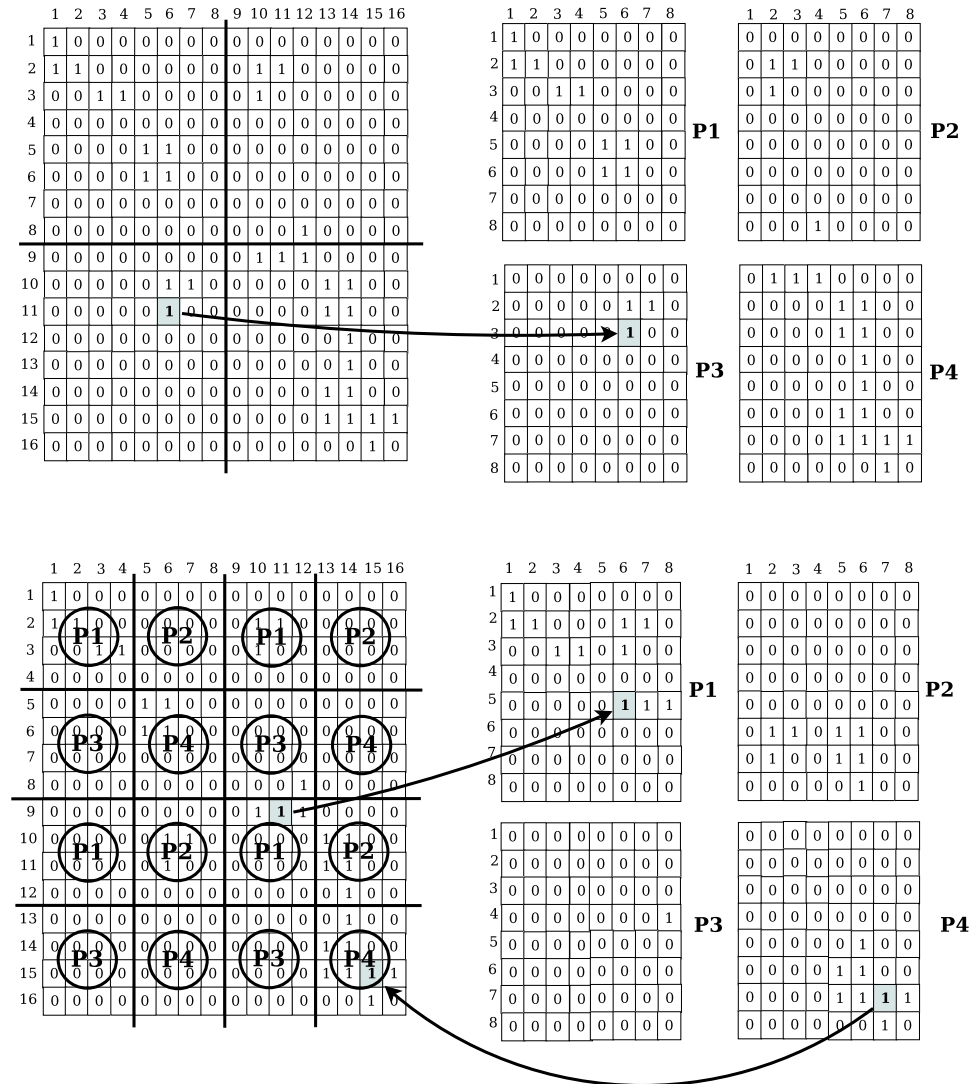


Figure 6.4: An example of basic grid partitioning (top) partitioning for 4 processors in a graph of 16 nodes and a multiple grid partitioning with L=2 (bottom)

matrix is firstly divided in 4 submatrices. Since the parameter L is set to 2, each one of these 4 matrices is divided in other 4 matrices.

The formulas to map the global adjacency matrix to each LM_k are a generalization of the proposals for the basic grid distribution:

- $sizeBlock' = \lceil \frac{|N|}{L * \sqrt{|P|}} \rceil$
- $gProcessorMap'(i, j) = (\lfloor \frac{i-1}{sizeBlock'} \rfloor) \bmod \sqrt{|P|} * \sqrt{|P|} + \lfloor \frac{j-1}{sizeBlock'} \rfloor \bmod \sqrt{|P|} + 1$
- $gRowMap'(i) = \lfloor \frac{i-1}{sizeBlock'|P|} \rfloor * sizeBlock' + (i - 1) \bmod sizeBlock' + 1$
- $gColMap'(j) = \lfloor \frac{j-1}{sizeBlock'|P|} \rfloor * sizeBlock' + (j - 1) \bmod sizeBlock' + 1$

In the same way, the reverse mapping is defined as follows:

- $revgRowMap'(i, k) = (\lfloor \frac{i-1}{sizeBlock'} \rfloor - 1) * sizeBlock' * \sqrt{|P|} + ((k - 1) \bmod \sqrt{|P|}) * sizeBlock' + (i - 1) \bmod sizeBlock' + 1$
- $revgColMap'(j, k) = (\lfloor \frac{j-1}{sizeBlock'} \rfloor - 1) * sizeBlock' * \sqrt{|P|} + (\frac{k-1}{\sqrt{|P|}}) * sizeBlock' + (j - 1) \bmod sizeBlock' + 1$

6.2.4.1 Adaptive multilevel grid distribution

Multilevel grid distribution in previous section tried to reduce the effects that a bad matrix distribution causes in the spatial balance. However, although the balance can be optimized, higher values of L can also produce a reduction of the compression capabilities of the K^2 -tree due to the breaking of its clustering properties. Furthermore, the matrix can present some dense areas but also some sparse areas, presenting a different optimal value for L .

We propose an alternative grid distribution that we called adaptive multilevel grid distribution, which recursively divides the matrix in regions, until obtaining a set of submatrices that does not exceed a given number of edges. Each final region will be represented with a K^2 -tree. An additional K^2 -tree will index these lists of K^2 -trees.

The data structures we need for this distribution are:

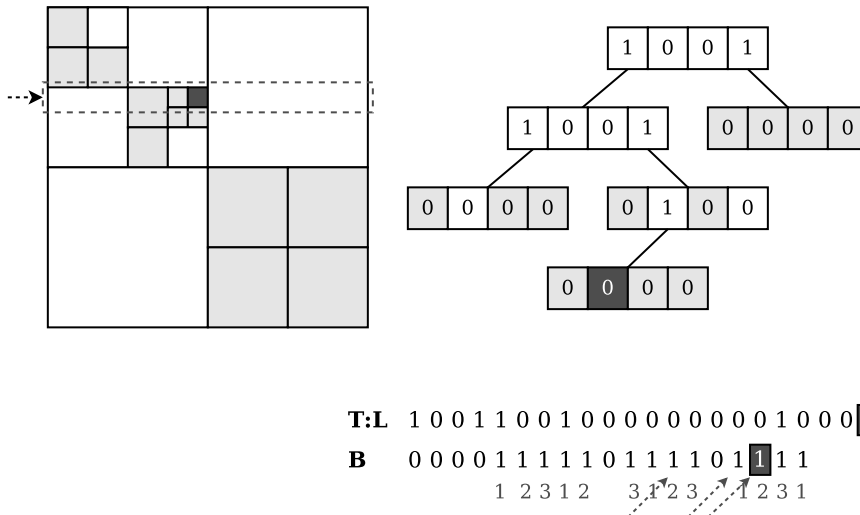


Figure 6.5: An example of adaptive grid partitioning for 4 processors in a graph of 16 nodes for a edge cut 6

K^2 -tree index A K^2 -tree is built representing the global adjacency matrix. The matrix is progressively divided in K^2 submatrices, where each submatrix is represented with a bit, just as the classical K^2 -tree. However, in this case, a *zero* in the tree can represent not only the empty matrix but also such matrices with a number of edges that does not exceed a given *limit*. Figure 6.5 shows an example of this K^2 -tree. Non-empty submatrices are highlighted in the global matrix. The *zeroes* of the tree that represent these non-empty submatrices are also highlighted. The subdivision continues until all the matrices have a number of edges less than the limit established.

Bitmap B An additional bitmap B stores the meaning of the bit with value *zero* of the original matrix. This bitmap will have as many bits as *0s* exist in the K^2 -tree index. The i -th bit in B will have value 1 if the i -th zero of $T : L$ represents a non-empty matrix. Otherwise, the bit is a zero. Figure 6.5 shows this bitmap. For instance, the 5-th position of the bitmap is a *one*, because the fifth zero of the tree (represented as a bitmap in $T : L$) corresponds with a non-empty matrix. However, the first bit of B is a one because the first zero of $T : L$ represents the top-right empty submatrix of the global adjacency matrix. We need *rank* access over the bitmap B , since the *ones* in these bitmaps will give the order of the submatrices represented with an individual K^2 -tree. This kind of representation, where each node of the K^2 -tree represents three different values, was originally proposed by G.

de Bernardo *et al.* [dB14,dBÁGB⁺13].

K^2 -trees Each non-empty submatrix represented as a *one* in the bitmap B , will be stored by a different K^2 -tree.

The K^2 -trees will be cyclically distributed among the different processors, according to their position in B . For instance, in Figure 6.5 the highlighted submatrix is stored in processor 2.

Each processor will store the K^2 -tree index and its corresponding K^2 -trees representing the non-empty submatrices. In this way, in order to answer a direct (or reverse) neighbor operation, the K^2 -tree index is queried (traversing the corresponding branches) just as in a traditional K^2 -tree. When a *zero* is reached (suppose it is the i -th zero of T:L), the i -th position in the bitmap B is checked. If it is a *one*, a $x = \text{rank}(B, i)$ operation is performed. That means that the x -th K^2 -tree has to be checked, which is the $(\lfloor \frac{x}{P} \rfloor + 1)$ K^2 -tree of the processor $(x - 1) \bmod P + 1$. The resulting K^2 -tree will be queried to finally answer the neighbour operation.

6.3 Proposals based on the K^2 -tree structure

In Section 6.2, some basic distributions are proposed: the partitioning of the adjacency matrix in blocks, cyclically by rows or columns and finally through a grid of sub-matrices. Those distributions were also adapted to balance the number of edges that each processor contains, expecting to improve the balance of space without deteriorating the compression of the matrix.

This section proposes a different strategy. We propose two new distributions, also focused on obtaining a good spatial balance. They are designed attending to the specific properties of the structure that finally stores the graph: that is, the K^2 -tree. Therefore, they try to balance the space according to the final space that each K^2 -tree needs. Consequently, they are distributions specifically designed for a K^2 -tree distributed system.

6.3.1 Edge-Balanced distribution

Consider the global K^2 -tree, representing the full adjacency matrix. Figure 6.6 shows the K^2 -tree (bottom) representing an adjacency matrix of 16 nodes (top). Each *one* at the bottom level of the tree represents an edge (a *one* in the adjacency matrix). It can be observed that the nearer two edges are in this bottom level, the more common ancestors they can share.

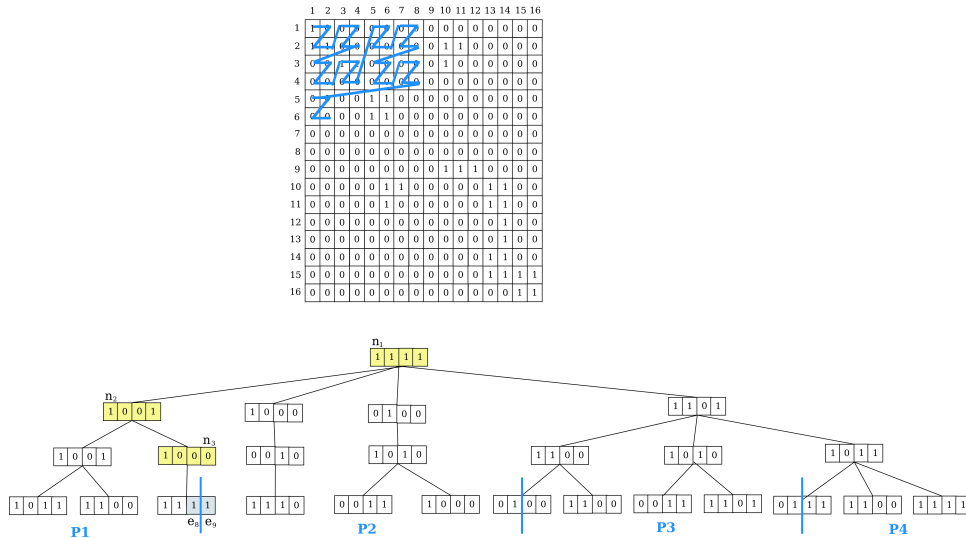


Figure 6.6: An example of an edge-balanced partition (bottom) for an adjacency matrix of 16 nodes (top), where z-ordering for first cells is also shown

The Edge-Balanced partition allocates the same number of edges to all the processors, trying to minimize the ancestors that have to be replicated in more than one processor. In this way, each processor stores its local K^2 -tree, with the same height than the global one. However, it only contains its corresponding edges. Note that the common ancestors whose edges are located in different processors are replicated and near edges in the bottom level of the tree share more common ancestors. Therefore, we allocate $\lceil \frac{|E|}{|P|} \rceil$ edges to each processor, following the order of appearance of the edges in the bottom level of the K^2 -tree.

The location of the edges in the bottom level follows a Z -ordering over the adjacency matrix. Figure 6.6 illustrates the Z -ordering of the first edges over the matrix. So, if we consider the list of edges in Z -order (E'), each processor k stores $\{E'[i], i = (k - 1) * \lceil \frac{|E'|}{|P|} \rceil + 1, \dots, k * \lceil \frac{|E'|}{|P|} \rceil\}$. The bottom of the figure shows how the edges of the K^2 -tree are distributed over 4 processors. The graph contains 24 edges, so 8 edges are represented in each processor.

In that way, the spatial overhead that produces the replication of the common ancestors in different processors is minimized. As an example of this replication, we can observe the edge e_8 and the edge e_9 in Figure 6.6, that are stored in different processors (P_1 and P_2). They produce a replication of their common ancestors (n_1, n_2 and n_3) which have to be stored in both processors. However, note that

the maximum spatial overhead between two consecutive processors P_i and P_{i+1} , which is produced when the last edge of P_i and the first edge of P_{i+1} belongs to the same node in the last level, is only $\lceil K^2 \log_K |N| \rceil$ bits. As consequence, the maximum spatial overhead (in bits) regarding to the global adjacency matrix in this distribution, for a network of $|P|$ processors, is $|P| * \lceil K^2 \log_K |N| \rceil$.

6.3.2 Perfect Spatial Balanced distribution

Edge-Balanced distribution equally allocates the same number of edges in each processor. It also minimizes the spatial overhead by distributing the edges following the appearance order in the bottom level of the global K^2 -tree, that is, following a Z -ordering over the adjacency matrix. However, it is easy to see that this distribution does not guarantee a spatial balance.

First of all, the cost of representing an edge in the K^2 -tree is given by the nodes that represent it in all levels (that is, the cost of representing the different sub-matrices of sizes $K^i \times K^i, i = 0 \dots \lceil \log_K |N| \rceil - 1$). Each of those sub-matrices is represented in a level of the tree with K^2 bits. Accordingly to that, the cost of e_i is defined as $C(e_i) = K^2 \log_K |N|$ bits. For instance, the cost of e_1 in the Figure 6.6 is 16 bits, since n_1, n_2, n_3 and n_4 are representing that edge, using K^2 bits for each one.

However, when the global K^2 -tree is considered, not all the edges cost the same. Consider two edges e_i and e_{i+1} that are located in the same node. They share common ancestors (nodes of the intermediate levels). If those two edges are finally located in the same processor, the final cost of storing the two edges is less than $C(e_i) + C(e_{i+1})$, since the shared bits are stored only once. So, in order to obtain a more fine-grain cost measure, we need to define a relative cost for each edge e_i denoted as $C'(e_i)$.

For that purpose, we first define the spatial cost of a graph $G = (N, E')$, denoted as $SC(G)$, as the number of bits that the K^2 -tree structure spends in order to represent G . Then, the relative cost of an edge e_i is $C'(e_i) = SC(G') - SC(G'')$, where $G' = (N, \{e_1, \dots, e_i\})$ and $G'' = (N, \{e_1, \dots, e_{i-1}\})$. The intuition under this definition is that the cost of storing e_i is the number of bits that costs to add the edge e_i to the K^2 -tree when all the previous edges (following a Z -ordering) are represented. That is, it is given by the additional bits that e_i needs to be represented in this tree. Those additional bits are such that e_i does not share with e_{i-1} . In that way, the cost of each intermediate node of the K^2 -tree is only computed for the first edge (always considering the Z -ordering or $E'[i]$) that needs it to be represented.

Given that $SC(G) = \sum_{i=1}^{|E'|} C'(e_i)$, the proposal for achieving a perfect spatial balanced partition consists in distributing the edges of the graph by storing in each

Algorithm 6.4 Perfect Spatial Balanced Algorithm

```

procedure ASSIGN( $C', P, SC(G)$ )
   $cost \leftarrow 0$ 
   $processor \leftarrow 1$ 
  for  $i = 1 \dots |C'|$  do
    if  $cost \geq processor * \lceil \frac{SC(G)}{|P|} \rceil$  then
       $mapProc[i] \leftarrow processor$ 
       $cost \leftarrow cost + C'[i]$ 
    end if
  end for
  return mapProc
end procedure

```

processor a K^2 -tree containing consecutive edges e_m, \dots, e_n , where $\sum_{i=m}^n C'(e_i) \simeq \frac{SC(G)}{|P|}$. The process that maps each edge e_i (that is, the i -th edge of the tree considering a Z -ordering) to the corresponding processor is shown in Algorithm 6.4. The algorithm receives as input the list C' representing in $C'[i]$ the relative cost of the i -th edge (following a Z -order), the set of processors P and the total cost of storing the full K^2 -tree ($SC(G)$) previously computed. It outputs the vector $mapProc$, where $mapProc[i]$ contains the processor in which the i -th edge is stored.

Figure 6.7 shows the partitions obtained by following a Perfect Spatial Balanced strategy for the example in Figure 6.6. Note that both edges e_8 and e_9 , which belonged to different processors in the Figure 6.6, belong to P_1 in this case. When edges are closer, sharing more common ancestors, the relative cost of each edge is reduced and more edges can be stored in the same processor. In the example, the processor P_1 stores 9 edges, while processor P_2 only store 5 edges, since the relative costs of those edges is greater than the costs of the edges of the processor P_1 .

6.3.3 Querying the Spatial Balanced partitions

For the basic partitioning algorithms described in Section 6.2 (excluding some of the adaptive approaches), each row and column could be mapped to $1 \dots |P|$ processors. Given a direct or reverse neighbor query, the processors and the location of the queried row or column in that processors can be computed through a simple formula.

But in this new family of distributions, this process cannot be performed in the same way. Since the edges are distributed following a Z -ordering, they are not distributed by their row and column location in the matrix. Therefore, given a direct or reverse neighbor, the processors that are involved in the operation are not easily

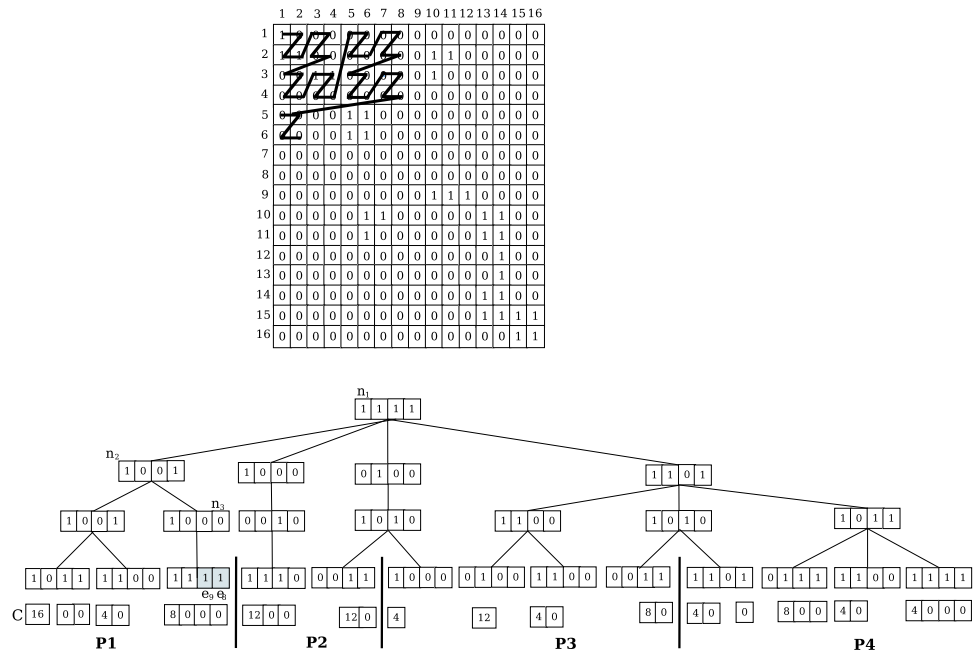


Figure 6.7: An example of a perfect spatial balanced distribution (bottom) for an adjacency matrix of 16 nodes (top), where relative cost C' for each edge is shown

computed. All processors will be checked for every query over the graph.

In return for not having a direct mapping, since each processor stores the original K^2 -tree (although pruning some branches), no mapping process has to be done to perform the query neither to transform the local results in global results. That is, an edge (i, j) of the global graph GM corresponds with the cell (i, j) in all LM_k , although at most one of the processors will have this cell set to *one*. The remaining processors will output 0 when this cell is queried.

Checking all processors for every query seems too costly. However, many of these operations are solved by an early prune in the first levels of the tree, resulting in a minimal overhead regarding to the execution in only the processors that can produce results.

6.4 Latin-Square partition

Basic distributions described in Section 6.2 propose partitions where the mapping of each cell to the processors only depends on the position of the cell in the adjacency matrix. Adaptive proposals use the information of the matrix to adapt a partition to a given input data but the queries are not homogeneously distributed among the processors. Spatial-balanced partitions described in Section 6.3 attend to the adjacency matrix properties. However, since the order of the map follows a Z curve is not guaranteed that each row or column will be homogeneously distributed over all processors.

The purpose of this new distribution is to homogeneously distribute the rows and columns over all processors, in the way that each query implies checking the same number of cells in all processors.

In order to obtain a distribution where all rows and columns have the same number of cells in each processor, an initial partition over the global adjacency matrix GA is performed. It consists in dividing the adjacency matrix in $|P| \times |P|$ squared regions, exactly as in a basic grid distribution for $|P|^2$ processors. Each squared region will be represented through an individual K^2 -tree. We denote as $T[i][j]$ the K^2 -tree representing the adjacency matrix located in the row i and column j of the $|P| \times |P|$ sized grid over the adjacency matrix. Each $T[i][j]$ has a spatial cost of $C[i][j]$ bits. The basic purpose of this partition consists in finding a way to distribute the grid of K^2 -trees by allocating to each processor one and only one of the K^2 -trees of each row and each column of the grid, while it tries to maintain a good spatial balance.

6.4.1 Latin Squares

To achieve the main constraint of this distribution, that is, to assign a grid $|P| \times |P|$ in the way that one cell of each row and each column is mapped to one processor, the mathematical Latin Square concept can provide support as an allocation template.

As briefly explain in Section 5.3.4, a Latin Square (LS) is a matrix of size $n \times n$ where each cell $LS[i][j]$ contains a value between $1 \dots n$ and each value appears once in each row and column. Any Latin Square can be modified by permuting its rows and its columns and it still is a Latin Square. Many other operations over a Latin Square produce other Latin Square. In particular, a Latin Square can be transformed to a normalized form accomplishing that $LS[1][j] < LS[1][j'] \Leftrightarrow j < j'$ and $LS[i][1] < LS[i'][1] \Leftrightarrow i < i'$.

6.4.2 Algorithm of distribution

We propose the distribution of a grid of $|P| \times |P| K^2$ -trees by using a Latin Square LS of size $|P| \times |P|$ as a template. Using an allocation based on a Latin Square, we have the initial constraint of this distribution. That is, each row and each column of the grid is homogeneously distributed over all processors. A random Latin Square could be used in order to obtain a valid partition.

However, we are also interested in balancing the space, so we will use the special properties of Latin Squares to obtain more balanced distributions by permuting the rows and the columns of the used Latin Square.

We start from an initial Latin Square, that we use as a template of assignation for the $|P| \times |P| K^2$ -trees. Then, we permute the rows of the Latin Square template, trying to balance the spatial cost of the different processors, while the initial constraint of this distribution is maintained. Algorithm 6.5 illustrates the process. First, the initial Latin Square template is set. Note that, although any other Latin Square could be used, we use as an example the normalized Latin Square shown in the top-right of the Figure 6.8. The total spatial cost for the row i (composed by $|P| K^2$ -trees) of the grid is stored in $RC[i]$. Figure 6.8 (middle) shows the spatial cost for each K^2 -tree (C) and for the K^2 -trees of each row (RC). For instance, the K^2 -tree representing the top-left adjacency matrix costs 12 bits, and the top row of 4 K^2 -trees costs 28 bits.

In the first step, the most costly row is allocated by using an available row (not used yet) of the Latin Square template. In the example, the third row with cost 40 is chosen, and it is assigned to the row 1 of the Latin Square template (LS). The process continues by selecting the most costly row in each step ($cRow$). $cRow$ will be assigned to the available row of LS which most balances the current spatial costs by processors ($bRow$). In the example of the Figure, the second step assigns the

Algorithm 6.6 Data distribution for the Latin-Square approach

```

procedure ASSIGN( $C, P$ )
    ▷ Latin Square Template
    for  $i \leftarrow 1 \dots |P|$  do
         $RC[i] \leftarrow \sum_{j=1}^{|P|} C_{i,j}$ 
         $PC[i] \leftarrow 0$ ;
        for  $j \leftarrow 1 \dots |P|$  do
             $LS(i, j) \leftarrow (i + j - 2 \bmod |P|) + 1$ 
        end for
    end for
    for  $i = 1 \rightarrow |P|$  do
         $cRow \leftarrow k | RC(k) = \max_{j=1}^{|P|} RC(j)$ 
         $maxBalance \leftarrow 0$ 
        ▷ Obtaining the best balance
        for  $j = 1 \leftarrow 1 \dots |LS|$  do
            for  $k \leftarrow 1 \dots |P|$  do
                 $PC'[LS[j][k]] \leftarrow PC[LS[j][k]] + C[cRow][k]$ 
            end for
            if  $balance(PC') > maxBalance$  then
                 $bRow \leftarrow j$ ;
                 $maxBalance \leftarrow balance(PC')$ 
            end if
        end for
        ▷ Allocating the best option
        for  $j \leftarrow 1 \dots |P|$  do
             $mapProc[cRow][j] \leftarrow LS[bRow][j]$ 
             $PC[LS[bRow][j]] \leftarrow PC[LS[bRow][j]] + C[cRow][j]$ 
        end for
         $remove(LS[bRow])$ 
         $RC[cRow] \leftarrow 0$ 
    end for
    return mapProc
end procedure

```

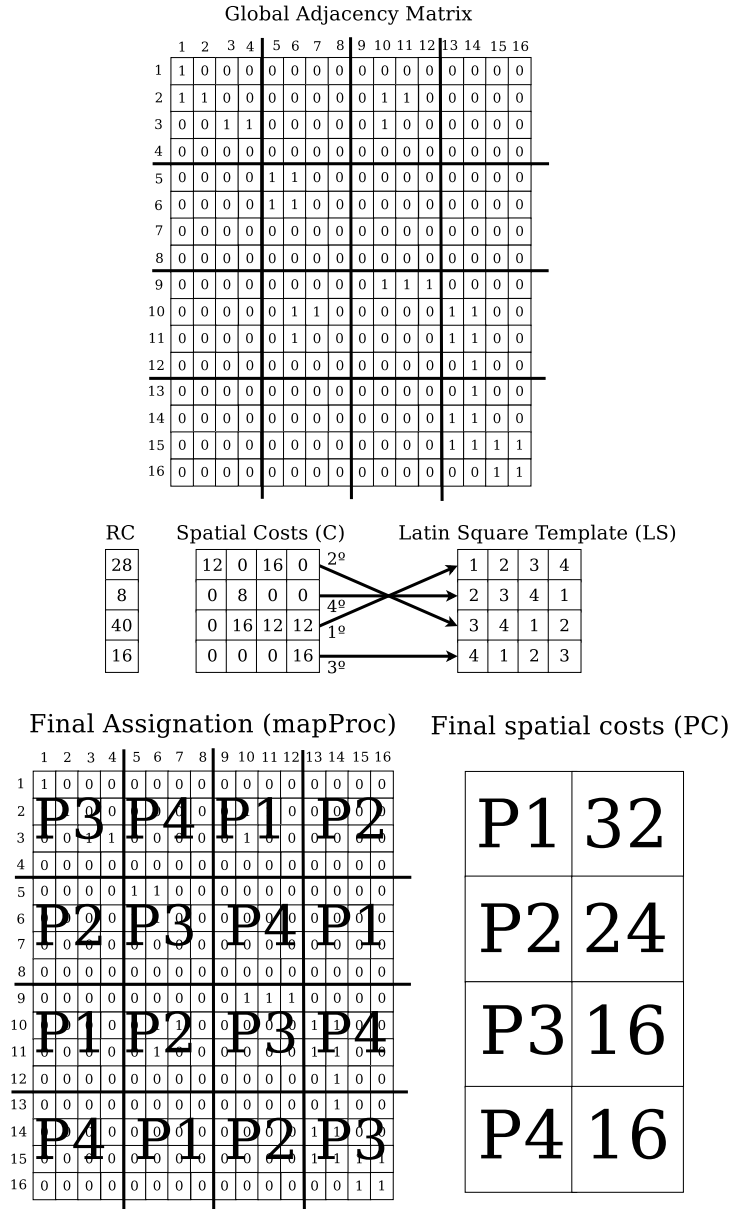


Figure 6.8: An example of a Latin Square partition showing the final assignation grid (bottom-left) for an adjacency matrix of 16 nodes (top-left), using a Latin Square template (top-right)

row 1 (which cost 28) of the grid to the row 3 of the Latin Square template. The current spatial costs in each processor are $PC[1] = 16$, $PC[2] = 16$, $PC[3] = 24$ and $PC[4] = 12$. The current spatial cost of each processor i is stored in $PC[i]$. The balance of each possible combination in each step is computed given by $\frac{PC}{\max_{i=1}^{|P|} PC[i]}$. The row of the Latin Square which most balance the distribution is allocated to the corresponding row of the grid of K^2 -tree. Then, this row of the Latin Square is disabled, since it cannot be used any more. This process is repeated $|P|$ times, until each $T[i][j]$ has its corresponding processor stored in $mapProc[i][j]$. The final grid assignation of the example can be seen in the Figure 6.8. For instance, we can see that the tree $T[3,3]$ with a spatial cost 12 will be stored in the processor 3. The final spatial costs by processor, that this algorithm tries to balance, is also shown in the figure (PC).

6.5 Execution Cycle

The previous sections describe different proposals to partition the graph in $|P|$ processors. The final purpose of this partitioning is to develop a full system, composed by $|P|$ processors, that receives queries and returns the answer, independently of the internal distribution of the data. With that purpose, we design a query system following the message passing paradigm and using synchronization barriers, in the way that a barrier at the end of each step synchronizes all the processors, where the messages that were sent during this step are received in the corresponding addressed processor.

All the processors execute the same algorithm, in the way that each processor is the responsible of sending the proper messages to other processors and it also receives the answers in order to solve the queries that it manages.

Therefore, the algorithm is composed of a set of steps, that ends with a sync barrier. In each superstep i of this algorithm, all processors execute three tasks:

- *Read n queries.* Each processor process its own pull of queries. For the sake of the simplicity, we assume this queue is never empty and all processors receive n requests in each step. In that way, we ensure that the observed load imbalance is not due to a imbalanced number of received queries. Each query will be sent to the corresponding processors, according to the rules of each partition. Some of the partitions, like multi-level grid distribution, need to map the global row or column to the corresponding local row or column through the explained formula. Other distributions, like the spatial-balanced partitions, need to send each query to all the processors, but no mapping is necessary, since each cell (i, j) corresponds with the cell (i, j) in all processors. For each step, all the

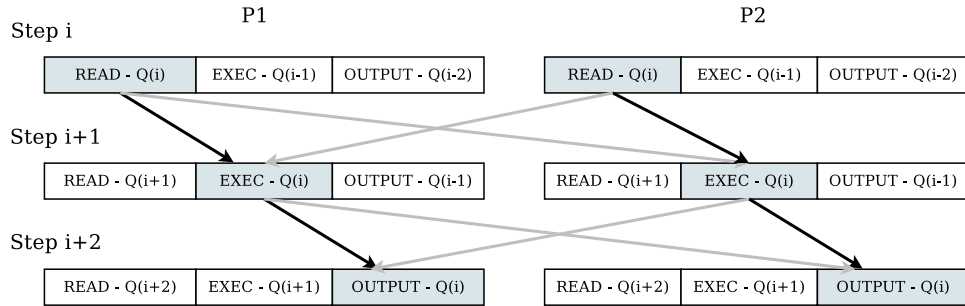


Figure 6.9: An example of three steps in the execution cycle for two processors and the external communication involved between the two processors needed to complete the queries received in the step i)

queries from a processor to another one are packed in the same message. So, in each step, depending on the distribution, each processor can receive up to $|P - 1| * n$ external queries (in $|P - 1|$ messages), which have to be computed and answered.

- *Query execution.* The external queries received from the other processors and the internal queries read in the previous superstep $i - 1$ are executed over the local data in this phase. New messages are created in order to answer the queries to the corresponding processor (that is, the processor which made the request).
- *Output n queries.* The n queries that the processor read in the superstep $i - 2$ were executed by the corresponding processors in the superstep $i - 1$, so in this superstep all the answers have been received and the processor is ready to output the answer, resulting of the union of the answers of all involved processors in that query. Depending on the distribution, this operation can involve the reverse mapping of the local results to the coordinates of the global matrix.

Figure 6.9 shows the phases along three supersteps for a system with two processors, and the communication involved to answer the queries read in the step i . The messages between different processors are highlighted.

6.6 Summary

This chapter presented the graph partitioning strategies we designed to distribute graphs by using the compact K^2 -tree structure. The first proposals are based on partitioning the adjacency matrix. In this way, we defined the basic, cyclic and grid distributions, which divide the matrix in P submatrices, each of one is represented with a K^2 -tree in a different processor.

These distributions do not take into account the distribution of edges into the adjacency matrix, so adaptive versions of these strategies were implemented. It is worth mentioning the adaptive multigrid distribution that, using another K^2 -tree as index, builds a set of K^2 -trees that are distributed over the processors.

A different kind of strategies was also proposed, specially focused on the structural characteristics of the K^2 -tree structure. The edge-balanced distribution allocates the same number of edges in all the processors, by performing a vertical partitioning over the conceptual K^2 -tree. Perfect spatial balanced distribution improved this strategy by considering the relative cost of storing each edge in order to achieve a perfect spatial balance.

In the Latin-Square partitioning we proposed a different approach to the problem, where the matrix is stored through a set of K^2 -trees, and the partitioning algorithm tries to distribute this set among the different processors by balancing the space as much as possible.

Finally, Section 6.5 describes the cycle of execution that follows all the distributions in order to solve the direct and reverse neighbor operations.

Next chapter experimentally evaluates all of these strategies by using Web and Social graphs, analysing its spatial and temporal balance and efficiency to expose use cases of each strategy.

Chapter 7

Experimental evaluation

In this chapter we present an experimental evaluation of the different partitioning algorithms described in Chapter 6 to distribute a graph using K^2 -trees as internal representation.

The purpose of this experimentation is to compare the spatial and temporal efficiency of the different partitioning strategies. The spatial balance is a good measure of the scalability of the distribution. All the distributions store the final graph in the main memory, so the more balanced the space is among the different processors, the bigger graphs can be managed by the same set of processors. However, some of the proposed distributions can break the compressibility of the data, so the total space used to store the distributed graph is also considered.

On the other hand, the temporal balance is measured, in order to analyze how the work load is distributed in the processors. The purpose is to minimize the time in which each processor is idle in each superstep of the querying cycle, waiting for the remaining processors in the synchronization barrier. However, the most important temporal measure is given by the *Speed-up* obtained for each distribution.

The chapter is structured as follows. First, the execution environment and the datasets that are used in the experimentation are presented. Then, the total space and the spatial balance for each distribution are compared. After that, the temporal results are described comparing the different distribution approaches. Finally, a global analysis of the results is presented.

7.1 Experimental Setup

7.1.1 Execution environment

We execute all the experiments of this section on a cluster composed by 67 processing nodes. Each node is equipped with two quad-core Intel Xeon E5555 processors running at 2.67GHz, and 24GB RAM. The cluster uses an Infiniband network to communicate the nodes for calculations and I/O purposes, that reaches a peak bandwidth of 40Gb/s per port and a latency of 100 nsec. The computing processors use a message passing communication library (MPI) to communicate. In the experiments, we ensure that each process is located in a different processor in the cluster, in order to obtain a fair measure of the communication costs between the different processors.

Datasets

We analyze the performance of all our implementations using well known graphs for experimentation from two different contexts: web graphs and social networks. The main purpose is to compare the different behavior of the partitioning proposals depending on the nature and the distribution of the dataset. Next, we describe the main characteristics of the datasets:

- **Epinions**¹ is a small social graph from the SNAP collection [Les] representing the trust relationship between the members of the network. It is a tiny graph which is included in the evaluation to analyse the behavior of the distributions in small graphs.
- **Live Journal** is a social graph, also obtained from the SNAP collection [Les] that represents the relationships between the users of this community.
- **EU** is a small Web crawl from the Web Graph project [BV04] [BRSV] [BCSV] that represents links between pages of the *.eu* domain.
- **UK** is a large Web crawl from the Web Graph project representing links between pages of the *.uk* domain from 2002.

Table 7.1 shows the number of nodes and edges of the graphs. It also shows the size (in Megabytes) which costs storing them in an individual K^2 -tree (using $K = 2$). Last column shows the number of bits per edge for each different dataset, giving a reference of the compressibility of the graphs with a K^2 -tree.

¹<http://www.epinions.com/>

Graph	Domain	# Nodes	# Edges	Size (MB)	BPE
Epinions	Social	75.888	508.837	0.89	14.67
LiveJournal	Social	4.847.571	68.993.773	167.29	20.34
EU	Web	862.664	19.235.140	11.86	5.18
UK-2002	Web	18.520.486	298.113.762	149.78	4.21

Table 7.1: Web and social graphs used in this experimentation

Queries

Each processor will execute 10.000 direct or reverse neighbor queries (depending on the experiment) in each superstep, that is, the querying system processes $10.000 * |P|$ queries per superstep. The performance is evaluated from the average time of 10 supersteps. We run experiments with 1 (sequential version), 4, 9, 16 and 25 processors.

7.2 Spatial evaluation

In this section the spatial cost of the different partitioning strategies is evaluated. First, the total spatial cost of storing the graph in the distributed environment is measured. After that, the balance of this spatial cost among the different processors is also evaluated.

7.2.1 Total spatial cost

Figure 7.1 shows the total space of the different graph partitioning strategies for Social Networks, while Figure 7.2 shows the total space achieved in the Web graphs. Each line of the Figure shows a different strategy implemented over a network with 4, 9, 16 and 25 processors. The total cost is the sum of the costs of the local K^2 -trees and the additional structures needed to map the cells from the global matrix to the cells in each local processor. In general, the cost of the additional structures are residual, with the only exception of the Adaptive MultiGrid distribution, where an additional K^2 -tree (used as index of the remaining K^2 -tree structures) is replicated in all processors.

The results show that all the distributions achieve a similar compression regarding to the sequential version. The only exception are the distributions based on cyclical

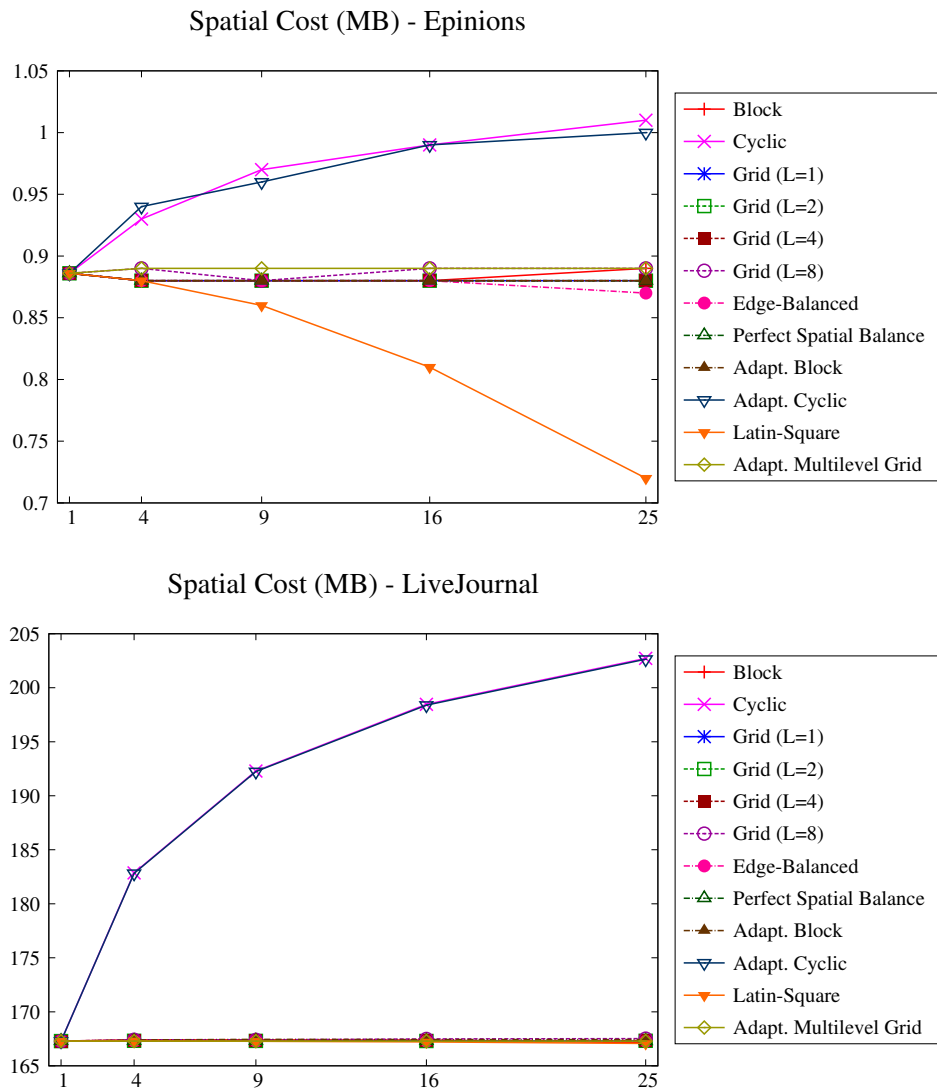


Figure 7.1: Epinions and Livejournal total space

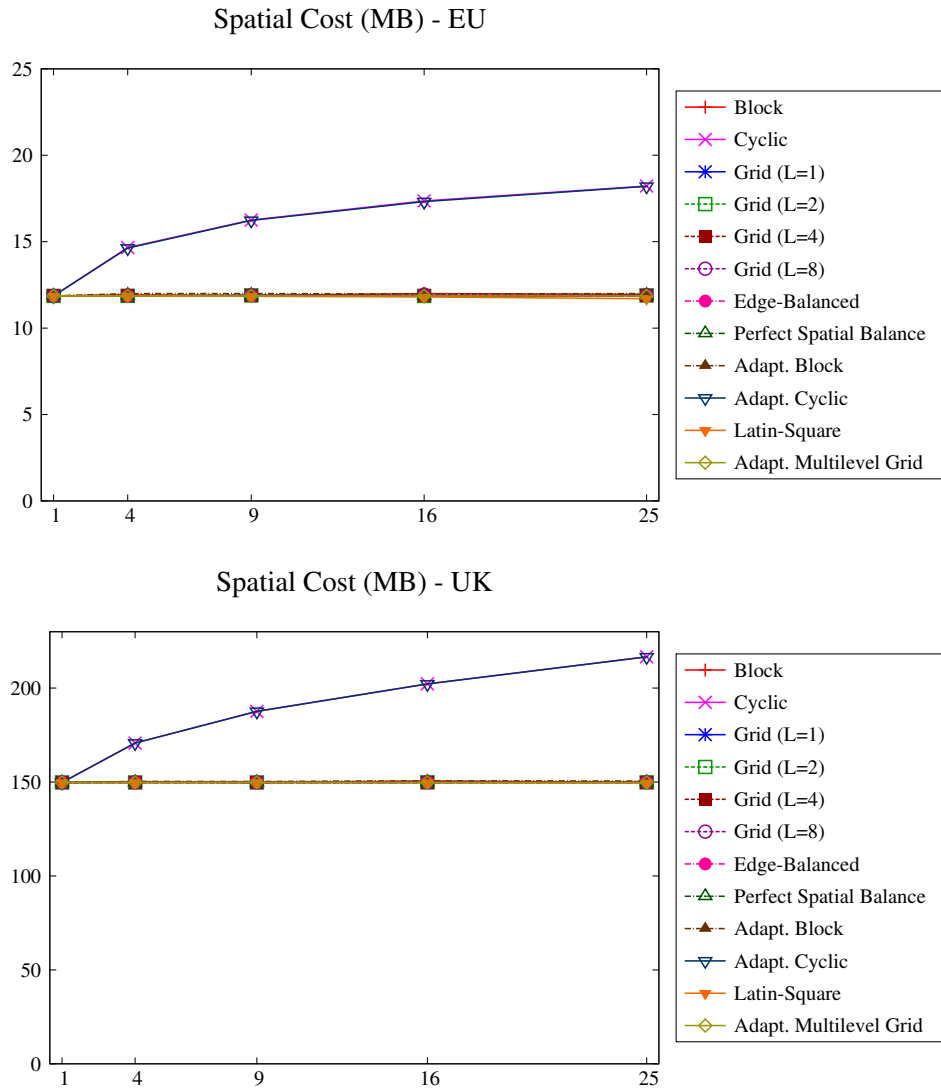


Figure 7.2: EU and UK total space

allocation, that is, the Cyclic distribution and the Adaptive Cyclic distribution. They reduce the compression of the K^2 -tree, because the locality of the data is lost, getting worse as the number of processors grows. For instance, we can observe that Cyclic distribution for the LiveJournal dataset with 25 processors has a 25% of extra cost regarding to the sequential version. Latin Square distribution also presents an interesting feature. When the number of processors grows, the number of K^2 -trees generated also grows, reducing the height of the resulting K^2 -trees. For instance, in the case of $|P| = 25$, a grid of 625 K^2 -trees is generated. This first partition of the data can produce an spatial saving that is specially meaningfull in tiny datasets like Epinions.

This behavior is repeated in the four analysed datasets, so we can conclude that Cyclic distribution and Adaptive Cyclic distributions are the only distributions that affect the total spatial cost of the graph, and this spatial cost grows with the number of processors.

7.2.2 Spatial efficiency

We analyze how the spatial cost is distributed among the different processors. The more balanced the spatial cost is, the bigger datasets will fit in the main-memory of each individually processor, so this is a relevant measure of the scalability of the graph partitioning strategy. Spatial efficiency is computed as $avgSpace/maxSpace$ where $avgSpace = \frac{\sum_{i=1}^{|P|} SC(LA_i)}{|P|}$ is the average spatial cost of the local subgraphs stored in each processor and $maxSpace = \max_{i=1}^{|P|} SC(LA_i)$ is the maximum space of all processors.

Figure 7.3 shows the spatial balance achieved for all the distributions. First, we can observe the behavior of the 1D-partitioning strategies. Block distribution achieves a poor balance of the space, because it is a distribution that strongly depends on the original data distribution. This effect is specially noted in the case of the evaluated social graphs, where the upper-left region of the adjacency matrix presents bigger edge density. Therefore, the first processor has to represent a great percentage of the edges of the total graph, producing the bad results observed in the figure. Furthermore, this effect degenerates when the number of processors grows, because the first processor still gathers most of the edges and the network is greater, causing a worse balance. However, the results obtained for the Adaptive Block Partitioning strategy clearly improves the balance of the previous distribution.

Distributions based on the K^2 -tree structure, that is, the Edge Balanced and the Perfect Spatial Balanced distributions achieve very good spatial results. Edge Balanced distribution balance is around 0.8, which is improved with the more fine-grain balance of Perfect Balanced distribution which achieves almost a perfect

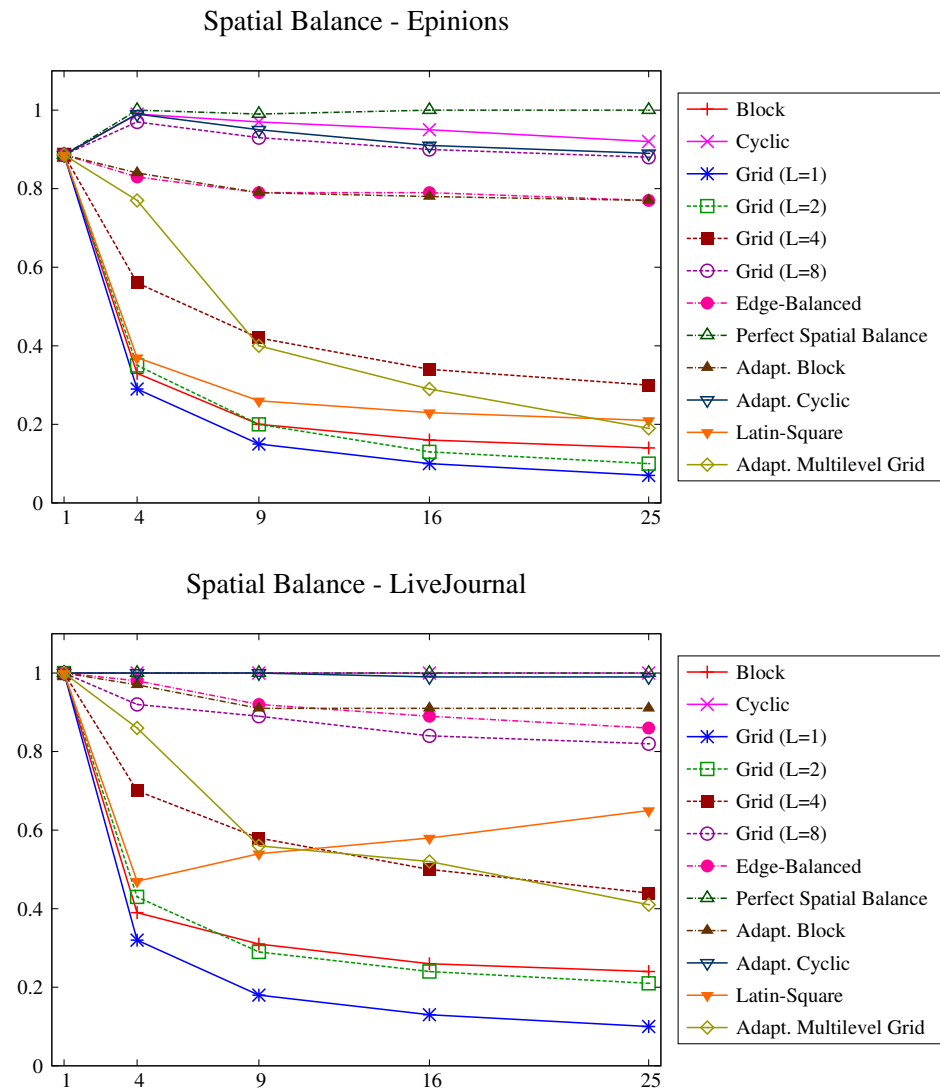


Figure 7.3: Epinions and Livejournal spatial efficiency

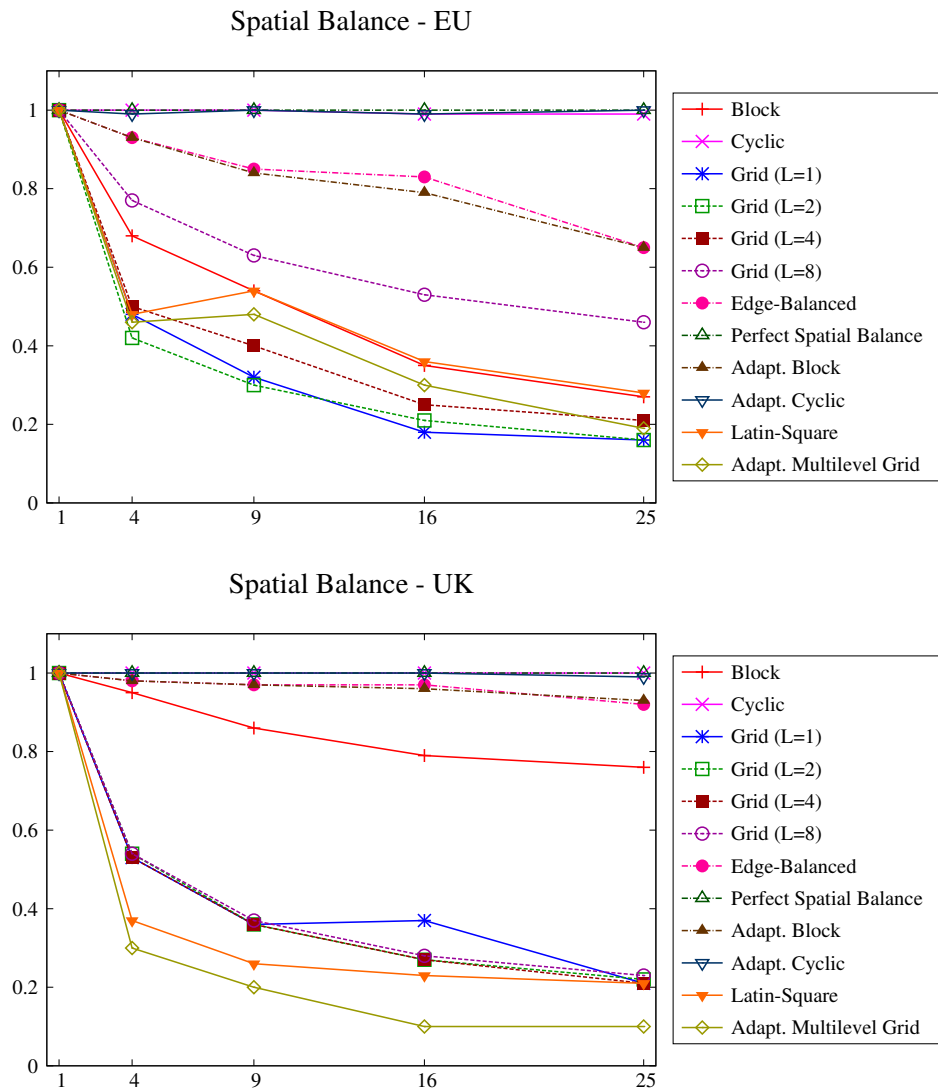


Figure 7.4: EU and UK spatial efficiency

balance (1).

Grid distributions with lower levels of granularity ($L = 1, 2, 4$) achieve bad spatial balance (in many cases, under 0.5), because they are very dependent on the graph distribution. However, a level of granularity $L = 8$ improves this efficiency, obtaining values around 0.9. Adaptive Multiple Grid distribution does not obtain a good spatial balance, and it is one of the distributions that get worst when the number of processor grows. Finally, Latin Squares also obtain a bad spatial balance, although in the case of *Livejournal* the balance is improved when the number of processors grows, because a bigger grid template produces a great number of K^2 -trees, which can be distributed better.

Figure 7.4 shows the spatial balance achieved in the Web graphs EU and UK. The results are quite similar to the obtained for the Social Graphs. However, in this case, the Multiple Grid distribution with parameter $L = 8$ does not obtain good results. Another difference is that the Block distribution works better in these Web graphs, because the distribution of the edges in the adjacency matrices of these Web graphs does not present most edges in the top-left of the matrix, as in the case of the studied social graphs.

Therefore, we can extract two main conclusions. First, the optimal parameter value for L depends on the graph, so the behavior of the grid distributions depends on the distribution of the edges in the adjacency matrix. On the other hand, the block distribution also obtain a good balance for some kinds of adjacency matrices, but it produces imbalanced distributions if a region of the matrix clusters most of the edges of the matrix.

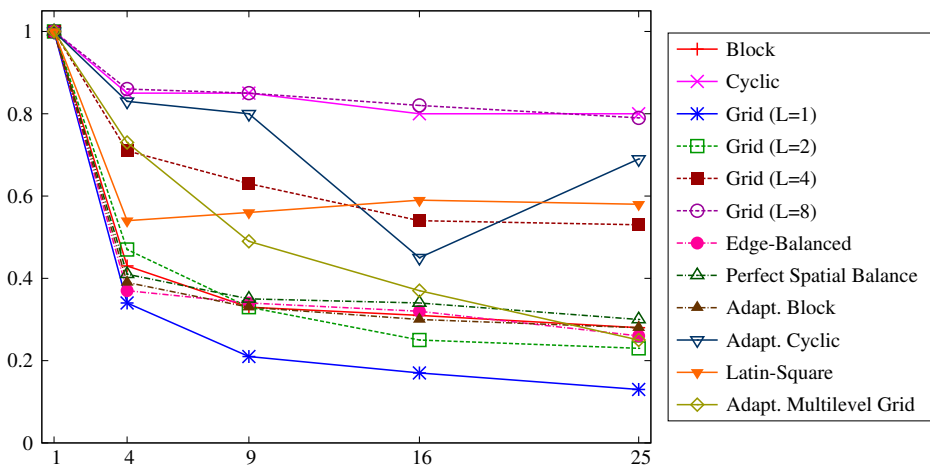
7.3 Temporal evaluation

We also analyze the running time performance of the different distributions.

7.3.1 Temporal efficiency

We first study the temporal efficiency of each distribution, which is computed as $avgTemp/maxTemp$ where $avgTemp = \frac{\sum_{i=1}^{|P|} TC(LA_i)}{|P|}$ is the average running time of processors per superstep and $maxTemp = \max_{i=1}^{|P|} TC(LA_i)$ is the maximum running time in any processor. The temporal efficiency is measured using the user time. It gives a measure of the time that processors are idle in each superstep, waiting for the remaining processors. A distribution that balances the work load, where all processors have a similar work load in each superstep, will achieve temporal efficiency with value 1. Figures 7.5 and 7.6 show the results for direct neighbor

Querying Efficiency - Direct Neighbors - Epinions



Querying Efficiency - Direct Neighbors - LiveJournal

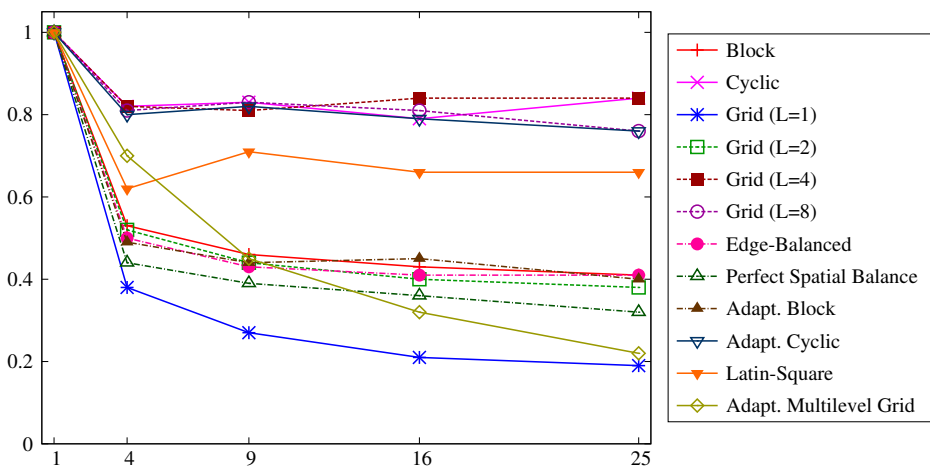


Figure 7.5: Epinions and Livejournal temporal efficiency for direct neighbors

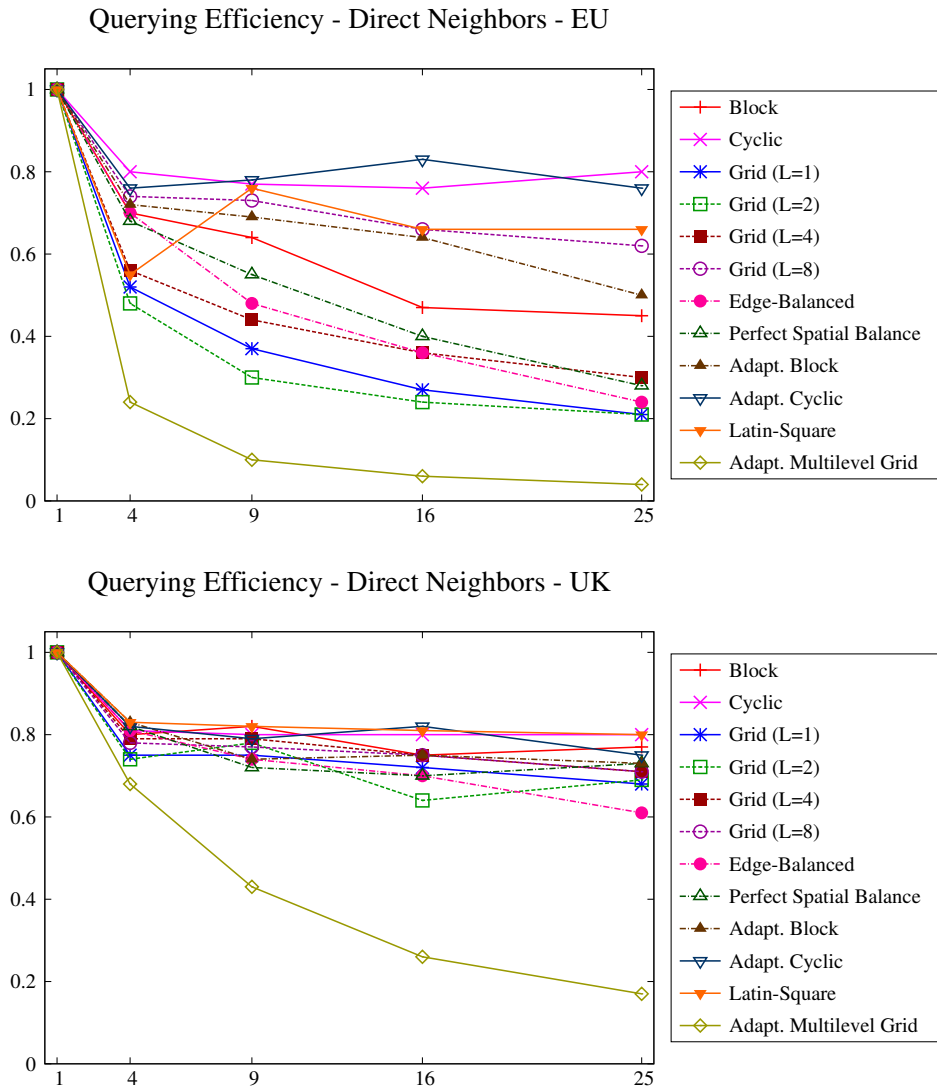
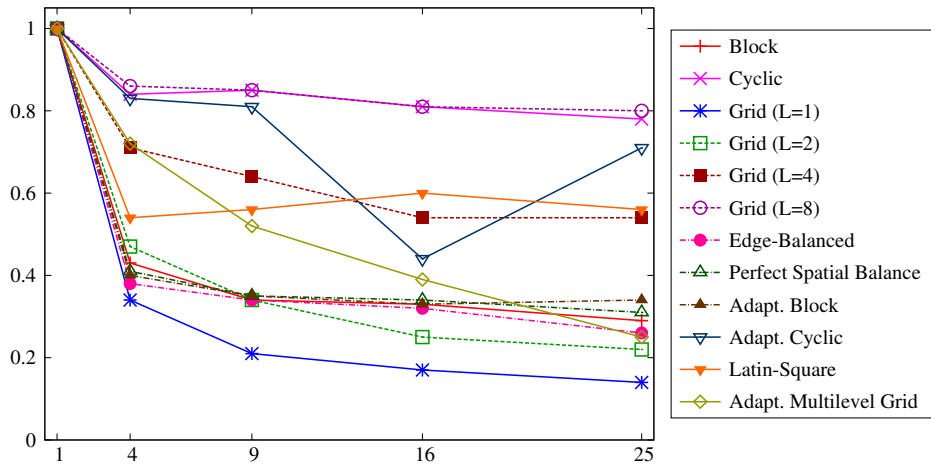
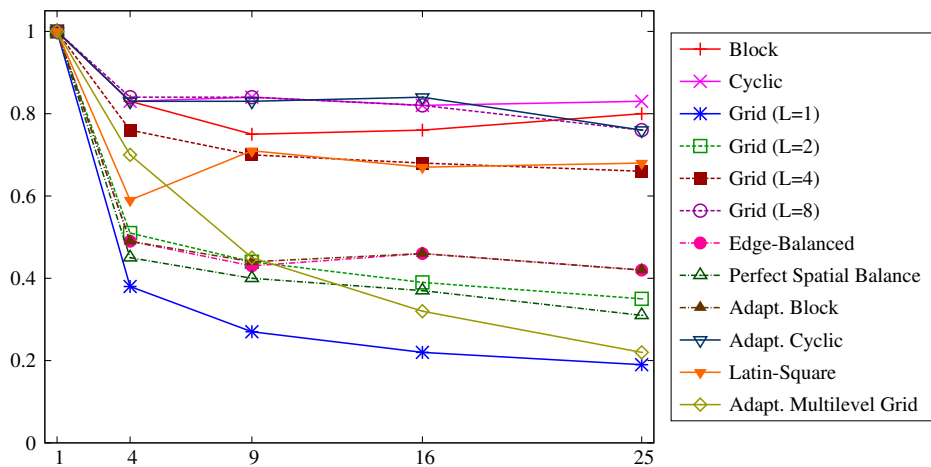


Figure 7.6: EU and UK temporal efficiency for direct neighbors

Querying Efficiency - Reverse Neighbors - Epinions



Querying Efficiency - Reverse Neighbors - LiveJournal

**Figure 7.7:** Epinions and Livejournal temporal efficiency for reverse neighbors

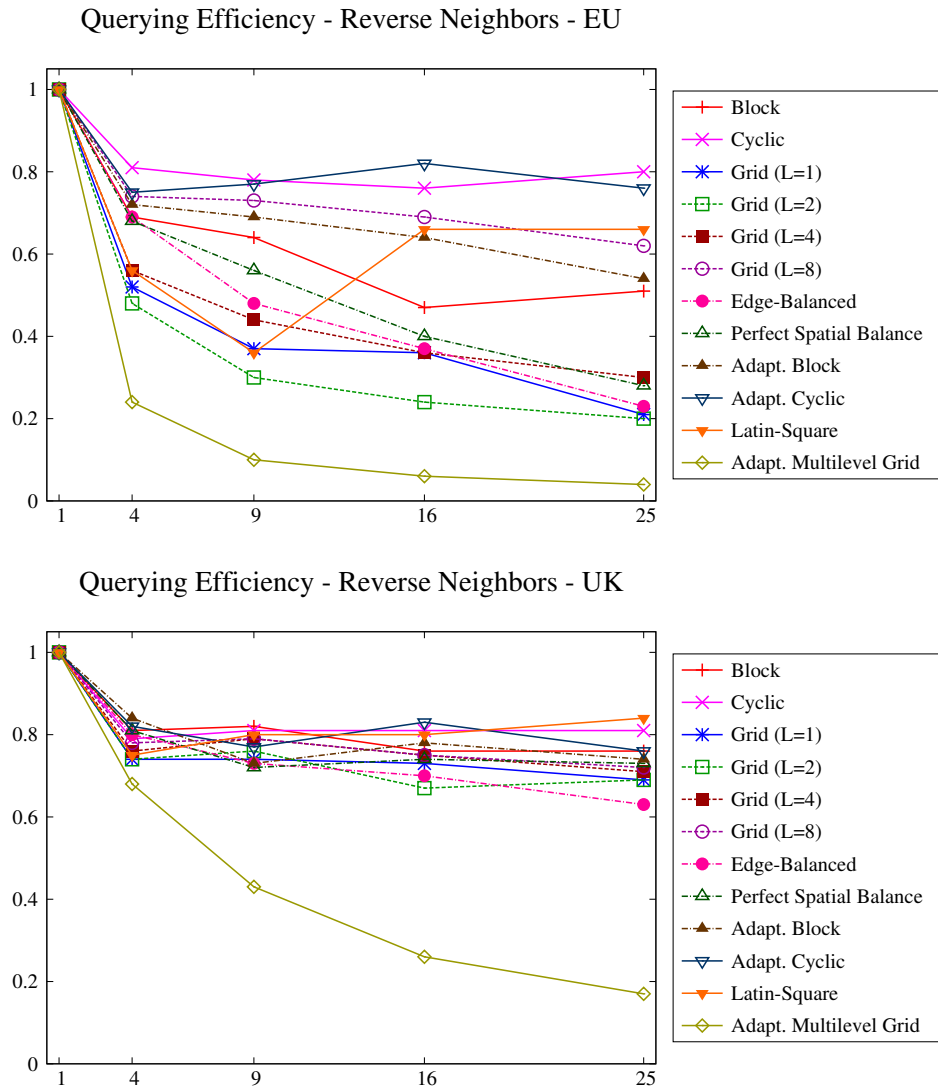


Figure 7.8: EU and UK spatial temporal efficiency for reverse neighbors

queries, while Figures 7.7 and 7.8 shows the results for reverse neighbor queries. Direct and reverse operations obtain quite similar results for each dataset. Grid distributions with a high level of L ($L = 8$) obtain a good temporal efficiency and cyclic (basic and adaptive) and Latin Square distributions also obtain good results. However, cyclic distributions obtain a good temporal efficiency in return of extra spatial requirements, as it was shown in the spatial evaluation. Grid distributions with low levels of L obtain poor temporal efficiency, produced by a bad spatial distribution.

7.3.2 Speed-up

Finally, we show the speed-up, which measures the real querying capacity of the system. The speed-up is defined as $\frac{T_s}{T_p}$ where T_s is the running time of a sequential algorithm for the problem and T_p the running time of a parallel algorithm. We use the real time of the processing in order to measure the total cost (including the processing and the message communication between the different processors).

Figures 7.9 and 7.11 show the speed-up obtained for the social datasets in direct and reverse neighbor operations. For the Epinions dataset, Latin Square and Multigrid ($L = 8$) are the best distributions, while Multigrid with lower levels obtains poor speed-ups ($L = 1, 2$). Note that for the Livejournal dataset, in addition to Multigrid distribution with $L = 8$, cyclic distribution also obtains a very good speed-up.

Figures 7.10 and 7.12 show the speed-up for the two Web graphs. Grid with $L = 8$, just as in the social graphs, obtains good results. However, in this case, adaptive block distribution also obtains a good speed-up. Note that block distributions are very dependent on the location of the edges in the adjacency matrix and they achieve a good spatial efficiency in these two datasets. Finally, Latin Square also obtains good results.

Therefore, grid distributions with high values of L obtain a good speed-up. Latin Squares are also a good choice. However, another distributions, like the Adaptive Block, have a behavior completely dependent of the graph nature.

7.3.3 Analysis

We reviewed the temporal and spatial results for the different distributions we proposed. In this section we analyze the strength points and the weakness of each distribution, according to the results observed in the four datasets we evaluated.

- Block distribution: the main strength of this distribution is its simplicity for

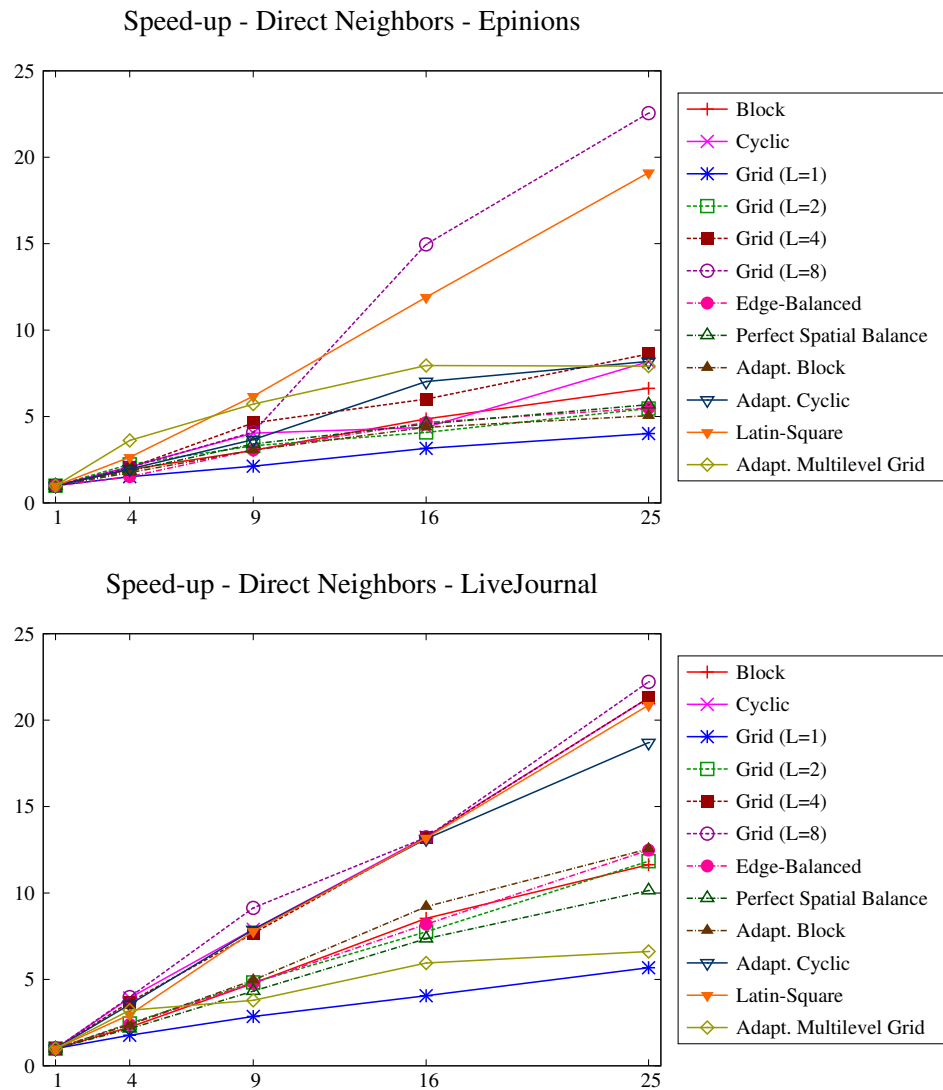


Figure 7.9: Epinions and Livejournal speed-up for direct neighbors

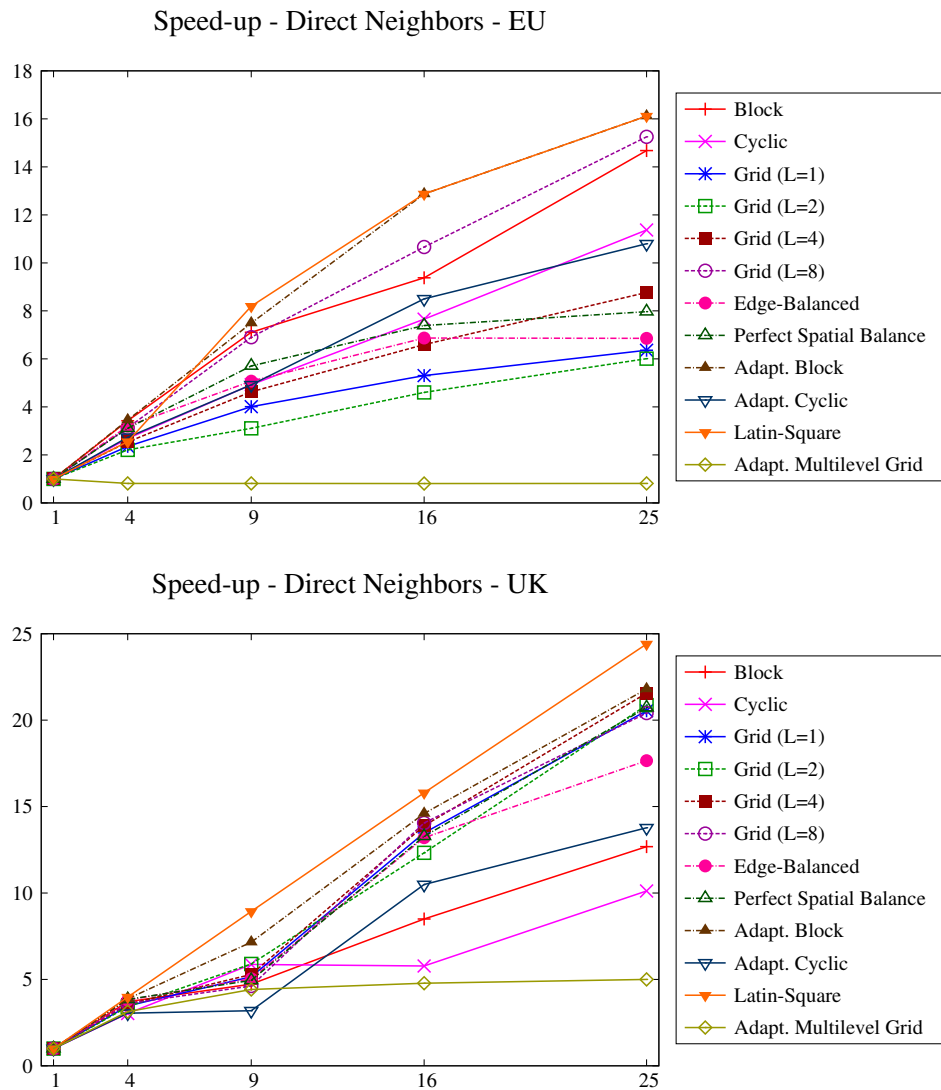


Figure 7.10: EU and UK speed-up for direct neighbors

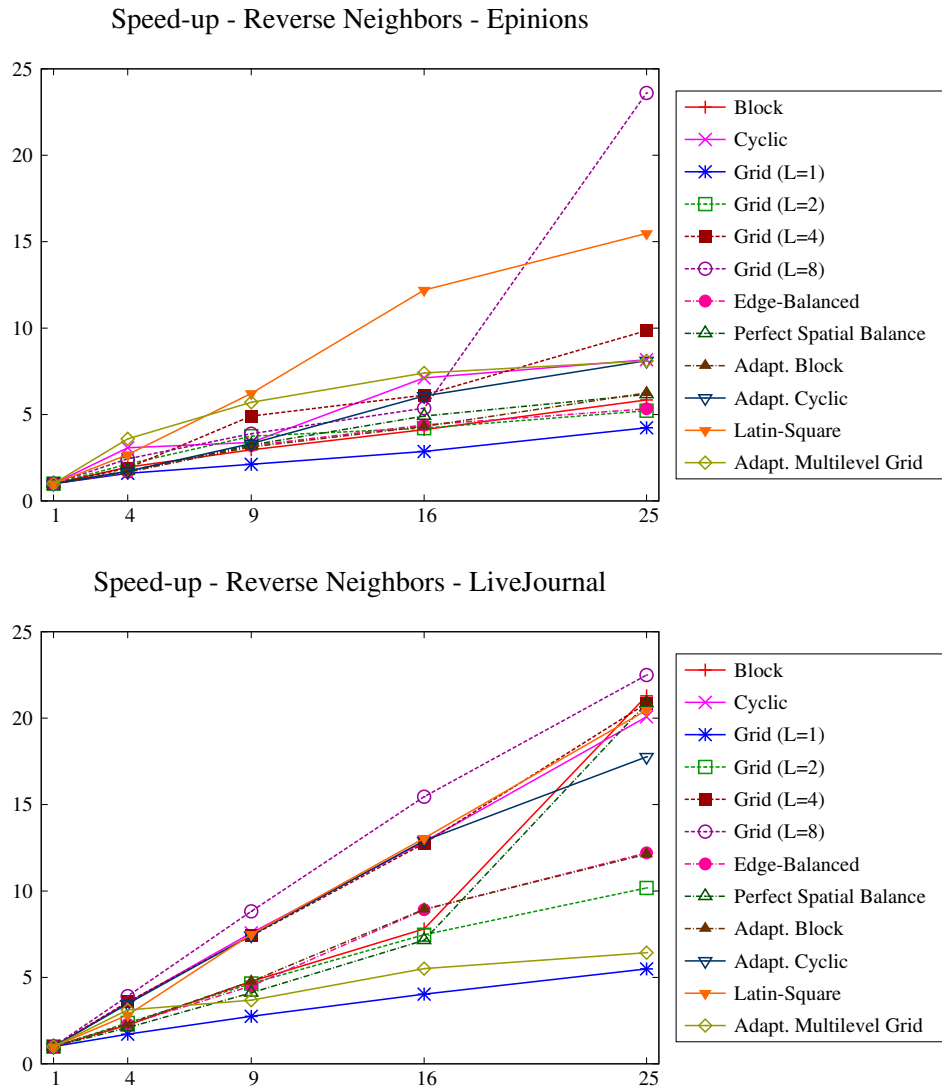


Figure 7.11: Epinions and Livejournal speed-up for reverse neighbors

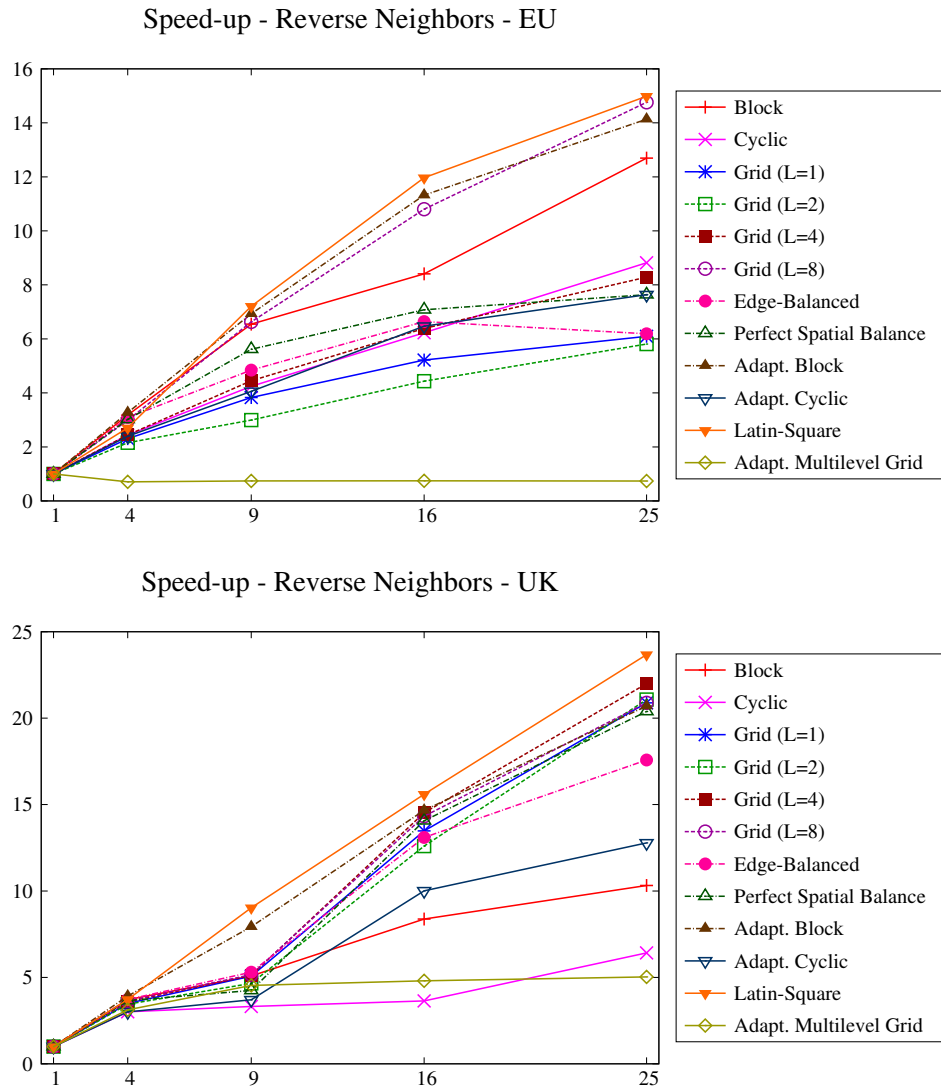


Figure 7.12: EU and UK spatial speed-up for reverse neighbors

computing the partitions. On the other hand, its strongly dependence on the original edge distribution can produce a poor spatial balance, specially in such datasets where edges are not equally distributed over the adjacency matrix, as we observed in the social network datasets evaluated like livejournal.

- Adaptive block distribution: this distribution still is a very lightweight partitioning algorithm that improves the spatial balance obtained with the previous distribution. However, for datasets where some regions of the adjacency matrix present a high concentration of edges, the rows with more edges will be stored in the same processor (although this processor will store less rows). Therefore, the processors storing dense areas will have more costly neighbor operations, producing in some cases a temporal imbalance.
- Cyclic distribution: although this distribution usually achieves a good spatial balance, independent to the distribution of the original matrix, it breaks the compressibility of the matrix, producing a meaningful spatial overhead. The work load balance is very good in this distribution, possibly due to the good spatial balance achieved. However, it obtains a moderate good speed-up, which can be caused by this lack of compressibility that produces the costs of the direct and reverse neighbor operations be more costly.
- Adaptive cyclic distribution: although this distribution was proposed to ensure the spatial balance of a cyclic distribution, the good results obtained by the cyclic distribution in terms of spatial balance makes the behavior of this new distribution quite identical to the non-adaptive version. Therefore, the main weakness still is the spatial overhead, that produces moderate speed-up values for this distribution.
- Grid distribution: this distribution was analyzed for different values of L , that determines the number of divisions of the global adjacency matrix in order to distribute it among the different processors. The results show that, in general, the higher the parameter L is, the more spatial balance the distribution will achieve. However, very high levels of L can break the compressibility of this distribution. Furthermore, the speed-up results are in general improved for higher L parameters, but the results show that it is very dependent on the dataset. As a consequence, the main weakness of this distribution is that there is no an optimal configuration of the distribution for every dataset.
- Adaptive grid distribution: this distribution presents the worst results, specially in terms of speed up, obtaining a very discrete increasing with the number of processors. However, in order to study the possibilities of this distribution is required a depth study of the parameter *limit*, that determines the minimum number of edges for a submatrix produce a new K^2 -tree allocated in a processor. Furthermore, in many cases, the problem is that many of the K^2 -trees created

are very small (containing only a few edges), so an alternative representation of this tiny trees could be studied. As conclusion, although the results of this distribution are not competitive with the current configuration, more studies could be performed in order to improve the results of the adaptive multiple grid distribution.

- Spatial balanced distributions: we proposed two alternatives that tries to balance the spatial balance taking into account the internal structure of the K^2 -tree. Perfect Spatial balanced distribution is an optimization of the edge balance distribution, so we only consider the optimized version in this analysis. It guarantees almost a perfect spatial balance, independently to the graph distribution. Furthermore, the spatial overhead is insignificant. Therefore, the spatial balanced distribution is the optimal distribution in terms of the spatial efficiency. However, the fact that each query has to be performed in every processor affects to the speed up of this distribution, specially for such datasets (like livejournal in our experimentation) where the edges are not homogeneously balanced.
- Latin Square presents the best results of this experimentation, since it is a commitment between the spatial and the temporal efficiency. The algorithm that permutes the rows of the Latin Square template obtains a reasonable good spatial balance, specially good in social graphs. On the other hand, it obtains the best speed-up values for web graphs and one of the best results in social graphs. Therefore, we can conclude that this distribution is the best choice among the implemented distributions to parallelize a graph using K^2 -trees.

7.4 Summary

In this chapter an experimental evaluation of the distributions designed in the previous chapter was presented. In general, the total cost of storing the distributed graph is near to the total cost of storing a graph as an individual K^2 -tree. Cyclic and Adaptive Cyclic distributions are the exception, because they break the compressibility of the graph and deteriorate the compression.

Edge balanced and Perfect Spatial balanced distributions achieve the best results in spatial efficiency. Cyclic distributions also obtain efficiency good results, but they need extra space. Grid distribution behavior depends on the graph distribution and the value of L .

In general, Grid (with $L = 8$), Cyclic and Latin Square distributions achieve the best temporal efficiency. However, when the speed up is analyzed, the Cyclic distribution is not a good solution. On the other hand, the Grid ($L = 8$) and Latin Square distributions achieve the best results.

To summarize, cyclic distribution achieves very good spatial and temporal efficiency but, when the speed-up and the total cost are analyzed, it does not obtain good results. The Grid distribution obtains good results in all measures. However, it is a parametrized distribution (with the value L) and the optimal value strongly depends on the graph distribution. As conclusion, the good results obtained for Latin Square distribution and the fact that this behavior is quite independent on the distribution, makes this distribution a good choice that balances as a commitment between the good spatial and temporal results.

Part III

Representation of RDF Graphs

Chapter 8

Representing RDF graphs with the K^2 -tree: K^2 -Triples

This chapter proposes K^2 -Triples, a new technique to store RDF datasets in a very compact way in main memory, providing at the same time efficient query algorithms. It is based on performing a vertical partitioning over the data, which is a very common strategy in the State of the Art about RDF stores. K^2 -triples splits an RDF dataset into $|P|$ binary relationships. Each binary relationship stores the relations between subjects and objects for a different predicate. Those binary relations are represented using the compact K^2 -tree data structure.

K^2 -Triples includes some additional indexes to reduce the main weakness of the vertical partitioning strategy: a poor efficiency in queries with unbounded predicate (that is, queries that do not specify a given predicate). A basic triple pattern resolution is provided and different join resolution strategies are implemented. The set of basic query patterns that K^2 -triples implements efficiently aims to set the basis for more complex queries, since an efficient SPARQL resolution strongly depends on the efficiency in these basic triple patterns.

The structure of this chapter is as follows. Section 8.1 gives a brief introduction to the RDF model and its standard query language SPARQL. Section 8.2 reviews the different RDF stores proposed in the State of the Art to manage RDF data. Next three Sections describe our proposal. Section 8.3 presents the internal storage of the K^2 -triples system. Section 8.4 describes the triple pattern resolution while Section 8.5 details the different join resolution strategies implemented for K^2 -triples. Finally, an exhaustive experimental evaluation is provided in Section 8.6.

8.1 RDF graphs

In this section we focus on describing the graphs used in a specific domain: RDF (*Resource Description Framework*). RDF models data in the form of triples which can be represented as a labeled graph, named RDF graph.

This section starts with a brief motivation that states the importance of RDF in the current conception of the Semantic Web. Next, the main features of the language representation (RDF) are described. This section ends with an overview of the SPARQL fundamentals, which have become the standard query language for RDF data.

8.1.1 The Web of Data

Nowadays, in the Big Data Era, big amounts of data are generated every day. They are usually published in a non-machine-readable way, because the web content is not well structured. Traditionally, information is mixed with style specifications for its visual representation, which is not suitable for an automatic processing. However, many of this information, which ranges from scientific data to more general knowledge like geographic information, is a very valuable source of information, which would have to be manually managed every day. Therefore, the automatic processing of this information will open many chances for its exploitation. In addition to the difficulties for an automatic processing, a new problem emerges when data sources from different providers have to be integrated. For instance, a publisher and a historic researcher can provide different information about the same city (a historic researcher provides events and a GIS publisher provides their location), which could be integrated. The detection of which information references the same entities in unstructured data is also an open research area.

The Web of Data appeared to deal with the management of this amount of data that is generated every day from very different information sources, supporting the principles of the Semantic Web [BLHL01]. It proposes a new publishing data format, which supports the connectivity between those heterogeneous data sources.

In this context, the Resource Description Framework (RDF) provides us with a common language to describe facts of the world in a structured and linked way. It enables data for its automatic processing and prepares it to be integrated in different data sources [MMM04]. The proposed model is quite simple: the data is structured as a set of triples (*subject, predicate, object*), which are identified with a Universal Resource Identifier (URI). The representation of the data by using URIs is fundamental in order to preserve the uniqueness of each data and, as a consequence, it makes possible to connect multiple data sources.

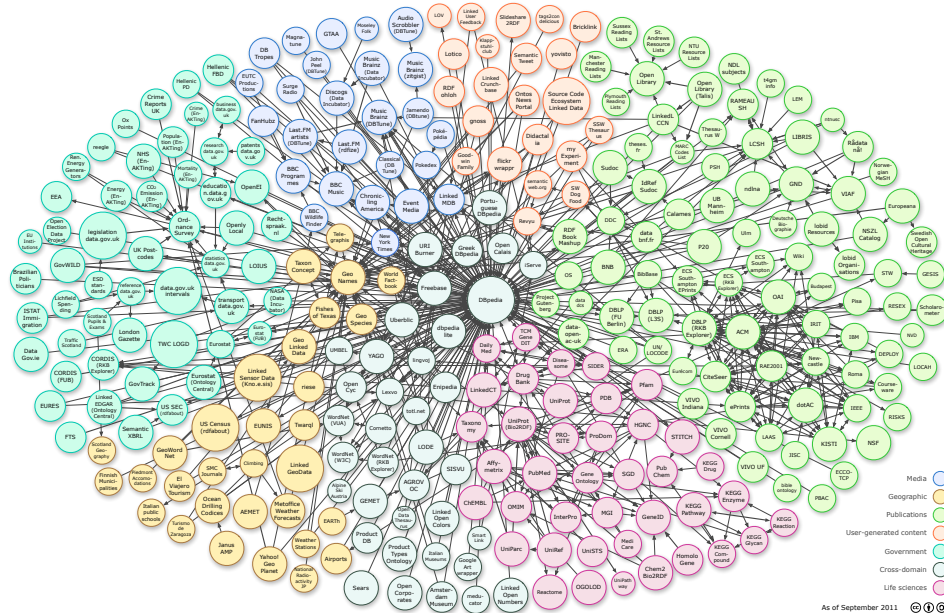


Figure 8.1: Web of Data in September of 2011

Internet, due to its own nature, is a context of distributed knowledge where multiple data sources publish data related to different aspects of shared entities (as in the given example of the cities). Although entities are represented in RDF as URIs, it is common to find the same entity identified with different URIs if it is used in different domains. For this reason, links that establish an identity relationship are needed. Those links can be also represented with RDF and they are called RDF links. They connect equal entities in different domains, creating a global data graph that integrates information from different sources [BHBL09].

RDF includes an important mechanism to describe new properties through the definition of new vocabularies. They can be defined using RDF syntax by the RDF vocabulary definition language RDF Schema (RDFS) or with the Web Ontology Language (OWL). Using those languages, each data provider can define its own vocabulary and, through RDF links, specify its equivalence with other vocabularies.

Linking Open Data Project has widely contributed to this integration of RDF data from multiple and heterogeneous data sources [CJ11]. Figure 8.2 shows the diagram of the different data sources and the relation between them, including data from media, geography, publications or even from government in 2011.

8.1.2 RDF as a data representation language

RDF is the standard language recommended by the W3C to describe and query semantic data. It provides a description framework that structures and links data as a set of *triples* (*subject, predicate, object*). A triple can be seen as a labeled edge in a graph, where the subject and the object are nodes and the predicate is the labeled edge that connects them. In this way, the set of triples that compose a dataset is called *RDF Graph*. The RDF graph describes resources (subjects of the triples) through properties (predicates or edges of the graph) set to a given value (objects of the triples). RDF uses Uniform Resource Identifiers to represent each resource, property and value. Objects (the values that a given resource takes for a given property) can also be literals. URIs can take part of different triples playing different roles: a URI that is the object of a triple can be described by other triples at the same time (acting in this case as subject).

Figure 8.2 (top) shows an example of an RDF graph that includes some of the information related to Madrid contained in the DBpedia dataset [LIJ⁺14], which is composed by structured information from Wikipedia. The graph contains three nodes representing *Madrid*, *Spain* and *Euro*; their URIs and one literal value (3265038). The entity *Madrid* is described by the property *country* with value Spain, forming a RDF triple. At the same time, *Spain* plays the role of subject in two triples: one describing the official currency in *Spain* (*Euro*) and another one showing the capital of *Spain* (*Madrid*). Finally, one triple is used to describe the total population of Madrid showing more than 3 millions of people.

We already explained how RDF datasets can be represented as a graph. However, another representation is possible. Figure 8.2 (in the middle) shows the same RDF graph in N-triples, a simple format where each triple is described in an individual line by its subject, predicate and object, represented by its URIs or literal values and followed by a *dot* at the end of the line. W3C recommends an RDF/XML syntax to encode the RDF graphs, which is shown in the bottom of the Figure.

The description of data in RDF is supported by the **RDF Schema** [BG04], used to define new concepts forming a vocabulary. The vocabulary is described using also RDF syntax. RDF Schema is a semantic extension of RDF that provides a set of classes and properties used to create new classes and properties. A fully review of the mechanisms of extension that RDF Schema provides is out of the scope of this introduction. However, some examples of the elements that it defines are given to illustrate the underlying idea under RDFS:

- **Classes** Resources are grouped in classes, which are also a special kind of resources. The belonging of a resource to a given class can be expressed with an RDF triple by using the property *rdf:type*. Continuing with the example of the Figure 8.2, the next triple specifies that *Madrid* belongs to the class *Place*:

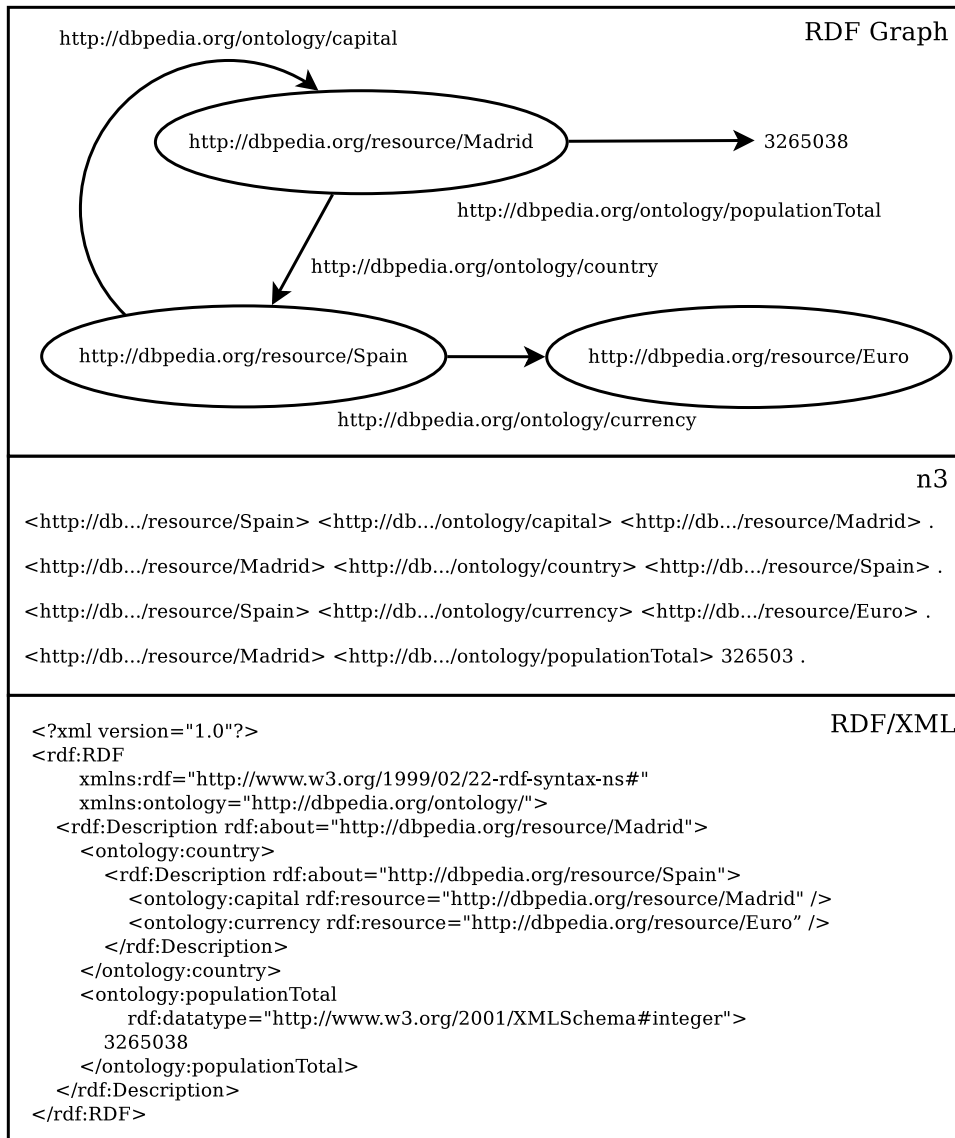


Figure 8.2: Web of Data in September of 2011

```
http://dbpedia.org/Page/Madrid rdf:type http://dbpedia.org/ontology/Place .
```

Resources that are classes, like `http://dbpedia.org/ontology/Place` are instances of `rdfs:Class`. It will be written with RDF syntax as:

```
http://dbpedia.org/ontology/Place rdf:type rdfs:Class
```

RDF Schema supports class inheritance, in the way that for an element which is an instance of a class C , having C a subclass of C' , then that element is also an instance of C' . The property `rdfs:subClassOf` is used to describe the subclass relationship between classes.

- **Properties** Properties are used to describe the subjects of the dataset (they are the predicates of the RDF triples). RDF Schema defines some properties with special characteristics. RDF Schema supports property inheritance, which can be specified through the special property `rdfs:subPropertyOf`. With the triple $P \text{ rdfs:subPropertyOf } P'$ we define that the property P is a subproperty of P' , meaning that for every triple (SPO) , $SP'O$ can be inferred. That is, two elements which are related to the property P will also be related through the property P' . Another special property included in RDF Schema is `rdfs:range`. It is used to describe the classes of the values of a property. That is, given the triple $P \text{ rdfs:range } C$, we can infer that every object value O for the property P (that is, every O for which a triple SPO exists), is an instance of the class C .

Through these simple mechanisms, new properties and new classes can be inferred, although they would not be explicitly represented in the RDF datasets. The ontology languages, like RDF Schema and *OWL* are the basis of the inference engines. They provide a rule definition mechanism to generate new knowledge which is not explicit in the RDF dataset.

8.1.3 SPARQL as a query language

SPARQL [PS08] is the W3C recommendation for querying RDF. It defines the queries by graph pattern matching, specifying the restrictions that the resulting RDF subgraphs have to accomplish. In this section we give a brief review of the most relevant aspects of SPARQL for the work developed in this thesis.

The most basic query we can define in SPARQL, that we called *Basic Graph Pattern*, is composed by a set of triple patterns. This set of triples is specified in Turtle format [BBLPC14], an extension of N-triples which additionally includes abbreviation techniques (like its support to factor common prefixes). Each element of each triple can be fixed (or **bounded**) to a value (URI or literal) or it can be variable (**unbounded**), which is marked with the special symbol `?` followed by a name. That means that eight different triple patterns are possible in SPARQL:

- (S, P, O) asks for the existence of a specific triple.
- $(S, P, ?)$ retrieves the objects that describe a subject S through the predicate P .
- $(S, ?, ?)$ retrieves all the triple patterns containing the given subject.
- $(S, ?, O)$ asks for the predicates that relate a given subject S to the object O .
- $(?, P, O)$ asks for subjects that are related to the predicate P and the object O .
- $(?, ?, O)$ searches all the triple patterns where the object O participates.
- $(?, P, ?)$ retrieves all the triples of a given predicate.
- $(?, ?, ?)$ obtains all the triples of the dataset.

Next example defines a simple SPARQL query composed by a Basic Graph Pattern of two triples:

```
SELECT ?place
WHERE {
  ?place http://dbpedia.org/ontology/country ?country .
  ?country http://dbpedia.org/ontology/currency http://dbpedia.org/resource/Euro .
}
```

This basic graph pattern (located after the special word *WHERE*) is composed by two triples: the predicate of the first triple is fixed, but no specific values are given for the subject and the object. On the other hand, the second triple fixes the predicate and the object, but the subject is variable (*country*), which at the same time played the role of object in the first triple. These two triples compose a pattern that the results of the query will have to accomplish. The special word *SELECT* defines how the results will be output. In this case, only the values that each result takes for the *place* variable are returned. Therefore, we can see that the previous query searches for the places that are located in a country whose official currency is the Euro. The query will be computed in the RDF dataset by pattern-matching, returning all the values that satisfied the conditions.

Basic Graph Patterns are the most simple queries that SPARQL defines, but another graph patterns are contemplated in the standard.

A more complex example is the Optional Graph Pattern, which allows us to specify optional conditions about new variables. If an optional condition is satisfied, the variables that compose that optional pattern are bounded and returned as a result. However, even if a result candidate does not satisfy an optional condition, this graph could still be a valid result (with the variables of the optional pattern

unbounded). The behaviour of an optional graph pattern can be observed in this example:

```
SELECT ?place ?population
WHERE {
  ?place http://dbpedia.org/ontology/country ?country .
  ?country http://dbpedia.org/ontology/currency http://dbpedia.org/resource/Euro .
  OPTIONAL ?place http://dbpedia.org/ontology/population ?population .
}
```

In this query, an optional pattern about the population of a place is specified. Consider a place (for instance, *Madrid*) which contains its population in the RDF dataset (that is, a triple `http://dbpedia.org/resource/Madrid http://dbpedia.org/ontology/population 3265038` exists). Then, a result for that query is `http://dbpedia.org/resource/Madrid, 3265038`. We now consider another place, like *Vigo*, whose population is not specified in our current RDF dataset. Then, since this pattern is optional, *Vigo* will be a result of that query, but with an unbounded population.

We defined the basic graph patterns, as a set of patterns that every resulting graph has to satisfy. SPARQL queries can also be composed by a combination of different graph patterns, through the use of different clauses. Clause *UNION* specifies that the final resulting graphs have to match with at least one of the alternative graph patterns. *FILTER NOT EXIST* is another interesting clause included in the SPARQL standard, used to specify conditions that resulting graphs must not match.

SPARQL standard also includes aggregation functions (like *COUNT*, *SUM* or *MIN*), grouping the results through the *GROUPBY* clause. Finally, SPARQL also supports complex restrictions over specific variables (like regular expressions) through the *FILTER* clause.

In conclusion, SPARQL language is a powerful language based on triple patterns, which can be joined to form more complex queries. As a consequence, the design of a SPARQL engine has necessarily to be built on top of an efficient triple pattern resolution and join implementation. Another aspects of the query resolution are query planners (which optimize the order of execution in a query) and additional features like the optional or union clauses. They are higher layers of SPARQL engines built over the basic core engine composed by a triple pattern and join resolution.

8.2 RDF stores

Given the increasing importance that RDF as data model and SPARQL as its query language is having in the Web of Data world, many specific RDF stores were proposed. Some of them adapt solutions from other fields of research, as is the case of the RDF engines based on relational solutions. Many other stores were natively designed to store RDF structured data and provide SPARQL language support. This section explains the two different philosophies followed in the State of the Art in RDF management and it describes the most successful RDF stores. Therefore, the descriptions of each RDF store are focused on their internal representation.

8.2.1 Relational Solutions

This section describes different proposals to store RDF data using a traditional relational scheme. Each method proposes different partitions of data, where the RDF triples are stored in one or many relational tables. Depending on this data distribution, querying will involve different operations over the relational scheme.

8.2.1.1 Single three-column table

This approach is the most straightforward way of storing RDF data over a relational physical storage. RDF triples are represented using a single table with as many rows as the number of RDF triples that the dataset contains. Subjects, predicates and objects are represented in three different columns of this table and each row stores the values of one triple. The main weakness of this solution is that the resolution of Basic Graph Patterns requires multiple joins over this huge table.

3-store [HG03] is one of the systems that implements this model. It uses MySQL as database engine and it defines a single table where the RDF triples are represented through their identifiers. An additional hash table maps the identifiers to its corresponding literal or URI value. Its query interface supports RDQL, a SQL-like query language for RDF. This system delegates the problem of designing a good query plan to the relational database, since it is the final physical implementation. With that purpose, it transforms the RDQL query into its equivalent SQL query. This step is quite straightforward due to the similarities between RDQL and SQL languages.

8.2.1.1.1 Virtuoso ¹ is a more evolved and modern RDF store that also implements this model. It includes support to load data in different formats (N3, Turtle and RDF/XML). Datasets can be queried using the standard SPARQL query language.

8.2.1.2 Property tables

Storing a RDF dataset in a huge three-column table presents the problem of the intensive usage of self-joins. This problem appears even in simple queries like patterns asking for a subject general description. *Property tables* groups the properties describing similar subjects in the same table, creating a kind of relational-like property tables for representing RDF data. In each table, a list of similar subjects are represented, where *similar subjects* means that they are from the same nature and they share most of the predicates. Each property table contains a column representing the subjects and several columns representing all the predicates that are related to those subjects, one for each predicate. Each row represents a subject, including the value (if known) it takes for each predicate represented in this property table. When subject is not described with some predicate, a NULL value is set in the corresponding column. In that way, each row represents as many RDF triples as columns containing not NULL values.

The main advantage of this model when compared to the single-three column table is that it is specially convenient for queries focused on describing resources with all their properties. However, the cost of the general query resolution remains high. An important issue of property tables is the management of NULL values. It affects the spatial storage cost, because no existing triples have to be explicitly stored. This problem is increased when the dataset is not well structured. Another important drawback is the management of multi-valued attributes, which are very common in semantic datasets. A multi-valued attribute entails the existence of several triples in the dataset related to the same subject and the same predicate. A simple example of multi-valued attribute could be two triples representing the different telephone numbers that a given user has. Finally, Property tables behaviour strongly depends on a good clustering algorithm which groups the subjects in property tables minimizing the NULL values. Clustering algorithm also needs to group the properties usually queried together in order to minimize the number of joins needed to solve the query.

Jena is a property table implementation proposed for the Semantic Web Framework [WW06]. Jena defines a property table schema over a relational database. It addresses the problem of the multi-valued attributes by storing them in special

¹<http://www.openlinksw.com>

property tables. Predicates that are specified by the user as multi-valued, are stored in two-column tables (representing the subject and the object). In this special kind of tables, as opposition to the property tables, each subject can appear in several rows, one for each different value that the subject takes for that predicate. Jena does not propose a clustering algorithm, which has to be defined by the user.

Sesame [BKVH02] implements property tables over the DBMS PostgreSQL, taking advantage of its power of expression, specially of its inheritance table support. Sesame is specially suitable for well structured datasets which includes metadata defined in RDF Schema. Sesame uses that RDF Schema to structure data in property tables. In that way, each class defined in the RDF Schema defines a new property table in the relational physical storage. A Subclass in the schema would be represented with new property tables which would be sub-tables of the table representing the parent class. Finally, each subject instance of a given class is stored in the property table that represents this class.

8.2.1.3 Vertical partitioning

The vertical partitioning scheme was proposed as an alternative to the property tables model [AMMH07], which maintains a similar philosophy (tables describing subjects). It aims to solve the main drawbacks of the property tables, like the NULL values and the multi-valued attributes. Vertical partitioning stores each predicate in a different table. Each row of a predicate table represents a pair *subject – object* related through that predicate.

Vertical partitioning model solves several of the main weakness identified for property tables. First of all, no NULL values are stored. Since each table only includes two columns (subject and object), and only the existing pairs for the corresponding predicate are stored as a row in the table, the subjects no related to a given predicate are not stored in that table. Vertical partitioning also addresses the multi-valued attributes issue. A multi-valued attribute is represented in this model by as many rows as different values the subject contains for a given predicate. Another important advantage when compared to the property table model is its simple design. Vertical partitioning does not depend on a clustering algorithm to design an efficient storage for each dataset.

Predicate tables in vertical partitioning are ordered by the subject column, to provide fast access to specific subjects. Although many joins are required in this approach to answer queries involving different predicates, fast merge joins can be used thanks to the subject column ordering.

Vertical partitioning suffers from an important lack of efficiency to solve queries with unbounded predicate. An example of that kind of queries are the ones in which

some entities (subjects) are asked for a general description. In this case, all tables must be queried, searching which properties describe each subject. Then, partial results must be merged to obtain the final result. The problem is increased when the number of different predicates of the dataset is high. Therefore, vertical partitioning is not a good choice for representing datasets with many predicates. Nevertheless, this lack of efficiency can be partially solved by using additional indexes.

Physical implementation of this data model fits better with the column-oriented databases principles, due to some of its characteristic features, like optimized merge joins or specific data compression by attribute. Consequently, column-oriented databases obtain better querying performance implementing vertical partitioning models than the proposals that are stored over traditional row-oriented databases. Results show a difference of one order of magnitude between the two alternatives [AMMH07].

SW-store [AMMH09] is one of the most representative RDF stores implementing a vertical partitioning model. It is physically represented over a column-oriented database, *C-store*, which stores each column of each table independently. SW-store uses a dictionary to encode the URIs and literals. Integer identifiers are stored in the columns of the tables. After performing the query, the resulting keys are decoded using index joins over the dictionary table. Since tables are indexed by subjects, SW-store has efficiency problems to perform queries that require subject-object joins, intensively used in path expressions. The solution that SW-store proposes is to materialize the most common path expressions, pre-calculating the subject-object joins. As a result, this materialization increments the cost of updating. SW-store also provides a single column alternative which can be more convenient for some predicate distributions. This approach uses one single-column table in which each row represents the value of a subject to this predicate. Values are ordered by subject identifier. The subjects for which this property is not defined are represented by NULL values. SW-store proposes different approaches to *compress* those NULL values depending on their density and distribution. A first approach consists in representing ranges of consecutive identifiers with NULL values. An alternative implementation stores bitmaps where i position takes value *one* if identifier i is described by this predicate (that is, it is not a NULL value). Figure 8.3 shows an example of dataset stored in SW-store. Two column tables are represented on the left. For instance, *capital* predicates contains two rows, representing identifier 1 (*Spain*) takes value *Madrid* and identifier 2 (*Italy*) takes the value *Rome*. Same predicate can be represented as a one column table (bottom of the Figure). No NULL values (*Madrid* and *Rome*) are represented ordered by subject. An additional bitmap represents which subject is represented (position 1 and 2 of the bitmap have a *one*, so identifier 1 and identifier 2 are represented in that table).

Another store was implemented in the context of the vertical partitioning philosophy. For instance, an in-memory storage based in MonetDB is proposed. It

Two-column storage

Capital	Currency
ID1 Madrid	ID1 Euro
ID2 Rome	ID2 Rome
Country	Population
ID3 Spain	ID3 3265038
ID4 Italy	ID4 2645907

One-column storage with compression of NULLs

Capital	Currency
Madrid	Euro
Rome	Rome
1100	1100
Country	Population
Spain	3265038
Italy	2645907
0011	0011

Figure 8.3: Vertical partitioning proposed by SW-store engine (top) and its alternative implementation in single column (bottom)

reduces the need of *merge joins* [SGK⁺08].

8.2.2 Native Solutions

Many stores are specifically designed to store and query RDF data addressing its own peculiarities. Some works [BHS03, HG04, AG05] proposed different graph-based models, although the most successful approaches are based in multi-indexes. This section briefly reviews some of the most relevant stores whose internal storage was designed to deal with the nature of RDF data.

Hexastore [WKB08] is an in-memory solution which stores RDF data using a set of six indexes. It is designed following the vertical partitioning model, which implicitly supports indexed access to the predicate.

In general, vertical partitioning is convenient to solve queries where the predicate is bounded (that is, a predicate is specified in the query). Furthermore, vertical partitioning solutions usually have each property table indexed by subject (however, searching by object becomes inefficient). To sum up, the main problem of the vertical partitioning comes from not treating the three dimensions in the same way. It prioritizes the access of the RDF triple in a given order (*predicate, subject, object*). Many SPARQL queries need a different access to the data, resulting in inefficient implementations over a vertical partitioning approach.

Hexastore aims to provide an equitable access to all the possible ways of asking for a triple, including any combination of bounded-unbounded variables in a triple pattern $((s, p, o), (? , p, ?), (? , p, o), \dots)$. For that purpose, Hexastore builds six indices that include all the possible access orderings: SPO, SOP, PSO, POS, OSP and OPS. The six-index structure is a conceptual evolution of the original proposal of Harth and Decker [HD05], but it goes one step beyond by providing efficient complex query resolution. Figure 8.4 represents this schema. Index *SPO* includes a first vector with all the subjects of the dataset. Each subject S_i has an associated vector that includes all the predicates describing this subject in the dataset, named $P_j(S_i)$. Finally, each one of those predicates contains object lists that are the value of the property P_j for the subject S_j , named $O_k(S_i P_j)$. Remaining five indexes are built in a similar way. Note that the object lists of *SPO*, located in the third level of the Figure 8.4, can be shared between the index SPO and the index PSO (the same happens for the pairs of indexes *SOP – OSP* and *POS – OPS*).

The main advantages of this approach are an efficient treatment of multi-attributed values (every object list of the third level containing more than one element is a multi-attributed value) and an efficient implementation of the object-subject joins needed to perform the path expressions queries (through merge joins, since all lists of the indexes are sorted). In this way, Hexastore overcomes one of the weakness of the vertical partitioning model. The main problem of Hexastore is its

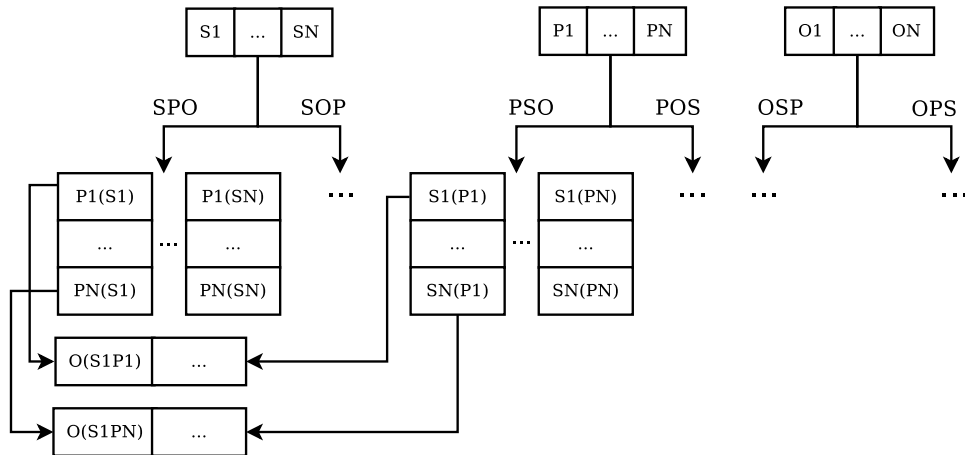


Figure 8.4: Internal representation of the six indices in Hexastore. List of objects shared by SPO and PSO indexes are shown

scalability, because of its high spatial requirements (5 times the space used to index the full dataset in a single three-column table), combined with the fact that it is stored in main-memory.

RDF-3X is a disk-based structure that continues with the philosophy of storing 6 indexes, including all the possible orderings of an RDF triple. It is focused on obtaining competitive query times while reducing the spatial needs for the index structures. The physical storage of an RDF dataset in the RDF-3X system is composed of three main components.

First of all, the triples are stored in a single big *triples table*. The implementation of this table is specifically designed for RDF-3X. It does not delegate its physical storage to any RDBMS (unlike the relational solutions). That triples are lexicographically sorted and encoded using a mapping dictionary.

A compressed *B+*-tree stores this ordered set of triples represented with identifiers. Over the triples table, the six possible orderings of the elements of the triple are represented with individual indexes, just as in Hexastore. But in this case, the indexes are compressed in a way that its total size does not exceed the size of the original data. Each index is individually compressed.

The construction of the six indexes will be illustrated through the *OPS* index, the other five indexes are built in the same way. Firstly, triples are ordered lexicographically according to *OPS* ordering and they are directly stored in the leafs of a *B+*-tree. Near triples according to this order have many chances to share the values for the object and the predicate (and even more chances to share at least

the object), so the triples are represented using a differential compression regarding to the previous element using a byte-level compression scheme, where differential values are represented with the minimum needed number of bytes. This byte-wise compression ensures a fast decompression for the querying process.

Additionally, *aggregated index* includes the number of triples that contains each one of the possible pairs of elements in an RDF triple (that is, SP, PS, SO, OS, PO, OP), which are also stored in $B+$ -trees. They provide fast access for queries that do not include all the elements in the variable selection. For instance, the aggregate index PS could be used to solve the query `SELECT ?S ?P WHERE { ?S ?P ?O}`. Object specific values are not returned as a result. However, each pair (*subject, object*) has to appear in the result as many times as triples in the datasource uses them. Therefore, the aggregated index PS can be used to count that number of occurrences. Another three *aggregated indexes* are included to count the occurrences of each subject, object and predicate in the dataset, for the same reason.

RDF-3X query processor relies on the intensive usage of merge joins over the sorted indexes. It includes a query optimizer mostly focused on join ordering for the generation of execution plans. It obtains very efficient results, outperforming the described SW-Store. The spatial requirements, although clearly overcoming other multi-index solutions like Hexastore, remain very high. This problem is increased for big datasets because large amounts of data need to be transferred from disk to memory, reducing the querying efficiency [SHK⁺08, ?].

Bitmat [ACZH10] establishes another alternative based on compressed indexes. It starts from modelling an RDF dataset as a three dimensional cube, where the dimensions represent subjects, predicates and objects. Each triple is conceptually represented in this cube by a 1 in its corresponding position. Remaining positions are represented by a 0. This 3D bit cube can be sliced in three different ways, one per dimension. For instance, if the cube is sliced along the dimension of the predicate, as many adjacency matrices as predicates are generated. Each adjacency matrix stores the binary relation between subjects and objects through the predicate it represents. In that way, BitMat considers the adjacency matrices for the pairs SO (through each predicate), the matrices PO (through each subject) and the matrices PS (one per object). Additionally, the inverted matrices OS are also considered due to its frequency of usage in the typical queries. BitMat stores all the adjacency matrices generated by the 4 slicing process (SO, PO, PS, OS) in a compressed way taking advantage of the sparsity of that matrices. Each individual matrix is compressed following a row-wise traversal, encoding the alternating run lengths of bits 0 and 1. Figure 8.5 shows an example of RDF dataset stored in BitMat. The top of the Figure describes the 3D Cube over its three axis (predicate, subject, object). The represented dataset is described with three predicates (*country, capital, population*) that defines three slices SO . That is, three adjacency matrix have to be compressed for that partitioning, represented in the bottom of the Figure. The same process

would be repeated for *OS*, *PO* and *PS* (its corresponding adjacency matrix are omitted in the Figure in favour of the simplicity). Figure also shows the run-lengths that finally will be stored representing that adjacency matrix. For instance, the predicate *country* is stored like the sequence [0]-1 1 8 1 1, where [0] means that the first run length is a zero, and the subsequent values represent the number of consecutive equal elements that the matrix contains, alternating zeros and ones. For instance, the third value 8 corresponds with the 8 zeros from the third position of the first row to the second position of the third row. Common operations like joins and filters are directly defined in terms of the compressed bit-arrays without needing to decompress the data, obtaining bit-arrays for the basic triple patterns and performing operations like unions or intersections in order to obtain the final result. This structure resides in main-memory and it only overcomes the State of the Art for low selectivity queries.

Many other full in-memory stores [JK05] emerged in this scenario, motivated by the fact that, if the entire dataset fits in the main memory, the queries can be performed orders of magnitude faster. Just to mention another example, **SpiderStore** is an in-memory RDF store that defines a pointer structure to store the RDF triples. Each subject, object and predicate is represented by an individual node in SpiderStore, containing information about all the edges in which it participates (whether playing the subject, predicate or object role). The information of each triple inside a node is composed by two pointers to the nodes representing the other members of the triple. This pointer representation is used to traverse the RDF graph in any direction (starting by subjects but also by objects or even predicates), specially convenient to solve path queries in a depth-first manner. SpiderStore determines the execution order depending on the selectivity of the nodes that participate in the query. This selectivity information is extracted from the amount of pointers that the node contains.

The main problem of full main-memory approaches is their lack of scalability, specially when data structures are not compressed, resulting in a high demand of memory. Because of that, the current results are limited to small or medium-size datasets. Some hybrid memory/disk stores were proposed in order to avoid this limitation, addressing the problem from different perspectives. G-SPARQL [SEH12] engine stores the entire graph in a relational database, maintaining an in-memory graph topology to speed up the traversal over the graph. On the other hand, Bypher [VMR⁺11] is another interesting store that proposed two different physical storages (one purely in-memory and another one disk-based), building a query engine that works over any of two physical representations.

New opportunities arise for in-memory stores thanks to the advances in distributed computing. This class of solutions, recently studied [HAR11b, UMB10] on the Map Reduce framework, allows arbitrarily large RDF data to be handled in main memory because more nodes can be added to a cluster when they are needed. The design

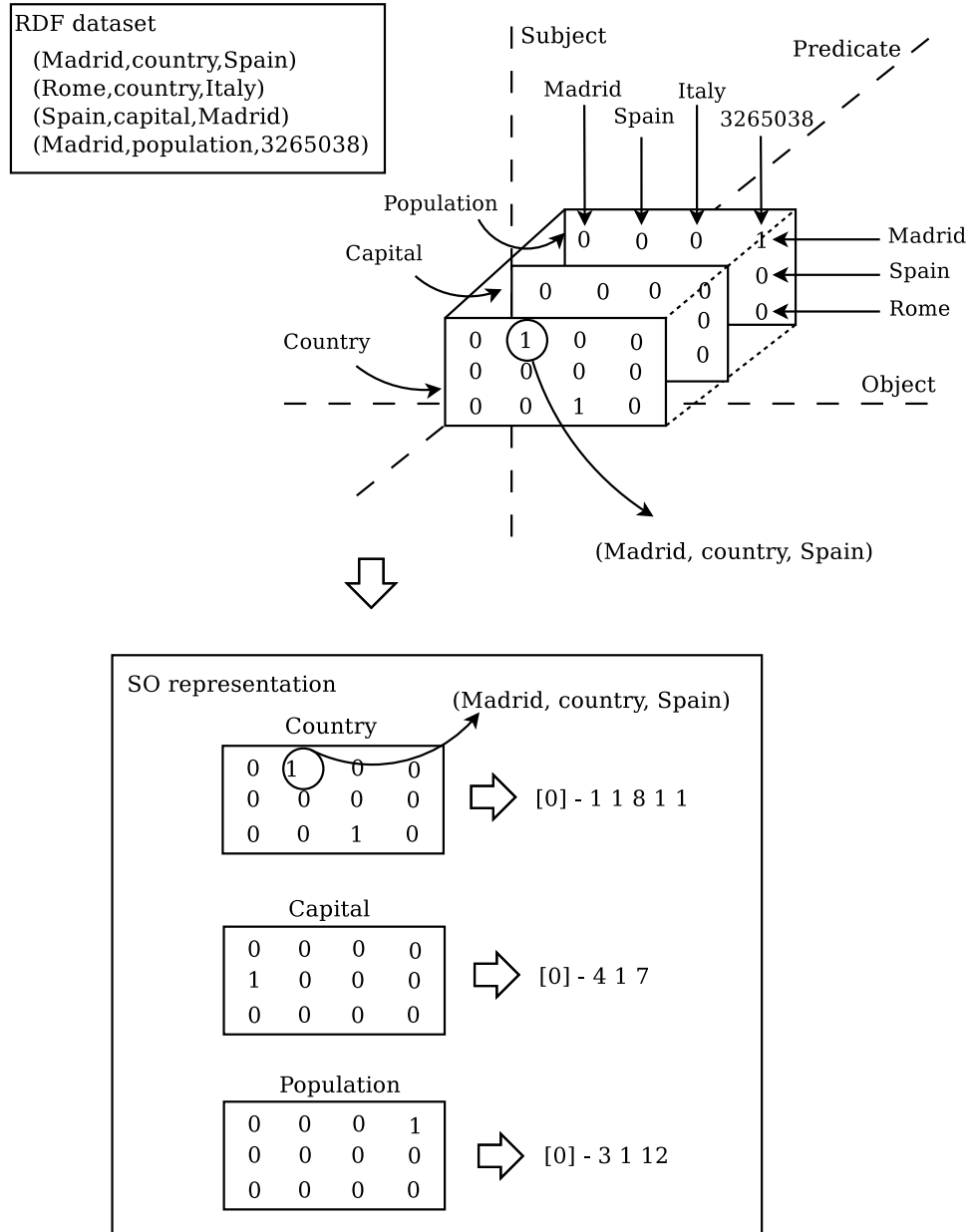


Figure 8.5: 3D cube model proposed in BitMat (top) and an extract of its physical representation with run-lengths (right-bottom)

of efficient RDF exchanging tools [FMPG⁺13, Bin11] is another important research line that complements those distributed RDF stores.

8.3 K^2 -triples structure

This section describes the internal representation of our proposal, named K^2 -Triples. This system performs a vertical partitioning over the RDF data and it relies on the massive usage of the K^2 -tree structure to represent the resulting binary relations. K^2 -triples manages an RDF dataset as a set of triples of numeric identifiers (id_1, id_2, id_3). With that purpose, the first step in order to manage an RDF dataset by using the K^2 -triples system consists in performing a *dictionary encoding* that converts each triple of the dataset composed by URIs and *literals* to a triple composed of three identifiers where id_1 represents the subject of the triple, id_2 identifies the predicate and id_3 corresponds to the object. This encoding simplifies the vertical partitioning represented through multiple K^2 -tree because, in this way, we can identify each row and each column of the adjacency matrix with a subject or an object of the dataset. The management of the dictionary (that maps the identifier to the corresponding URI or literal) is out of the K^2 -triples system purposes. Several State of the Art purposes specifically designed to manage RDF dictionaries could be used [MPFC12].

In Section 8.3.1 we introduce the Dictionary Encoding technique, that processes the dataset in order to make data suitable of being stored in K^2 -triples. Section 8.3.2 describes how vertical partitioning will be applied over the data. Finally, additional indexes that improve the performance of queries with unbounded predicates are presented in Section 8.3.3.

8.3.1 Dictionary encoding

Dictionary encoding consists in mapping the URI identifiers and the *literals* that compose each triple to correlative numeric identifiers. This encoding starts with extracting the different terms that appeared in the dataset, assigning a different numeric value for each element. Then each triple is transformed using its corresponding numeric identifiers. The main advantage of this simple technique is that it allows to save space, because each different long term (as URIs usually are) is stored only once in the dictionary, and it is referenced through its numeric identifier in the several triples it belongs. Furthermore, in the particular case of the K^2 -tree, this encoding technique will be very useful to perform the assignation of rows and columns with the different elements (just as the original K^2 -tree does to store web graphs).

In order to encode the RDF dataset preparing it to the K^2 -triples system, we

propose a dictionary divided in four categories, where for each category, terms are ordered in lexicographic order [MPFC12, ACZH10]:

- **Common subjects and objects** (SO) includes the terms that play both subject and object roles in the dataset. They are mapped to the range $[1, |SO|]$
- **Subjects** (S) includes the remaining subjects that are not included in the **SO** category, because they do not play the role of object in any triple of the dataset. They range from $[|SO| + 1, |SO| + |S|]$.
- **Objects** (O) contains the objects that are not subjects in any triple, so they do not belong to the **SO** category. They are mapped to consecutive identifiers from $|SO| + 1$ to $|SO| + |O|$.
- **Predicates** (P) includes the elements that represent predicates of the dataset. They have numeric identifiers in the range $[1, |P|]$.

Note that each different term (literal or URI) included in the RDF dataset can belong to one or two categories depending on the different roles it plays in the triples. For instance, an element that is the subject in a triple and the predicate in another triple is stored in both **Subjects** and **Predicates**. However, if a term is subject and object in different triples, it is only stored once (because it is stored in **Common subjects and objects**). Therefore, the redundant information is minimal ($|P|$ elements at most, that is expected to be small because predicates is usually a dimension with few elements). In exchange for a minimal redundancy, this classification in four dictionary categories will be very convenient to perform the vertical partitioning of the data used in the K^2 -triple system. First of all, it can improve the performance of the subject-object joins because all the possible results are mapped, by definition, in the range of identifiers $[1, |SO|]$. Consequently, the join resolution can be concentrated on the area of SO in order to avoid querying the remaining subjects and objects.

Figure 8.6 shows an example of dictionary encoding for a dataset of 10 triples. The full URI has been replaced by a shorter term in favor of the clarity of the example. Four dictionaries are built with the different elements of the dataset. **SO** dictionary contains 4 elements in the interval of identifiers $[1, 4]$, so **S** and **O** dictionaries start from the identifier 5. Finally, P follows a different numeration starting from 1 to 5, since the dataset is composed by 5 different predicates. The final dataset that will be stored in the K^2 -triples system is shown on the right of the Figure. For instance, the triple $(5, 2, 1)$ is representing the triple $(Carcassone, Country, France)$, because the subject 6 is located in the S category, 2 represents the predicate $Country$ and the object 1 is located in the SO category corresponding to $France$ (it is a term that appears as subject and as object in the dataset).

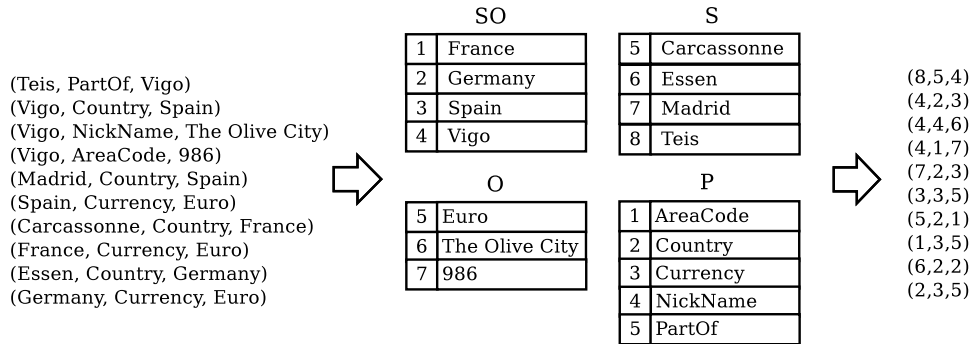


Figure 8.6: Example of the simple dictionary encoding for a RDF dataset

The management of the dictionary and how the mapping between terms and identifiers is implemented is out of the purpose of K^2 -triples system. Any technique within the State of the Art could be used as an independent module of the RDF system. However, it is important to note that RDF dictionaries usually take up to 3 times more space than the triples structure [MPFC12] so the usage of compressed and efficient dictionary indexes is crucial to the global performance of a RDF store. From this point, only the triples represented with their numeric identifiers will be considered.

8.3.2 Vertical partitioning

K^2 -triples follows a *vertical partitioning* strategy to model the RDF datasets. The set of triples (s, p, o) is divided in $|P|$ sets, one for each predicate. In this way, for each predicate, a binary relation composed by the pairs (s, o) related through this predicate can be represented. Therefore, each triple (s, p, o) of the original dataset belongs to the binary relation induced by its p value. The RDF dataset is fully represented by storing the $|P|$ binary relations.

Each binary relation associated to a predicate value, which is composed by a set of pairs (s, o) can be represented through an adjacency matrix where the rows in the interval $[1, |SO| + |S|]$ represent the subjects of the dataset and the columns included in $[1, |SO| + |O|]$ represent the objects. Each adjacency matrix associated to a predicate value is then stored through a single K^2 -tree in a very compact way.

At this point, note that the lexicographic order of the dictionaries implies that consecutive terms in the dictionary have consecutive rows or columns in the adjacency matrix, so the compression of the K^2 -tree strongly depends on the correlation between the lexicographic order of the terms and its relation distribution for a given predicate.

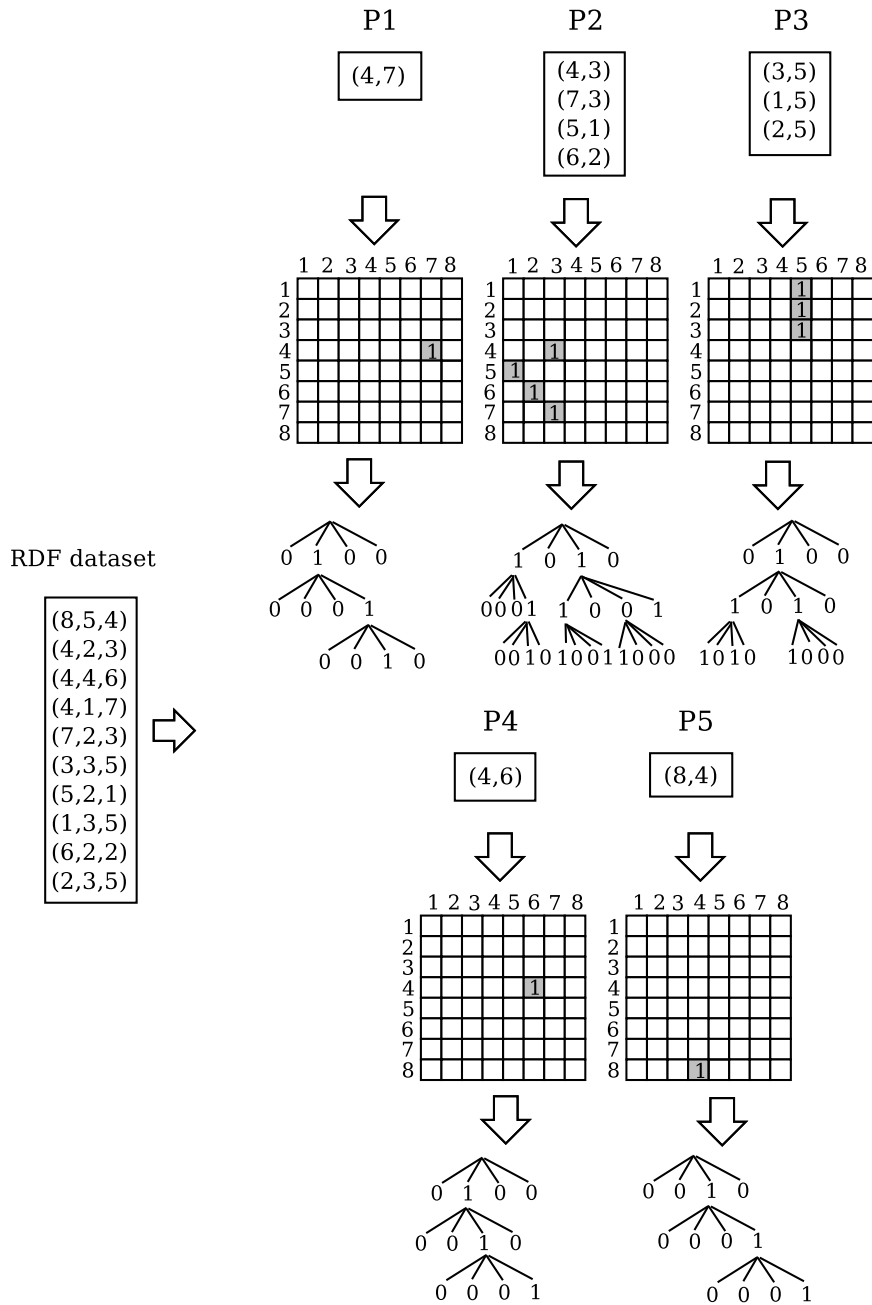


Figure 8.7: Internal representation of K^2 -triples system using K^2 -trees

Figure 8.7 shows an example of a dataset represented in K^2 -triples. In the top-right of the Figure, the triples are partitioned in 5 sets, one for each different predicate value, inducing 5 binary relations. For instance, the binary relation of the predicate 1 is composed by the pair (4, 7), meaning that triple (4, 1, 7) belongs to the dataset. Each binary relation can be represented as a 8×8 adjacency matrix (shown in the middle-right of the Figure), where the rows represent the subjects and the columns represent the objects. For instance, adjacency matrix of predicate 4 contains a 1 in the cell (4, 6) meaning the triple (4, 4, 6) is in the dataset. The individual K^2 -tree built for each adjacency matrix is shown in the bottom of the Figure. Those five K^2 -trees compose the internal representation of the dataset in K^2 -triples.

8.3.3 Indexing predicates

It is well known that the main weakness of an RDF store that partitions the data by predicate is the performance of queries with the predicate unbounded. This kind of queries usually involves, in the vertical partitioning approaches, checking all the individual structures stored for each predicate. In the case of K^2 -triples, a query with unbounded predicate needs all the K^2 -trees to be traversed, reducing meaningfully the efficiency of the query when the number of different predicates is large. In this section, we propose additional indexes to improve the efficiency of that queries, that reduce the search space by limiting the number of K^2 -trees that have to be checked depending on the information provided in the query (the subject or the object that the query specifies, or even both).

Consider the simple triple patterns with unbounded predicates: $(?S, ?P, ?O)$, $(?S, ?P, O)$, $(S, ?P, ?O)$, $(S, ?P, O)$. The first pattern, $(?S, ?P, ?O)$, asks for all the triples of the dataset, so the traversal over all the K^2 -tree is compulsory in order to return the results. However, in the pattern $(?S, ?P, O)$ only the predicates related to a given object can produce results, so the other K^2 -trees could be avoided. The same happens with the pattern $(S, ?P, ?O)$, where the given subject can be used to limit the searched K^2 -trees. This optimization is carried out by the construction of two additional indexes:

- **SP(subject-predicate) index** contains the list of predicates related to each subject of the dataset.
- **OP(object-predicate) index** contains the list of predicates related to each object of the dataset.

Empirical results [FMPG⁺13] show that the average size of these lists of predicates for each different subject and object are an order of magnitude less than the number

of total predicates used in real-world datasets. This fact is crucial to obtain good results by using that indexes due to several reasons. Firstly, by using those indexes, the final number of the traversed K^2 -trees will be meaningfully smaller, improving the efficiency of the unbounded predicate queries. Furthermore, this fact also implies that those index **SP** and **OP** can be stored in a limited size. In that way, K^2 -triples can include those indexes that improve the querying efficiency while the structure continues being very compressed.

We propose a compact representation of that indexes. Note that both **SP** and **OP** indexes are representing a *predicate list* for each subject and each object. For instance, in the case of the **SP** index, many subjects will have the same predicate list (the same happens with the predicate lists of the objects). Our proposal tries to take advantage of that common predicate lists, through the construction of a *predicate list vocabulary*, where predicate lists are sorted by its number of occurrences, in the way that smallest codewords are assigned to the most common *predicate lists*. A different vocabulary is built for the predicate lists of the subjects (V_{sp}) and the predicate lists of the objects (V_{op}).

Figure 8.8 shows an example of how the predicate lists are computed for the dataset described in Figures 8.7 and 8.6. The predicate lists for all the subjects and the objects are shown in the center of the figure. For instance, subject 4 is related to the predicates 1, 2 and 4 in several triples, so its *predicate list* is $\{1, 2, 4\}$. A dictionary of the different predicate lists of the subjects is built and ordered by frequency (that is, by the number of subjects represented by this predicate list).

The different predicate lists are represented in two succinct vocabularies V_{sp} and V_{op} that can be implemented through an array of integers where predicate lists are located consecutively and ordered by its frequency. An additional index array *Vocabulary* (IV_{sp} and IV_{op}) locates where the first predicate of each predicate list is, in order to provide indexed access to the Vocabulary. In that way, the i -th most frequent predicate list of the subjects is composed by $V_{sp}[IV_{sp}[i]], V_{sp}[IV_{sp}[i] + 1], \dots, V_{sp}[IV_{sp}[i + 1] - 1]$.

Figure 8.8 shows the *Vocabulary* and the *Index Vocabulary* for the predicate lists of the subjects and the objects. For instance, the 4-th predicate list of the subjects (that is, $\{1, 2, 4\}$), is located from the 4-th to the 6-th position in V_{SP} , because $VI_{SP}[4] = 4$ and $VI_{SP}[5] = 7$. The most frequent predicate list of the objects $\{2\}$ is located in the first position of V_{OP} .

This basic predicate list vocabulary can be implemented in many different ways. An alternative way of storing the vocabulary index is using a bitmap B with the same length of the vocabulary array, where the position i of the bitmap has a 1 if the element $V[i]$ is the first predicate of a predicate list. Otherwise, if it is an intermediate element of a predicate list, it has a *zero*. In that way, the p -th predicate list can be located through two *select* operations over this bitmap: the

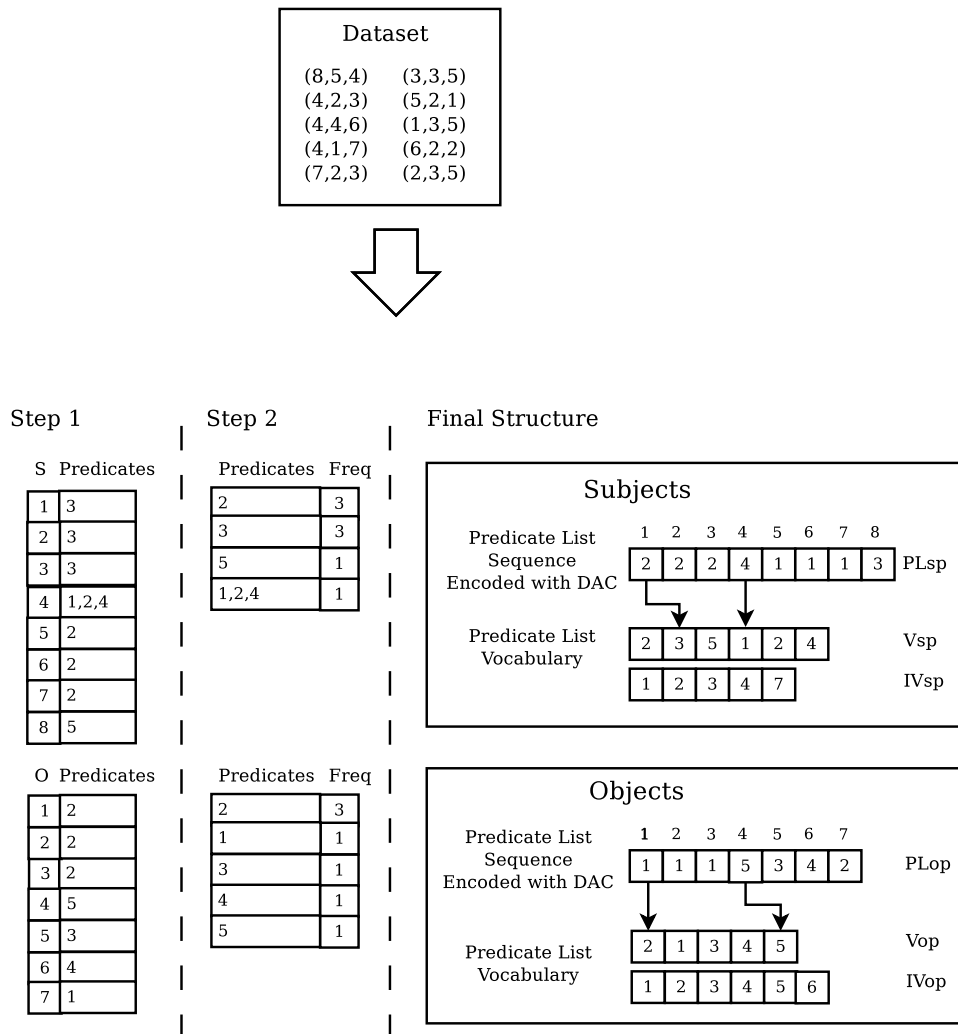


Figure 8.8: SP and OP indexes for K^2 -triples

values of the i -th predicate list are stored between the positions $select_1(B, i)$ and $select_1(B, i + 1) - 1$ of the V_{SP} array.

With the dictionary explained above, the different predicate lists are stored so they can be accessed in a fast way. The identifiers of the predicate list assigned to each subject and object are encoded with DAC by assigning shorter codes to the most frequent predicate lists. In that way, DAC represents the encoded sequence of predicate list identifiers ordered by subject identifier.

Figure 8.8 shows the predicate lists sequences encoded with DAC, named PL_{SP} for the subjects and PL_{OP} for the objects. For instance, $PL_{SP}[4] = 4$, meaning that the subject 4 has assigned the 4-th predicate list in the vocabulary. The components of the predicate list 4 start in $V_{SP}[IV_{SP}[4]] = V_{SP}[4]$ and finishes with the element $V_{SP}[IV_{SP}[5] - 1] = V_{SP}[6]$, obtaining the predicate list $\{1, 2, 4\}$. That means that all the triples with subject 4 have as a predicate value 1, 2, or 4. Then, for the query $(4, ?, ?)$, is guaranteed that the check of only the K^2 -trees $\{1, 2, 4\}$ obtains the full result. Therefore, we can avoid the check of all the remaining K^2 -trees because they will not contain any valid triple. Similar process can be followed for queries that specify a given object, by using the index OP .

The indexes **SP** and **OP** improve the performance of many queries with unbounded predicate. They can be applied in many triple patterns but also in join patterns as an early filter of the query space. Specific details of the query implementations are given in next sections, where the usage of that indexes is also described.

8.4 Triple pattern resolution in K^2 -triples

This section describes the triple pattern resolution in the K^2 -triples system. Triple patterns are the most simple queries that can be solved in a RDF store, and their importance relies on the fact that more complex queries are implemented over the basis of this triple pattern resolution. These triple patterns can be natively solved in K^2 -triples as a combination of basic queries over a K^2 -tree, like checking a cell, a row or a range in the K^2 -tree. They are efficient operations because the original K^2 -tree structure was specifically designed to solve that kind of queries.

Next, the implementation of the triple patterns in K^2 -triples is explained. For queries with unbounded predicates, the possible usage of the indexes SP and OP is also introduced:

- (S, P, O) The cell (S, O) of the K^2 -tree that corresponds to the predicate P is queried. The result of this query is the triple (S, P, O) when it exists. Otherwise, no results are returned. Query 1 in Figure 8.9 shows an example

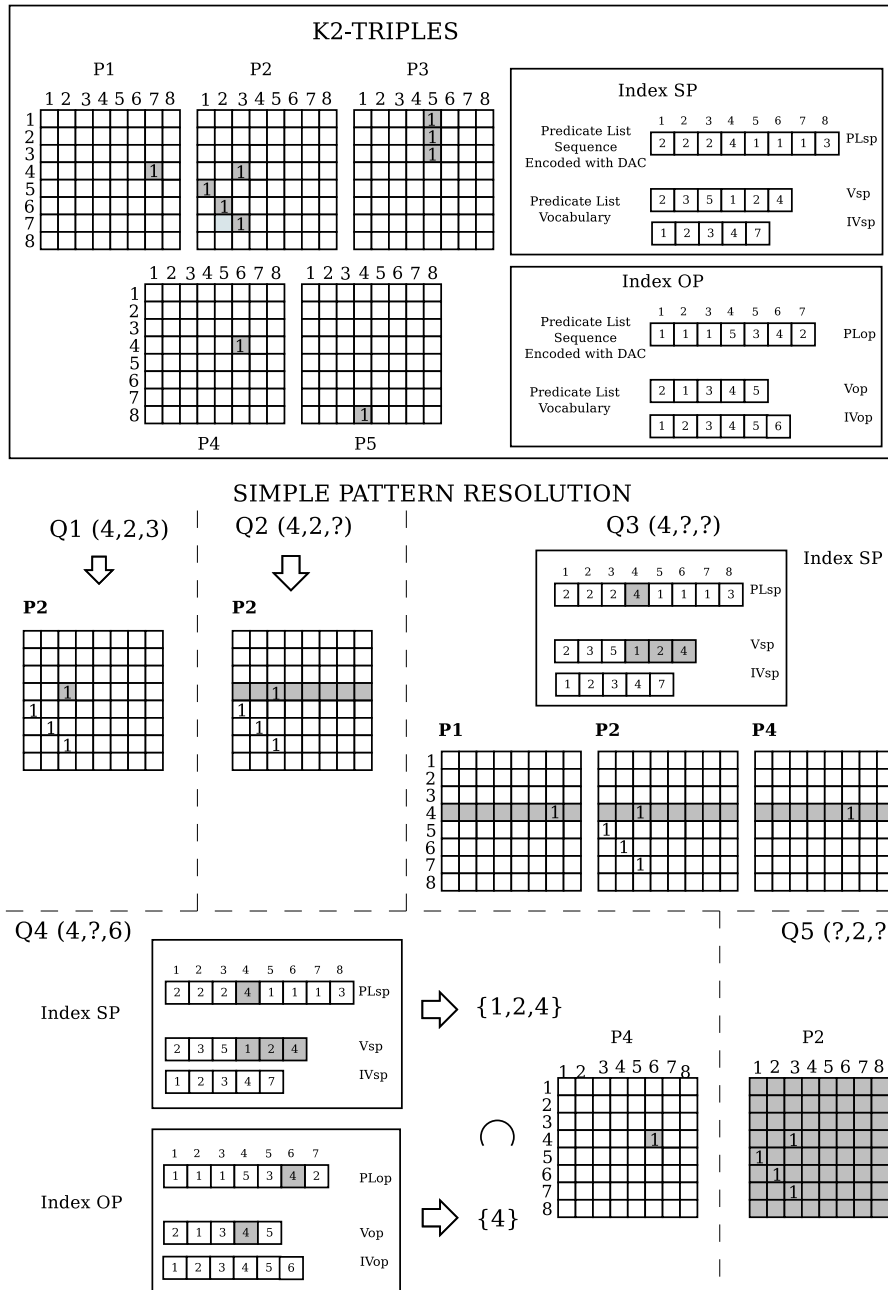


Figure 8.9: Simple pattern resolution

of resolution of a triple pattern included in this category. The pattern $(4, 2, 3)$ is solved by checking the value of the cell $(4, 3)$ in predicate 2, that produces $\{(4, 2, 3)\}$ as result.

- $(S, P, ?)$ In this pattern the predicate is bounded, so the operation is solved by querying a single K^2 -tree, where the full row for the subject S will be queried. Query 2 in Figure 8.9 shows the resolution of $(4, 2, ?)$ that checks the row 4 of the K^2 -tree for the predicate 2. The cell $(4, 3)$ has value 1, therefore, the result to the query $(4, 2, ?)$ is a set that contains one triple $\{(4, 2, 3)\}$.
- $(S, ?, ?)$ It is a pattern with an unbounded predicate, therefore a row operation has to be performed over all the K^2 -trees of the dataset. The performance of this operation can be improved by using the index SP in order to limit the number of K^2 -trees queried. The index SP provides the predicates which subject S is related so only the predicates in the predicate list of S are queried. Query 3 shows an example of this pattern where the index SP is used. Instead of querying the row 4 in all the K^2 -trees, only the predicates 1, 2 and 4 are checked. They will contain, by definition, at least one result of this triple pattern. The result of the pattern $(4, ?, ?)$ is the set $\{(4, 2, 3), (4, 4, 6), (4, 1, 7)\}$.
- $(S, ?, O)$ The basic implementation of this pattern checks the cell (S, O) in all the K^2 -trees of the dataset. However, this implementation can be improved by using both indexes SP and OP to limit the query space. Given the predicate lists $PL_{SP}[S]$ and $PL_{OP}[O]$ of the subject S and the object O respectively, only the predicates $PL_{SP}[S] \cap PL_{OP}[O]$ can return results to the simple pattern. Query 4 of the Figure 8.7 shows an example of this operation using the indexes SP and OP . The predicate list of the subject 4 is $\{1, 2, 4\}$ while the predicate list of the object 7 is $\{4\}$. Therefore, given that $\{1, 2, 4\} \cap \{4\}$, only the cell $(4, 6)$ of the predicate 4 is checked. It is a 1, so the query result is $\{4, 4, 6\}$.
- $(?, P, O)$ It is solved similarly to the $(S, P, ?)$ operation.
- $(?, ?, O)$ First of all, the index OP is checked, in order to limit the number of K^2 -trees queried. Then, for each predicate in the predicate list of the object O the column O is queried and each value 1 composes a new triple included in the final result.
- $(?, P, ?)$ That pattern is solved by checking all the cells of the given K^2 -tree that stores P . Query 5 in Figure 8.9 shows how the pattern $(?, 2, ?)$ is solved, producing three results: $\{(4, 2, 3), (5, 2, 1), (6, 2, 2)\}$
- $(?, ?, ?)$ It is solved by checking all the K^2 -trees of the dataset.

8.5 Join Resolution in K^2 -triples

The previous section describes how the simple triple patterns are implemented in K^2 -triples. They are the most basic operations that can be queried in SPARQL. In general, complex queries in SPARQL are composed by a conjunction of Basic Graph Patterns (BGP), which *join* triple patterns by sharing variables. K^2 -triples supports the joins composed by a pair of triple patterns in an efficiently-native way. This pair join resolution is implemented with the purpose of setting the basis over which more complex SPARQL queries could be implemented. However, providing a full support of SPARQL is out of the scope of this work. We aim to support an API of simple operations (triple patterns and pair joins) in order to allow external SPARQL libraries to be implemented over it.

We define a pair join query in K^2 -triples as two triple patterns where one variable is shared between the two triple patterns, which can be placed in the subject or the object of the triple patterns (it is not allowed in the predicate position). We called this variable *join variable*. The remaining elements of the pair join can be given (specified with an identifier) or be unbounded. Depending on the position of the *join variable*, the join is named *subject-subject*, *subject-object* or *object-object*. For instance, the pair join $(S, ?, ?X) \bowtie (?X, ?, ?)$ is a *subject-object join*.

Subject – object join resolution is one of the main weakness of the vertical-partitioning approaches, because they usually support only *subject-subject* joins and they need additional indexes to perform joins over the object variables. However, K^2 -triples overcomes this problem and it solves *object – object* and *subject – object* joins in a native way because of two main factors. First of all, object indexation is guaranteed in the system thanks to the structural properties of the K^2 -tree, that provides both row and column indexation capabilities. Secondly, the rules of encoding used to assign the identifiers to the subjects, objects and predicates enclose the range of valid identifiers for the *subject-object* joins. The range of valid values for a join variable of a *subject-object* join is $1 \dots |SO|$. Therefore, the search space can be limited to the rows $1 \dots |SO|$ and columns $1 \dots |SO|$. Therefore, *cross-joins* are natively supported in K^2 -triples, which are the basis of the common *path expression queries*, traditionally considered in the vertical-partitioning approaches as inefficient and expensive operations [AMMH07].

Note that due to the vertical-partitioning nature of K^2 -triples system, pair joins over predicate variable are not natively supported. However, it is worth noting that the operations involving predicates as join variables are not frequent in real contexts.

This section is structured as follows. First, a classification of the different pair-joins supported in K^2 -triples is given. Then, the join strategies that are implemented are discussed. Finally, the convenience of applying the strategies for each kind of join is analysed in the last section.

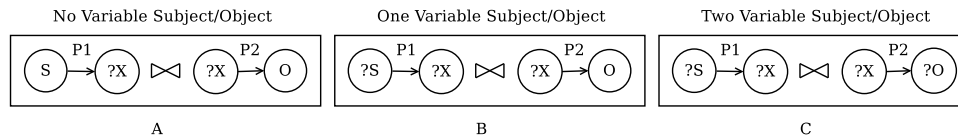
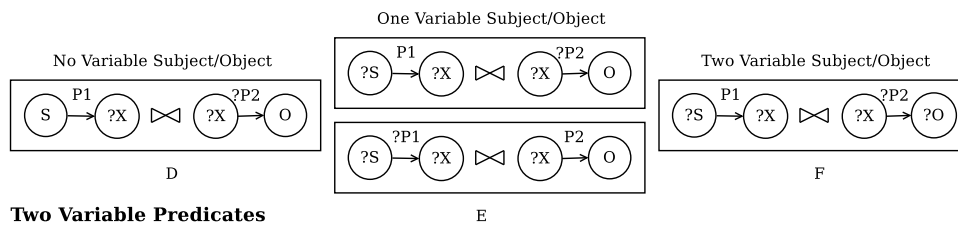
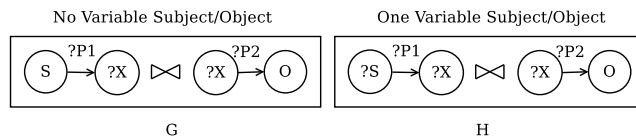
No Variable Predicates**One Variable Predicate****Two Variable Predicates**

Figure 8.10

8.5.1 Join classification

Figure 8.10 shows the different kinds of pair joins classified by two criteria: the number of *bounded/unbounded* variables (columns of the table) and the number of *bounded/unbounded* predicates (rows of the table). This figure shows the join pattern in its *subject-object* form representing each join class, but for each pattern, symmetrical *subject-subject* and *object-object* joins are included in the same class. Join variable is denoted as $?X$.

Therefore, the joins can be classified according to the specified values for the predicates in three rows:

- Row with no variable predicates: it involves the classes *A*, *B* and *C* representing join patterns where a predicate value is specified in both patterns of the join.
- Row with one variable predicate: it includes the classes *D*, *E* and *H*, where one pattern of the pair join has unbounded predicate and the other pattern asks for an specific predicate value.
- Row with two variable predicates: it includes the classes *G* and *F*, where both patterns have unbounded predicates.

Each one of these rows determines the number of K^2 -trees involved in the query. First row is a join between regions included in two K^2 -trees (one per pattern). In the second row, a specific K^2 -tree is joined with the $|P|$ K^2 -trees of the second pattern (although this number can be reduced in some cases using the indexes SP and OP). Finally, the third row is a join of $|P|$ K^2 -trees in both sides of the join (again, it can be reduced using additional indexes).

Columns divide the joins according to the subject and object values that are not the join variable. They determine the query space for the K^2 -trees involved in the query:

- Column with bounded subject/object values (classes A , D and G). It reduces the query space to rows or columns for each involved K^2 -triple.
- Column with one unbounded subject/object (classes B , E and H). It involves full K^2 -trees in one side of the join and rows or columns in the other pattern of the join.
- Column with two unbounded subject/object (classes C and F). Full K^2 -trees are explored in both sides of the join.

Two additional comments can be done about the join classification explained above. First, in the case of the class E , which contain one unbounded predicate and one unbounded subject/object, two configurations are possible, $E.1$ where the predicate and the subject/object variable is in the same side, and $E.2$, where each pattern contains one unbounded variable. Finally, the figure does not include the theoretical class I , $(?, ?, ?X) \bowtie (?X, ?, ?)$ where all the non-join variables are unbounded, which is clearly impractical and therefore is not studied.

8.5.2 Join algorithms

Join algorithms have been widely studied in the relational database world [RG00] and they have been also studied from a perspective of semantic Web databases [Gro11]. Inspired in this experience, we design three different join strategies for K^2 -triples:

- **Chain evaluation** This algorithm follows the philosophy of the traditional *index join*. In a first step, the first pattern of the join is solved (considering the join variable as a classical unbounded variable). Then, for each different value for the join variable obtained as result, the second pattern is *inflated* with this join variable and it is executed as a simple triple pattern. Therefore, the execution of a join in a chain evaluation strategy consists of executing a simple pattern, obtaining the different values for the join variable (it can also need an additional ordering of the resulting triples, depending on the query) and a

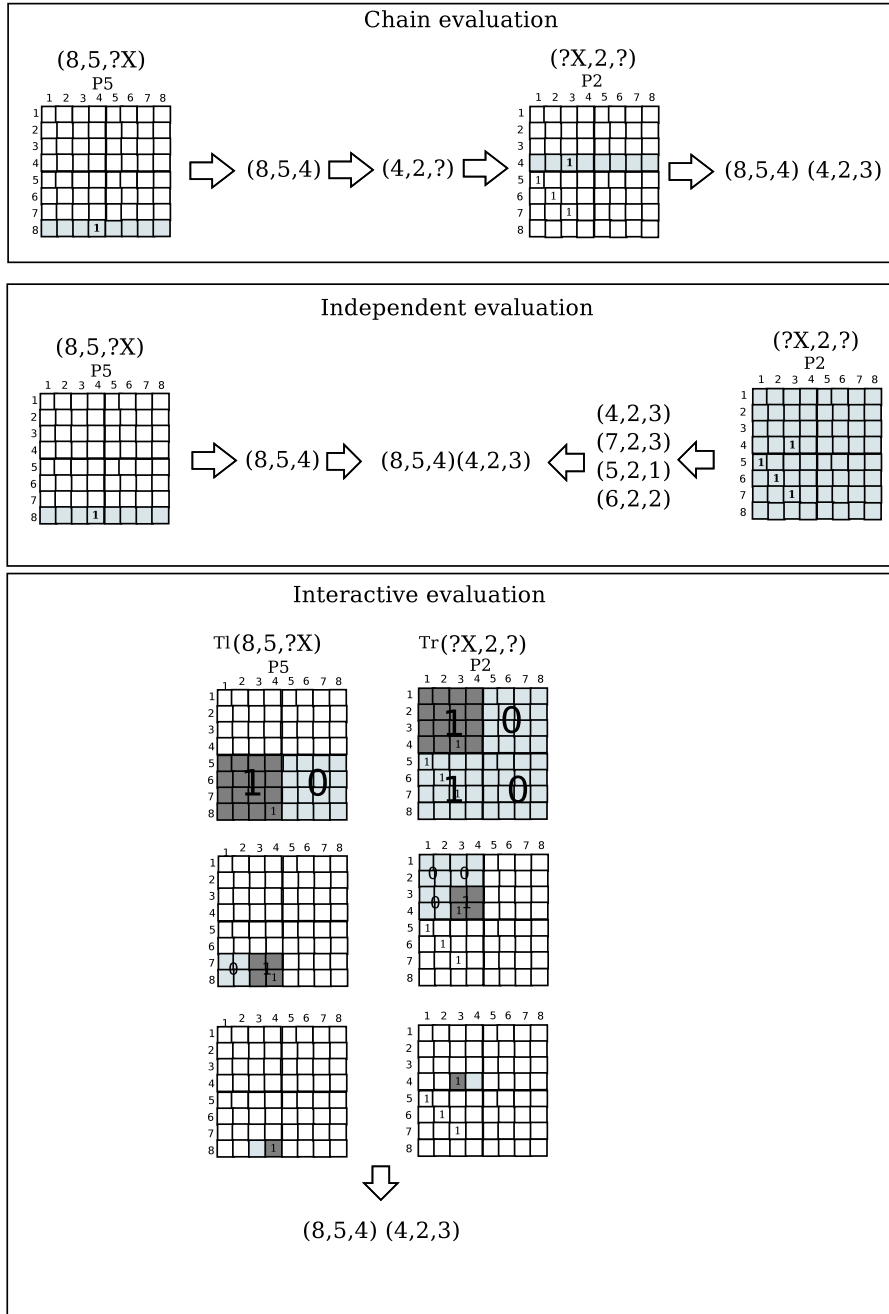


Figure 8.11: Classification of the different pair joins

simple triple pattern for each different value for the join variable. Figure 8.11 shows an example of chain evaluation for the join query $(8, 5, ?X) \bowtie (?X, 2, ?)$. First, the left pattern is executed as a row operation in the predicate 5. Then, for each different object obtained (in this case, the object 5), the second pattern is completed and executed (in that case, the $(4, 2, ?)$ query). In this example, the final result set is composed by only one element $\{(8, 5, 4)(4, 2, 3)\}$.

- Independent evaluation** This algorithm is based on the *merge join* concept. It consists in executing individually the two simple patterns and the results are joined by its join variable. Again, depending on the query, an additional ordering of the results for each pattern can be necessary to the execution of the posterior merge. Figure 8.11 (middle) shows how the query of the previous example $(8, 5, ?X) \bowtie (?X, 2, ?)$ is executed with an independent evaluation philosophy. A row operation is executed for the left pattern and a range operation is executed for $(?, 2, ?)$. The results of each query are ordered by the join variable and then are joined in order to obtain the final result set.
- Interactive evaluation** This algorithm is strongly inspired on the Sideways Information Passing (SIP) mechanism proposed by Neumann and Weikum [NW09], where information is passed between the operands involved in the query to provide a mutual feedback. In interactive evaluation, the K^2 -trees involved in both patterns of the join are executed progressively by levels. As usual, K^2 -trees involved in a query are navigated top-down selecting the explored branches depending on the specified values in the query. But, when the first levels of all the trees are executed, we check what branches remain active, in order to prune such branches that can not take part of the final result because the values for the join variable that they range were eliminated from the other pattern of the query. That is, all the sub-interval of identifiers which are pruned in one pattern of the join will not continue being explored in the second pattern. This checking is performed after the execution of each level, providing additional information to prune more branches of the tree. Figure 8.11 shows an example of the process step by step. At the first level of the tree, sub matrices of size 4×4 are explored. Two ranges for the join variable are defined: $1 - 4$ and $5 - 8$. Note that the join is a subject-object join, so the join variable is located in the columns of T_l and in the rows of T_r . T_l is a row operation, so only two submatrices are explored. However, T_r has unbounded object, so the four submatrices are checked. When the first level of the tree is explored, we observe that the range of values $5 - 8$ for the join variable, T_l has not contain any result. Therefore, although the bottom-left matrix in T_r contain results, it will not be explored because it belongs to a invalid range of values for T_l ($5 - 8$). T_l contains valid values for the range $1 - 4$, so the submatrices of T_r with *one* values continues in that range its execution (in this case, the top-left matrix). The second level of the tree manages smaller

submatrices, so the ranges 1 – 2 and 3 – 4 are inspected. In the case of T_l , only the second interval is represented by a *one* value, so the submatrices of T_r located in that rows (that is, the top submatrices) are discarded. The next step only explores the active submatrices for the region of values 3 – 4. Final result is obtained for that join variables with cells in both patterns (in that case, (8, 5, 4)(4, 2, 3) is the only result).

8.5.3 Join implementation

In this section we detail how the different join classes can be implemented using the join strategies explained in the previous section. We will refer to T_r and T_l as the first and second patterns of the join. In general, the three strategies can be applied in all the join classes, although in some cases some of them will be impractical. The best strategy in terms of temporal efficiency but also in spatial efficiency (intermediate results that need to be generated for performing the query) depends on the join class, but it also depends on the features of the dataset. However, as a general heuristic, chain evaluation usually performs better than the independent evaluation when the query space of one of the triple patterns is smaller than the corresponding to the other one. Interactive evaluation usually is a good choice but, for joins with a great query space (that is, with many unbounded variables), it can produce an spatial overflow due to the produced intermediate results. Note that the indexes SP and OP are used in the three strategies whenever the predicate is variable and the independent subject/object is specified. Next, details about the possible implementations in each join class are given. We explain them using the subject-object class. However, the implementation is completely analogous for subject-subject and object-object joins.

8.5.3.0.1 Joins with no variable predicates These first joins specify the predicate values, so indexes SP and OP are not need and each pattern T_l and T_r involves one K^2 -tree. They are located in the first row of the Figure 8.11. Interactive evaluation works over two K^2 -trees, pruning branches when no results are obtained for one of the trees in a sub-interval of values. Next, we describe how the chain and independent approaches are implemented:

- **Join A** $(S, P_1, ?X) \bowtie (?X, P_2, O)$ It is the simplest pattern, where all the values are provided (except the join variable). Independent evaluation involves two row/column operations and a *merge* of the results, while chain evaluation first executes a row/column operation and then a cell operation is executed for each result of T_l . Independent evaluation can perform better when the rows involved in the operation contain many *ones*.

- **Join B** $(?, P_1, ?X)(?X, P_2, O)$ contains an unbounded variable in T_l . Chain evaluation starts by T_r , executing a column operation. Then, for each result, a column operation is performed over P_1 . Independent evaluation involves a range operation for T_l and a column operation for T_r . After that the results of both sides are merged by the join variable. The column operation already provides the results ordered by the join variable. However, the results for T_l are ordered in z -order, so a previous ordering by the join variable has to be performed to prepare the result data to the merge operation.
- **Join C** $(?, P_1, ?X)(?X, P_2, ?O)$. Chain evaluation starts by performing a range query operation, which can produce several results for each different join variable value. Then, an ordering of the results is performed to extract the different values to fill and execute T_r as a row operation. Independent evaluation in this case execute two range operations, order the results and performs the merge join over the ordered results.

8.5.3.0.2 Joins with one variable predicate Next class joins contain one of the patterns with the unbounded predicate. As a consequence, that pattern involves several K^2 -trees which number, in most cases, can be reduced by using the index SP or OP . In the case of the interactive evaluation, several K^2 -trees are involved in the operation, but the process works in the same way: a sub-interval for a join variable is discarded when any of the K^2 -tree of one of the patterns contain results. Chain and independent strategies can be also applied:

- **Join D** $(S, P_1, ?)(?, ?, O)$ Chain evaluation starts by solving the T_l pattern as a row operation. Then, each result produces $|P|$ cell operations in T_r , which can be limited using the indexes SP and OP . Independent evaluation consists in a row operation for T_l and $|P|$ column operations in T_r (which can be also reduced using index OP). T_r produces many lists of the results with the independent evaluation (one for each checked predicate), so a previous ordering of the lists is need to the merge join operation.
- **Join E.1** $(?, P_1, ?)(?X, ?, O)$ Chain operation starts with T_r , performing several column operations whose results are ordered by adaptive sorting and next, each different result produces a column operation in the second pattern. Independent evaluation intersects a range operation with the multiple column operations, where additional orderings are need, as usual, previously to the merge join.
- **Join E.2** $(?, ?, ?)(?X, P, O)$ Chain evaluation starts with T_r (column operation), where each result produces $|P|$ column operations for T_l . In that case, independent operation is impractical because T_l is all unbounded, implying the retrieval of the full database.

- **Join F** $(?, P, ?X)(?X, ?, ?)$ Again, independent evaluation is not possible, while chain evaluation starts with T_l , implemented as a range operation, whose results have to be ordered by the join variable. Each different value for the join variable produces $|P|$ row operations in T_l (they can be reduced using the information provided by the index SP).

8.5.3.0.3 Joins with two variable predicates Both sides of the join have the predicates unbounded, so several K^2 -tree are involved in each pattern. This means that interactive evaluation manages several K^2 -trees for both patterns. Chain and independent evaluation can proceed as follows:

- **Join G** $(S, ?, ?X)(?X, ?, O)$ Independent evaluation consists in performing $|P|$ row/column operations for each pattern (reduced when indexes SP and OP are used). Resulting lists for each pattern are ordered and then a merge join is performed over the two ordered lists of results. Chain evaluation is implemented starting with T_r , composed by $|P|$ column operations. When the resulting lists are ordered, each different result is used to fill T_l , performing several cell operations to obtain the final result.
- **Join H** $(?, ?, ?X)(?X, ?, O)$ Independent evaluation is obviously impractical for that case, while chain evaluation starts with T_r , performing several column operations in T_l for each different value for the join variable in T_r .

Table 8.1 summarizes the described implementations for each class of join. The first and second column described each join class. The third column shows the order of pattern execution following the chain evaluation strategy, denoting with a * when the results of a pattern have to be ordered. The next column is checked when independent evaluation is a practical solution of that join class (denoting again with a * the additional orderings). Last columns show the initial region of space that is explored in the interactive evaluation (for pattern T_l and T_r respectively). Finally, note that although the explanation of the join implementation is provided for *subject-object* joins, the subject-subject and the object-object joins are implemented exactly in the same way.

8.6 Experimentation

In this section we analyze the performance of K^2 -triples measuring this efficiency in real RDF datasets that represent very different areas of knowledge. We measure the spatial requirements to store those real datasets and the temporal efficiency to query them through simple triple patterns and joins between pairs of triples. We compare our results against some relevant implementations proposed in the State of the Art.

Table 8.1: Summary of joins resolution in k^2 -triples (* means that removing duplicates is required for join resolution)

Join Class	Example	Chain	Independent	Interactive	
				T_l	T_r
A	$(S, P_1, ?X) (?X, P_2, 0)$	$T_l \rightarrow T_r$ $T_r \rightarrow T_l$	\checkmark	Direct	Reverse
B	$(?S, P_1, ?X) (?X, P_2, 0)$	$T_r \rightarrow T_l$	\checkmark^*	Range	Reverse
C	$(?S, P_1, ?X) (?X, P_2, ?0)$	$T_l^* \rightarrow T_r$ $T_r^* \rightarrow T_l$	\checkmark^*	Range	Range
D	$(S, P_1, ?X) (?X, ?P_2, 0)$	$T_l \rightarrow T_r$	\checkmark^*	Direct	Reverse ($\times preds$)
E.1	$(?S, P_1, ?X) (?X, ?P_2, 0)$	$T_l^* \rightarrow T_r$ $T_r^* \rightarrow T_l$	\checkmark^*	Range	Reverse ($\times preds$)
E.2	$(?S, ?P_1, ?X) (?X, P_2, 0)$	$T_r \rightarrow T_l$		Range ($\times preds$)	Reverse
F	$(?S, P_1, ?X) (?X, ?P_2, ?0)$	$T_l^* \rightarrow T_r$		Range	Range ($\times preds$)
G	$(S, ?P_1, ?X) (?X, ?P_2, 0)$	$T_l^* \rightarrow T_r$ $T_r^* \rightarrow T_l$	\checkmark^*	Direct ($\times preds$)	Reverse ($\times preds$)
H	$(?S, ?P_1, ?X) (?X, ?P_2, 0)$	$T_r^* \rightarrow T_l$		Range ($\times preds$)	Reverse ($\times preds$)

8.6.1 Experimental Setup

We run all the experiments on an AMD-PhenomTM-II X4 955@3.2 GHz, quad-core (4 cores - 4 siblings: 1 thread per core), 8GB DDR2@800MHz, running Ubuntu 9.10. All the implementations were developed in C and we compile them using *gcc* (version 4.4.1) parametrized with the optimization *-o9*. We show the results of our proposal for the two different alternatives described in previous sections:

- K^2 -triples, our vertical partitioning approach that uses one K^2 -tree structure for each different predicate included in the dataset.
- K^2 -triples⁺ that, starting from the same vertical partitioning strategy, it includes additional indexes *SP* and *OP* used to improve the temporal efficiency of such queries with unbounded predicates.

8.6.1.0.4 Datasets We choose a collection of datasets from different domains with the purpose of testing our proposal in RDF data following different distributions. We are interested on studying the scalability of our solution, so our experimental framework includes datasets from different sizes, from 1 million of triples to 232 millions. Our proposal consists in a vertical partitioning of the data, so a high number of predicates could be one of the most important issues in our proposals,

Table 8.2: Statistical dataset description

Dataset	Size (MB)	# Triples	# Predicates	# Subjects	# Objects
jamendo	144.18	1,049,639	28	335,926	440,604
dblp	7,580.99	46,597,620	27	2,840,639	19,639,731
geonames	12,347.70	112,235,492	26	8,147,136	41,111,569
dbpedia	33,912.71	232,542,405	39,672	18,425,128	65,200,769

specially in K^2 -triples version (which does not include the indexes SP and OP). Therefore, although we study datasets with a reduced number of predicates (over 20 predicates), we will pay special attention to such cases with a large number of predicates (almost 40,000), which allow us to analyze the impact of the improvement achieved with the indexes SP and OP. We choose the following datasets:

- jamendo² is a large repository of Creative Commons licensed music, that includes information about artists and their records, tracks and performances. It is a dataset from a very specific domain, so the triples are described by using very few different predicates.
- dblp³ provides information on Computer Science journals and proceedings and it was also used as use case for GraphGen on Chapter 3.
- geonames⁴ is a geographical database covering all countries and containing a large number of places, with predicates that describe characteristics of the places like its location or population as well as relationships between the places (like a belonging relation). It is a dataset very used to enhance the information of datasets from very different domains, which connect with them by using geonames resources as objects in such triples that specify a place.
- dbpedia⁵ is the semantic evolution of Wikipedia, an encyclopedic dataset. dbpedia is considered the “nucleus for a Web of Data” [ABK⁺07] and it is linked from many different datasets. It contains a large number of predicates due to the fact that it includes very general knowledge, using very different descriptors to represent all of this information.

Table 8.2 shows the main information of these datasets. We processed all the datasets as a previous step to transform them to the K^2 -triples representation.

²<http://dbtune.org/jamendo/>

³<http://dblp.13s.de/dblp++.php>

⁴<http://download.geonames.org/all-geonames-rdf.zip>

⁵<http://wiki.dbpedia.org/Downloads351>

First, we represent all the datasets in N-Triples [GB04] format, where each triple is represented in a different line. We use Any23 tool⁶ to transform the datasets published in a different format to N-triples. Finally, the dataset file is lexicographically sorted and duplicated triples are discarded. Table 8.2 shows the size in N-Triples and the number of triples, different predicates, subjects and objects of each dataset.

We include datasets of different sizes in this evaluation. We include jamendo in order to observe the behavior of K^2 -triples in a small dataset. Additionally, it allow us to compare it with other solutions indexing uncompressed data in memory. On the other hand, we choose large datasets to show the scalability of the proposal, managing from 46 Million triples in dblp up to more than 232 Million triples in dbpedia.

K^2 -triples represents subjects and objects as rows and columns of the adjacency matrices. The number of different subjects and objects is shown in the last columns of Table 8.2. The number of different subjects is significantly lower than the number of objects. It is due to the fact that subjects describe the resources in RDF (usually appearing in multiple triples) whereas objects represent the values of the descriptions, which are used in many cases in an unique triple for all the dataset (e.g., a concrete timestamp, a textual description, an ID field, etc.).

Table 8.2 also shows that the number of predicates is usually low, including three datasets with 26, 27 and 28 different predicates. The only exception is the dbpedia dataset, an extreme case in which the number of predicates grows to the order of thousands due to the variability of the represented information. It allows us to analyze the performance of K^2 -triples when the number of predicates increases. This is the worst case for queries with unbounded predicate, which is the main weakness of the vertical partitioning approaches.

8.6.1.0.5 RDF Stores We compare our results with three representative techniques in the State of the Art.

- A *vertical partitioning* solution following the vertical partitioning approach of [AMMH07] which was described in Section 8.2.1.3. We implement it over MonetDB (MonetDB Database Server v1.6, Jul2012-SP2) because it achieves better performance than the original C-Store based solution [SGK⁺08].
- **Hexastore**⁷, a memory-based system that was described in Section 8.2.2

⁶<http://any23.apache.org/> (version: any23-0.6.1)

⁷Hexastore has been kindly provided by its authors.

- **RDF3X**⁸, a highly-efficient store, described in Section 8.2.2 that was recently reported as the fastest RDF store [HAR11a].

All these techniques had been tested following the configurations and parameterization provided in their original sources.

8.6.1.0.6 Queries We design experiments focused on demonstrating the efficiency of all RDF stores included in this experimental evaluation. We run triple pattern and join experiments in order to predict the core performance for basic graph pattern solution in SPARQL.

Our experiment set is composed by randomly generated queries⁹ that covers all the possible kind of triple patterns and pair joins. For each dataset, we consider 500 random triple patterns of each type. Note that in all datasets, except for dbpedia, the triple pattern (**?S,P,?O**) is limited by the number of different predicates.

Join tests are generated by following the classification described in Figure 8.10. For each different kind of join we consider Subject-Object (SO), Subject-Subject (SS), and Object-Object (OO) joins. We generate 500 random queries of each join and perform a big-small classification according to the number of intermediate results: for each join we take the product of the number of results for the first triple pattern and the results of the second triple pattern in the join. According to this value we classify the queries, randomly choosing 25 queries with a number of intermediate results over the mean (joins **big**) and other 25 queries with fewer results than the mean (joins **small**).

We design two evaluation scenarios to analyze how I/O transactions penalize on-disk RDF stores included in our setup. The *warm* evaluation is designed to cause that the query results be available in main memory before the query execution. It was implemented taking the mean solution time of six consecutive repetitions of each query. On the other hand, the *cold* evaluation simulates a real scenario in which queries are independently performed.

8.6.2 Compression Results

In this section we compare the results achieved by K^2 -triples against the other approaches. This comparison involves on-disk based representations, MonetDB and RDF3X, and memory-based ones, Hexastore and our two K^2 -triples based approaches. In these cases, we consider the space required for operating the representations in main memory. Table 8.3 summarizes the results for all stores and all datasets.

⁸<http://code.google.com/p/rdf3x/>

⁹The full testbed is available at <http://dataweb.infor.uva.es/queries-k2triples.tgz>

Table 8.3: Space requirements (all sizes are expressed in MB)

	On-disk		Memory-based		
	MonetDB	RDF-3X	Hexastore	K^2 -triples	K^2 -triples ⁺
jamendo	8.76	37.73	1,371.25	0.74	1.28
dblp	358.44	1,643.31	×	82.48	99.24
geonames	859.66	3,584.80	×	152.20	188.63
dbpedia	1,811.74	9,757.58	×	931.44	1178.38

We observe that MonetDB is more compressed than RDF3X and Hexastore. This is an expected result according to the compressibility of column-oriented representations [AMF06]. MonetDB uses up to 5.4 times less space than RDF3X. On the other hand, Hexastore reports the worst results for jamendo and cannot index the other datasets in our configuration.

Nevertheless, K^2 -triples requires much less space on all the datasets, taking advantage of its compact data structures. This result can be analyzed from three complementary perspectives:

- K^2 -triples achieves better results than column-oriented compression for vertically partitioned representations. The comparison between our approach and MonetDB shows that K^2 -triples requires several times less space than the column-oriented database. The space used by MonetDB for the largest datasets is 2–5.5 times larger than K^2 -triples and 1.5–4.5 times larger than K^2 -triples⁺.
- K^2 -triples allows many more triples to be managed in main memory. If we divide the number of triples in jamendo (1,049,644) by the space required for their memory-based representation in Hexastore (1,371.25 MB), we obtain that it represents roughly 765 triples/MB. This same analysis, in our approaches, reports that K^2 -triples manages almost 1.5 million triples/MB, and K^2 -triples⁺ represents more than 800,000 triples/MB. Although this rate strongly depends on the dataset, its lowest values (reported for dbpedia) are $\approx 200,000$ triples/MB. This means that K^2 -triples increases by more than two orders of magnitude the number of triples that can be managed in main memory on Hexastore because of its compression ability.
- K^2 -triples provides full RDF indexing in a space significantly smaller than that used for systems based on sextuple indexing. This difference also depends on

the dataset; for instance, RDF3X uses roughly 8–10 times the space required by our techniques for representing dbpedia.

K^2 -triples achieves compression ratios between 10 and 40 bits per triple (bpt) in the datasets analyzed in this experiment (excluding the tiny dataset jamendo, where a ratio of 5 bpt is obtained). However, meaningful differences are observed depending on the features of the dataset. In dbpedia we need almost 34 bpt, as opposed to the 12–15 bpt in geonames and dblp. The main reason for this difference seems to be the high number of predicates used in dbpedia (39,672) in contrast to the other datasets. Many of these K^2 -trees are very sparse (about 57% of the predicates contain less than 10 edges, and roughly 81% less than 100 edges). These K^2 -trees get worse the overall compression, due to the overhead of storing a full K^2 -tree for only a few edges.

We can also see in the table the additional space required by the indexes SP and OP (K^2 -triples⁺) over the original K^2 -triples representation. The extra cost ranges from $\approx 20\%$ for dblp to $\approx 26.5\%$ for dbpedia. The only exception is the jamendo dataset, where indexes SP and OP almost double the original space. Therefore, the use of the additional SP and OP indexes produces an acceptable space overhead considering that our representation remains the most compressed one even adding these new indexes. The spatial cost of the indexes SP and OP depends on two factors. First of all, it depends on the different number of subjects and objects of the dataset. An individual entry representing its corresponding *predicate list* is maintained for each subject and object, so the cost of storing the indexes SP and OP is expected to be high for datasets where each element appears in a few triples. Another factor is the predicate distribution. A dataset with many different predicates will probably have a number of different predicate lists high. As a result, the size of the predicate list vocabulary is incremented, resulting in a higher space overhead. This factor can be observed in the indexes SP and OP for dbpedia, where the cost per element (subject or object) is about 24 bits, in contrast to the other datasets where the cost is about 6 bits per element. However, we have to take into account that the SP and OP indexes are specially useful for such datasets with a large number of predicates.

8.6.3 Query Performance

This section evaluates the query time performance, showing the results for triple patterns and for the pair joins.

8.6.3.0.7 Triple patterns These experiments measure the capabilities of all stores for RDF retrieval through triple pattern solution. These are the atomic SPARQL queries, and are massively used in practice [AFMP11].

Table 8.4: Solution time (in milliseconds) for the patten $(?, P, ?)$ on dbpedia (warm scenario)

	K^2 -triples ⁺	RDF3X	MonetDB
small	0.09	2.53	3.77
big	24.57	14.88	6.14

Figure 8.12 compares these times for jamendo (top) and dbpedia (bottom) in the warm scenario, which is the most favorable for on-disk systems. It shows the average query times for all the possible triple patterns¹⁰ in milliseconds.

The comparison for jamendo includes the system Hexastore. As can be seen, this is never the best choice and it only outperforms MonetDB in patterns with unbounded predicate. According to these results, we discard it because of its lack competitiveness in the current setup. K^2 -triples⁺ is the most efficient choice, and only MonetDB slightly outperforms it for $(?, P, ?)$ in all collections but dbpedia.

Figures from 8.13 to 8.16 summarize the complete triple pattern experiments for all the datasets in our setup. We provide the figures for cold (left column) and warm (right column) scenarios.

In general, our approach reports the best overall performance for RDF retrieval. This can be analyzed in more detail:

- Our approach overcomes the main vertical partitioning drawback and provides high performance for solving *patterns with unbounded predicate*. This is studied on dbpedia because in these queries scalability is more seriously compromised due to the large number of predicates. K^2 -triples⁺ obtains the best results, although RDF3X is close for $(S, ?, ?)$. As expected, a larger improvement is achieved by K^2 -triples⁺ with respect to the original K^2 -triples (between 1 and 3 orders of magnitude).
- MonetDB excels above the other systems in solving the pattern $(?, P, ?)$. The only exception is in dbpedia, due to the fact that in dbpedia some predicates are overused and the remaining ones are scarcely used. Thus, the performance of the query $(?, P, ?)$ depends on the predicate evaluated. Table 8.4 summarizes solution times for predicates returning *small* and *big* result sets. As can be seen, K^2 -triples⁺ is better for less used predicates, whereas MonetDB is better when

¹⁰The pattern $(?, ?, ?)$, which returns all triples in the dataset, is excluded because it is rarely used in practice.

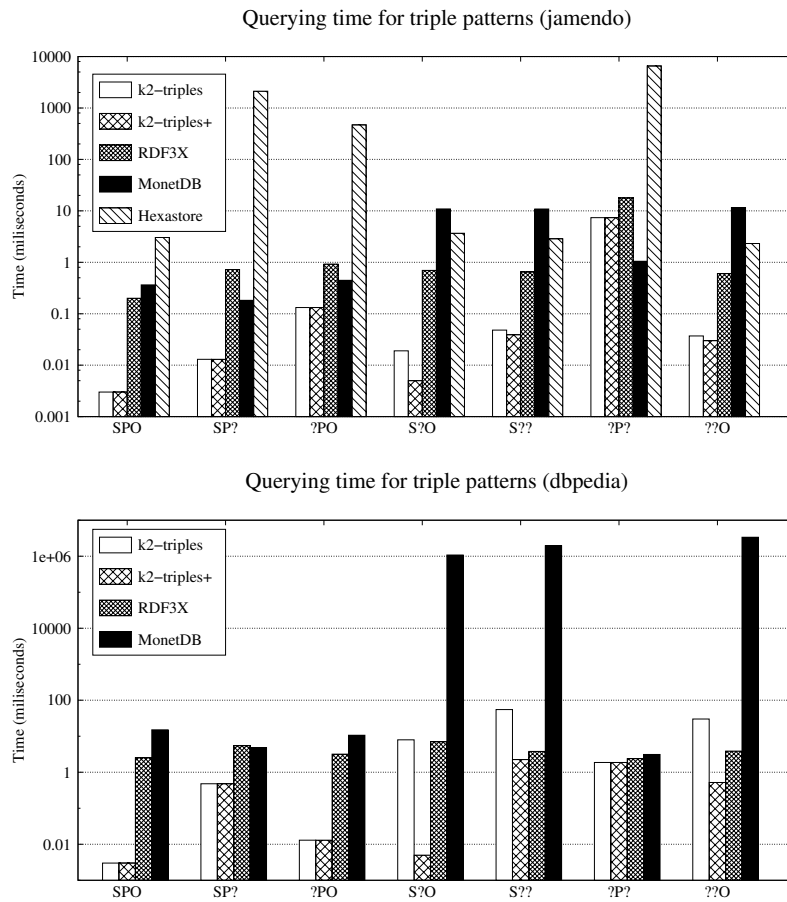


Figure 8.12: Solution time (in milliseconds) for triple patterns in jamendo and dbpedia (warm scenario).

more results are retrieved. Thus, the optimized column-oriented representation provides the fastest solution when the predicate is used in numerous triples, whereas K^2 -triples⁺ outperforms it for more restrictive predicates.

Finally, it is worth noting that K^2 -triples⁺ obtains a competitive advantage over the original K^2 -triples for datasets involving many predicates. For instance, in the case of dbpedia (containing a high number of predicates), the performance of the query pattern (S,?,?) is improved by 2 orders of magnitude. This difference owes to the fact that, in K^2 -triples, 39,672 row queries have to be performed in order to answer that pattern. However, considering that the average number of different predicates describing a subject (that is, its predicate list size) is about 6 in this dataset, the number of K^2 -trees that have to be queried is significantly reduced in K^2 -triples⁺. However, in the other datasets, the maximum number of predicates that are checked in K^2 -triples is already less than 30. As a result, the improvement in those datasets is lower and both techniques achieve comparable performance, yet K^2 -triples uses slightly less space. Nevertheless, we will use K^2 -triples⁺ in all the remaining experiments.

8.6.3.0.8 Joins After studying triple pattern performance, the next stage focuses on join solution. We analyze the results obtained for the three evaluation algorithms we implemented: *chain*, *independent* and *interactive*, comparing them with respect to RDF3X and MonetDB. All these experiments are performed in the warm scenario in order to avoid penalizing on-disk solutions.

Figures 8.17 and 8.18 summarize join results for dbpedia. Figure 8.17 includes the joins from A to E2, while Figure 8.18 shows the joins F,G and H, according to the classification described in Figure 8.10. Each plot comprises three subsets of joins: *Subject-Object* (SO), *Subject-Subject* (SS), and *Object-Object* (OO). The left group considers joins generating a *small* amount of intermediate results, whereas the right group gives equivalent results for joins involving *big* intermediate result sets. Solution times are reported in milliseconds. Times over 10^7 milliseconds are discarded in all the experiments.

- K^2 -triples⁺ is the fastest technique for solving joins in which the value of the two not joined nodes (subjects and objects) are provided (classes A, D and G). This is mainly because all these classes are solved using, exclusively, direct and reverse neighbors queries, which are very efficient in practice. Both *chain* and *interactive* evaluation algorithms are the best choices for the simplest join, the **Join A**. They report, at least, one order of magnitude of improvement with respect to RDF3X and MonetDB. *Chain* evaluation is slightly faster in **Join D**, improving upon RDF3X by more than one order of magnitude (except for OO big). Note that, in this case, MonetDB is no longer competitive since

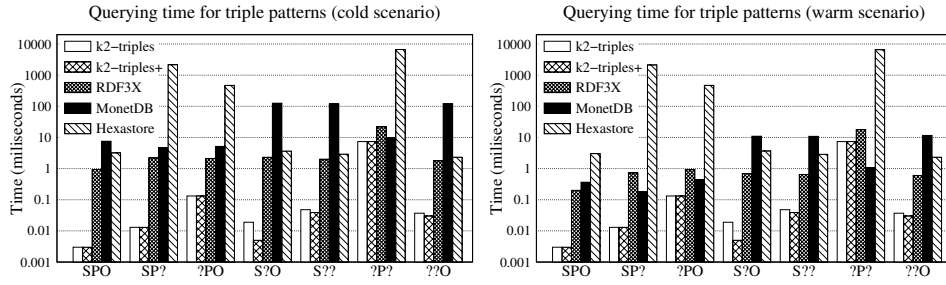


Figure 8.13: Solution time (in milliseconds) for triple patterns in jamendo.

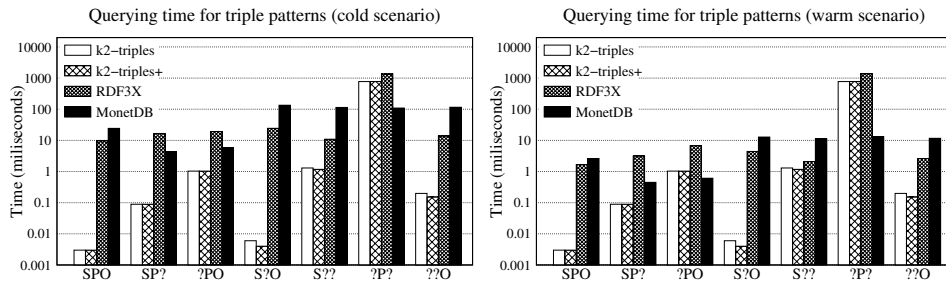


Figure 8.14: Solution time (in milliseconds) for triple patterns in dblp.

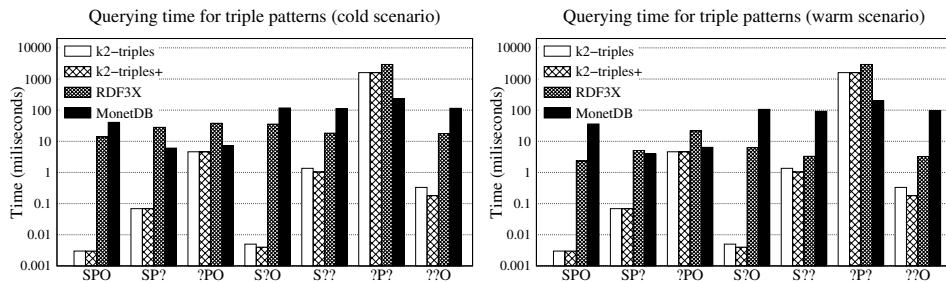


Figure 8.15: Solution time (in milliseconds) for triple patterns in geonames.

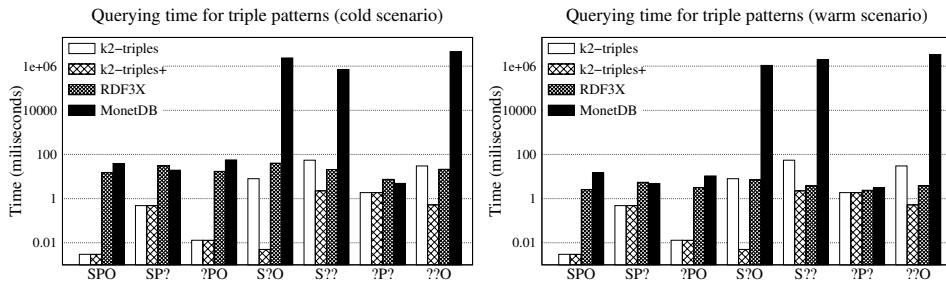


Figure 8.16: Solution time (in milliseconds) for triple patterns in dbpedia.

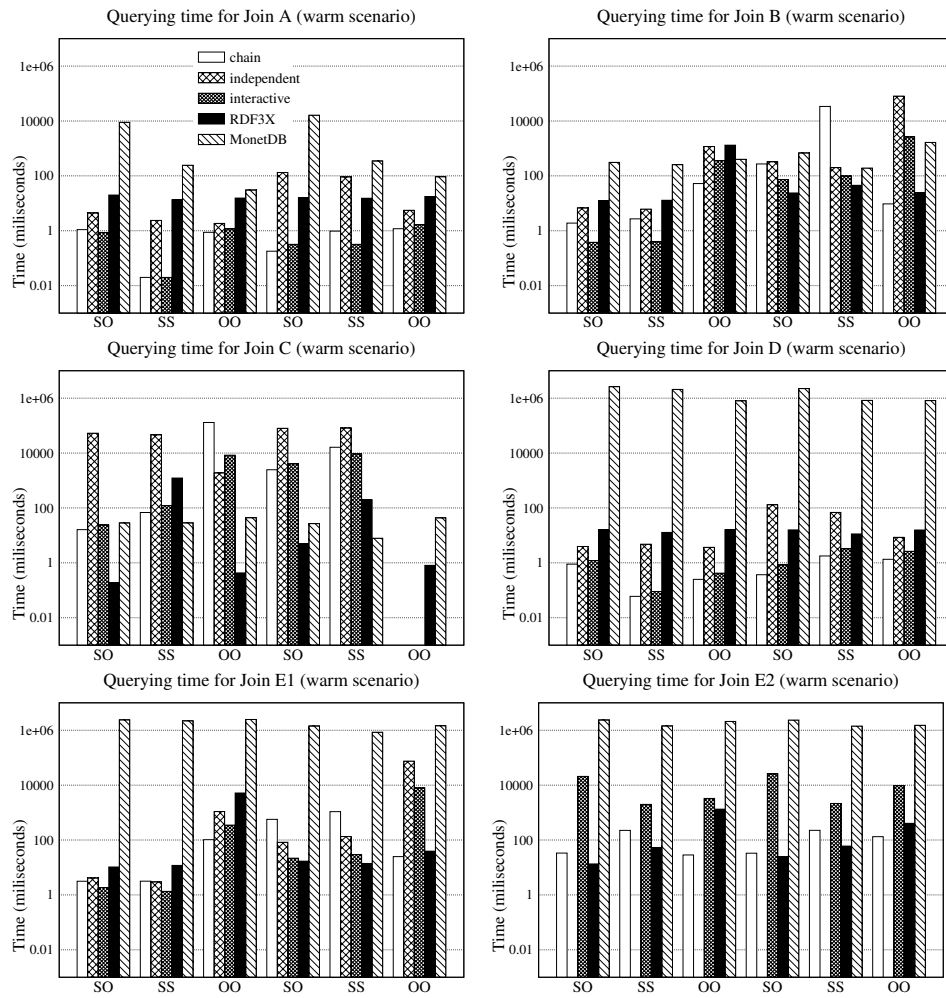


Figure 8.17: Solution time (in milliseconds) for joins A-E2 in dbpedia (warm scenario).

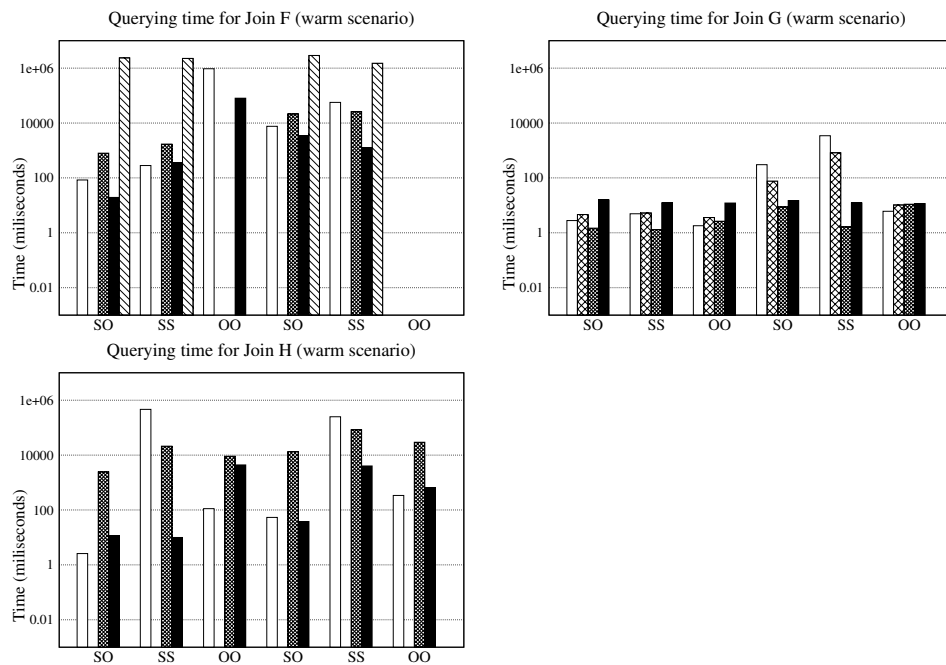


Figure 8.18: Solution time (in milliseconds) for joins F-H in dbpedia (warm scenario).

it pays the penalty of solving a pattern with unbounded predicate. Finally, *interactive* is the fastest choice for **Join G**, although *chain* overcomes it for OO joins. While K^2 -triples⁺ is always faster than RDF3X in all cases, it is worth noting that differences are reduced due to the need of solving two patterns with unbounded predicate. The performance of the vertical partitioning in MonetDB collapses (no times are drawn in this class).

- The joins B, E1 and E2 have a not joined node (subject or object) unbounded. K^2 -triples⁺ and RDF3X share the lead in these experiments, whereas MonetDB remains competitive only in Join B, although it is never the best choice. For **Join B** and **E1** K^2 -triples⁺ is the best choice for joins generating *small* intermediate result sets: *chain* is fastest for OO, and *interactive* for SO and SS. RDF3X overcomes K^2 -triples⁺ when *big* intermediate result sets are obtained, although our *chain* evaluation obtains the best performance for OO joins. On the other hand, **Join E2** and **H** give similar conclusions. RDF3X always achieves the best times, except for OO joins, in which *chain* evaluation is the most efficient choice again. In this case, *interactive* evaluation is less competitive because it performs multiple range queries.
- Finally, joins C and F represent queries with two unbounded nodes. In **Join C**, RDF3X is the best choice for SO and OO joins, whereas MonetDB wins for SS. Note that our approach remains competitive for SS and SO, but its performance is significantly degraded for OO. In **Join F**, our *chain* evaluation competes with RDF3X for the best times, overcoming it for SS *small*. However, this turns out to be the most costly query; note that no technique finishes on OO joins involving *big* intermediate results.

Summarizing, K^2 -triples⁺ excels when triple patterns provide values for the non-joined nodes, and it is clearly scalable when predicates are provided as variables. Thus, in general terms, a query optimizer using K^2 -triples⁺ must favor firstly joins A, D or G; then joins B, E, and H; and finally joins C and F. In any case, joins involving small intermediate result sets are always preferable over those generating big intermediate results.

These findings also apply, in general form, for the remaining datasets in our setup. We show the join performance figures for the remaining datasets in our setup: jamendo in Figures 8.19 and Figure 8.20 (that discards all times over 100,000 milliseconds), dblp in Figure 8.21 and Figure 8.22 (discarding all times over 10⁶ milliseconds) and geonames in Figures 8.23 and Figure 8.24 (that discards all times over 10⁶ milliseconds). All these numbers are obtained in warm state because solution times for RDF3X and MonetDB are less competitive in cold scenarios.

It is worth noting that K^2 -triples⁺ overcomes the other techniques for the smallest dataset (jamendo), dominating the comparison in most cases, and coming

very close to RDF3X in Joins C and F. Another interesting aspect is that MonetDB is the one profiting most from the reduced number of predicates; it reports the best performance in some particular cases.

8.7 Summary

In this chapter we presented a new indexing technique for RDF datasets, that we called K^2 -triples. It follows a vertical partitioning approach, a very common strategy used in other RDF stores in the State of the Art. We represent the triples of each predicate with a K^2 -tree (a data structure that stores binary relationships in a very compact way). We also present some additional indexes to improve the efficiency of the queries with unbounded predicate, which are the main weakness of the vertical partitioning approaches.

We first review the State of the Art in RDF stores. Then, we presented our structure K^2 -triples and the additional index SP and OP. We implemented algorithms to solve simple triple patterns and pair joins in an efficient way. We experimentally evaluated the spatial and temporal results of our system compared with relevant RDF stores in the State of the Art. The results show that our proposal is the best compressed approach and it obtains very competitive results in triple and join query operations.

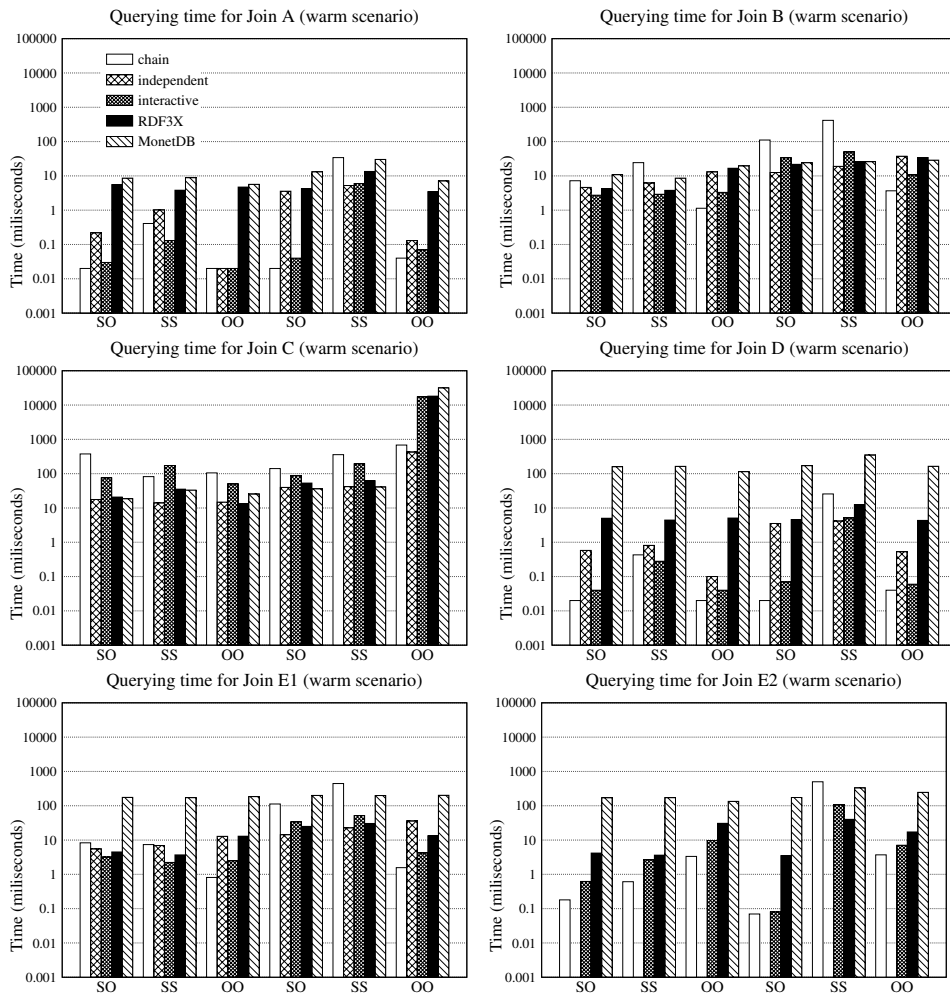


Figure 8.19: Solution time (in milliseconds) for joins A-E2 in jamendo (warm scenario).

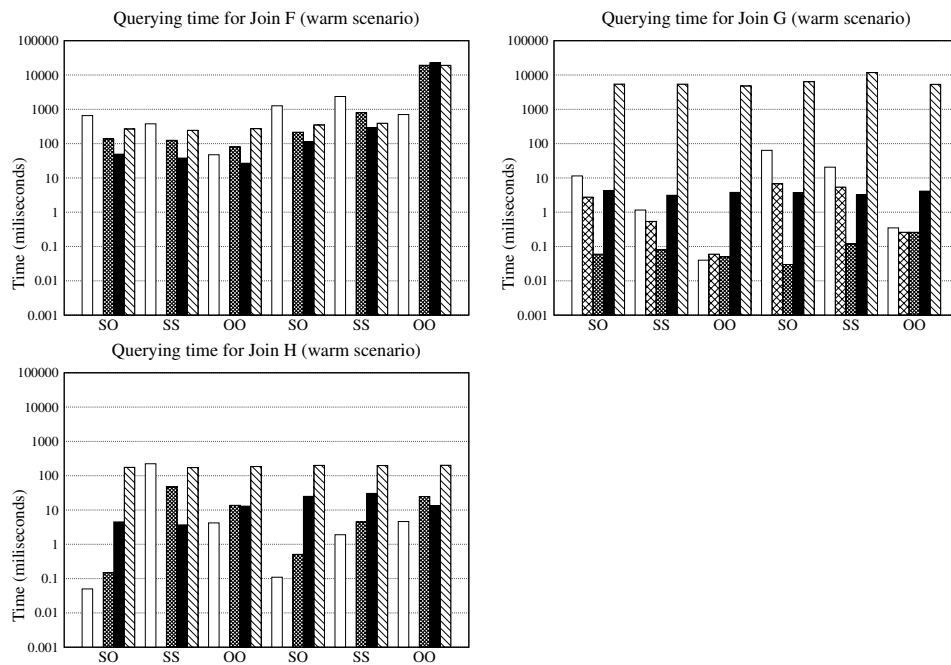


Figure 8.20: Solution time (in milliseconds) for joins F-H in jamendo (warm scenario).

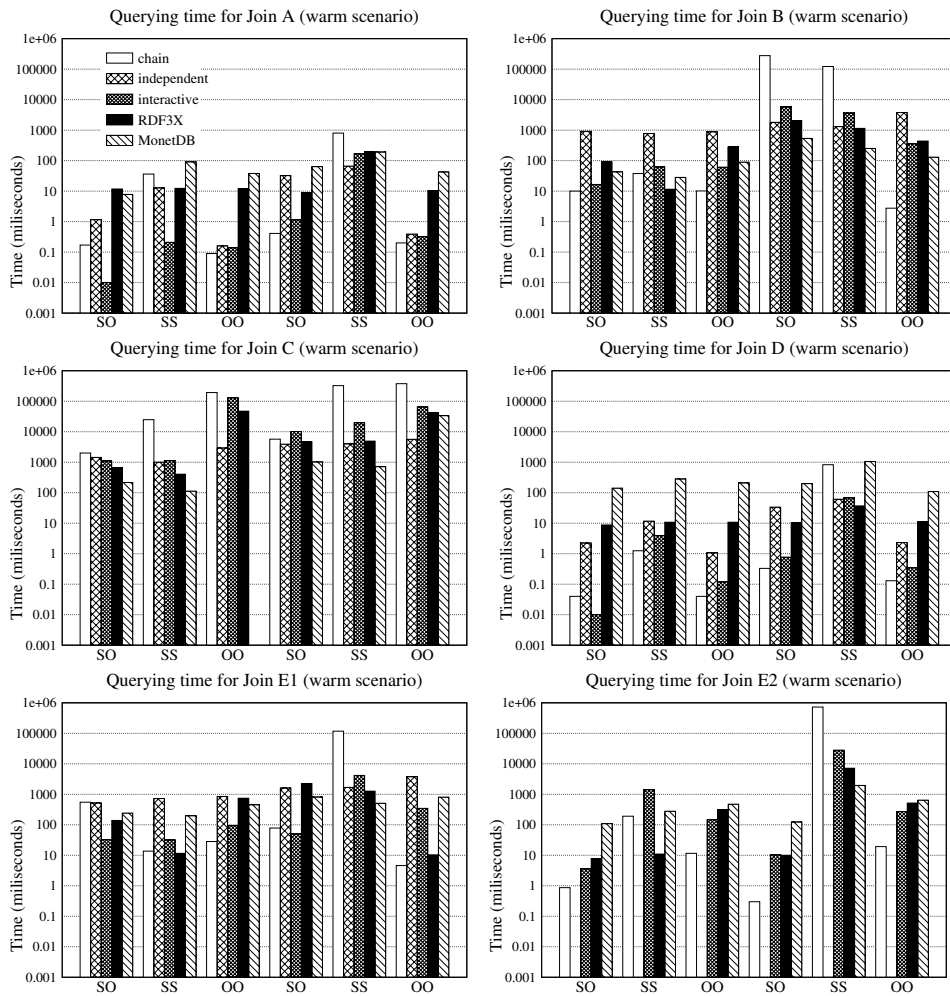


Figure 8.21: Solution time (in milliseconds) for joins A-E2 in dblp (warm scenario).

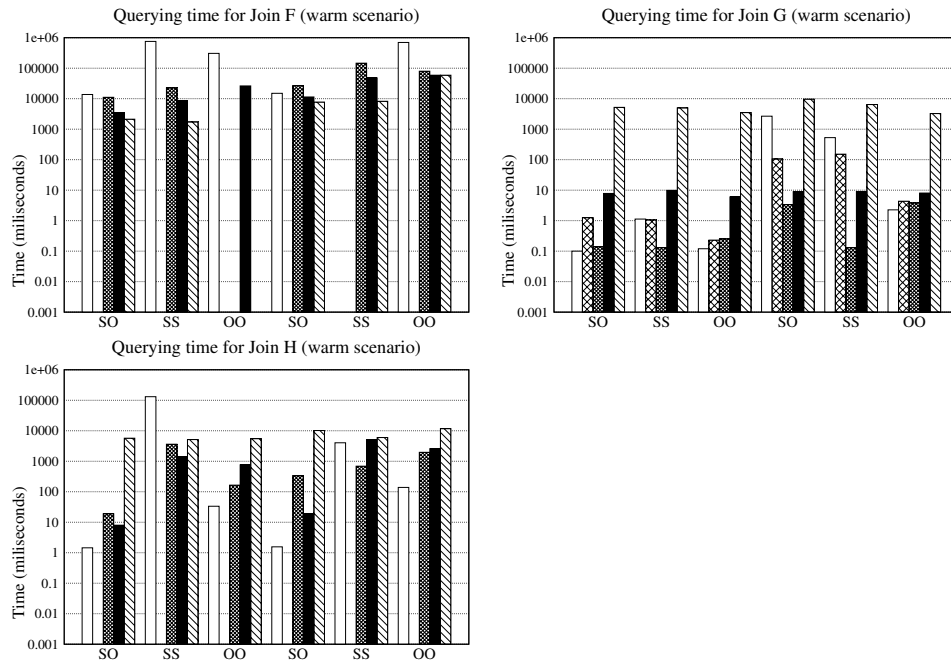


Figure 8.22: Solution time (in milliseconds) for joins in dblp (warm scenario).

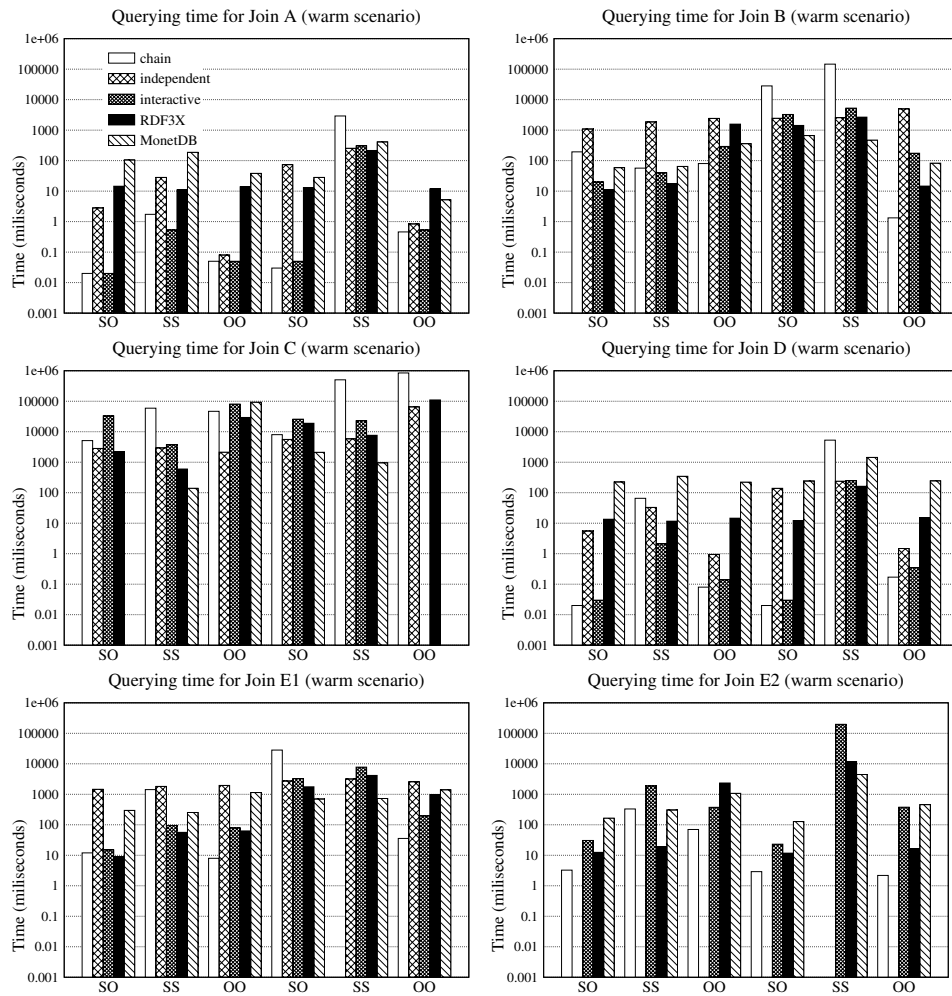


Figure 8.23: Solution time (in milliseconds) for joins A-E2 in geonames (warm scenario).

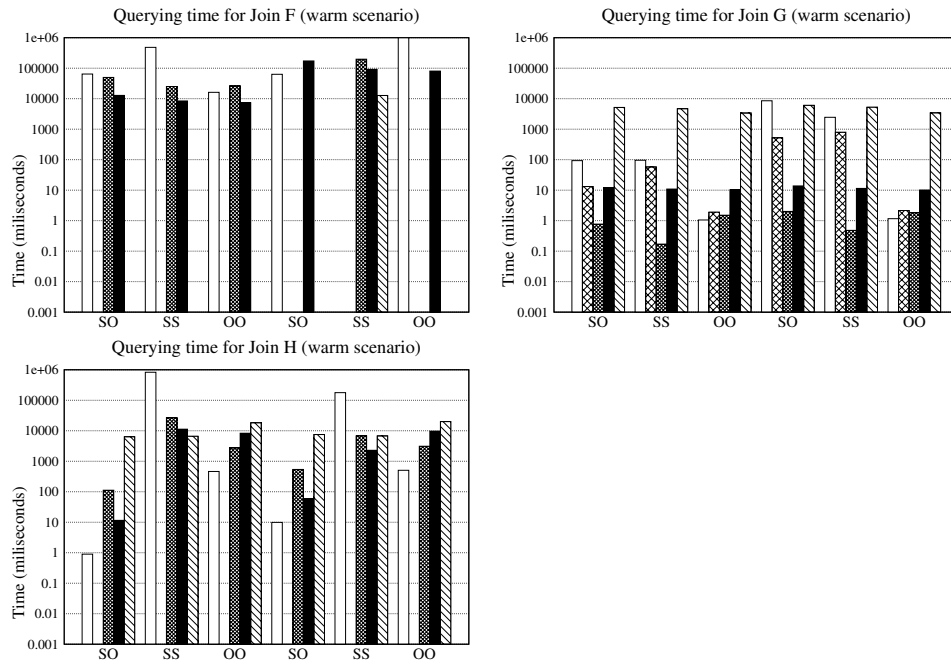


Figure 8.24: Solution time (in milliseconds) for joins F-H in geonames (warm scenario).

Chapter 9

Interleaved K^2 -tree

In this chapter we propose a compact structure to store and query ternary relationships. This new structure is an evolution of the K^2 -tree that provides indexing capabilities over the three dimensions. It is specially designed to represent and manage data where the three dimensions are not equally sized, being one of them smaller (with a lower number of different values) than the others.

Many contexts can be modelled as a ternary relation. For instance, the representation of the temporal evolution of a simple graph can be seen as a ternary relation $N \bowtie N \bowtie T$, where a triple (n_i, n_j, t_k) represents that an edge starting in the node n_i and pointing to the node n_j exists in the temporal instant t_k . Raster data that represents a matrix where each cell (x, y) has associated a value, can also be modelled as a ternary relation. They are commonly used in Geographic Information Systems to store different attributes of a region, like the elevation of the land or the atmospheric pressure. Therefore, a raster is a ternary relation $X \times Y \times Z$ where dimensions X and Y are the coordinates of the region and dimension Z corresponds to the attribute that the raster data describes. The standard RDF is also a good example of ternary relation, where data is structured in triples $(s, p, o) \in \textit{Subject} \bowtie \textit{Predicate} \bowtie \textit{Object}$ representing that the subject s takes value o for the predicate p .

Previous examples are ternary relations that present an asymmetric distribution along the spatial domain $(X \times Y \times Z)$ defined by the three dimensions. For instance, in raster data, each coordinate (x, y) is only related to a single value in the Z dimension (by definition). However, given an $x \in X$ and $z \in Z$ values, many $y \in Y$ can be related to those two values. In addition to that, due to the different nature of the dimensions taking part in the ternary relation, their size (number of different values) are seemingly different. In Raster Data, the number of different values that

the attribute describing the region can take (Z) is expected to be smaller than the coordinates of the space (X and Y dimensions). Furthermore, the access pattern to the data of a ternary relation will be different in the three dimensions. For instance, queries asking for values inside a squared subregion of the space are more common than queries asking for which $y \in Y$ values take the attribute z in the row x .

The solution proposed in this chapter is specially designed to store such asymmetric ternary relations, where one of the dimensions has a small number of different values.

We designed this compact structure to store RDF data sets, which usually contain a reduced number of predicates regarding to the subjects and the objects that the collection describes. However, the design of the structure and the generic navigation over the three dimensions of the relation makes this structure a useful tool in other applications where data can be also modelled as an asymmetric ternary relation with one of the dimensions smaller than the other two.

Several approaches were followed to represent ternary relations in the State of the Art. Most of them are focused on solving the characteristics of a specific domain. That is the case of RDF graphs, which we reviewed in Section 8.2. Many RDF stores follow a vertical partitioning approach, like SW-store described in Section 8.2.1.3. In that kind of approaches, the problem of storing a ternary relation is reduced to storing several binary relations, one for each value of the partitioning variable.

In this chapter we propose the Interleaved K^2 -tree, a compact structure that, starting from a vertical partitioning of the data, considers the representation of each binary relation in a single K^2 -tree. However, instead of storing those K^2 -trees independently, it gathers the three dimensions in a single tree which improves the indexation of the partitioning variable, keeping the indexation over the other two dimensions of the data.

9.1 Our proposal

Interleaved K^2 -tree (IK^2 -tree) represents a ternary relation defined as a set of triples $\{(x_i, y_j, z_k)\} \subseteq X \times Y \times Z$. This structure is designed for ternary relations where one of the dimensions is smaller than the other two, understanding the size of a dimension as the number of different values that a triple can take for that dimension (also named variable).

Considering Y is the smaller variable, the Interleaved K^2 -tree starts from the representation of the ternary relation in $|Y|$ binary relations, one for each different value $y_j \in Y$. This is the same vertical partitioning that was performed for K^2 -triples, explained in Chapter 8. Rows of that adjacency matrix will represent the values of

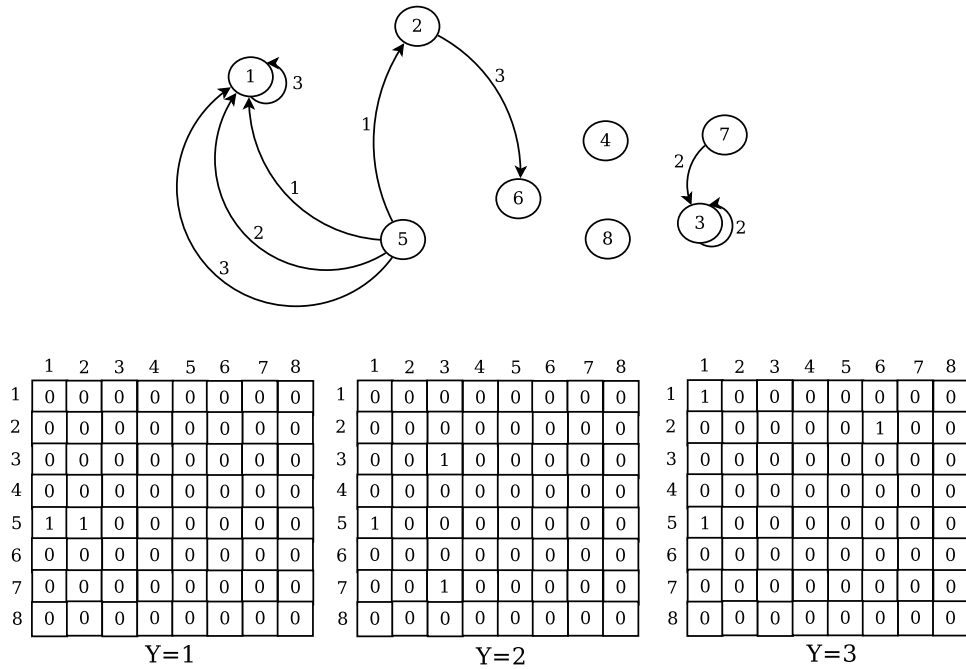


Figure 9.1: Vertical partitioning over a labelled multigraph

the variable X , while columns represent the values of the variable Z . A one in a cell (x_i, z_k) of the adjacency matrix y_j implies the existence of the triple (x_i, y_j, z_k) . Figure 9.1 (top) shows an example of weighted multi-graph, which produces three different binary relationships, one for each different value of Y .

Each adjacency matrix could be stored through an independent K^2 -tree in order to compose a complete system containing the full ternary relation. However, instead of building those multiple K^2 -trees, we propose a new structure. It is an evolution of the K^2 -tree that, also following the vertical partitioning philosophy, divides the ternary relation in several binary relations. However, it gathers all of them in a unique tree, providing indexing capabilities in the three dimensions.

9.1.1 Data structure

Interleaved K^2 -tree starts from the $|Y|$ adjacency matrices that represent a ternary relation. However, instead of storing each adjacency matrix in a different K^2 -tree, this structure will gather the full ternary relationship in just one tree. Interleaved K^2 -tree will be a K^2 -ary tree, where each node of this new tree will contain one bit

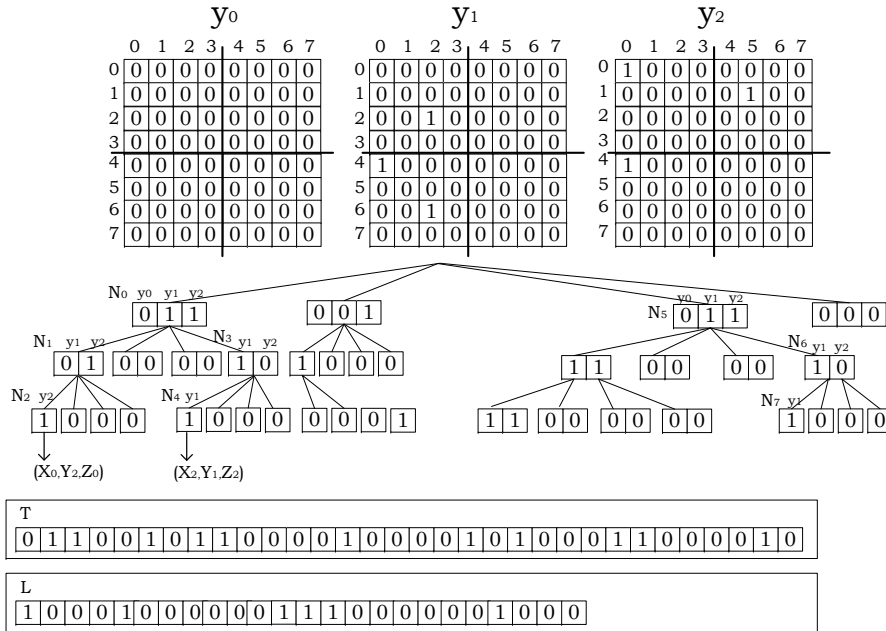


Figure 9.2: A ternary relation represented with the Interleaved K^2 -tree

for each different value of the partitioning variable.

The K^2 nodes of the first level contain $|Y|$ bits, one per each different value $y_j \in Y$. Inside a node, the i -th bit will be a 1 if the submatrix which represents that node has at least a 1 for the y_i value. Otherwise, this position will contain a 0.

Each node n_i will have K^2 children nodes, and the number of bits of each children is given by the number of bits with value 1 of its parent. So, for a node N_i with m ones, each one of their K^2 children nodes contains m bits, one per value y_j that has at least a one in the matrix corresponding to N_i .

Figure 9.2 shows an example of the IK^2 -tree structure representing the graph shown in the Figure 4.3. The partitioning variable Y takes three different values, so three adjacency matrices represent the binary relations that each different value of variable Y produces. Bottom of the figure shows the IK^2 -tree structure. A value $K = 2$ was chosen for that example, so 4 root nodes appear in the first level of the tree. Each node is composed by three bits, one per value of Y . For instance, the first bit of the first root node (N_0) contains a 0, meaning the top-left submatrix of the relation y_0 does not contain any one. However, the second bit of N_0 is a one because the top-left submatrix of y_1 contains at least a one (actually in the cell

(2, 2)). The other three root nodes also contain three bits with the same meaning regarding to the other three submatrices of that division.

It was shown how the first level of the tree is computed. Next, given a node of this tree, we describe how its children are built. Suppose a root node of size $|Y|$ with m bits with one value. Their K^2 children nodes will be created as a result of the matrix division, but each children will contain only m bits, that is, the number of *active* values of the parent, since the remaining $|Y| - m$ elements are empty matrices that are not represented in lower levels. If all values of a node are *zero*, then that node will not produce any child. Otherwise, the process continues with a recursive division of the matrix (as in the original K^2 -tree), building the tree in a top-down process until the leaves are reached. In that way, any non empty node produces K^2 children nodes with as many bits as ones the parent contains. Since the elements in a node are always ordered by its corresponding value for the variable Y , by performing a top-down traversal will be easy to know which value of Y is representing each bit of a node.

Figure 9.2 shows that the children of node N_0 contain 2 bits each, since only the values y_1 and y_2 are *ones* in N_0 . Thus, the first bit of N_1 represents the value y_1 , which is a *zero*, showing that this sub-matrix does not contain any value for y_1 . However, the second bit is a *one*, representing the sub-matrix corresponding to y_2 contains at least a one in the node, which is in the cell $(0, 0)$. Consequently, each children of N_1 (like N_2) only contains one bit for the y_2 value. Just as in the original K^2 -tree, the coordinates that represent each bit can be inferred by its position in the topology of the tree (which provides information about X and Z variables).

The resulting tree is stored exactly in the same way as the original K^2 -tree. The last level will be stored as a bitmap L and the rest of the levels are stored in a bitmap T following a level-wise traversal over the tree. An additional structure to perform *rank* operations is built over T . As in the K^2 -tree, we store the tree in two different bitmaps: *rank* operation supports is only needed to the intermediate levels, since *rank* operation is used to search the children of a node.

The original K^2 -tree structure has the interesting property that the position of the first children of a node can be computed directly. Since each *one* of the tree produces K^2 children, and the elements are stored in the bitmap ordered by levels, the first child of a node in the position i in the bitmap is in the position $rank_1(T, i - 1) * K^2$ of the bitmap $T : L$.

The IK^2 -tree also presents this direct access to the children of a node. Although nodes in IK^2 -tree have a variable size (depending on the *active* values in the parent), the navigation over the tree is quite similar. Observe that a *one* in a node is still producing K^2 bits in the lower level, although those bits can be not consecutive (because each bit is placed in a different node and nodes can have several bits). As a result, given a node, the position of their children in the bitmap can be computed

through a simple formula. Consider a node starting at position i with $m > 0$ one bits. Then, its first child node of size m starts in the position $rank_1(T, i - 1) * K^2 + adjust$. Starting at this position, the next $K^2 * m$ bits are representing the children of that node (K^2 nodes of size m). Note that the formula is exactly the same as in the original structure, except for the value $adjust = |Y| * K^2$ which is a correction factor needed as a consequence of having a first level of size $|Y| * K^2$ instead of K^2 as in the original structure.

9.1.1.1 Leaves compression with DAC

Original K^2 -tree structure proposes an interesting improvement to represent last levels of the tree in a very compact way. An arbitrary number of last levels can be collapsed and statistically compressed using a submatrix vocabulary, as Section 2.3.4.2 describes.

This improvement also be also applied in the Interleaved K^2 -tree structure. Note that in Interleaved K^2 -tree structure each node has an arbitrary number of bits and each *one* of that node produces K^2 elements in next level. Therefore, it is not possible to encode the submatrix vocabulary of complete nodes, since a node in the third level (starting from the bottom of the tree) collapses a cube of dimensions $K^2 \times K^2 \times m$ where m is the number of *ones* of that node. The suitable way of having a vocabulary of equally-sized submatrices is encoding the $K^2 \times K^2$ submatrix that each *one* of a node would individually produce. In that way, the considered submatrices are a subregion of an individual adjacency matrix. All of these submatrices are used to compute a global submatrix vocabulary. They are represented with DAC.

Note that, taking into account that each individual submatrix corresponds to a specific value of the variable Y , those submatrices could be encoded using a different vocabulary for each y_i . However, in order to avoid storing $|Y|$ vocabularies, all the matrices are encoded with the same vocabulary.

9.1.2 Navigation

The previous section describes the internal representation of the IK^2 -tree structure through the bitmaps T and L (with *rank* access to the bitmap T). Next, the implementation of the basic navigation operations over this structure is described.

Basic operations are denoted by a triple pattern (x, y, z) . We use the same notation used to describe the simple triple patterns of the K^2 -triples structure in Chapter 8. If $x \in X$, a triple will be a result of that pattern only if it takes the value x for the variable X . On the other hand, if $x = ?$, the pattern does not restrict

the valid values of X (is unbounded). Then, any value for X is a valid solution for that query pattern. The same constraints can be specified in Y and Z .

Using this query pattern notation, 9 different query patterns are defined based on whether a variable is bounded or unbounded. The query pattern groups range from the query that searches for one triple (x_i, y_j, z_k) to the totally unbounded query $(?, ?, ?)$ that returns all the triples contained in the dataset. Given the nature of our structure, the value taken for the partitioning variable Y strongly determines the pattern resolution. Therefore, two big families of query patterns will be considered: query patterns with bounded partitioning variable (that is, queries asking for a specific value of the partitioning variable) and query patterns with unbounded partitioning variable (searching for triples with any value for the partitioning variable). The next sections provide implementation details of these two kinds of operations.

9.1.2.1 Query patterns with bounded partitioning variable

This query pattern family includes queries that specify a fixed value for the partitioning variable. The query patterns that belong to this group are (x_i, y_j, z_k) , $(?, y_j, z_k)$, $(x_i, y_j, ?)$ and $(?, y_j, ?)$. First of all, the implementation of the query (x_i, y_j, z_k) will be illustrated. After that, details about how this algorithm is extended to the remaining query patterns will be provided.

The pattern (x_i, y_j, z_k) searches for an individual triple of the dataset so, as in the original K^2 -tree, a single branch is explored in each level of the tree. In the corresponding node for each level of the tree, the bit corresponding to the y_j value is checked. If it is a *one*, the process continues going down the tree. Otherwise, the operation is finished and no results are returned. In order to know the position that the bit of the variable y_j occupies in each node, additional data has to be managed. In this way, in the first level of the tree the y_j value is in the j position of the node. In its children, the location of the bit y_j depends on the number of *ones* in the parent. Suppose that the root contains m bits between the first position and the position of the element y_j (that is, the one of y_j is the m one in the parent node). Then, the m -th position of its children correspond with y_j . The operation continues in the same way traversing down the tree, so in each level a count of the previous *ones* to the element y_j in the node has to be computed in order to know the position of y_j in the next level.

Figure 9.2 shows the nodes involved to solve the query (x_6, y_1, z_0) . In the root, the node N_5 is explored (because it corresponds to (x_6, z_0)) and the second bit (corresponding to y_1) is checked. It is a *one* and it is the first one of the node, so in its children the position for y_1 is the first one. The corresponding node for (x_6, z_0) is N_6 and the first bit is checked. The process continues until the leaves, where

the only bit of N_7 is checked and it is a *one*, so the triple (x_6, y_1, z_0) exists in the dataset.

This is the basic mechanism of the query patterns with bounded partitioning variable. It can be generalized to the other query patterns in this group. The query pattern $(?, y_j, z_k)$ searches for the reverse neighbors of z_k in the relation of y_j . It is solved similarly to the (x_i, y_j, z_k) pattern. The only modification is that two branches have to be checked for each explored node with a *one* in the value corresponding to y_j . This change is due to the X variable (which usually prunes some of the branches) is unbounded. The query pattern $(x_i, y_j, ?)$ is totally symmetrical: two branches are also explored for each valid node since one of the variables used to prune the branches is also unbounded. Finally, the query pattern $(?, y_j, ?)$ is executed following the same philosophy but, since the variables X and Z are unbounded, the K^2 children are explored for any valid node, only discarding branches when the bit corresponding to y_j is *zero* in a specific node.

9.1.2.2 Query patterns with unbounded partitioning variable

The second kind of query patterns presents the variable Y unbounded, so it does not restrict the value that the partitioning variable Y has to take. The patterns included in this group are $(x_i, ?, z_k), (?, ?, z_k), (x_i, ?, ?)$ and $(?, ?, ?)$. The common characteristic of that group is that, since no value is given for the partitioning variable, all values for each explored node have to be checked, maintaining the list of active Y values in each step. As in the previous explanation of the bounded partitioning variable query patterns, the algorithm of the most restricted query is given first and the remaining query patterns will be described as an extension of the same mechanism.

The query pattern $(x_i, ?, z_k)$ searches for the values of Y that relate the pair of elements (x_i, z_k) . That is, it asks for the binary relations that relate those elements. Given that specific values for variables X and Z are provided, the process starts from the top of the tree descending for one branch in each step, which is the child that the given values x_i and z_k fit with. The process continues down the tree until we reach the leaves unless no *ones* appear in the node explored for a previous level of the tree. Note that in each level, the size of the node can be reduced, because only the values for the variable Y with a *one* value in the parent are represented in the children. So, in order to know the remaining active values for Y when the leaves are reached, a list of the active values has to be managed and updated in each level. We name A_j the list of active values in the node N_j . So, for a root node N_j , its list A_j always has $|Y|$ elements, where $A_j[i] = y_i$ for each $i \in 1 \dots |Y|$. The list of active values A_k of a child of the node N_j , N_k , contains m elements where m is the number of ones in the node N_j . In that way the list of active values is built as $A_k[i] = A_j[\text{rank}(N_j, i)]$. It means that the i -th position of the children active list contains the value corresponding to the i -th *one* in the parent node. It is easy to see

that, given the active list of the parent (A_j) and the node N_j the list of the active values for the children of the nodes can be computed by a sequential checking over the bits of the node.

For instance, returning to the example of the Figure 9.2, the query $(x_2, ?, z_2)$ starts by checking the first node of the tree, N_0 , with a list of active values $A_0 = \{0, 1, 2\}$, because it is a root node (so it initially contains all the different values for the variable Y). The values for the variables X and Z determine that the explored child is N_3 . Since in N_0 only the second and third bit have value *one*, the list of active values for N_3 is $A_3 = \{1, 2\}$, which determines that the first bit of N_3 (which is a *one*) corresponds with y_1 and the second bit (a *zero*) corresponds to y_2 . As N_3 continues having at least a *one* value, the process continues traversing down the tree until the leaves level, where node N_4 is explored. The list of active values of the node N_4 is computed from the bitmap of N_3 and the list of active values A_3 . In that way, only the first bit of N_3 is a *one*, so $A_4 = \{A_3[1]\} = \{1\}$. Then, the bit *one* located in N_4 corresponds to y_1 and the final result of the query $(x_2, ?, z_2)$ is $\{(z_2, y_1, z_1)\}$.

The remaining query patterns in this group follow the same philosophy of maintaining the list of active values of each explored node. Query pattern $(?, ?, z_k)$ recovers the reverse neighbors of the node z_k for any value of the partitioning variable. It is performed as a top-down traversal where two children are explored for each valid node of the tree. Since the partitioning variable is unbounded, a checked node is valid to continue the exploration when at least a bit is set to *one* in that node. The query pattern $(x_i, ?, ?)$ follows the same idea, exploring always two children of each valid node. Finally, the query $(?, ?, ?)$ consists in exploring all the dataset, checking all the nodes and descending for all the branches of the tree, taking count of the list of active values for each node of the tree.

9.1.3 Analysis

Several details can be given about the properties of this new structure. First of all, Interleaved K^2 -tree is an aggregation and reorganization of the same bits used in a multiple K^2 -tree representation where an individual K^2 -tree represents the adjacency matrix of each different value of Y . Thus, the spatial cost of both structures is exactly the same. However, the way in which this structure aggregates in a same node the same submatrix for all the different values of Y will have important consequences in the efficiency of some operations. An analysis will be given in the Experimental Evaluation, but the intuitive idea comes from taking advantage of the fact that $|Y|$ submatrices for a range of X and Z values are consecutively joined in the node that represents that matrix. Therefore, in opposition to the $|Y|$ rank operations for obtaining the children of those nodes in a vertical partitioning environment, a single access in the tree is performed to obtain all the values for that submatrix. However,

as a return, some additional operations are needed in this structure to manage the variable size of the nodes.

Note that although the three dimensions are represented in the tree, the indexation capabilities are different in the three variables. X and Z dimensions directly support the navigation over the tree, because the queried values for those variables determine the branches of the tree that have to be consulted, easily pruning the tree (branches that are out of the scope of the query are discarded). Therefore, X and Z have an efficient indexation. Considering the dimension Y , we can observe that pruning the tree by the Y variable is not straightforward, because the different values of Y are distributed over all the nodes so no branches can be filtered only by the Y values. Instead, the specific values of each node of the tree have to be checked in order to know if a branch is valid for the queried value. Furthermore, since the nodes have a variable size, the active values for the ancestor nodes have to be taken into account to know the bit which corresponds to the queried Y value. Therefore, the indexation of that dimension is not as efficient as in the case of X and Z . This concept is important to decide the role of the variables of a ternary relation in the Interleaved K^2 -tree. In general, the most convenient partitioning variable will be the smallest one, but this criterion can change depending on the needs of the domain and the priority and frequency of usage of each kind of query in that context.

9.2 Experimental evaluation

In this section we analyze the spatial and temporal results achieved with the data structure we proposed (Interleaved K^2 -tree). We test this data structure with the four RDF dataset used in Chapter 8: Jamendo¹, Dblp², Geonames³ and DBpedia⁴. We implement the 7 simple triple patterns: (S, P, O) , $(S, P, ?)$, $(?, P, O)$, $(?, P, ?)$, $(S, ?, O)$, $(S, ?, ?)$, $(?, ?, O)$. Our machine is an AMD-PhenomTM-II X4 955@3.2 GHz, quad-core, 8GBDDR2, running Ubuntu 9.10. The code was developed in C, and compiled using gcc (version 4.4.1) with optimization -O9.

Table 9.1 shows the size of the different RDF datasets and the compression results obtained by the IK^2 -tree. We compare our structure against our proposal for RDF described in chapter 8 K^2 -triples and K^2 -triples⁺ and RDF-3X. First columns show the number of triples and the number of predicates that each dataset contains. It determines the maximum number of bits of the nodes in the IK^2 -tree and the number of individual K^2 -trees that have to be created for the K^2 -triples and K^2 -triples⁺.

¹<http://dbtune.org/jamendo>

²<http://dblp.l3s.de/dblp++.php>

³download.geonames.org/all-geonames-rdf.zip

⁴wiki.dbpedia.org/Downloads351

Dataset	$ Triples $	$ Predicates $	K^2 -triples	K^2 -triples ⁺	IK^2 -tree	RDF-3X
Jamendo	1,049,639	28	0.74	1.28	0.74	37.73
Dblp	46,597,620	27	82.48	99.24	84.04	1,643.31
Geonames	112,235,492	26	152.20	188.63	156.01	3,584.80
DBpedia	232,542,405	39,672	931.44	1178.38	788.19	9,757.58

Table 9.1: Space comparison for different RDF datasets (in MB)

We can observe that K^2 -triples is the most compressed structure for Jamendo, DBLP and Geonames; while for the DBpedia dataset the IK^2 -tree achieves the best compression. Since the IK^2 -tree contains just a reordering of the bits of the individual K^2 -tree, their space differences are due to the multiple K^2 -tree compresses the last levels of each tree independently, and IK^2 -tree uses a global vocabulary, as explained. In DBpedia, the unique vocabulary saves some redundancy, but in the smaller datasets the specific vocabularies obtain better compression. On the other hand, K^2 -triples⁺ achieves worse compression than K^2 -triples and IK^2 -tree, because it includes the index SP and OP additionally. The three representations based on K^2 -trees are much more compressed than RDF-3X.

Regarding to the query efficiency, we test all the basic triple patterns. Table 9.2 shows the results for Geonames (as a representative domain with few predicates) and for DBpedia (as an example with many predicates). For each simple pattern, we show the average time per query (500 queries were executed for each pattern).

Results show that, for bounded predicates, K^2 -triples is the fastest, and the IK^2 -tree is about twice as slow. This is expected, because the navigation in the IK^2 -tree is slightly more costly: it must execute additional *rank* operations to compute the number of active bits in each child. In general, RDF-3X is slower.

For patterns with unbounded predicates, instead, the IK^2 -tree obtains better results than K^2 -triples, especially for datasets with many predicates like DBpedia, because it partially solves the problem of the vertical partitioning. However, K^2 -triples⁺ obtains the best results, specially in DBpedia, due to the optimization carried out by the indexes SP and OP , but in return of an additional spatial cost. RDF-3X is far less efficient.

9.3 A lazy evaluation

Section 9.1.2 showed how the different basic query patterns are executed depending on the partitioning variable (it can be unbounded when it can take any value or bounded when a specific binary relation for a given partitioning variable value is

		Geonames			
Category	Pattern	K^2 -triples	K^2 -triples ⁺	IK^2 -tree	RDF-3X
Bounded Predicate	(S,P,O)	1.8	1.8	3.9	2,346.5
	(S,P,?)	64.9	64.9	110.4	4,882.3
	(?,P,O)	0.1	0.1	0.3	0.6
	(?,P,?)	0.4	0.4	0.5	0.7
Unbounded Predicate	(S,?,O)	5.3	3.6	4.4	6,118.6
	(S,?,?)	95.0	70.0	69.7	229.7
	(?,?,O)	240.0	125.2	187.0	2,473.1

		DBpedia			
Category	Pattern	K^2 -triples	K^2 -triples ⁺	IK^2 -tree	RDF-3X
Bounded Predicate	(S,P,O)	3.2	3.2	6.2	2,532.4
	(S,P,?)	358.7	358.7	608.5	4,117.3
	(?,P,O)	0.6	0.6	1.6	143.9
	(?,P,?)	0.7	0.7	1.6	0.9
Unbounded Predicate	(S,?,O)	7,186.1	4.7	155.2	6,330.6
	(S,?,?)	3,925.2	161.6	911.2	272.3
	(?,?,O)	10,918.1	186.0	1,444.6	1,377.9

Table 9.2: Time evaluation of simple patterns for RDF, in μ s per result

given). The main difference of implementation between the two kind of queries is that the queries with unbounded partitioning variable check all the bits of each explored node (maintaining the list of active values of each explored node, which is used to compute the list of active values of its children). In that way, when the level of the leaves is reached, each leaf node (which can be composed by one or more bits set to *one* in a node) is translated by using its associated list of values in order to know which values of Y relate the element (x_i, z_k) that is represented in that leaf. Therefore, some list of active values are used to compute the value of Y suitable for each resulting triple.

There are datasets that contain many different values for the partitioning variable. In that case, experimental evaluation proves that this process can be very expensive, because it involves a sequential check over the nodes to compute their corresponding list of active values. Furthermore, this costly operation may be finally not useful many times, because the branch that is being explored would be finally discarded in lower levels of the tree. Therefore, all this greedy process of computing, for each level, which value of Y corresponds with each bit of the current nodes, could be avoided for such branches that finally do not produce any result.

In this section, we propose a *Lazy Evaluation*, which consists in postponing the computation of the corresponding value for the variable Y until making sure that this calculus is necessary for some result of the query. This navigation strategy is designed as an alternative to the greedy evaluation of the query patterns with unbounded partitioning variable for datasets with a large $|Y|$.

In the lazy approach a top-down traversal is performed, as usual, navigating over the branches determined by the values of the variables X and Z . The difference will be that the list of active values is not computed for each node. In order to continue the navigation over the children nodes, only the number of bits set to *one* in the parent is necessary (but no its correspondence to Y values). This count of the number of *ones* in a node N_i can be performed by a sequential check of the bits or through two rank operations $rank_1(N_j) = rank_1(T, init + |N|) - rank_1(T, init)$ where *init* is the position of the first bit of N_i in the tree (T). Then, the top-down traversal only takes count of how many values of Y remain actives in each node, but no which values are.

Once the leaves are reached, the real needed branches for the result are known. Then, the specific values for the variable Y are computed but only for the branches involved in the result. The process of mapping the Y values starts from the bottom of the tree, computing the list of *relative* values within that node. In that way, a bit *one* in the position m of a leaf node has initially associated the value y_m , which will be transformed in a bottom-up traversal until its real value. Given an intermediate node with a list of *relative* values, they are transformed by checking the bitmap of the parent node. In that way, a *relative* value y_j in a children is mapped to the

position of the j -th *one* in the bitmap of the parent node. The process continues by recursively updating the lists of *relative* values, mapping each value with its position in the parent, until the root is reached. At this moment, the current *relative* values are the real values for the variable Y corresponding to each result.

An optimization of this process can be performed for such queries where dimensions X , Z or both are unbounded. In those query patterns, several branches are explored, so many results could have common ancestor nodes, which have to be explored many times (once for the mapping of relative values of each result). In order to avoid this replication of computation, in each level of the tree where the relative values for results in nodes share the same parent, a previous *merge sort* is performed, obtaining a single list of relative values which is mapped by recovering once the bitmap of the parent. In that way, in each step several lists of results sorted by their Y value are maintained, which are progressively merged when they belong to the same parent node. As a consequence, redundant mappings for the same node are avoided.

We show an example of lazy evaluation for a dataset with three different values for the variable Y through a set of figures. Figure 9.3 shows the initial top-down traversal performed over the tree. For each node, only the number of active values are stored (in order to know the size of the children). In this way, node N_0 contains 2 bits with *one* value so its children has 2 bits each one. N_1 contains one bit with *one* value so its leaves only contain one bit. On the other hand, the two bits with value *one* of N_2 produce leaves with two bits each one. At the end of this process, four results are obtained, which are highlighted in the figure. Their values for the variables X and Z are given by the branch of the tree which they are located. Their values for the Y variable are unknown and a *relative* value is computed as their position in the leaf node they belong. For instance, the leaf (X_2, Y_1, Z_2) is related to the second value of Y because is located in the second position of the leaf node. In this point the down-top traversal starts, shown in Figure 9.4. First, a merge sort is performed to join the lists of results which belong to the same parent node. The element (x_0, y_0, z_0) is in an independent branch. However, the other three results $\{(x_2, y_1, z_2), (x_2, y_0, z_3), (x_2, y_1, z_3)\}$ are merged in the same list: y_0 has one result associated (x_2, z_3) and y_1 contains two results $(x_2, z_2), (x_2, z_3)$. Next step consists of transforming the current relative values in basis to the parent nodes. Then, as the Figure 9.5 shows, the list y_0 with the element (x_0, z_0) is transformed to y_1 since the first bit one in the parent N_1 is in the second position. However, the list $y_0 : (x_2, z_3)$ continues associated to the y_0 value since the first one in the parent N_2 is in the first position. The same happens with $y_1 : (x_2, z_2)$ which is transformed to y_1 because the second one of the parent node N_2 is in the second position. The three lists of this level share the same parent node so they are merged by Y value. Finally, in Figure 9.6, the lists are transformed with the information of N_0 . The first one of N_0 is in the second position and the second one is in the third position, so the final

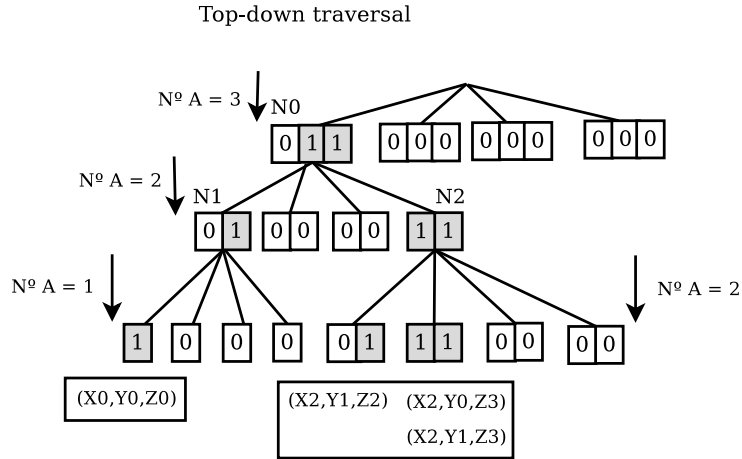


Figure 9.3: First step of the top-down traversal of the lazy evaluation

Simple pattern	K^2 -triples	K^2 -triples ⁺	IK^2 -tree	Lazy IK^2 -tree	RDF-3X
(S,?,?)	3,925.2	161.6	911.2	232.7	272.3
(?,?,O)	10,918.1	186.0	1,444.6	430.8	1,377.9

Table 9.3: Time, in μ s per result, of basic and lazy evaluation in patterns with unbounded predicate on DBpedia

results will be $(x_2, y_1, z_3), (x_0, y_2, z_0), (x_2, y_2, z_2), (x_2, y_2, z_3)$.

We analyzed the overall improvement obtained with the Lazy evaluation strategy. Table 9.2 shows the results obtained for triple patterns with unbounded predicate in DBpedia. Lazy evaluation improves significantly the results for those queries, being up to 5 times faster than the eager algorithm. However, the lazy implementation still is slower than the K^2 -triples⁺.

9.4 Summary

In this chapter we present a new compact data structure to represent ternary relationships, specially designed to such relationships where one of the variables has a smaller number of different values than the other two variables of the relation. This data structure follows a vertical partitioning of the data, transforming a ternary relationship in multiple relationships. Each resulting binary relationship can be represented with a K^2 -tree, just as in the K^2 -triples system described in Chapter 8. However, in this case, all these trees are gathered in a unique tree structure, which

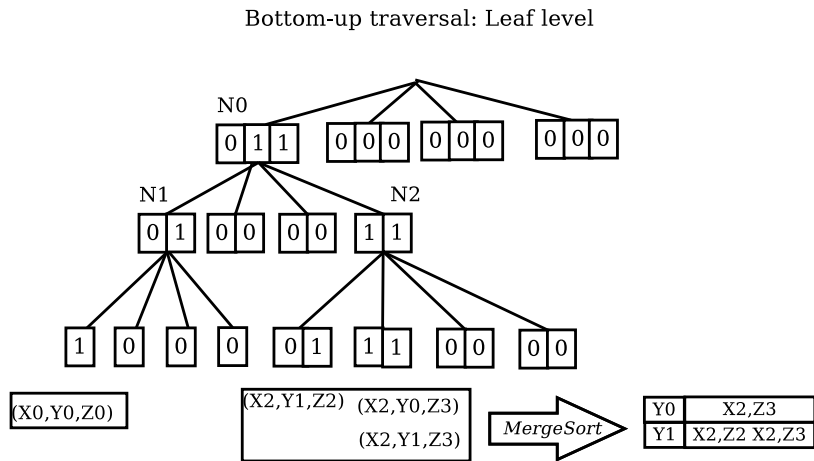


Figure 9.4: Reaching the leaf level of the lazy evaluation

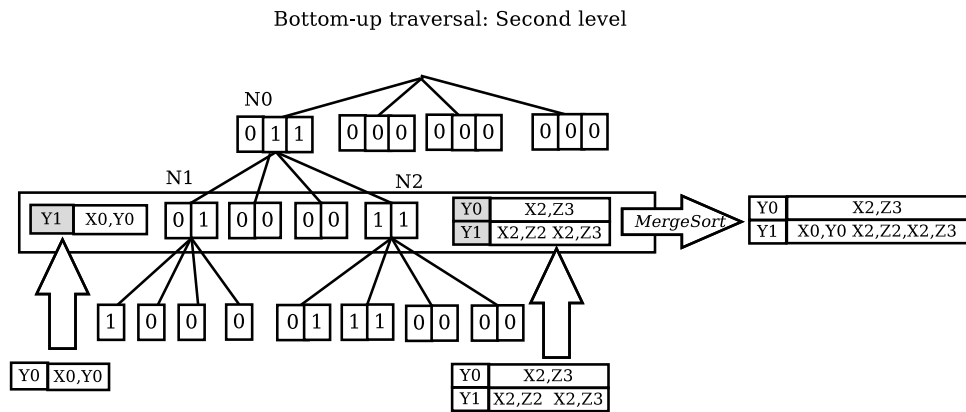


Figure 9.5: Starting the bottom-up traversal of the lazy evaluation

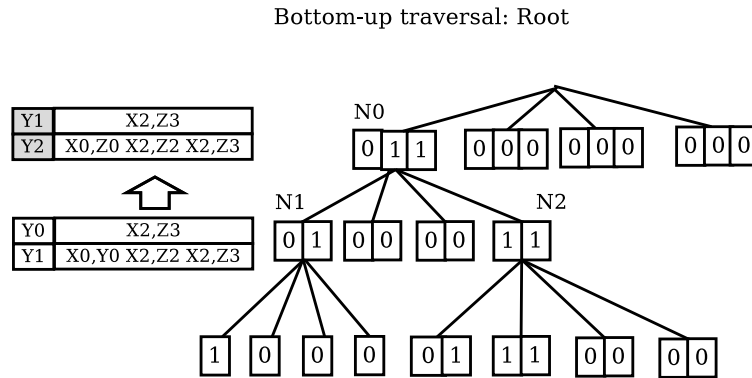


Figure 9.6: Reaching the root in lazy evaluation

we called Interleaved K^2 -tree.

Since Interleaved K^2 -tree is a reordering of the bits of the multiple K^2 -tree solution, it obtains exactly the same spatial cost. Furthermore, it maintains most of the navigation characteristics of the K^2 -tree. On the other hand, it allows to explore different values of the partitioning variable with one traversal over the tree. Consequently, the structure improves the results obtained for a representation with multiple K^2 -trees in such triple patterns where the partitioning variable is unbounded. A lazy evaluation is also proposed, specially designed to solve this kind of queries for datasets with a large number of different values for the partitioning variable. It improves the results obtained by the eager evaluation.

We study the behavior of this structure in RDF datasets, comparing it against our structures for RDF K^2 -triples and K^2 -triples⁺ proposed in Chapter 8. The results obtained for this structure improve the results of K^2 -triples for the queries with the partitioning variable unbounded, specially using the lazy evaluation. However, the alternative K^2 -triples⁺, which includes the use of indexes SP and OP, outperforms the results obtained with the Interleaved K^2 -tree, although in return of a higher spatial cost.

To summarize, Interleaved K^2 -tree is a very compact structure to efficiently store and query any kind of ternary relation, which is specially designed to represent relations where one of the variables has a reduced number of different values. It improves the results obtained by the multiple K^2 -tree alternative for queries with the partitioning variable unbounded, where a lazy evaluation was implemented improving the results of the classic eager evaluation.

Chapter 10

Conclusions and Future work

10.1 Summary of contributions

Graphs have become an interesting way of representing data where the relationships among the different entities are as relevant as the entities themselves. Specially, some kind of graphs, which are called *simple directed graphs*, have become extremely popular because of their ability for representing binary relationships, due to the fact that these binary relationships have emerged everywhere in the new applications for the Big Data Era. Web graphs, social networks or RDF are good examples of the proliferation of data with an underlying graph nature.

However, although a lot of methods, strategies and tools have been developed to represent, visualize, analyze and mine graphs, two main problems that need to be faced in order to work efficiently with graphs still remain. The first problem consists in the need of designing tools that automatically transform collections of complex data into a graph, following a specific graph model according to the needs of the exploitation of the domain. On the other hand, given the huge amount of information that usually composes those generated graphs, the design of very compact graph representations using self-indexed and compressed data structures is crucial in order to manage those huge graphs in main memory in an efficient way.

Two of the contributions of this thesis aim to advance in both scenarios. We presented ***GraphGen***, a completely developed tool available to the community for automatic transformation of any collection of data (using CSV, plain, HTML or XML format) into a specific graph previously defined by the user through the intuitive user interface that GraphGen provides. As explained, GraphGen also offers capabilities for the visualization of the defined graph model and for the exportation

of the built graph to GraphML, the standard language for graph representation. In this way, graphs generated using the GraphGen application can be used for the graph management tools supporting this format, such as the visualization tool Gephi.

On the other hand, we adapted a well-known compressed and self-indexed data structure, the K^2 -tree, to represent labelled and directed attributed graphs, showing how graphs management can take advantage of compressed representations.

This thesis also studied the possibilities for parallel processing of graphs represented with the K^2 -tree data structure. In this way, different distributions of the K^2 -tree cells over a cluster of processors were designed, studied and experimentally compared in terms of space and time.

Finally, we devote a complete section of the thesis to the compact representation of huge collections of RDF data using different variants of the K^2 -tree specifically designed for this purpose. We designed the K^2 -triples and the *Interleaved K^2 -tree*. In both cases we experimentally checked not only the required space but also the answer time of the systems for basic queries (triples queries) over the RDF representation. But, in order to have a more complete idea of the potential of our proposal consisting in an in-memory, compact and self-indexed representation of the RDF data, we developed over K^2 -triples the algorithms required, not only to answer basic queries but also *Join* queries. We compared our proposal with the most well-known and representative Data Base Management Systems for RDF management, such as RDF3x or MonetDB, showing the benefits of our approach.

Although more work needs to be done to transform graphs into a common tool as general and universally used as relational databases, this thesis advances ideas, tools, strategies, data structures and algorithms to use and process graphs in an efficient way, taking advantage of compression technology and distributed processing strategies.

10.2 Future work lines

This section describes the future work lines we expect to address in order to continue the work that has been developed in this thesis. We describe, for each contribution, the most relevant possibilities:

- In this thesis we proposed GraphGen, a tool to generate graphs from arbitrary data. The tool currently exports generated graphs into the standard GraphML format. In this way, the graph can be used in other applications that support this input format. An interesting extension for GraphGen could be the functionality of exporting the final graph in a K^2 -tree format. In this way, if labels and weights of the final graph are relevant to the domain, the graph can

be firstly exported to be imported afterwards in $\text{Att}K^2$ -tree, which supports the mining of attributed and labelled graphs. On the other hand, if the graph generated by the user has not included relevant information in the weights and types of nodes and edges, the graph could be exported as a simple graph to be exploited with a simple K^2 -tree. GraphGen could be also expanded to generate ternary relationships that could be managed with the Interleaved K^2 -tree. In other words, a future work line consists in the integration of GraphGen with the graph structures proposed in this thesis. In that way, GraphGen would prepare any kind of arbitrary data to be processed with the $\text{Att}K^2$ -tree, Interleaved K^2 -tree or even the K^2 -triples, depending on the needs of each application domain.

- In this thesis we study the possibility of representing labelled and weighted graphs in a very compact way, and we provide a basic set of operations to query the data structure we designed. Given that the results obtained with our structure for these basic operations are competitive, a possible extension would be to offer extended operations, in order to provide high level graph mining algorithms for $\text{Att}K^2$ -tree. In addition to that, the storage of the sparse attributes can be optimized using more sophisticated representations of variable-length strings existing in the State of the Art.
- The proposed algorithms to partition a graph using the K^2 -tree structure still have unexplored work lines. First of all, the adaptive multigrid distribution proposed is a very configurable algorithm, whose behavior can strongly depend on the chosen configuration. Further study of the possibilities of this distribution could be carried out, exploring also other variants for the index of this structure (currently implemented by another K^2 -tree). In addition to that, a different problem that is not studied in this thesis could be explored: the distribution of multiple K^2 -trees in order to parallelize the RDF querying system K^2 -triples. The challenge of this new research would be to find the most balanced and efficient way of distributing a set of K^2 -trees, taking into account the effect of the distribution in the different join implementations (specially for the interactive join evaluation).
- An important contribution of this thesis is the new technique to store RDF datasets, which we called K^2 -triples (and its variant K^2 -triples⁺). We implement the simple triple patterns and the different pair joins, which compose a basis over which more complex queries could be implemented. Therefore, the next step to obtain a complete RDF query engine would consist of supporting queries composed by a combination of joins. In order to implement these joins composed by more than two triple patterns, a study of the appropriate metrics to design the order of execution of the pair joins for each query and the kind of strategy used in each case is needed.

- In this thesis we also proposed a new structure to represent ternary relationships. The simple triple patterns were implemented over this structure. The different join algorithms designed for the K^2 -triples could be adapted to work with this new structure, extending its current functionality. In addition to that, the possibility of using the indexes SP and OP designed for the K^2 -triples⁺ in combination with this new structure deserves to be studied.

Appendix A

Publications and other research results

Publications

JCR Journals

- Álvarez-García, S.; Baeza-Yates, R.; Brisaboa, N. R.; Larriba-Pey, J.; Pedreira, O.: Automatic Multi- Partite Graph Generation from Arbitrary Data, en *Journal of Systems and Software*, (to appear), Estados Unidos, 2014.
- Álvarez-García, S.; Brisaboa, N. R.; Fernández, J.D.; Martínez Prieto, M.A.; Navarro, G.: Compressed Vertical Partitioning for Efficient RDF Management, en *Knowledge and Information Systems*, (to appear), Estados Unidos, 2014.

International conferences

- Álvarez-García, S.; Brisaboa, N. R.; de Bernardo, Guillermo; Navarro, G.: Interleaved K2-tree: Indexing and Navigating Ternary Relations, en *Proceedings of the 2014 Data Compression Conference (DCC 2014)*, Snowbird, Utah (Estados Unidos), 2014, pp. 342-351.
- de Bernardo, Guillermo; Álvarez-García, S.; Brisaboa, N. R.; Navarro, G.; Pedreira, O.: Compact Queriable Representations of Raster Data, en *Proceedings of 20th International Symposium (SPIRE 2013) - LNCS 8214*, Springer, Tel Aviv (Israel), 2013, pp. 96-108.

- Álvarez-García, S.; Brisaboa, N. R.; Gomez-Pantoja, C.; Marín, M.: Distributed Query Processing on Compressed Graphs Using K2-Trees, en *Proceedings of 20th International Symposium (SPIRE 2013)* - LNCS 8214, Springer, Tel Aviv (Israel), 2013, pp. 298-310.
- Álvarez-García, S.; Brisaboa, N. R.; Fernández, J.D.; Martínez Prieto, M.A.: Compressed k2-Triples for Full-In-Memory RDF Engines, en *Proc. of the Seventeenth Americas Conference on Information Systems (AMCIS 2011)*, AIS Electronic Library, Detroit, Michigan (Estados Unidos), 2011.
- Álvarez-García, S.; Baeza-Yates, R.; Brisaboa, N. R.; Larriba-Pey, J.; Pedreira, O.: GraphGen: A Tool for Automatic Generation of Multipartite Graphs from Arbitrary Data, en *Procs. of Latin American Web Congress (LAWEB'12)*, IEEE Press, Cartagena (Colombia), 2012, pp. 87-94.
- Álvarez-García, S.; Brisaboa, N. R.; Ladra, S.; Pedreira, O.: A Compact Representation of Graph Databases, en *Proc. of the 8th Workshop on Mining and Learning with Graphs (MLG 2010)*, ACM Press, Washington D.C. (Estados Unidos), 2010, pp. 18-25.

National conferences

- Álvarez-García, S.; Folgar, S.; Ladra, S.: Resúmenes de grafos usando k2-trees, en *Actas del II Congreso Español de Recuperación de Información (CERI 2012)*, Publicacions de la Universitat Jaume I, Valencia (España), 2012, pp. 37-48.
- Fernández, J.D.; Martínez Prieto, M.A.; Arias, M.; Gutiérrez, C.; Álvarez-García, S.; Brisaboa, N. R.: Lightweighting the Web of Data through Compact RDF/HDT, en *Advances in Artificial Intelligence*. LNAI 7023, Springer-Verlag, San Cristóbal de la Laguna (España), 2011, pp. 483-493.
- Álvarez-García, S.; Brisaboa, N. R.; Ladra, S.; Pedreira, O.: Almacenamiento y explotación de grandes bases de datos orientadas a grafos, en *Actas de las XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2010)*, Ibergaceta Publicaciones, Valencia (España), 2010, pp. 187-198.
- Álvarez-García, S.; Cerdeira-Pena, A.; Fariña, A.; Ladra, S.: Desarrollo de un compresor de textos orientado a palabras basado en PPM, en *Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD'09)*, San Sebastián (España), 2009, pp. 237-248.

Research stays

- *September, 2011 - December, 2011.* Research stay at Yahoo! Research Latin America (Santiago de Chile), under the supervision of Prof. Mauricio Marín Cahuin.

Appendix B

Descripción del Trabajo Realizado

El trabajo desarrollado a lo largo de esta tesis se centra en el diseño e implementación de representaciones compactas y eficientes de grafos. Los grafos son una forma natural de modelar conjuntos de datos de diversos dominios en los que la información de interés reside principalmente en las relaciones entre las entidades. La teoría de grafos ha sido ampliamente estudiada desde una perspectiva matemática. En la actualidad, y especialmente desde la aparición de Internet, donde multitud de datos de diferente naturaleza son generados diariamente, esta estructura se ha utilizado como forma de representación de grandes volúmenes de datos.

La web es un ejemplo de información que habitualmente se representa mediante grafos, donde las diferentes páginas son representadas como nodos, y las aristas relacionan cada página Web con todas aquellas páginas que son enlazadas desde ella. Otro ejemplo muy representativo son los grafos sociales, en los que los nodos son los usuarios del grafo, mientras que las aristas representan en este caso relaciones de amistad.

En este contexto, y teniendo en cuenta el gran volumen de estos datos, el diseño de estructuras de datos para representar grafos que puedan ser consultados de forma eficiente se vuelve fundamental para el tratamiento de los mismos. Por este motivo, se ha convertido en los últimos años en un importante campo de investigación, en el que se han realizado numerosas aportaciones.

En esta tesis estudiamos diferentes aproximaciones a este problema, diseñando e implementando estructuras para la representación de grafos de diferente naturaleza. Estas estructuras han sido diseñadas con el objetivo de representar de forma compacta

y eficiente grafos de multitud de dominios. Además, se ha estudiado el problema no sólo desde una perspectiva monoprocesador, sino que se ha abordado el diseño de una representación de grafos en entornos distribuidos. Esta nueva perspectiva permite la escalabilidad de los sistemas, permitiendo aplicar las mismas técnicas a conjuntos de datos de un mayor volumen.

Las estructuras y algoritmos que se presentan en esta tesis pueden ser agrupados en cinco grandes apartados. Los dos primeros abordan la representación y generación de grafos con atributos, en los que los nodos incluyen información detallada a lo largo de una serie de propiedades. El tercer apartado de esta tesis se centra en el estudio de representaciones de grafos comprimidas y distribuidas, utilizando para ello como base el K^2 -tree, una estructura compacta diseñada originalmente para grafos Web. Los dos últimos apartados se centran en la representación compacta de grafos RDF, un tipo especial de grafo en el que los nodos origen (sujetos) se relacionan con los nodos destino (objetos) a través de aristas etiquetadas (predicados). RDF se ha convertido en un estándar de facto para la descripción de metadatos que permita el intercambio de información entre fuentes heterogéneas. Dos estructuras compactas y eficientes se han propuesto en esta tesis para el manejo de fuentes de datos RDF: la primera de ellas enfocada exclusivamente en este problema, mientras que la segunda ha sido diseñada para representar relaciones ternarias en multitud de dominios, aunque su evaluación se ha centrado exclusivamente en el dominio RDF.

A continuación, se detallarán cada uno de estos cinco grandes apartados que constituyen las principales aportaciones de esta tesis, indicándose los objetivos perseguidos, así como los resultados obtenidos con el trabajo realizado.

B.1 Modelado de grafos con GraphGen

Como ya se ha descrito, son numerosos los dominios en los que representar la información mediante un grafo nos permite destacar las relaciones entre los diferentes elementos y consultarlos de forma eficiente. Algunos de los casos citados (como es el caso de los grafos Web o las redes sociales) y otros muchos como los grafos de consultas de usuario (al interactuar con un motor de búsqueda) pueden resultar muy útiles para poder explotar la información que contienen.

Sin embargo, la mayoría de los datos de entrada que producirían estos grafos están habitualmente disponibles en otros formatos estándar como XML, CSV o incluso en bases de datos relacionales. De esta forma, todo investigador o investigadora que requiera representar esta información mediante un grafo, deberá realizar la tediosa tarea de implementar un algoritmos de transformación manual del formato de entrada a la estructuración mediante un grafo. Aunque existen en el estado del arte algunas propuestas para realizar este tipo de transformaciones, la mayoría han

sido diseñadas ad-hoc para un dominio específico.

En esta tesis proponemos GraphGen, una herramienta que permite al usuario automatizar esta transformación de fuentes de datos arbitrarias a grafos estándar, para que posteriormente puedan ser gestionados con cualquier herramienta de visualización o de consulta de grafos.

En primer lugar, proponemos un modelo teórico que da soporte a la posterior herramienta que hemos desarrollado. En GraphGen, cada nodo representará una cantidad de información, que podrá descomponerse en diferentes nodos más simples, a través de reglas de transformación que serán definidas por el usuario. Las aristas relacionarán los nodos con aquellos nuevos elementos en los que se descompone progresivamente.

GraphGen proporciona un conjunto de reglas de extracción predefinidas, concebidas para la generación de grafos a partir de los formatos más comunes de entrada, como es el caso del XML. Además de reglas de descomposición, que permiten la creación de nuevos nodos y aristas (que asocian el nodo original con todos aquellos que hayan sido extraídos desde él), se han definido reglas de relación, que permiten establecer relaciones entre diferentes nodos del árbol en base a la existencia de una relación topológica entre ellos, u otro tipo de criterios que podrán ser definidos por el usuario.

Hemos evaluado la versatilidad de GraphGen a lo largo de 3 escenarios diferentes: grafos extraídos a partir de bases de datos bibliográficas, la transformación de query logs en formato CSV y la creación de grafos sociales a partir de fuentes de datos en XML. Los resultados obtenidos muestran que GraphGen permite definir reglas que se adecúan a multitud de dominios y que genera grafos con millones de elementos en minutos. Los resultados de este trabajo han sido publicados en la revista indexada JCR Journal of Systems and Software y en un workshop internacional.

B.2 Grafos con atributos

En multitud de ocasiones, los grafos que se necesitan para representar la información de un dominio son grafos simples dirigidos, cuyas aristas pueden incluir una etiqueta o un peso. Ejemplos de este tipo de grafos pueden ser los grafos Web o los grafos sociales. Sin embargo, en muchos otros casos, este modelo de grafos no es suficiente para representar toda la información del dominio, ya que los nodos y las aristas contienen datos adicionales. Estos dominios en los que nodos y aristas están descritos por una serie de atributos (clave/valor) definen un nuevo modelo de grafos, que se denominan grafos con atributos.

En los últimos años, han surgido numerosos modelos teóricos para representar

este nuevo tipo de grafos con atributos. Soportados por estos nuevos modelos, han surgido en el estado del arte multitud de propuestas para almacenar y consultar de forma eficiente grafos con atributos. Estos nuevos sistemas se denominan Bases de Datos orientadas a grafos, y entre las más reconocidas podemos destacar DEX y Neo4J.

Estos grafos con atributos representan generalmente dominios en los que el volumen de datos suele ser muy elevado, por lo que la eficiencia, tanto espacial como temporal, se vuelve un aspecto de vital importancia a la hora de diseñar un sistema de estas características.

En esta tesis presentamos $\text{Att}K^2\text{-tree}$, una representación compacta en memoria principal para grafos con atributos, que está basada en la estructura compacta para grafos simples $K^2\text{-tree}$.

Nuestro objetivo al desarrollar este trabajo ha consistido en extender esta estructura compacta $K^2\text{-tree}$, que puede ser utilizada para relaciones binarias, para representar los grafos con atributos.

La idea principal de $\text{Att}K^2\text{-tree}$ consiste en representar los atributos densos (que son aquellos en los que numerosos nodos o aristas comparten el mismo valor) como la nacionalidad de las personas, mediante relaciones binarias que pueden ser representadas como el $K^2\text{-tree}$. Los atributos esparsos (como el DNI de una persona o el título de un libro) serán representados mediante listas doblemente indexadas (por identificador de nodo y por orden lexicográfico).

Las aristas entre los diferentes nodos también se representarán mediante un $K^2\text{-tree}$, para el que se ha propuesto una extensión que permite representar multiaristas (es decir, la existencia de más de una arista por cada nodo), además de almacenar los identificadores de cada una de estas aristas, que servirán como referencia para posteriormente poder recuperar su restante información adicional (es decir, los atributos que contiene).

Se han propuesto una serie de operaciones básicas para consultar el sistema propuesto, que incluyen la consulta del valor de un atributo para un nodo, los nodos que cumplan una determinada condición o los nodos que se relacionan con otros nodos del grafo. El objetivo perseguido con el diseño de este conjunto de operaciones básicas es el de obtener una base sobre la que se puedan implementar operaciones más complejas.

La eficiencia temporal y espacial de esta estructura se ha evaluado en diferentes conjuntos de prueba, comparando los resultados obtenidos con dos de las aproximaciones más relevantes en el estado del arte: DEX y Neo4J. Los resultados obtenidos prueban que $\text{Att}K^2\text{-tree}$ obtiene muy buenos resultados de eficiencia espacial y que obtiene tiempos competitivos a la hora de implementar las operaciones básicas que soporta en la actualidad. Los resultados obtenidos con este trabajo han

sido publicados en un Workshop internacional.

B.3 Representación de grafos distribuidos

En la actualidad, la cantidad de datos que se genera diariamente es tan elevada que dificulta su procesamiento con los sistemas de procesamiento tradicionales. En los últimos años, se han propuesto en multitud de campos alternativas distribuidas a las tradicionales estructuras y algoritmos mono-procesador, que presentan limitaciones para tratar con grandes volúmenes de datos. En este contexto, el paralelismo supone una buena alternativa a los problemas de escalabilidad de los sistemas de explotación de datos, y surge un nuevo campo de investigación, que tiene como objetivo diseñar nuevas estructuras paralelas que proporcionen escalabilidad a la explotación de datos de gran volumen.

En este apartado de la tesis se propone el diseño e implementación de diferentes algoritmos de particionamiento para distribuir grafos simples dirigidos, utilizando como base la estructura K^2 -tree, que permite la representación compacta de grafos simples dirigidos.

En este apartado se comienza realizando una revisión exhaustiva del estado del arte en algoritmos de particionamiento para grafos. En un primer lugar, se revisan las estrategias tradicionales, en las que cada procesador representa un conjunto de nodos disjuntos del grafo. Entre estas técnicas destacan los algoritmos de particionamiento geométricos, que utilizan la información topográfica de los nodos o los algoritmos estructurales, o el algoritmo clásico de Kernighan and Lin. Posteriormente, se realiza una descripción de las principales técnicas de particionamiento de matrices, que tiene una gran relación con nuestro problema de estudio, puesto que el K^2 -tree, que es la estructura que hemos utilizado de base para nuestra estructura, parte de una representación del grafo en forma de matriz de adyacencia. Entre los algoritmos para particionamiento de grafos destacan los particionamientos rectilíneos o los métodos de particionamiento utilizados para distribuir páginas de disco.

Tras revisar las principales técnicas del estado del arte, se han propuesto nuevas técnicas de particionamiento de grafos basados en la utilización de la estructura K^2 -tree. En primer lugar, se han aplicado técnicas de división de la matriz de adyacencia básicas, como el particionamiento en bloques, el particionamiento cíclico o el particionamiento en rejilla simple y recursivo. Pero además, se han diseñado versiones adaptativas de estos algoritmos que tienen en cuenta la distribución del grafo a lo largo de la matriz de adyacencia, para distribuir en mayor medida aquellas regiones con una gran concentración de aristas.

Además de estas propuestas, se han diseñado algunas estrategias de particionamiento que logran alcanzar un balance espacial perfecto, utilizando para ello las

características propias del árbol K^2 -tree que será distribuido finalmente. Finalmente, se propone una alternativa diferente basada en cuadrados latinos, inspirada en los particionamientos habituales en la distribución de páginas de disco.

Se ha realizado una evaluación exhaustiva de la eficiencia espacial y temporal de cada una de estas distribuciones, a la hora de representar y consultar grafos Web y grafos sociales. Además, se ha prestado especial atención al balance espacial y de carga obtenido por cada una de ellas, así como la evaluación de la aceleración conforme aumenta el número de procesadores utilizados. Algunas de las distribuciones propuestas han obtenido resultados de eficiencia muy competitivos, que demuestran que la distribución de grafos utilizando una estructura tan compacta como el K^2 -tree proporciona la escalabilidad necesaria a esta estructura, que trabaja en memoria principal, para poder gestionar grafos con un mayor volumen de nodos y aristas.

Los resultados obtenidos en este campo se han publicado en la conferencia SPIRE en el año 2013.

B.4 Representación de grafos RDF: K^2 -triples

En este apartado hemos estudiado la representación de un tipo de grafo muy específico: el grafo RDF (Resource Description Framework), sobre el que se construyen los principios de la Web semántica y consiste en un nuevo formato de publicación que pretende servir de lenguaje automático de intercambio de información en la llamada Web de datos. RDF proporciona un lenguaje común para definir *hechos* del mundo de forma estructurada. Un conjunto de datos RDF está formado por un conjunto de tripletas (sujeto, predicado, objeto), que indican que el sujeto toma el valor del objeto en un determinado predicado. Un conjunto de datos RDF puede ser modelado como un grafo etiquetado, donde el sujeto y el objeto son nodos del grafo y el predicado son aristas del mismo.

En los últimos años, con la aparición de la denominada Web de datos, el formato RDF ha ido creciendo en importancia, por lo que multitud de sistemas de almacenamiento y consulta de datos RDF han aparecido en el estado del arte. En este apartado de la tesis hemos realizado una revisión exhaustiva de las diferentes aproximaciones que se han seguido al abordar la gestión de datos RDF.

Una de las principales contribuciones de esta tesis es una nueva técnica que hemos diseñado para almacenar conjuntos de datos RDF de una forma muy compacta en memoria principal, implementando diferentes algoritmos de búsqueda eficientes. Nuestra estructura sigue un particionamiento vertical, en el que se utilizará un K^2 -tree para representar los datos de cada uno de los predicados.

El principal problema del particionamiento vertical es la eficiencia a la hora de

realizar consultas en las que se desconoce el predicado, puesto que se vuelve necesario consultar los diferentes elementos que representan cada uno de los predicados. Para resolver este problema, hemos propuesto unos índices compactos sobre los sujetos y los predicados (que han sido implementados con DAC), y que mejoran la eficiencia de este tipo de consultas mediante la minimización del número de predicados a consultar.

Hemos diseñado la implementación de los patrones de consulta básicos, así como el desarrollo de algoritmos de joins. En este último apartado, hemos propuesto tres estrategias alternativas para resolver cada uno de los posibles joins existentes, cuyos resultados se han comparado en la evaluación experimental.

Los resultados espaciales y temporales obtenidos con esta nueva estructura que proponemos han sido evaluados contra las principales alternativas del estado del arte, obteniendo resultados muy competitivos tanto en tiempo como en espacio.

Los resultados de este trabajo han sido publicados en la conferencia internacional AMCIS (2011) y en la revista indexada JCR Knowledge and Information Systems (por aparecer).

B.5 Representación de relaciones ternarias

Los grafos RDF que hemos descrito en el apartado previo pueden ser considerados como un grafo etiquetado; pero también pueden ser interpretados como una relación ternaria. Existen multitud de relaciones ternarias en otros dominios, como es el caso de los rasters espaciales. A lo largo de los años han surgido algunas estrategias para almacenar y consultar estas relaciones ternarias, pero la mayoría de estas aproximaciones están enfocadas al almacenamiento y consulta de un conjunto de datos específico.

En esta tesis proponemos el Interleaved K^2 -tree, una estructura compacta para representar relaciones ternarias de una forma muy compacta y eficiente. Esta nueva estructura es una evolución del K^2 -tree, que representa en un solo árbol una relación binaria etiquetada, proporcionando capacidades de indexación sobre las tres dimensiones que representa. De esta forma, intenta resolver el problema del particionamiento vertical, agrupando todas las relaciones en un único árbol, y manteniendo muchas de las propiedades de acceso que contenía el K^2 -tree.

Esta estructura está especialmente diseñada para la representación de relaciones ternarias donde una dimensión tiene un número de valores diferentes muy inferior al de las otras dos dimensiones. Sobre esta estructura, hemos propuesto una serie de operaciones básicas que han sido implementadas de forma eficiente. Para aquellos casos en los que la tercera dimensión tiene un gran número de valores y en las

operaciones en las que esa tercera dimensión no está ligada (cualquier valor es válido como resultado), hemos propuesto una estrategia de evaluación *lazy*, que mejora los resultados obtenidos para estas operaciones.

Esta estructura ha sido evaluada para grafos RDF como alternativa a la estructura K^2 -triples desarrollada en el apartado anterior, que mejora los resultados obtenidos para la misma en este tipo de operaciones que incluyen la tercera dimensión libre.

Los resultados obtenidos con este trabajo han sido publicados en la conferencia internacional DCC (2014).

Para concluir, cabe destacar que los trabajos realizados a lo largo de estos cinco apartados pretenden proporcionar nuevas soluciones al problema de la representación eficiente de grafos de gran volumen en diferentes dominios. Por una parte, se ha propuesto una herramienta que facilita el modelado de grafos a partir de fuentes de datos heterogéneas. Además, se ha propuesto una nueva estructura para gestionar grafos con atributos. Para entornos distribuidos, se han propuesto diferentes estrategias para particionar grafos y representarlos con estructuras compactas y eficientes. Finalmente, los dos últimos apartados de la tesis se centran en grafos para dominios más concretos. En primer lugar, se propone un sistema de almacenamiento y consulta para grafos RDF y finalmente, se propone una estructura para almacenar relaciones ternarias asimétricas que puede ser también utilizada para representar grafos RDF. Todas las contribuciones de esta tesis han sido publicadas en conferencias o revistas indexadas.

Bibliography

- [ABK⁺07] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: a nucleus for a Web of open data. In *Proc. 6th International Semantic Web (ISWC) Conference and 2nd Asian Semantic Web Conference (ASWC)*, pages 722–735, 2007.
- [ACZH10] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix bit loaded: a scalable lightweight join query processor for rdf data. In *Proceedings of the 19th international conference on World wide web*, pages 41–50. ACM, 2010.
- [AFMP11] M. Arias, J. D. Fernández, and M. A. Martínez-Prieto. An empirical study of real-world SPARQL queries. In *Proc. 1st International Workshop on Usage Analysis and the Web of Data (USEWOD)*, 2011. Available at <http://arxiv.org/abs/1103.5043>.
- [AG05] R. Angles and C. Gutierrez. Querying rdf data from a graph database perspective. In *The Semantic Web: Research and Applications*, pages 346–360. Springer, 2005.
- [AG08] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.
- [ÁGBdBN14] S. Álvarez-García, N. R. Brisaboa, G. de Bernardo, and G. Navarro. Interleaved k2-tree: Indexing and navigating ternary relations. In *Data Compression Conference (DCC), 2014*, pages 342–351, Snowbird, Utah, 2014.
- [ÁGBF⁺14] S. Álvarez-García, N. R. Brisaboa, J.D. Fernández, M.A. Martínez Prieto, and G. Navarro. Compressed vertical partitioning for efficient rdf management. *Knowledge and Information Systems (to appear)*, 2014.

- [ÁGBFMP11] S. Álvarez-García, N. R. Brisaboa, J.D. Fernández, and M.A. Martínez Prieto. Compressed k2-triples for full-in-memory rdf engines. In *Americas Conference on Information Systems*, 2011.
- [ÁGBGPM13] S. Álvarez-García, N. R. Brisaboa, C. Gomez-Pantoja, and M. Marín. Distributed query processing on compressed graphs using k2-trees. In *String Processing and Information Retrieval*, pages 298–310, Tel Aviv, 2013.
- [ÁGBLP10a] S. Álvarez-García, N. R. Brisaboa, S. Ladra, and O. Pedreira. Almacenamiento y explotación de grandes bases de datos orientadas a grafos. In *Jornadas de Ingeniería del Software y Bases de Datos*, pages 187–198, 2010.
- [ÁGBLP10b] S. Álvarez-García, N. R. Brisaboa, S. Ladra, and O. Pedreira. A compact representation of graph databases. In *Workshop on Mining and Learning with Graphs*, pages 18–25, Washington D.C., 2010.
- [ÁGBYB⁺12] S. Álvarez-García, R. Baeza-Yates, N. R. Brisaboa, J. Larriba-Pey, and O. Pedreira. Graphgen: A tool for automatic generation of multipartite graphs from arbitrary data. In *Latin American Web Congress*, pages 87–94, 2012.
- [ÁGBYB⁺14] S. Álvarez-García, R. Baeza-Yates, N. R. Brisaboa, J. Larriba-Pey, and O. Pedreira. Automatic multi-partite graph generation from arbitrary data. *Journal of Systems and Software (to appear)*, 2014.
- [AMF06] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. 33th International Conference on Management of Data (SIGMOD)*, pages 671–682, 2006.
- [AMMH07] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.
- [AMMH09] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 18(2):385–406, 2009.
- [AW10] C. C. Aggarwal and H. Wang. *Managing and mining graph data*, volume 40. Springer, 2010.

- [BB87] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *Computers, IEEE Transactions on*, 100(5):570–580, 1987.
- [BBC⁺08] P. Boldi, F. Bonchi, C. Castillo, D. Donato, A. Gionis, and S. Vigna. The query-flow graph: model and applications. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 609–618. ACM, 2008.
- [BBLPC14] D. Beckett, T. Berners-Lee, E. Prud’hommeaux, and G. Carothers. Terse rdf triple language, feb 2014. <http://www.w3.org/TR/turtle/>.
- [BCSV] P. Boldi, B. Codenotti, M. Santini, and title= Vigna, S.”.
- [BDBY10] I. Bordino, D. Donato, and R. Baeza-Yates. Coniunge et impera: Multiple-graph mining for query-log analysis. In *Machine Learning and Knowledge Discovery in Databases*, pages 168–183. Springer, 2010.
- [BEF⁺09] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. Cophir: a test collection for content-based image retrieval. *arXiv preprint arXiv:0905.4627*, 2009.
- [BFNP07] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Lightweight natural language text compression. *Information Retrieval*, pages 1–33, 2007.
- [BG04] D. Brickley and R. V. Guha. Resource description framework (rdf) schema specification 1.0, feb 2004. <http://www.w3.org/TR/rdf-schema/>.
- [BHBL09] C. Bizer, T. Heath, and T. Berners-Lee. Linked data-the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, 2009.
- [BHS03] V. Bonstrom, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *Web Congress, 2003. Proceedings. First Latin American*, pages 27–36. IEEE, 2003.
- [Bin11] RDF Binary. Representation for publication and exchange (hdt). *W3C Member Submission*, 2011.
- [BKVH02] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *The Semantic Web—ISWC 2002*, pages 54–68. Springer, 2002.

- [BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific american*, 284(5):28–37, 2001.
- [BLN09] N. R. Brisaboa, S. Ladra, and G. Navarro. K2-trees for compact web graph representation. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE'2009)*, pages 18–30, 2009.
- [BLN13] N. R. Brisaboa, S. Ladra, and G. Navarro. Dacs: Bringing direct access to variable-length codes. *Information Processing and Management*, pages 392–404, 2013.
- [BM76] J. A. Bondy and U. S. R. Murty. *Graph theory with applications*, volume 6. Macmillan London, 1976.
- [BRSV] P. Boldi, M. Rosa, M. Santini, and title = Vigna, S.”.
- [BV04] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.
- [BY04] R. Baeza-Yates. Web usage mining in search engines. *Web mining: applications and techniques*, pages 307–321, 2004.
- [BY07] R. Baeza-Yates. Graphs from search engine queries. In *SOFSEM 2007: Theory and Practice of Computer Science*, pages 1–8. Springer, 2007.
- [BYBLP10] R. Baeza-Yates, N. Brisaboa, and J. Larriba-Pey. A model for automatic generation of multi-partite graphs from arbitrary data. In *Web-Age Information Management*, pages 49–60. Springer, 2010.
- [Cha98] B. L. Chamberlain. Graph partitioning algorithms for distributing workloads of parallel computations. Technical report, University of Washington, 1998.
- [CJ11] R. Cyganiak and A. Jentzsch. Linking open data cloud diagram. *LOD Community* (<http://lod-cloud.net/>), 2011.
- [CJL94] P. Ciarlet Jr and F. Lamour. Recursive partitioning methods and greedy partitioning methods: a comparison on finite element graphs. *UCLA CAM Report*, pages 94–9, 1994.
- [Cla98] D. Clark. *Compact Pat nes*. PhD thesis, University of Waterloo, 1998.

- [dB14] Guillermo de Bernardo. *New data structures and algorithms for the efficient management of large spatial datasets*. PhD thesis, Universidade da Coruña, Spain, 2014.
- [dBÁGB⁺13] Guillermo de Bernardo, S. Álvarez-García, N. R. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In *String Processing and Information Retrieval*, pages 96–108, Tel Aviv, 2013.
- [Els97] U. Elsner. Graph partitioning. Technical report, a survey, 1997.
- [Fie75] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633, 1975.
- [Fja98] P. O. Fjallstrom. Algorithms for graph partitioning: A survey. *Linköping Electronic Articles in Computer and Information Science*, 1998.
- [FL93] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, 36(5):745–764, 1993.
- [FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference, DAC '82*, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.
- [FMPG⁺13] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary rdf representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web*, 2013.
- [FP10] N. Fan and P. M. Pardalos. Linear and quadratic programming approaches for the general graph partitioning problem. *Journal of Global Optimization*, 48(1):57–71, 2010.
- [GB04] J. Grant and D. Beckett. Rdf test cases, 2004. <http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/>.
- [GEP12] The Open Graph Viz Platform, 2012. [Online; accessed 26-May-2014].
- [GGMN05] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA '05) (Greece)*, pages 27–38, 2005.

- [GHSV06] A. Gupta, W. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 213–222. IEEE, 2006.
- [GMT98] J. R. Gilbert, G. L. Miller, and S. H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.
- [GPVdBVG94] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *Knowledge and Data Engineering, IEEE Transactions on*, 6(4):572–586, 1994.
- [GRA09] Graphml specification, 2009. Online; accessed 26-May-2014.
- [Gro11] S. Groppe. *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011.
- [Gro14] Grouplens. MovieLens dataset, 2014. [Online; accessed 19-July-2014].
- [GVSMGV⁺08] S. Gómez-Villamor, G. Soldevila-Miranda, A. Giménez-Vañó, N. Martínez-Bazan, V. Muntés-Mulero, and J. L. Larriba-Pey. Bibex: a bibliographic exploration tool based on the dex graph query engine. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 735–739. ACM, 2008.
- [HAR11a] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.
- [HAR11b] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.
- [HD05] A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *Web Congress, 2005. LA-WEB 2005. Third Latin American*, pages 10–pp. IEEE, 2005.
- [HG03] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. *1st International Workshop on Practical and Scalable Semantic Systems*, 2003.
- [HG04] J. Hayes and C. Gutierrez. Bipartite graphs as intermediate model for rdf. In *The Semantic Web-ISWC 2004*, pages 47–61. Springer, 2004.

- [Ior10] B. Iordanov. Hypergraphdb: a generalized graph database. In *Web-Age Information Management*, pages 25–36. Springer, 2010.
- [Jac88] G. J. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.
- [JK05] M. Janik and K. Kochut. Brahms: A workbench rdf store and high performance memory system for semantic association discovery. In *The Semantic Web-ISWC 2005*, pages 431–445. Springer, 2005.
- [JLNL13] D. Jiang, K. W. Leung, W. Ng, and H. Li. Beyond click graph: Topic modeling for search engine query log analysis. In *Database Systems for Advanced Applications*, pages 209–223. Springer, 2013.
- [KK98a] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.
- [KK98b] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.
- [KL70] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2), 1970.
- [Lad11] S. Ladra. *Algorithms and compressed data structures for information retrieval*. PhD thesis, Universidad de A Coruña, 2011.
- [Lan50] C. Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office, 1950.
- [Les] L. Leskovec. Snap: Stanford network analysis platform. [Online].
- [Ley09] M. Ley. Dblp: some lessons learned. *Proceedings of the VLDB Endowment*, 2(2):1493–1500, 2009.
- [LIJ+14] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 2014.
- [LP90] M. Levene and A. Poulouvasilis. The hypernode model and its associated query language. In *Information Technology, 1990.'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No. 90TH0326-9)*, pages 520–530. IEEE, 1990.

- [LSR92] J. Li, J. Srivastava, and D. Rotem. Cmd: A multidimensional declustering method for parallel database systems. In *Proceedings of the 18th VLDB Conference*, pages 3–14. Citeseer, 1992.
- [MA91] B. Mohar and Y. Alavi. The laplacian spectrum of graphs. *Graph theory, combinatorics, and applications*, 2:871–898, 1991.
- [MAB⁺10] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.
- [MBÁLMM⁺12] N. Martínez-Bazan, M. A. Águila-Lorente, V. Muntés-Mulero, D. Dominguez-Sal, S. Gómez-Villamor, and J. L. Larriba-Pey. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Symposium*, pages 110–119. ACM, 2012.
- [MBMMGV⁺07] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M. A. Sánchez-Martínez, and J. L. Larriba-Pey. Dex: high-performance exploration on large graphs for information retrieval. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 573–582. ACM, 2007.
- [MM14] C. Mukherjee and G. Mukherjee. Role of adjacency matrix in graph theory. *IOSR Journal of Computer Engineering*, 2014.
- [MMM04] F. Manola, E. Miller, and B. McBride. Rdf primer. *W3C recommendation*, 10:1–107, 2004.
- [MN07] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
- [MPFC12] M. A. Martínez-Prieto, J. D. Fernández, and R. Cánovas. Querying rdf dictionaries in compressed space. *ACM SIGAPP Applied Computing Review*, 12(2):64–77, 2012.
- [MS96] F. Manne and T. Sørsvik. *Partitioning an array onto a mesh of processors*. Springer, 1996.
- [MTTV98] G. L. Miller, S. H. Teng, W. Thurston, and S. A. Vavasis. Geometric separators for finite-element meshes. *SIAM Journal on Scientific Computing*, 19(2):364–386, 1998.

- [Mun96] J. I. Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer, 1996.
- [NDC11] M. Nascimento and A. De Carvalho. Spectral methods for graph clustering—a survey. *European Journal of Operational Research*, 211(2):221–231, 2011.
- [NEO14] Neo4J Graph Database, 2014. [Online; accessed 26-May-2014].
- [New13] MEJ Newman. Spectral methods for community detection and graph partitioning. *Physical Review E*, 88(4):042822, 2013.
- [Nic94] D. M. Nicol. Rectilinear partitioning of irregular data parallel computations. *Journal of Parallel and Distributed Computing*, 23(2):119–134, 1994.
- [NW09] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 627–640. ACM, 2009.
- [OBM06] X. Ou, W. F. Boyer, and M. A. McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345. ACM, 2006.
- [OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*. SIAM, 2007.
- [PA97] A. Pinar and C. Aykanat. Sparse matrix decomposition with optimal load balancing. In *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, pages 224–229. IEEE, 1997.
- [Pag99] R. Pagh. *Low redundancy in static dictionaries with $O(1)$ worst case lookup time*. Springer, 1999.
- [Pot97] A. Pothen. Graph partitioning algorithms with applications to scientific computing. In *Parallel Numerical Algorithms*, pages 323–368. Springer, 1997.
- [PS08] E. Prud’hommeaux and A. Seaborne. Sparql query language for rdf, jan 2008. <http://www.w3.org/TR/rdf-sparql-query/>.
- [PSL90] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990.

- [RG00] R. Ramakrishnan and J. Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.
- [RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics, 2002.
- [RZ95] L. F. Romero and E. L. Zapata. Data distributions for sparse matrix vector multiplication. *Parallel Computing*, 21(4):583–605, 1995.
- [Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
- [Sam06] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.
- [SEH12] S. Sakr, S. Elnikety, and Y. He. G-sparql: a hybrid engine for querying large attributed graphs. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 335–344. ACM, 2012.
- [SGK⁺08] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008.
- [SHJ⁺02] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 273–284. IEEE, 2002.
- [SHK⁺08] M. Schmidt, T. Hornung, N. Kuchlin, G. Lausen, and C. Pinkel. An experimental comparison of rdf data management approaches in a sparql benchmark scenario. In *The Semantic Web-ISWC 2008*, pages 82–97. Springer, 2008.
- [SK12] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.
- [UMB10] J. Urbani, J. Maassen, and H. Bal. Massive semantic web data compression with mapreduce. In *Proceedings of the 19th*

-
- ACM International Symposium on High Performance Distributed Computing*, pages 795–802. ACM, 2010.
- [VMR⁺11] M. E. Vidal, A. Martínez, E. Ruckhaus, T. Lampo, and J. Sierra. On the efficiency of querying and storing rdf documents., 2011.
- [WKB08] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.
- [WW06] K. Wilkinson and K. Wilkinson. Jena property table implementation, 2006.
- [WZ99] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.
- [ZSC94] Y. Zhou, S. Shekhar, and A. Coyle. Disk allocation methods for parallelizing grid files. In *Data Engineering, 1994. Proceedings. 10th International Conference*, pages 243–252. IEEE, 1994.

