

Aggregated search in Distributed Graph Databases

Rihab Ayed

▶ **To cite this version:**

Rihab Ayed. Aggregated search in Distributed Graph Databases. Databases [cs.DB]. Université de Lyon; Université de Carthage (Tunisie), 2019. English. NNT : 2019LYSE1305 . tel-02520460v2

HAL Id: tel-02520460

https://theses.hal.science/tel-02520460v2

Submitted on 31 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre NNT :

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
l'Université Claude Bernard Lyon 1

Ecole Doctorale 512
(InfoMaths)

Spécialité de doctorat :
Discipline : Informatique

Soutenue publiquement le 16/12/2019, par :
Rihab Ayed

Recherche d'Information Agrégative dans des Bases de Graphes Distribuées

---

Devant le jury composé de :

Nom, prénom	**grade**	**établissement/entreprise**	
Kheddouci, Hamamache	Professeur	Université de Lyon (LIRIS)	**Président**
Rocancio, Claudia	Professeur	Université de Grenoble (LIG)	**Rapporteure**
Faiz, Rim	Professeur	Université de Carthage (LARODEC)	**Rapporteure**
Mephu Nguifo, Engelbert	Professeur	Université Clermont-Auvergne (LIMOS)	**Examinateur**
Khalgui, Mohamed	Professeur	Université de Carthage (INSAT)	**Examinateur**
Hacid, Mohand-Said	Professeur	Université de Lyon (LIRIS)	**Directeur de thèse**
Jemai, Abderrazak	Professeur	Université de Carthage (INSAT, SERCOM/EPT)	**Directeur de thèse**
Haque, Rafiqul	Co-fondateur	Cognitus	**Invité**

P H D T H E S I S

to obtain the title of

PhD of Science

Specialty : Computer Science

Defended by

Rihab Ayed

Aggregated Search in Distributed Graph Databases

prepared at Université de Lyon and Université de Carthage

defended on 16 December, 2019

Jury :

President :	Pr. Hamamache Kheddouci	Université de Lyon (LIRIS)
Reviewers :	Pr. Claudia Roncancio	Université de Grenoble (LIG)
	Pr. Rim Faiz	Université de Carthage (LARODEC)
Examinators :	Pr. Engelbert Mephu Nguifo	Université Clermont-Auvergne (LIMOS)
	Pr. Mohamed Khalgui	Université de Carthage (INSAT)
Advisors :	Pr. Mohand-Said Hacid	Université de Lyon (LIRIS)
	Pr. Abderrazak Jemai	Université de Carthage (INSAT, SERCOM/EPT)
Invited :	Dr. Rafiqul Haque	Cognitus

To all those who will find help in this work,
To all those who are available to recognize themselves,

To 3 continents, 2 roads and 1 path,

Acknowledgments

This work has been elaborated as a part of the CAIR[1] project.

Special thanks are addressed to Frequent Subgraph Mining authors especially to Xifeng Yan, Thorsten Meinl, Andrés Gago-Alonso, Sebastian Nowozin, Christian Borgelt, Mohammad Al Hasan and Sabeur Aridhi for sending us software and datasets, for providing clarifications and for their availability. Also, we would like to thank Sébastien Harispe and Jérôme David for answering our questions in Semantic Web field.

Members of the LIRIS laboratory such as Emmanuel Coquery, Jean-Claude Iehl, Romain Chanu are thanked for the some help they afforded in our technical processes.

A part of this work (*i.e.,* Frequent Subgraph Mining Study) has been elaborated under the supervision of Rafiqul Haque[2]. Thanks are addressed to him.

Collaborations have been done with a trainer Slim Abid[3]. Slim participated in a part of a state of the art about query decomposing and a primal version of query parsing implementation of our approach. In the same scope, we collaborated with a Post-doctorant student Mokhtar Sellami[4] who implemented the query decomposition approach and contributed in the primitive implementation of our structural clustering. Thanks are addressed to them.

Gratitude is finally addressed to all members that participated in the administrative going and financial support of this PhD, mainly in :

- LIRIS laboratory and Université de Lyon 1
- SERCOM laboratory, Ecole Polytechnique de Tunisie, Université de Carthage
- Project CAIR

[1] CAIR home page: `www.irit.fr/CAIR/fr/`

[2] Rafiqul Haque Page : `https://ie.linkedin.com/in/rafiqul-haque-9a2413a`

[3] Slim Abid Page : `https://fr.linkedin.com/in/slim-abid`

[4] Mokhtar Sellami researches `www.researchgate.net/profile/Sellami_Mokhtar3`

Abstract

In this research, we are interested in investigating issues related to query evaluation and optimization in the framework of aggregated search. Aggregated search is a new paradigm for accessing massively distributed information. It aims to produce answers to queries by combining fragments of information from different sources. The queries ask for objects (documents) that do not exist as such in the targeted sources, but are built from fragments extracted from different sources. The sources might not be specified in the query expression, they are dynamically discovered at runtime.

In our work, we consider data dependencies to propose a framework for optimizing query evaluation over distributed graph-oriented data sources. For this purpose, we propose an approach for the document indexing/organizing process of aggregated search systems. We consider information retrieval systems that are graph oriented (more specifically RDF graphs). Using graph relationships, our work is within relational aggregated search where relationships are used to aggregate fragments of information. Our goal is to optimize the access to source of information in a aggregated search system. These sources contain fragments of information that are relevant partially for the query. We aim at minimizing the number of sources to ask, also at maximizing the aggregation operations within a same source. For this, we propose to reorganize the graph database(s) in clusters, dedicated to aggregated queries. We use a semantic or strucutral clustering of RDF predicates. For structural clustering, we propose to use frequent subgraph mining algorithms, we performed for this a comparative study of their performances. For semantic clustering, we use the descriptive metadata of RDF predicates and apply semantic textual similarity methods to calculate their relatedness. Following the clustering, we define query decomposing rules based on the semantic/structural aspects of RDF predicates. Preliminary experiments show that our semantic clustering optimizes the query decomposing and the number of sources to ask.

Keywords

Relational aggregated search, Semantic partitioning, Clustering, Graph mining, Frequent subgraph mining, Query decomposition, Distributed search

Résumé

Le sujet de cette thèse s'inscrit dans le cadre général de la Recherche d'Information et la gestion des données massivement distribuées. Notre problématique concerne l'évaluation et l'optimisation de requêtes agrégatives (*Aggregated Search*). La Recherche d'Information Agrégative est un nouveau paradigme permettant l'accès à l'information massivement distribuée. Elle a pour but de retourner à l'utilisateur d'un système de recherche d'information des objets résultats qui sont riches et porteurs de connaissances. Ces objets n'existent pas en tant que tels dans les sources. Ils sont construits par assemblage (ou agrégation) de fragments issus de différentes sources. Les sources peuvent être non spécifiées dans l'expression de la requête mais découvertes dynamiquement lors de la recherche. Nous nous intéressons particulièrement à l'exploitation des dépendances de données pour optimiser les accès aux sources distribuées.

Dans ce cadre, nous proposons une approche pour l'un des sous-processus de systèmes de RIA, principalement le processus d'indexation/organisation des documents. Nous considérons dans cette thèse, les systèmes de recherche d'information orientés graphes (plus spécifiquement les graphes RDF). Utilisant les relations dans les graphes, notre travail s'inscrit dans le cadre de la recherche d'information agrégative relationnelle (*Relational Aggregated Search*) où les relations sont exploitées pour agréger des fragments d'information. Nous proposons d'optimiser l'accès aux sources d'information dans un système de recherche d'information agrégative. Ces sources contiennent des fragments d'information répondant partiellement à la requête. L'objectif est de minimiser le nombre de sources interrogées pour chaque fragment de la requête, ainsi que de maximiser les opérations d'agrégations de fragments dans une même source. Nous proposons d'effectuer cela en réorganisant les bases de graphes dans plusieurs clusters d'information dédiés aux requêtes agrégatives. Ces clusters sont obtenus à partir d'une approche de clustering sémantique ou structurel des prédicats des graphes RDF. Pour le clustering structurel, nous utilisons les algorithmes d'extraction de sous-graphes fréquents et dans ce cadre nous élaborons une étude comparative des performances de ces algorithmes. Pour le clustering sémantique, nous utilisons les métadonnées descriptives des prédicats dont nous appliquons des outils de similarité textuelle sémantique. Nous définissons une approche de décomposition de requêtes basée essentiellement sur le clustering choisi. Nos expérimentations préliminaires montrent que l'approche de clustering sémantique utilisée réduit le nombre de sous-requêtes et le nombre de sources à interroger.

Mots clés

Recherche d'information agrégative, Partitionnement sémantique, Fouille de graphes, Extraction de sous-graphes fréquents, Décomposition de requêtes, Recherche distribuée

التلخيص

في هذا البحث ، نحن مهتمون بالتدقيق في المسائل المتعلقة بتقييم الاستعلام وتحسينه في إطار البحث المجمّع . البحث المجمّع هو نموذج جديد يمكّن من الوصول إلى معلومات موزّعة على نطاق واسع، ويهدف إلى إنتاج إجابات على الاستفسارات من خلال الجمع بين أجزاء من المعلومات من عدة مصادر. البحث يكمن في ايجاد مكوّنات (وثائق) غير موجودة كما هي في المصادر ، انما هي تبنى من أجزاء مستخرجة من هاته المصادر. قد لا يتم تحديد المصادر في تعبير الاستعلام ، ويتم اكتشاف هذه المصادر ديناميكيًا في وقت التشغيل. في هذه الأطروحة سننظر في تبعيات البيانات في ما بينها لتصميم إطار يحسّن من تقييم الاستعلام في إطار مصادر البيانات الموزّعة.

الكلمات المفاتيح

بحث مجمّع ، التقسيم الدلالي ، تعدين الرسم البياني ، رسم بياني فرعي متكرّر ، تحليل استعلام ، بحث موزّع

Publications & Workshops

National Conferences and Workshops

- Ayed Rihab, Mohand-Said Hacid, Rafiqul Haque and Abderrazek Jemai, "An updated dashboard of Complete Search FSM Implementations in Centralized Graph Transaction Databases", 32ème Conférence sur la Gestion de Données, Principes, Technologies et Applications (BDA), Poitiers, France, 15 - 18 Novembre, 2016

- Ayed Rihab, Mohand-Said Hacid, Rafiqul Haque and Abderrazak Jemai, "An updated survey of Complete Search FSM Implementations in Centralized Graph Transaction Databases", Atelier ACI, 17ème conférence Extraction et Gestion des Connaissances (EGC), Grenoble, France, 23-27 janvier 2017

International Conferences

- Ayed Rihab, Mohand-Saïd Hacid, Rafiqul Haque, and Abderrazek Jemai. "An Intra-algorithm Comparison Study of Complete Search FSM Implementations in Centralized Graph Transaction Databases." In International Symposium on Methodologies for Intelligent Systems, pp. 78-88. Springer, Cham, 2018.

Journals

- Ayed Rihab, Mohand-Saïd Hacid, Rafiqul Haque, and Abderrazek Jemai. (2019). "An updated dashboard of complete search FSM implementations in centralized graph transaction databases". Journal of Intelligent Information Systems. 10.1007/s10844-019-00579-4.

Contents

Introduction **1**

1 Frequent Subgraph Mining **4**
1.1 Introduction . . . 4
1.2 The theory of frequent subgraph mining . . . 5
1.2.1 Data Mining . . . 5
1.2.2 *Frequent* Pattern Mining . . . 5
1.2.3 *Subgraph* Pattern Mining . . . 5
1.2.4 Frequent Subgraph Mining (FSM) . . . 6
1.3 FSM approaches . . . 6
1.3.1 Graph Representation . . . 7
1.3.2 Candidate Generation . . . 9
1.3.3 Subgraph Counting . . . 10
1.4 FSM algorithms . . . 15
1.4.1 FSG . . . 15
1.4.2 gSpan . . . 16
1.4.3 MoFa/MoSS . . . 16
1.4.4 FFSM . . . 16
1.4.5 Gaston . . . 17
1.4.6 DMTL . . . 17
1.4.7 Comparison of FSM algorithms approaches . . . 17
1.5 Main Application Fields of FSM . . . 19
1.6 Conclusion . . . 19

2 Experimental Study of FSM algorithms **20**
2.1 Introduction . . . 20
2.2 Review of State of the Art FSM studies . . . 21
2.2.1 Targeted Categories of FSM algorithms . . . 22
2.2.2 List of FSM Algorithms . . . 23
2.2.3 Performance of FSM Algorithms . . . 25
2.2.4 Specific Cases . . . 26
2.2.5 Availability of Software . . . 27
2.2.6 Experimental setting in literature . . . 27
2.3 Experimental Study . . . 29
2.3.1 Experimental Setup . . . 30
2.3.2 Intra-Algorithm Performance Study . . . 38
2.3.3 Comparison with the State of the Art . . . 51
2.3.4 An Inter-Algorithms Performance Study . . . 61
2.3.5 Discussion . . . 82

2.3.6 Impacts of the Environment variations on the results . 85
2.3.7 Other Options of tested FSM Implementations 90
2.4 Conclusion 91

3 Aggregated Search in RDF Graph Databases 93
3.1 Introduction 94
3.2 Aggregated Search in Information Retrieval 95
3.2.1 Classical Information Retrieval 95
3.2.2 Federated Search 97
3.2.3 Cross-Vertical Aggregated Search 97
3.2.4 Composite Retrieval 99
3.2.5 Relational Aggregated Search 99
3.2.6 A Comparison of Aggregation-Oriented Information Retrieval Approaches 100
3.2.7 Relational AS and Semantic Graphs 101
3.3 Semantic Graph-based Search 102
3.3.1 Graph search 102
3.3.2 RDF graph : A Knowledge representation model . . . 102
3.3.3 SPARQL queries 103
3.4 Used techniques in RDF Federated Search dedicated to Aggregation 105
3.4.1 Aggregated Query Decomposition Strategies 106
3.4.2 RDF Graph Partitioning 111
3.5 Query/Fragment Matching 112
3.6 Relational Aggregated IR Scenario : What is done ? 113
3.6.1 Aggregation of fragments in Exact graph search 115
3.6.2 Aggregation of fragments in Inexact graph search . . . 115
3.6.3 Entity-oriented search and Structured result of aggregation 116
3.6.4 Assembling fragments from hetergeneous sources . . . 116
3.7 Conclusion 116

4 Predicate-based Clustering Approach 118
4.1 Introduction 118
4.2 Clustering Approach for Aggregated Search 118
4.2.1 Semantic Clustering 119
4.2.2 Structural Clustering 126
4.3 Partition-Aware Query Decomposition 126
4.4 Architecture 130
4.5 Conclusion 131

5 Experimental Study of Clustering Approach 132
5.1 Introduction . . . 132
5.2 Definition of Experimental Notions . . . 133
5.2.1 RDF Benchmarks . . . 133
5.2.2 Triple Stores . . . 138
5.2.3 Query Processing Engines and SPARQL servers . . . 139
5.2.4 Query Shapes . . . 139
5.3 Experimental Setup . . . 140
5.3.1 DBPSB Benchmark Dataset . . . 140
5.3.2 RDF Framework . . . 141
5.3.3 Machines Configuration . . . 141
5.3.4 Clustering Configuration . . . 141
5.3.5 DBPSB Queries Characteristics . . . 143
5.3.6 Evaluation Metrics of our Aggregated Search System . 145
5.4 Experimental Results . . . 146
5.4.1 Predicate Relatedness Results . . . 146
5.4.2 Query Decomposing Results . . . 146
5.5 Conclusion . . . 150

Conclusion 151

A Appendix 153
A.1 CAIR Project . . . 153
A.2 FSM Keywords . . . 156
A.3 DBSPB Query Characteristics . . . 158
A.4 DBSPB Query 2012 Decomposing with semantic Clustering . 163
A.5 DBSPB Query 2011 Decomposing with semantic Clustering . 171
A.6 Semantic relatedness of predicates . . . 186

Bibliography 205

List of Figures

1.1 Market Basket Analysis 5
1.2 An example of a graph database [Ke 2008] 6
1.3 Isomorphic graphs [Isom 2015] 9
1.4 A lattice of candidate subgraphs (molecules) for a chemical database [Meinl 2007] 11
1.5 Subgraph occurrences in single graph setting [Abdelhamid 2016] 12
1.6 Example of all Frequent Subgraphs (Exact Matching) 13
1.7 Example of all Frequent Patterns (Approximate Matching) . 14
1.8 Example of a subset of Frequent Subgraphs (Exact Matching) 15

2.1 Examples of SDF file errors - AID2DA99 dataset 33
2.2 gSpan Runtime (Low Support Threshold) - HIV-CA 42
2.3 gSpan Runtime (Low Support Threshold) - PTE 42
2.4 Memory Consumption of Gaston Original versions - AID2DA99 49
2.5 Memory Consumption of Gaston - PTE 50
2.6 Gaston Runtime - AID2DA99 52
2.7 GSpan ParMol Runtime : Comparison of our results (Left) with the Literature [Gago-Alonso 2008, Gago-Alonso 2010a, Gago-Alonso 2010b] (Right) - AID2DA99 54
2.8 gSpan ParMol Memory Consumption : Comparison of our results (Left) with the Literature [Gago-Alonso 2008] (Right) - AID2DA99 54
2.9 Number of Duplicates with gSpan ParMol : Comparison of our results (Left) with the Literature [Gago-Alonso 2008, Gago-Alonso 2010b, Gago-Alonso 2010a] (Right) - AID2DA99 55
2.10 Gaston ParMol Runtime: Comparison of our results (Left) with the Literature [Gago-Alonso 2008, Gago-Alonso 2010a] (Right) - PTE 57
2.11 Gaston ParMol Memory Consumption (GB) : Comparison of our results (Left) with the Literature [Gago-Alonso 2008] (Right) - AID2DA99 58
2.12 FSG Original Runtime : Comparison of our results (Left) with the Literature [Inokuchi 2005] (Right) - PTE 59
2.13 FFSM ParMol Runtime : Comparison of our results (Left) with the Literature [Gago-Alonso 2010a] (Right) - FFSM ParMol - PTE 60
2.14 Classification of FSM Implementation according to the Number of Frequent Subgraphs 68
2.15 FSM Algorithm Runtime (PTE) - (H strategy) 70

2.16 FSM Algorithm Runtime (PTE) - (L strategy) 71
2.17 FSM Algorithm Runtime (AID2DA99) - (H strategy) 72
2.18 FSM Algorithm Memory Consumption (PTE) - (H strategy) 75
2.19 FSM Algorithms Memory Consumption (PTE) - (L strategy) 76
2.20 FSM Algorithms Memory Consumption (AID2DA99) - (H strategy) 79
2.21 Dataset Variation Effect on the Runtime - Gaston Original (L) 86
2.22 Dataset Variation Effect on the Memory - Gaston Original (L) 86
2.23 Dataset Variation Effect on the Number of FS - Gaston Original (L Strategy) 87
2.24 gSpan ParMol Runtime performance by OS and IDE - AID2DA99 dataset 88
2.25 gSpan ParMol Memory Consumption performance by OS and IDE - AID2DA99 dataset 88
2.26 gSpan ParMol Runtime performance by argument variation and OS - AID2DA99 dataset 89

3.1 An associative map about our concerned fields 94
3.2 The Architecture of an IRS 95
3.3 Federated search scenario [simplified] [Nassopoulos 2016] . . . 97
3.4 An example of cross-vertical Aggregated Search Results (Google) [Lalmas 2011] 98
3.5 Examples of Relational aggregated search from two engines (Left : Google, Right : Wolfram Alpha) for "Albert Einstein" query 100
3.6 Aggregation of subgraphs from two graphs *g1* and *g3* responding to the query *q* [Le 2012] 102
3.7 A RDF graph example [Przyjaciel-Zablocki 2011] 103
3.8 Four quadrants of RDF systems [Hammoud 2015] 105
3.9 Result Aggregation with no query decomposing (Case 1) . . . 112
3.10 Result Aggregation with query decomposing (Case 2) 113
3.11 Approaches related to Relational Aggregated Search (July 2018) 114

4.1 From Resources Relatedness to Predicates Relatedness 119
4.2 Use of metadata in our semantic clustering approach 122
4.3 Our Clustering Process of Predicates 124
4.4 Our Aggregated Information Retrieval System 131

5.1 SPARQL BGP Query shapes 139

A.1 A Conceptual Map of Aggregated Search Literature described in CAIR project [French Version] 154
A.2 The axis of study in the CAIR project synthesized in a Conceptual Map [French Version] 155

List of Tables

1.1 Comparison of FSM Algorithms approaches 18

2.1 FSM Algorithms with specific graphs (March 2016) 23
2.2 An exhaustive list of FSM Centralized Algorithms (Complete Search) (March 2016) 24
2.3 The usage of Centralized FSM algorithms (Complete Search) (March 2016) 25
2.4 Contextual performance of FSM algorithms 26
2.5 FSM Algorithms with specific uses 26
2.6 Unavailable FSM algorithms 27
2.7 Available Implementations of FSM Algorithms (Complete Search) (March 2016) 28
2.8 FSM Implementations with Technical Drawbacks (Complete Search) (March 2016) 28
2.9 Characteristics of Tested Centralized Graph Transaction Datasets in the Literature [2016] 29
2.10 Available Datasets used in the Literature (March 2016) . . . 32
2.11 Algorithms' strategy of Minimum Support/Frequency Input . 34
2.12 Machine Characteristics 34
2.13 Framework characteristics 35
2.14 Code Accessibility 36
2.15 Our Estimated Parsing Time of the FSG Algorithm 36
2.16 Abbreviations of implementations in Tables 37
2.17 Number of Frequent Subgraphs by gSpan (L strategy) - HIV-CA 38
2.18 gSpan Original *vs.* gSpan Original 64bit : Number of Frequent Subgraphs Comparison 39
2.19 gSpan Original *vs.* gSpan ParMol : Number of Frequent Subgraphs Comparison 40
2.20 Examples of Memory Consumption of two gSpan versions (L strategy) 41
2.21 Minimal Support threshold value reached by gSpan versions (L strategy) 41
2.22 gSpan Implementations Runtime Comparison (gSpan ParMol *vs.* gSpan Original) 43
2.23 gSpan Implementations Runtime Comparison (gSpan Original versions) 44
2.24 gSpan Implementations Runtime Comparison (gSpan ParMol *vs.* gSpan Original 64bit) 46
2.25 Gaston Original *vs.* Gaston Original RE : Number of Frequent Subgraphs Comparison 47

2.26 Gaston Implementations: Number of Frequent Subgraphs Comparison (Gaston ParMol *vs.* Gaston Original) - (L/H strategy) 48
2.27 Examples of Memory Consumption and Number of Frequent Subgraphs of Gaston versions 49
2.28 Limits of Memory Consumption (KB) of Gaston versions for low support threshold 51
2.29 Number of Frequent Subgraphs of gSpan : Comparison of our results (Left) with the Literature [Nijssen 2004, Nijssen 2006] (Right) - PTE 53
2.30 Memory Consumption (MB) of Gaston Original : Comparison of our results (Left) with the Literature [Nijssen 2004, Nijssen 2006] (Right) - PTE 55
2.31 Gaston Original Runtime : Comparison of our results (Left) with the Literature [Nijssen 2004, Nijssen 2006] (Right) - PTE 56
2.32 Number of Frequent Subgraphs with Gaston ParMol : Comparison of our results (Left) with the Literature [Gago-Alonso 2010a] (Right) - AID2DA99 57
2.33 Number of Frequent Subgraphs with FSG : Comparison of our results (Left) with the Literature [Kuramochi 2002] (Right) - PTE 58
2.34 FSG Runtime (sec) : Comparison of our results (Left) with the Literature [Nijssen 2003] (Right : Nijssen et *al*) - PTE . . 59
2.35 Gaston ParMol *vs.* gSpan ParMol: Number of Frequent Subgraphs Comparison 62
2.36 Number of Frequent Subgraphs (H strategy) - PTE 63
2.37 $gSpan^Z$ *vs.* Gaston Original : Number of Frequent Subgraphs Comparison 63
2.38 Number of Frequent Subgraphs by FSM solutions - (L strategy) - PTE 64
2.39 Gaston Original *vs.* gSpan Original: Number of Frequent Subgraphs Comparison 64
2.40 FSG Original *vs.* Gaston Original : Number of Frequent Subgraphs Comparison 66
2.41 DMTL Original *vs.* Gaston Original (or FSG Original) : Number of Frequent Subgraphs Comparison 66
2.42 MoSS ParMol *vs.* MoFa ParMol : Number of Frequent Subgraphs Comparison 67
2.43 Number of Frequent Subgraphs by MoFa Implementations - (H strategy) - AID2DA99 68
2.44 FSG Original *vs.* gSpan versions (L/H strategy): Runtime Comparison 73
2.45 FSG Original *vs.* gSpan versions (L strategy): Number of Frequent Subgraphs 73

2.46 gSpan *vs.* FFSM ParMol (L strategy): Runtime Comparison 74
2.47 gSpan ParMol *vs.*. Gaston Original versions (L strategy): Memory consumption comparison 77
2.48 gSpan ParMol *vs.* FFSM ParMol (L/H strategy): Memory consumption comparison 78
2.49 Minimal Support threshold value reached by FSM Algorithms- (L strategy) 78
2.50 Memory Consumption (MB) of two MoFa implementations (AID2DA99) - (H strategy) 80
2.51 Bottleneck Experiment of Complete Search FSM Algorithms (Gaston) 80
2.52 Execution of Implementations with Very Dense Datasets - PI 81
2.53 Mining Performance of Gaston RE with Very Dense Datasets - PI (Incomplete Search) 81
2.54 FSM Algorithms with performance drawbacks 83
2.55 FSM Algorithms with performance advantages 84
2.56 FSG Original - Number of Frequent Subgraphs - DS3 *vs.* DS3M 90
2.57 FSG Original Runtime - DS3 *vs.* DS3M 90
2.58 Optional settings for FSM Implementations 91

3.1 A comparison of IR Approaches for aggregation (July 2018) . 101
3.2 An example of triple RDF 103
3.3 An example of SPARQL-like Query and its BGP decomposed 106
3.4 Query federation approaches in RDF search (November 2018) 109

4.1 ADW Results of 5 most related predicates to "artist" 125
4.2 UMBC Results of 5 most related predicates to "artist" 126

5.1 Centralized RDF benchmarks (June 2018) 135
5.2 RDF datasets characteristics in Benchmarks from [Kim 2015] (modified) 137
5.3 DBPedia10 (10% of DBPedia) Characteristics 141
5.4 DBPedia10 Predicate Metadata Ontologies 142
5.5 Query Shapes of QuerySet2011 144
5.6 Query Keywords of QuerySet2011 144
5.7 Query Shapes of QuerySet2012 145
5.8 Query Keywords of QuerySet2012 145
5.9 Predicate relatedness statistics 147
5.10 Semantic relatedness in Queries2012 set 147
5.11 Queries 2012 and the number of subqueries 148
5.12 Queries 2011 and the number of subqueries 149
5.13 Decomposing approaches of Queries2012 set : Targeted sources149
5.14 Decomposing approaches of Queries2011 set : Targeted sources150

A.1 FSM Keywords 157

A.2 DBSPB Queries2011 Characteristics 158
A.3 DBSPB Queries2012 Characteristics 161
A.4 DBSPB Queries2012 Decomposed
(Number of clusters : 10) 164
A.5 DBSPB Queries2011 Decomposed
(Number of clusters : 10) 172
A.6 Semantic relatedness in Queries2011 set 186

Introduction

Background and Context

Information Retrieval mechanisms are evolving with the changing of the Web information from an information space of linked documents to information objects of linked data and heterogeneous information [Haase 2010]. *Aggregated Search (AS)* [Lalmas 2011, Kopliku 2011] is an approach that aims to aggregate the information results from the different sources and return aggregated information richer than the basic list of homogeneous results returned by classic Information Retrieval systems. Sources of information are not necessarily specified in the query but they discovered dynamically in the execution.

AS is an approach with steps and processes relatively newly defined in the literature [Lalmas 2011, Kopliku 2011]. One of its defined subcategories : *Relational Aggregated Search [Kopliku 2011]* aims to aggregate fragments of information of different sources and create objects of information that do not exist as-such in the Web. A result of an aggregated query is obtained after a chain of complex operations serving to aggregating the relevant fragments of information. Each of these fragments constitutes a partial result to the query.

Several recent studies have been proposed in Relational AS such as the study [Kopliku 2011] which aims to aggregate information about an entity (*e.g.,* Albert Einstein) and return a structured table of data about it. The study [Echbarthi 2017] tended to define a query processing strategy for graph aggregation and the study [Elghazel 2011] proposed an indexing strategy based on relational databases dedicated to optimizing graph aggregation. Most of the few studies in Relational AS exploited the *semantic web* to try to form aggregates of data.

Objectives/Motivation

The goal of the CAIR[5] project is to contribute in defining better the main processes (*i.e.,* Query Formulation, Indexing, Query/Document Matching) of an Aggregated Search System. Our goal in this PhD is to define one of two sub-processes of Relational Aggregated Search (*i.e.,* Query and Document Indexing) using the richness of the metadata and relationships in the semantic web (*i.e.,* RDF graph databases) and graphs in general. Our approaches that are dedicated to these two subprocesses try to optimize the

[5]CAIR home page: `www.irit.fr/CAIR/fr/`

aggregation in a context of several sources.[6] The optimization consists on trying to reduce the number of requests to sources of information and the number of aggregations to perform between fragments.

Summary of Contributions

We study in this PhD at first the strategies that could be used for indexing graphs. We select one of the most used approaches : Frequent Subgraph Mining. We elaborate (a) a state-of-the art of all existing algorithms and their implementations in Centralized graph transaction databases. Due to a lack of information about the cases of performance of FSM algorithms, we elaborate (1) an experimental study of FSM available implementations and select some implementations that can be used according to an end-user context.

We are further interested in organizing the graph database by clustering RDF graphs in the way that related fragments (*i.e.,* subgraphs) would be accessed together. We find it interesting to use for clustering, the semantic relatedness of units of graphs (*e.g.,* predicates) and see its effect on aggregated queries compared to a structural aspect. The structural aspect is based on Frequent Subgraph Mining. For clustering the graphs, we studied (b) the state of the art of clustering and partitioning strategies in RDF graphs. These strategies were mainly discussed in Federated Search field. We also elaborate (c) a state of the art of studies in Relational Aggregated Search. We conclude that the existing studies lack the use of the metadata of semantic web for optimizing aggregation. We propose (2) our semantic clustering approach using meta-metadata of RDF graphs. We also define (3) query decomposing approach which is dependent of the clustering. (4) Finally, we propose to experiment our approaches using the DBPSB benchmark.

Thesis Outline

The rest of this dissertation is subdivided into five Chapters :

Chapter 1 presents Frequent Subgraph Mining (FSM) and its approaches and algorithms.

Chapter 2 describes the state-of-the art of FSM algorithms in Centralized graph transaction databases. Also, it proposes an experimental study of available FSM implementations.

Chapter 3 defines what is aggregated search and a comparison of its annex categories. Also, it defines RDF search. A state-of-the art of techniques dedicated to RDF graph aggregation from different sources of information

[6]Sources of information can be created during the indexing from the distribution of one source or extracted from existing distributed sources

is proposed. These techniques concern graph partitioning and query decomposing. We also state the contributions of studies of relational aggregated search in literature.

Chapter 4 proposes our semantic and structural clustering approaches for a relational aggregated graph search system. Also, it defines our query decomposing approach in the search system. The architecture of our system is also illustrated.

Chapter 5 describes the experimental setup and the impact of our clustering approaches on the quality of query decomposing.

Chapter 1

Frequent Subgraph Mining

Contents

1.1 Introduction 4
1.2 The theory of frequent subgraph mining 5
1.2.1 Data Mining 5
1.2.2 *Frequent* Pattern Mining 5
1.2.3 *Subgraph* Pattern Mining 5
1.2.4 Frequent Subgraph Mining (FSM) 6
1.3 FSM approaches 6
1.3.1 Graph Representation 7
1.3.2 Candidate Generation 9
1.3.3 Subgraph Counting 10
1.4 FSM algorithms 15
1.4.1 FSG 15
1.4.2 gSpan 16
1.4.3 MoFa/MoSS 16
1.4.4 FFSM 16
1.4.5 Gaston 17
1.4.6 DMTL 17
1.4.7 Comparison of FSM algorithms approaches 17
1.5 Main Application Fields of FSM 19
1.6 Conclusion 19

1.1 Introduction

This chapter introduces from a conceptual point of view, the frequent subgraph mining algorithms and their respective approaches. The frequent subgraph mining constitutes an important approach for capturing the frequent structure representing an identity for a set of data.

The remainder of this chapter is organized as follows: Section 2 defines the frequent subgraph mining task (FSM). Section 3 describes the approaches used by different FSM algorithms. Section 4 defines available FSM algorithms in literature and compares their approaches. In Section 5, application fields of FSM are mentioned.

1.2 The theory of frequent subgraph mining

We define, in this section, the basic notions around frequent subgraph mining mainly *data* mining (or pattern mining), *frequent* pattern mining, *subgraph* pattern mining and finally frequent subgraph mining.

1.2.1 Data Mining

Data mining defines the process of producing an enumeration of patterns or models from the data, while considering an reasonable computational efficiency. Data mining comprises four main category of mining problems : clustering, classification, outlier analysis and frequent pattern mining [Aggarwal 2014]. We are interested in this chapter by the last one.

1.2.2 *Frequent* Pattern Mining

Frequent pattern mining problem was basically defined on sets. It was further extended to various advanced data types such as spatial-temporal data, graphs, and uncertain data [Aggarwal 2014]. Classical "data-mining" (*i.e.,* frequent pattern mining task) is referring to frequent data values of items and their association rules (*e.g.,* milk and butter in market basket analysis, see Figure[1] 1.1) [Agrawal 1993]. Further, in semi-structured and graph data mining, it focuses on frequent labels and common specific topologies [Inokuchi 2003, Gudes 2006].

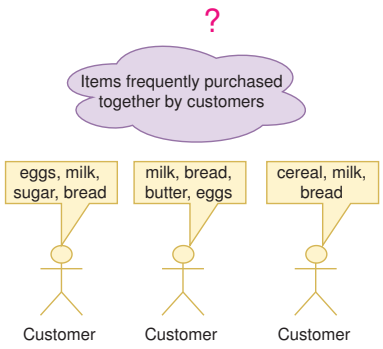


Figure 1.1: Market Basket Analysis

We are interested in graph mining, and in subgraphs as a common specific topology.

1.2.3 *Subgraph* Pattern Mining

In subgraph pattern mining, the interestingness of a subgraph is defined by the task of usage. For exploratory graph mining, *frequent subgraphs* with

[1]Inspired from `https://vnktrmnb.wordpress.com/tag/market-basket-analysis/`

high redundancy are selected. However, for a classification task, *discriminatory subgraphs* with a high quality are considered [Al Hasan 2009b].

We are interested in this chapter by exploratory graph mining.

1.2.4 Frequent Subgraph Mining (FSM)

Mining frequent subgraphs is defined as finding subgraphs that appear frequently in a database according to a given frequency threshold [Inokuchi 2000]. Given a graph database G (*e.g.,* see Figure 1.2) and a minimum support s, the task of a frequent subgraph mining algorithm is to obtain the set of frequent subgraphs that have a support above the defined minimum support threshold [Inokuchi 2000, Kuramochi 2001].

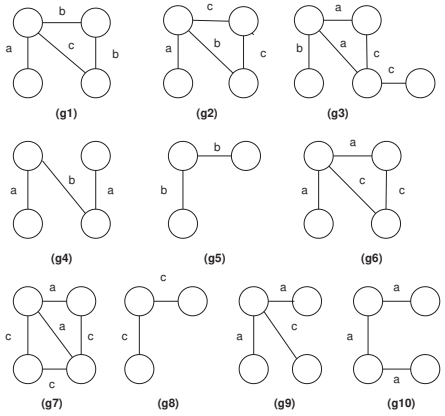


Figure 1.2: An example of a graph database [Ke 2008]

A support of a subgraph g is typically defined by the Equation (1.1).

$$sup = \frac{freq}{|G|} \tag{1.1}$$

freq : number of supergraphs[2] *of g in the database*
$|G|$ *: the number of graphs in the database*

We define in the following the process of FSM with its different approaches.

1.3 FSM approaches

The steps of FSM consist of: *(i)* representing graphs, *(ii)* generating candidate subgraphs, *(iii)* determining the frequency of occurrence of the candidates by performing subgraph ismorphism and further *(iv)* checking and

[2]graphs containing g

filtering the redundant subgraphs [Krishna 2011, Jiang 2013]. The computationally expensive tasks of FSM algorithms are the candidate generation and support computation tasks. The latter is considered as the most computationally expensive for FSM algorithms [Jiang 2013] due to subgraph isomorphism known to be NP-complete.

1.3.1 Graph Representation

We introduce in what follows, the nature of input and output graphs in frequent subgraph mining. Also, we define the database setting. We present graph representation schemes. In addition, we define the canonical labeling strategy which allows to uniquely represent a graph by a chosen scheme.

Graph Topology

In centralized graph transaction mining, the input graphs which are used in most of the FSM algorithms are assumed to be *labeled (vertices and edges), simple*[3]*, connected and undirected* graphs and the output are *connected subgraphs*. However, there are some few algorithms developed for specific graphs (*e.g.,* complex graphs [Acosta-Mendoza 2015], unconnected subgraphs [Skonieczny 2009], vertex labeled graphs [Zeng 2006]). We consider in our study only static graphs (*vs.* stream graphs [Ray 2014]).

Database Setting

There are two distinct problem formulations for frequent subgraph mining in graph datasets: (i) graph-transaction setting and (ii) single-graph setting.

- *Graph-Transaction Setting:* In this case, the input is a collection of moderate sized graphs (transactions). For example, Figure 1.2 illustrates the graph transaction setting with 10 graphs in the database. A subgraph is considered frequent if it appears in a large number of graphs. A subgraph occurrence is counted only once per transaction, independently of the possible multiple occurrences in the same transaction [Inokuchi 2000]. Graph Transaction mining is applied in, *e.g.,* biochemical structure analysis, program control flow analysis, XML structure analysis, image processing and analysis [Aggarwal 2010, Jiang 2013].

- *Single-Graph Setting:* This setting involves mining frequent subgraphs in different regions of one large sized graph. The frequency of a subgraph is based on the number of its occurrences (*i.e.,* embeddings)

[3] A simple graph is "an un-weighted and undirected graph with no loops and no multiple links between any two distinct nodes" [Gibbons 1985]

in the large graph. Special support metrics are used, by considering, for example, the overlapping of two subgraphs [Kuramochi 2005]. Single Graph mining is dedicated to applications such as social networks, citation graphs, or protein-protein interactions in bioinformatics [Elseidy 2014].

The mostly used schemes by frequent subgraph discovery algorithms are adjacency matrix, adjacency list, hash tables and trie data structures [Krishna 2011, Gholami 2012]. In the following, we specify the cases of use of these structures.

Adjacency Matrix

The easiest mechanism whereby a graph structure can be represented is an adjacency matrix where the rows and columns represent vertices, and the intersection of row i and column j represents a potential edge connecting the vertices v_i and v_j [Kuramochi 2001, Jiang 2013].

Adjacency List

Sparse graphs would have several "zeros" in an adjacency matrix. To avoid this waste of memory, adjacency lists are used as they assign memory dynamically [Yan 2002b, Krishna 2011, Dinari 2014].

Hash Table

For very large graphs, hash tables could be used in order to avoid enumerating all possible subgraph isomorphisms for a new subgraph discovery. A hash table scheme uses a hash function which maps keys to their corresponding values [Nguyen 2004, Krishna 2011]. An example of using hash tables is illustrated in [Luo] where subgraphs are ordered by their number of nodes and edges. A level for each size (number of nodes) is created. A key in the hash table corresponds to the labels of nodes of (n-1)-sized subgraphs and the value is a subgraph (containing these nodes) in the level n.

Trie

A Trie [Fredkin 1960] (also named prefix table) is an ordered tree, where all descendants of a node have the same common prefix. Basically, tries are used for strings. In order to avoid redundancy in storing, the trie stores the common prefixes once. Tries can be also managed for graphs where common prefixes represent common subgraphs between the graphs of the database. For example in [Ribeiro 2010], the authors define a concept of G-trie (*i.e.,* a trie for representing graphs). Each node in the trie stores a single vertex from a subgraph and its corresponding edges (coded by boolean values) to ancestor nodes.

Canonical Labeling Strategy

A graph can be represented in different ways depending on how the vertices and edges are ordered (see Figure 1.3). It is important to adopt a labeling strategy ensuring that two identical (*i.e.,* isomorphic) graphs are labeled in the same way [Washio 2003, Nijssen 2004, Jiang 2013]. The basic idea for generating a canonical labelling is to flatten the associated adjacency matrix by concatenating rows or columns to produce a code. Different codes are generated for different adjacency matrices. The canonical form of representation is the maximal or minimal code. The minimal (maximal) code is imposed by the lexicographical ordering [Yan 2002b]. Various canonical labeling schemes have been proposed. The three most significant ones are : *Minimum DFS Code (M-DFSC)* proposed in gSpan algorithm [Yan 2002a], *Canonical Adjacency Matrix (CAM)* proposed in AGM [Inokuchi 2000] and FSG [Kuramochi 2001] algorithms and *Canonical Spanning Tree (CST)* in SPIN algorithm [Huan 2004].

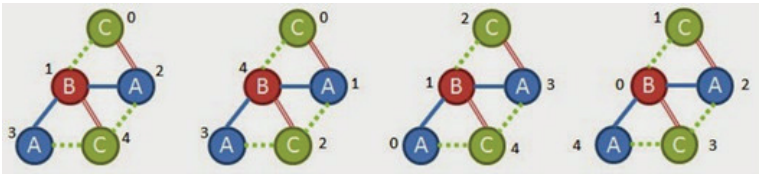


Figure 1.3: Isomorphic graphs [Isom 2015]

1.3.2 Candidate Generation

Candidate generation in a graph dataset poses two main challenges : *(i)* a generation with no redundancy where each subgraph should be generated only once and *(ii)* a generation of candidates that only exist in the dataset [Keyvanpour 2012, Jiang 2013]. Two techniques are defined for candidate generation : *level-wise join* which is related to an *Apriori approach* and a *right-most path extension* technique which is related to *Pattern-growth* approach.

Apriori approach & Level-wise join technique

An apriori approach [Agrawal 1994] consists of generating a new candidate by merging two candidate subgraphs that have been already found and have a common subgraph. This merging approach may generate subgraphs that do not exist in the database [Meinl 2007, Jiang 2013]. The level-wise join technique was introduced by [Kuramochi 2001]. Basically, a *(k + 1)* subgraph candidate is generated by combining two frequent k subgraphs which share the same *(k - 1)* subgraph [Kuramochi 2001].

Pattern-growth approach & Right-most path extension technique

The pattern-growth approach [Han 2000] consists of extending a subgraph candidate by an edge (and a node if no cycle is closed). This edge (and node) has to exist in the database. In data mining, the generation using an increasing order of candidate size is referred to as *level-wise* exploration [Aggarwal 2014]. This method generates only candidates that exists in the database but may generate redundant subgraphs [Meinl 2007]. The right-most path extension technique [Yan 2002b, Borgelt 2002] eliminates the redundancy by adding an extra-edge only on the rightmost path.

Pattern-growth approach was developed to avoid the overhead of the candidate generation by the Apriori Approach. This is done by trying to grow the pattern from a single pattern, instead of joining two patterns [Rehman 2014].

1.3.3 Subgraph Counting

Conceptually, candidates are searched and counted in the underlying graph database. The ones satisfying a minimum support constraint are retained. Practically, candidates need to be tested according to a *search space structure* and a *traversal strategy*. We define these two notions in the following and we also introduce how the subgraphs are counted using the subgraph isomorphism and the support measure. We conclude this subsection by defining the subgraph search and matching strategies.

Search Space & Traversal Strategy

A search space structure is used in order to optimize the exploration of frequent subgraphs [Aggarwal 2014]. The structure should be explored in a level-wise way. If a k-pattern is not frequent then all of its supersets (k+n)-patterns should not be tested. This property is named the *Downward Closure Property (DCP)* or *anti-monotonicity* [Agrawal 1993]. Some search space structures used in literature are enumeration trees (named also lexicographical trees) [Yan 2002b, Aggarwal 2014], lattices [Meinl 2007] and G-tries [Ribeiro 2010]. Some other structures are cited in [Nadimi-Shahraki 2015].

For example, each layer L of a lattice structure contains all L-edge subgraphs (see Figure 1.4) and their frequencies. A connection between two items in the lattice is an extension of a subgraph by an edge and a node (if no cycle is closed). Further, frequent subgraph mining consists of traversing the lattice, reporting all frequent candidates and pruning infrequent ones [Nijssen 2004]. A traversal strategy (called also *enumeration strategy*) for the lattice has to be chosen [Nijssen 2004, Meinl 2007].

There are two main traversal strategies : *breadth-first* (BFS, named also *horizontal* support counting strategy) or *depth-first* search (DFS, also named

vertical support counting strategy). An hybrid strategy can also be performed [Jiang 2013, Aggarwal 2014].

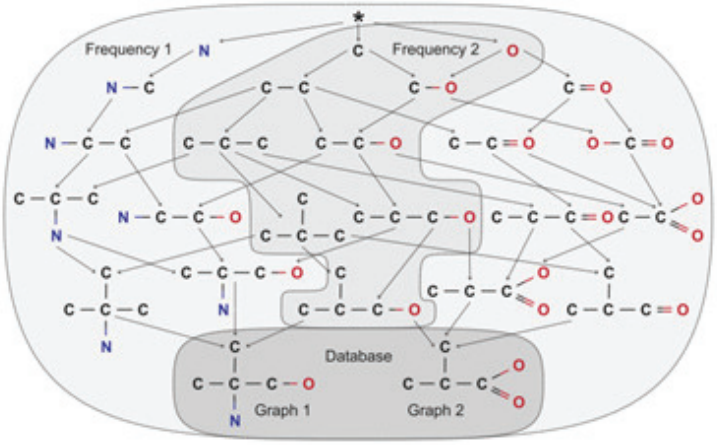


Figure 1.4: A lattice of candidate subgraphs (molecules) for a chemical database [Meinl 2007]

The BFS tends to be more efficient in the pruning of infrequent subgraphs at an early stage in the mining process and this causes higher I/O, memory usage and execution time. However, DFS requires less memory usage but it performs less efficient pruning [Krishna 2011, Jiang 2013].

The memory usage of BFS is due to the number of lists stored in memory [Krishna 2011]. In fact, in case of the DFS strategy, the number of lists is proportional to the depth of the graph (for graph transaction databases, it is equal to the depth of the biggest graph). However, for BFS, it is proportional to the width of the graph (*i.e.,* the maximal number of subgraphs in one level) [Wörlein 2005]. The majority of algorithms traverse the lattice by using a DFS approach since it requires less memory compared to BFS [Meinl 2007].

Subgraph Isomorphism tests & Minimum Support Threshold

The subgraph isomorphism problem is NP-Complete [Cook 1971, Ke 2008]. For counting the support of a pattern, a trade-off is proposed between using explicit subgraph isomorphism (*e.g.,* using a *Transaction Identifier list (TID)* [Yan 2002b] or keeping embeddings of a pattern (*e.g.,* using an *embedding list* [Borgelt 2002]). In a transaction list, each frequent subgraph has a list of transaction identifiers containing the subgraph [Yan 2002b]. The support of a k subgraph is computed using the intersection of the TID lists of (k-1) subgraphs. An embedding list consists of a mapping of the vertices and edges of a candidate to the corresponding vertices and edges in the graph it occurs in. Embedding lists reduce the subgraph isomorphism tests. They

are supposed to reduce the runtime. As a trade-off (time versus memory), they consume a lot of memory [Lakshmi 2012, Jiang 2013, Douar 2014].

Support Computation & Minimum Support Threshold The frequencies of generated candidates in database are counted. The approach of this step is different depending on the configuration of the graph database.

- *Occurrence-based counting*: If the database is a single large graph, the number of occurrences of subgraph is counted by taking in consideration the graph overlapping cases [Vanetik 2002]. For example, in Figure 1.5, the support of the subgraph (A,B,C) is 6, there is an overlapping between subgraphs (*e.g.,* subgraphs (u11,u10,u12) and (u8,u9,u10)), the counting algorithm take into consideration these overlapping.

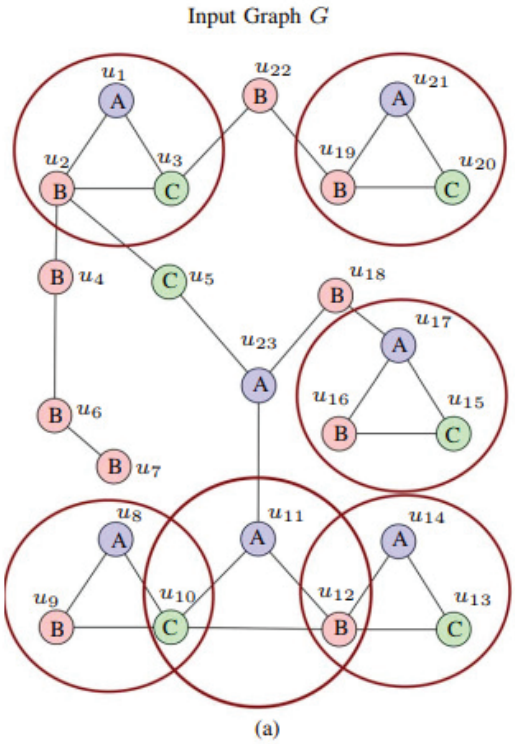


Figure 1.5: Subgraph occurrences in single graph setting [Abdelhamid 2016]

- *Transaction-based counting*: It is considered if the database consists of multiple graphs, the number of graph occurrences is counted once in each graph [Inokuchi 2000].

A subgraph is considered to be frequent if its occurrence count is higher than a predefined threshold value. The absolute occurrence is referred to as

its frequency and the relative occurrence is referred to as its *support*. The threshold is referred to as the *minimum support threshold* [Inokuchi 2000]. The key parameter of the graph mining results is the minimum support threshold used to discover the frequent subgraphs [Douar 2014].

For transaction-based counting, the Downward Closure Property [Agrawal 1993] is associated with the support metric. This property states that if a graph is frequent then all of its subgraphs must also be frequent [Jiang 2013].

For occurrence-based counting, several anti-monotone support metrics are proposed in the literature and they define the support in presence of the overlaps [Vanetik 2002, Kuramochi 2005, Patel 2013, Elseidy 2014].

Subgraph Search & Matching Strategy

FSM algorithms can be classified according to search strategy : *complete* and *incomplete* (or heuristic) search. Also, they can be classified according to the type of isomorphism test (matching) performed between the mined subgraphs : *exact* and *inexact* matching. We describe these categories in what follows.

Complete Search The complete search[4] algorithms perform a complete mining *i.e.,* it guarantees to find all frequent subgraphs from the input data, above a minimum frequency threshold [Kuramochi 2001, Inokuchi 2003].

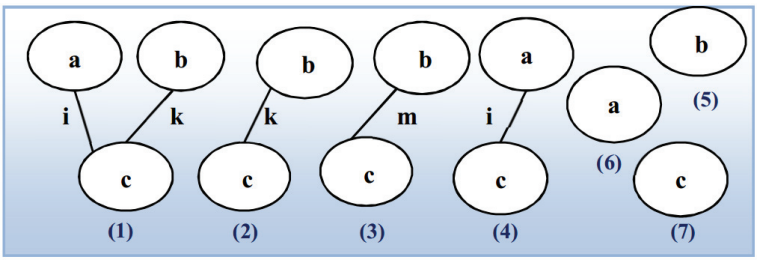


Figure 1.6: Example of all Frequent Subgraphs (Exact Matching)

(a) **Exact Matching:** It consists in finding all possible frequent subgraphs as they appear in the input data [Kuramochi 2001, Inokuchi 2003]. The complete search must return a frequent subgraph (*e.g.,* a subgraph (1) shown in Figure 1.6) and all of its possible subgraphs that are necessarily frequent as well (*e.g.,* subgraphs (2), (4), (5), (6) and (7) shown in Figure 1.6).

(b) **Approximate Matching:** It consists in finding all frequent subgraphs, with an assumption that subgraphs having the same structure and different labels, will all be returned as the same subgraph [Li 2009]. This

[4]Complete search is also called "exact search" [Jiang 2013, Saha 2014], we will use, in this manuscript, only the designation "complete search".

is considered as a complete search because all possible frequent subgraphs could be verified in the output set with the abstraction of labels (edges or vertices). Figure 1.7 illustrates the approximate matching where graphs with different edge labels are considered the same. For example, the subgraphs (2) and (3) in Figure 1.6 could be represented with the approximate matching by one subgraph (2') in Figure 1.7.

Incomplete or Heuristic Search The incomplete and heuristic search algorithms discover a set of frequent subgraphs whose cardinality is greater or lower than the one returned by the complete search. This category of FSM search is used to : (i) reduce the set of frequent subgraphs (use of exact [Yan 2003] or approximate matching [Cook 1994]), or (ii) add more frequent subgraphs than the complete search in order to consider the inaccuracy or uncertainty of the input data (use of approximate matching) [Zou 2010].

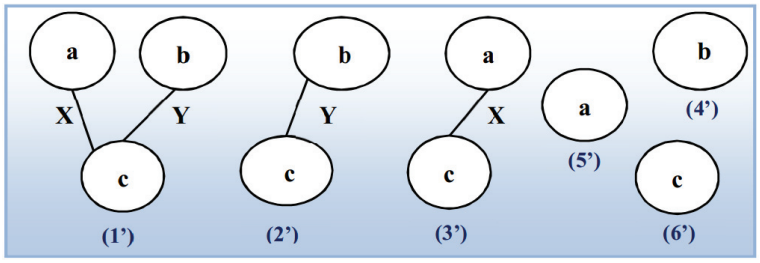


Figure 1.7: Example of all Frequent Patterns (Approximate Matching)

(c) **Exact Matching:** It consists in returning a subset of frequent subgraphs [Wang 2005] by setting a supplementary calculable parameter (*e.g.,* maximum size of frequent subgraphs, closed subgraphs, maximal subgraphs, maximum support threshold) [Yan 2003, Huan 2004, Al Hasan 2009a], besides the minimum support threshold. Figure 1.8 shows an example returning a subset of frequent subgraphs (see all frequent subgraphs, Figure 1.6) where the set parameter is the maximum size of frequent subgraphs (set to 2 edges).

(d) **Approximate Matching:** It consists in either (i) reducing the output by returning a set of representative frequent patterns or (ii) enriching the frequent subgraphs output by considering the inaccuracy or uncertainty of data [Zou 2010, Jia 2011]. For the first case, a representative frequent pattern is a frequent subgraph similar to a set of other frequent subgraphs. In other words, frequent subgraphs that have some differences regarding edges, vertices and labels are represented by one pattern in the output [Al Hasan 2007]. For the second case, it consists in adding infrequent subgraphs that are similar to frequent subgraphs with respect to the structure or labels [Acosta-Mendoza 2012].

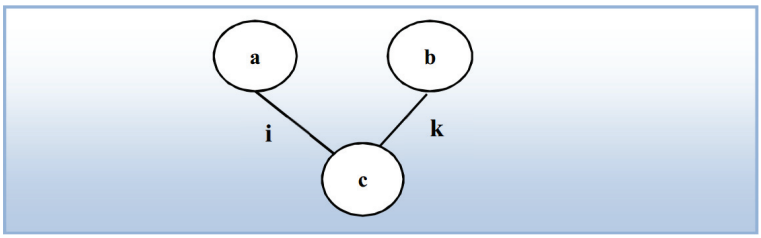


Figure 1.8: Example of a subset of Frequent Subgraphs (Exact Matching)

1.4 FSM algorithms

FSM algorithms have been proposed from a period since 1994 to the present. Since 2007, the proposed algorithms represent a variation of existing algorithms. The FSM field is reaching its maturity [Jiang 2013], so a synthesized study about existing algorithms should be completed. Through our study of FSM algorithms, we collected the characteristics of many of them.[5]

In this section, we briefly describe some algorithms. The selected algorithms are those which will be studied in chapter 2. The justification of the selection of these algorithms will also be provided in chapter 2.

An FSM algorithm can be considered as efficient according to the used strategies for the mining subtasks [Jiang 2013]: (i) the graph representation structure (*e.g.,* adjacency list, adjacency matrix, hash table, tries, see Section 1.3.1), (ii) subgraph candidate generation (*i.e.,* extending, joining or combinational, see Section 1.3.2) using a search approach (*i.e,* apriori or pattern-growth), (iii) canonical graph representation for filtering duplicates (*i.e.,* the two main representations are CAM : Canonical Adjacency Matrix and M-DFSC : Minimum DFS Code, see Section 1.3.3), (iv) subgraph isomorphism detection strategy to compute the support (*i.e.,* keeping embedding of patterns or explicit subgraph isomorphism, see Section 1.3.3).

We describe the six algorithms we selected. The description is done according to strategy used for each subtask.

1.4.1 FSG

FSG (Frequent Subgraph Discovery) [Kuramochi 2001] uses adjacency lists for storing graphs [Krishna 2011]. It uses an Apriori approach. It requires a large amount of memory because it employs BFS and generates a large volume of candidate patterns. Consequently, it scans many times the database and examines a large number of candidates [Nadimi-Shahraki 2015]. It uses the CAM canonical representation [Wörlein 2005]. It generates candidates using the level-wise join technique. It uses transaction list for support counting. It has a bad performance on graphs with many vertices and edges of

[5]Please refer to this open access document `https://docs.google.com/document/d/1qy2OEDSMy5jUinTEJXBIpvwUYr7daNjO7_Lum-vWQ8I/edit?usp=sharing`

identical labels and could be inefficient for mining large-sized subgraph patterns.[6]

1.4.2 gSpan

GSpan (Graph-based Substructure Pattern Mining) [Yan 2002b] uses adjancency matrix. It uses M-DFSC as a canonical representation. It uses a DFS lexicographic ordering to construct a tree-like lattice over all possible patterns, resulting in a hierarchical search space called a DFS code tree [Douar 2014]. It performs a rightmost path expansion as subgraph extension [Wörlein 2005] which means that the k subgraphs are generated by one edge expansion from the k-th level of the DFS tree. Unlike embedding list saving algorithms, gSpan saves transaction list for each discovered pattern which saves on memory usage. GSpan, with some minor changes, can be used for directed graphs [Jiang 2013].

1.4.3 MoFa/MoSS

MoFa (Molecular Frequent Miner) or MoSS (Molecular SubStructure miner) [Borgelt 2002] is a specialized miner for molecular data. It enables to find frequent molecular substructures and discriminative fragments. However, it can also work on arbitrary graphs. The algorithm is inspired by the Eclat algorithm[7] for frequent item set mining. MoFa stores graphs in adjacency matrices. It follows the pattern growth approach. It uses a rightmost path extension. New subgraphs are built by extending former subgraphs with an edge (and a node if necessary). It uses embedding lists to remove duplicates [Meinl 2007]. It is able to mine directed graphs [Jiang 2013].

1.4.4 FFSM

FFSM (Fast Frequent Subgraph Mining) [Huan 2003] is based on gSpan. It uses adjacency matrix for graphs. It follows pattern-growth approach. FFSM uses the CAM representation for canonical graph representation [Wörlein 2005]. The CAM tree of the database is built dynamically using two matrix operations of join and extension [Gago-Alonso 2010a]. FFSM completely avoids subgraph isomorphism testing by maintaining an embedding set for each frequent subgraph [Huan 2003]. The embedding lists allow to avoid excessive subgraph isomorphism tests and therefore avoid exponential runtime (see Section 1.3.1). However, as a trade-off, FFSM faces exponential memory consumption instead [Douar 2014]. FFSM cannot be used in the context of directed graphs due to its use of triangle matrices [Wörlein 2005, Jiang 2013].

[6] http://web.ecs.baylor.edu/faculty/cho/4352/

[7] Eclat webpage: http://www.borgelt.net/eclat.html

1.4.5 Gaston

Gaston (GrAph/Sequence/Tree extractiON) [Nijssen 2004] is based on gSpan. It uses a hash table representation which pleads for its performances over the other algorithms [Krishna 2011]. It follows the pattern-growth approach. Also, Gaston is the fastest among other algorithms [Nijssen 2004] due to the fact that it performs subgraph extension using a quick-start principle where paths and trees are considered at first, and general graphs with cycles are enumerated at the end [Krishna 2011]. To detect the duplicate subgraphs, a well-known algorithm, namely Nauty [McKay 1981] is used to deal with the NP-complete subgraph isomorphism problem [Han 2007]. Gaston scans the database only once because it uses embedding lists stored in main memory [Lee 2012]. Gaston cannot be used in the context of directed graphs unless considering major changes [Wörlein 2005, Jiang 2013].

1.4.6 DMTL

DMTL (Data Mining Template Library) [Al Hasan 2005] is a library for frequent pattern mining. It offers implementations to mine four types of patterns - itemsets, sequences, trees and graphs - in a unified platform. It performs the join of two patterns to generate one or more new candidates. It counts support by using a vertical representation of patterns named Vertical Attribute Table (VAT) (*i.e.,* a list of transactions in which the pattern occurs). This vertical representation is typically faster than the horizontal representation of the database due to I/O cost reduction. The join of patterns is associated with a back end operation : the intersection of two VAT tables of patterns.

1.4.7 Comparison of FSM algorithms approaches

Several comparative tables of the FSM algorithms approaches exist in the literature [Al Hasan 2010, Krishna 2011, Keyvanpour 2012, Lakshmi 2012, Jiang 2013, Dinari 2014, Acosta-Mendoza 2015, Ramraj 2015]. We summarize, here, the comparison of the available FSM algorithms (see chapter 2) according to their used strategies. The comparison concerns only algorithms performing complete search in Centralized graph transaction databases. The comparison of the six algorithms (*i.e.,* FSG [Kuramochi 2001], gSpan [Yan 2002b, Yan 2002a], MoFa [Borgelt 2002], FFSM [Huan 2003], Gaston [Nijssen 2004, Nijssen 2005b] and DMTL [Al Hasan 2005]) approaches is presented in Table 1.1. AGM [Inokuchi 2000] and its extensions (*i.e.,* AcGM [Inokuchi 2002], B-AGM [Inokuchi 2003, Inokuchi 2005]) are added in the comparison since AGM is one of the pioneers in the FSM field.

It is expected that algorithms which use a DFS strategy (*e.g.,* gSpan, Gaston, DMTL, MoFa, see Table 1.1) will be more efficient in terms of time and memory than the ones that use BFS (*e.g.,* FSG, see Table 1.1).

Table 1.1: Comparison of FSM Algorithms approaches

Algorithm	**Graph Typology**	**Graph Representation**	**Candidate Generation**	**Subgraph Counting**
AGM & B-AGM	- Directed or undirected - Labeled - Graphs with loops	Adjacency matrix & CAM	level-wise join	Database scan & BFS
AcGM	- Directed or undirected - Labeled - No self-loop of vertices	Adjacency matrix & CAM	level-wise join	Database scan & Search tree & BFS
FSG	- Undirected (extendable to directed) - Labeled	Adjacency list & CAM	level-wise join	Transaction list & BFS
gSpan	- Undirected (extendable to directed) - Labeled	Adjacency list & M-DFSC	Rightmost path extension	Transaction list & DFS
MoFa	- Undirected (extendable to directed) - Labeled	Adjacency matrix	Rightmost path extension	Embedding list & DFS
FFSM	- Undirected (hardly extendable to directed [Jiang 2013]) - Labeled	Adjacency matrix & CAM	Join & Extension	Embedding list & DFS
Gaston	- Undirected (hardly extendable to directed [Jiang 2013]) - Labeled	Hash table	path, tree and graph enumeration	Embedding list & DFS
DMTL	- Directed or undirected - Labeled - Acyclic graphs	Adjacency matrix & CAM	Join of same type of pattern	Transaction list (VAT table) & DFS

1.5 Main Application Fields of FSM

Mining frequent subgraphs in graph databases is a relevant task for several application fields (*e.g.,* Process models, financial processes, Biochemistry) and goals (*e.g.,* indexing, sampling). However, the most tested datasets in the literature characterize data from the field of *cheminformatics* and *bioinformatics*. In cheminformatics, frequent molecular fragments help finding new drugs [Wörlein 2005]. We tried to enumerate the number of tested real and centralized graph transaction datasets in the literature. We found around thirty-one datasets. 78% of the tested datasets (24 out of 31) are chemical and protein datasets. The rest are of different other fields (*e.g.,* US stock market database [Wang 2006], Money Laundering Case dataset [Li 2010], COIL image database [Acosta-Mendoza 2015] and dataset from the UCI KDD archive [Thomas 2010]). This intensive use of chemical and protein datasets has been criticized by some authors [Saha 2014] since these kind of datasets tend to be tree-like graphs and so performance results tend to be better for the presented algorithms.

1.6 Conclusion

In this chapter, we presented the main notions concerning the frequent subgraph mining, the used approaches and the most known relative algorithms. It was important to define the different approaches in order to categorize FSM algorithms and be able to detect the ones that are useful for our context (*i.e.,* Aggregated Information Retrieval System). Also, the presented techniques give expectations about the algorithm efficiency (Hash table for Gaston *vs.* adjacency matrix for AGM, see Table 1.1). In the following chapter, we will present the most efficient FSM algorithm solutions (from the literature experiments and our proper experiments).

Chapter 2

Experimental Study of FSM algorithms

Contents

2.1 Introduction 20
2.2 Review of State of the Art FSM studies 21
2.2.1 Targeted Categories of FSM algorithms 22
2.2.2 List of FSM Algorithms 23
2.2.3 Performance of FSM Algorithms 25
2.2.4 Specific Cases 26
2.2.5 Availability of Software 27
2.2.6 Experimental setting in literature 27
2.3 Experimental Study 29
2.3.1 Experimental Setup 30
2.3.2 Intra-Algorithm Performance Study 38
2.3.3 Comparison with the State of the Art 51
2.3.4 An Inter-Algorithms Performance Study 61
2.3.5 Discussion 82
2.3.6 Impacts of the Environment variations on the results 85
2.3.7 Other Options of tested FSM Implementations . . 90
2.4 Conclusion 91

2.1 Introduction

In this chapter, we propose to elaborate a synthesis regarding the algorithms and their existing solutions for frequent subgraph extraction. In the framework of the CAIR[1] project, we are in a context of aggregated relational information retrieval system, where retrieved documents are graphs. For this, we are mainly interested by the algorithms that perform in a bunch of labeled graphs (instead of single large garph). Also, we consider static graphs (*vs.* Dynamic graphs). Our objective is to conduct an investigation on implementations of complete search FSM algorithms. The goal is to find

[1]CAIR home page: `www.irit.fr/CAIR`

the most efficient implementation that would be used to cluster graphs in order to optimize the performance of an aggregated search system.

Frequent subgraph mining algorithms are widely used in various areas for complex analysis. As yet, a handful number of algorithms have been proposed in literature. Several experimental studies were reported; however, these experiments lack some critical details which are vital to select an implementation of an algorithm for a specific purpose.

For this, we elaborated an experimental study with implementations of complete search Frequent Subgraph Mining (FSM) algorithms in centralized graph databases. Thirteen working implementations are experimented. In what follows, we provide details of the experimental results in terms of performance metrics and input variation effect. We propose a preliminary selection of the most efficient FSM solutions (*i.e.,* implementations) for end users based on the literature datasets. We attempted to compare our results with state of the art.

The remainder of this chapter is organized as follows: Section 2.2 describes our approach for selecting FSM algorithms and their respective implementations. Section 2.3 describes the evaluation of the selected FSM algorithms' implementations and discusses the results.

2.2 Review of State of the Art FSM studies

A large volume of literature was dedicated to FSM algorithms. These algorithms can be classified according to the search type (*complete* or *incomplete*) and matching strategy (*exact* or *approximate*). Several studies (see, *e.g.,* [Saha 2014, Wörlein 2005, Nijssen 2006, Rehman 2014, Gago-Alonso 2008, Krishna 2011]) were devoted to benchmarking these algorithms. These studies revealed different aspects mainly strength, and weaknesses - of these algorithms - that are critical to select potential candidates for a specific need. However, our investigation summarizes the following shortcomings of these studies : (i) the conclusions about algorithms do not explicitly consider the effects of variability of inputs on performance. The variability includes the characteristics of datasets (*e.g.,* size, density) and the minimum support threshold interval (low or high values) ; (ii) two different implementations of a given algorithm - provided by original authors and the third party implementers - reported different performance results ; (iii) the most recent experimental comparisons (2014) [Rehman 2014, Saha 2014, Aridhi 2015, Douar 2014] are concerned with at most four algorithms. These algorithms are relatively old (proposed between 2001 and 2007). About thirteen new algorithms of the same category have been proposed since 2007 ; (iv) no comparison of currently available FSM algorithms is provided in literature ; (v) the implementations of some algorithms are refined without any experimental study regarding their per-

formance (*e.g.,* gSpan (2002) [Yan 2002b] release v.6 2009).

To the best of our knowledge, no exhaustive list of FSM algorithms has been provided so far. Also, there is no study that cites all the currently available FSM implementations. In this section, we provide a list of all algorithms for the Complete search category 1.3.3 in centralized graph transaction databases (see Section 1.3.1) and highlight their availability and usefulness. We justify this choice in the following. Then, we select a few of them. To establish our selection process, we defined a set of criteria which includes: *performance reported in literature*, *availability of implementation*, and *specific cases of use*. We also point out the ambiguities, found in state of the art regarding the most efficient algorithm to use. We provide details of the experiments settings reported in literature, in order to make our further experimental configurations understandable.

2.2.1 Targeted Categories of FSM algorithms

We target algorithms performing complete search in centralized graph transaction databases according to our context of use.

Centralized graph transaction databases

We are interested in this work by algorithms performing on *centralized graph transaction databases* (see Section 1.3.1). Since the scope of our use concerns the application of Information Retrieval Systems where the database consists, traditionally, in a set of documents (medium sized graphs). In a preliminary way, algorithms performing on distributed databases or consisting of parallel processes are not included in our work.

General graphs

Input graphs are supposed to be *labeled, static and general graphs*. Output subgraphs are supposed to be *connected*. Algorithms developed only for specific graphs (*e.g.,* complex graphs [Acosta-Mendoza 2015], unconnected subgraphs [Skonieczny 2009], vertex labeled graphs [Zeng 2006], see Table 2.1), are not considered in our work.

Complete search category

As mentionned in chapter 1, FSM algorithms output different types of results (*i.e.,* complete/incomplete set, exact/approximate subgraphs) according to the search need. For the four subcategories of FSM search strategy and matching (*i.e.,* a, b, c and d, see Section 1.3.3), there are 31, 1, 22 and 15 algorithms respectively.[2]

[2]Please refer to `http://liris.cnrs.fr/rihab.ayed/ACFSM.pdf` to have the list of all FSM algorithms in centralized graph transaction databases.

Table 2.1: FSM Algorithms with specific graphs (March 2016)

Input Cases	**Algorithms**
Complex graphs	MgVEAM [Acosta-Mendoza 2015]
Directed graphs	mSpan [Li 2009]
Directed Acyclic graphs	DIGDAG [Termier 2007]
Unlabeled graphs	The smoothing-clustering framework [Chen 2008]
Vertex-labeled graphs	Cocain [Zeng 2006], TSMiner [Jin 2005]
Relational graphs	CODENSE [Hu 2005], CLOSECUT & SPLAT [Yan 2005], Fp-GraphMiner [Vijayalakshmi 2011]
Geometric graphs	gFSG [Kuramochi 2005], MaxGeo [Arimura 2007], FREQGEO [Nowozin 2008]
Uncertain graphs	Monkey [Zhang 2007], RAM [Zhang 2008], MUSE [Zou 2009]
Output Cases	**Algorithms**
Cliques and quasi-cliques from dense graphs	CLAN [Wang 2006], Cocain [Zeng 2006]
Unconnected subgraphs	UGM [Skonieczny 2009]

In our work, we are interested in algorithms that perform a complete search (subcategories a, b). Our main objective is to identify an efficient FSM algorithm for generating subgraphs which will be used to index large repositories. The incomplete search algorithms (subcategories c, d) that are available and usable for general purposes propose to return : (i) closed subgraphs ([Yan 2003, Takigawa 2010]), (ii) maximal subgraphs [Huan 2004, Al Hasan 2009a], (iii) significant subgraphs [Yan 2008], (iv) sample of fixed size subgraphs [Saha 2014] or (v) approximate subgraphs [Jia 2011]. The closed and maximal subgraphs could not be used for the purpose of indexing [Yan 2004]. The sampling and approximation of subgraphs can be used for indexing. However, we did not select probabilistic or approximation algorithms to avoid the impact of their output set (*i.e.,* frequent subgraphs) on our indexing approach.

In this study, we include algorithms with all types of approaches (*e.g.,* BFS/DFS, Apriori/Pattern-growth, see Section 1.3) with no restriction.

2.2.2 List of FSM Algorithms

We identified thirty-two algorithms (in the literature) designed to extract all possible frequent subgraphs above a minimum support threshold (see Table 2.2). Before studying the performance and availability of these algorithms, we investigated their usage. We define the usage of an algorithm in accor-

dance with three facets: (i) *the number of experiments*[3] *performed with the algorithm for centralized graph transaction datasets*, (ii) *the number of real datasets* used for testing, and (iii) *the most recent experiment (i.e., paper*[4]*) with the algorithm*. In Table 2.3, E, D and R denote these facets, respectively. We found that eleven out of the thirty two algorithms are relatively more popular. Table 2.3 shows that the most tested algorithms in the literature are: gSpan [Yan 2002b], Gaston [Nijssen 2004], FSG [Kuramochi 2001] and FFSM [Huan 2003].

Additionally, Table 2.3 illustrates that the recent FSM algorithms (*e.g.,* LC-Mine [Douar 2014]) are compared with the least recent algorithms (*e.g.,* gSpan [Yan 2002b], FSG [Kuramochi 2001]), instead of the most recent ones. Questions are raised about the availability and performances of each algorithm among the 32 ones proposed.

In what follows, we discuss the outcome of our investigations in terms of performance, availability and specific cases of use.

[3]We counted the number of distinct authors experiments. Authors that experimented the algorithm in many papers are counted once

[4]Original paper of the algorithm is not considered

Table 2.2: An exhaustive list of FSM Centralized Algorithms (Complete Search) (March 2016)

Algorithm	**Author**	**Algorithm**	**Author**
WARMR	[Dehaspe 1998]	ADI-Mine & GraphMiner	[Wang 2005, Yan 2008]
AGM	[Inokuchi 2000]	TSMiner	[Jin 2005]
FARMER	[Nijssen 2001]	FSP	[Han 2007]
MOLFEA	[Kramer 2001]	DMTL	[Al Hasan 2005]
AcGM	[Inokuchi 2002]	gRed	[Gago-Alonso 2008]
B-AGM	[Inokuchi 2003, Inokuchi 2005]	FSMA	[Wu 2008]
FSG	[Kuramochi 2001]	mSpan	[Li 2009]
FREQGEO	[Nowozin 2008]	SyGMA	[Desrosiers 2007]
MoFa/MoSS	[Borgelt 2002]	CGM & UGM	[Skonieczny 2009]
DPMine	[Gudes 2006]	gdFil	[Gago-Alonso 2010a]
gSpan	[Yan 2002b, Yan 2002a]	grCAM	[Gago-Alonso 2010b]
Topology	[Hong 2003]	ADI-Minebio	[de Sousa Gomide 2011]
FFSM	[Huan 2003]	Fp-GraphMiner	[Vijayalakshmi 2011]
DSPM	[Cohen 2004]	FSMA	[Gao 2012]
AGM-H	[Nguyen 2004]	LC-Mine: FGMAC & AC-miner	[Douar 2014]
GASTON	[Nijssen 2004, Nijssen 2005b]	IDFP-tree	[Nadimi-Shahraki 2015]

Table 2.3: The usage of Centralized FSM algorithms (Complete Search) (March 2016)

Algorithm	**E**	**D**	**R**
gSpan [Yan 2002b]	25	25	[Nadimi-Shahraki 2015]
Gaston [Nijssen 2004]	11	14	[Saha 2014]
FSG [Kuramochi 2001]	9	11	[Douar 2014]
FFSM [Huan 2003]	5	10	[Rehman 2014]
AcGM [Inokuchi 2002]	4	3	[Saha 2014]
MoFa [Borgelt 2002]	3	6	[Skonieczny 2009]
FSP [Han 2007]	2	3	[Rehman 2014]
ADI-Mine [Wang 2004]	2	3	[Wang 2006]
FSMA [Wu 2008]	2	0	[Vijayalakshmi 2011]
MOLFEA [Kramer 2001]	2	2	[Inokuchi 2005]
WARMR [Dehaspe 1998]	2	1	[Nijssen 2004]
LC-Mine [Douar 2014]	1	10	-
The remaining 20 algorithms	1	<5	-

2.2.3 Performance of FSM Algorithms

Studies in the literature reported that the performance of four algorithms, namely WARMR [Dehaspe 1998], FARMER[5] [Nijssen 2001], UGM & CGM[6] [Skonieczny 2009] and MOLFEA [Kramer 2001], is commonly poor. Also, we found that FSMA algorithm [Gao 2012] was experimented moderately and was not compared with any FSM algorithm. Therefore, we removed these five algorithms from the list of potential candidates.

It is worth noting that we found performance ambiguities in several experiments of well-known FSM algorithms. This led to a confusion for choosing the best candidates.

Table 2.4 shows some examples of ambiguities, which include: *(i)* no general conclusion determines which of the two algorithms FFSM and gSpan is the most efficient (see *case b* in Table 2.4); *(ii)* the performance comparison of Gaston and gSpan depends on the dataset (*e.g.,* large NCI dataset [Wörlein 2005]) and the used implementation (Gaston or Gaston RE) (see *case a* in Table 2.4). The contexts of the experiments (*e.g.,* FSM implementation, the support threshold interval, datasets characteristics) were not defined adequately in order to have a complete view of the FSM solutions performance. In Section 2.3, we conduct such a study, and highlight the best implementations with the specification of their performance cases.

[5]WARMR and FARMER were both used mainly for itemsets and complex relations

[6]MoFa is competitive with UGM&CGM. MoFa has a poor performance compared to Gaston, gSpan, FFSM [Nijssen 2003, Wörlein 2005]

Table 2.4: Contextual performance of FSM algorithms

a) Is Gaston or gSpan a more efficient algorithm?	
It is Gaston	*It is gSpan*
* Gaston was the fastest graph mining algorithm compared to gSpan and FSG [Nijssen 2004] * Gaston RE was the best memory consumer over Gaston, FFSM and gSpan [Nijssen 2006]	* For the large dataset NCI and for low support threshold, Gaston was slower than gSpan [Wörlein 2005] * GSpan was the best memory consumer comparing to Gaston and FFSM [Wörlein 2005]
b) Is FFSM more efficient than gSpan ?	
Yes	*No*
* FFSM outperformed gSpan [Huan 2003] * FFSM achieved a considerable performance gain over gSpan [Patel 2013]	* GSpan was slightly faster than FFSM. GSpan was the best algorithm regarding its memory requirements compared to FFSM, MoFA, Gaston [Wörlein 2005] * GSpan was almost as competitive as Gaston and FFSM, at least with not too big fragments [Douar 2014]
c) Is FSG an efficient algorithm to use?	
No	*Yes*
* GSpan outperformed FSG by an order of magnitude in terms of runtime [Yan 2002a]. * AcGM was faster than FSG [Inokuchi 2002]	* GSpan and FSG are placed among the most efficient graph miners in their respective categories [Douar 2014]

2.2.4 Specific Cases

In this work, we intend to study algorithms that propose generic usage. We removed four algorithms (FREQGEO [Nowozin 2008], TSMiner [Jin 2005], SyGMA [Desrosiers 2007] and ADI-MineBio [de Sousa Gomide 2011]) due to their usability for specific cases of input graphs (*e.g.,* SyGMA [Desrosiers 2007] requires that graphs have few labels, see Table 2.5). Twenty-three algorithms for general use are kept for comparison.

Table 2.5: FSM Algorithms with specific uses

Algorithm	**Input Graphs Case**
FREQGEO [Nowozin 2008]	Geometric Graphs (2D or 3D)
TSMiner [Jin 2005]	Graphs with unlabeled edges
SyGMA [Desrosiers 2007]	The number of labels has to be small
ADI-MineBio [de Sousa Gomide 2011]	The input data is relational tables Dedicated for specific biomedical data

2.2.5 Availability of Software

We tried to collect the implementations of the twenty-three algorithms. However, only one-third implementations (7 out of 23) are publicly available.

According to our study, the reasons of unavailability are (see Table 2.6): *(i)* legal constraint (intellectual property right), *(ii)* codes are lost, *(iii)* no response from the authors following our requests.[7]

Table 2.6: Unavailable FSM algorithms

Algorithms	**Unavailability**
AGM [Inokuchi 2000], Topology [Hong 2003], AGM-H [Nguyen 2004], B-AGM [Inokuchi 2003], ADI-Mine [Wang 2004], FSP [Han 2007], FSMA [Wu 2008], mSpan [Li 2009], LC-Mine framework [Douar 2014], IDFP-tree [Nadimi-Shahraki 2015]	No answer from authors
gRed [Gago-Alonso 2008], gdFil [Gago-Alonso 2010a], grCAM [Gago-Alonso 2010b]	Under intellectual properties
DPMine [Gudes 2006], DSPM [Cohen 2004], Fp-GraphMiner [Vijayalakshmi 2011]	The code is lost

There are different implementations of the seven remaining algorithms (see Table 2.7). AcGM and four implementations of gSpan, FFSM and Gaston were removed from the list due to technical shortcomings (see Table 2.8 for the details). We could have tried to debug the implementations but our main objective is to use and compare existing implementations as such, without making any changes. The final list of candidates contains six algorithms with their thirteen implementations. We performed an experimental study with these implementations.

In the following, we will present the experimental setting used in the literature. Our setting choices consider the literature setting.

2.2.6 Experimental setting in literature

We found different experimental settings in literature used for testing FSM algorithms. In this section, we briefly describe those settings.

Datasets

For experimenting FSM implementations in centralized environment, the largest datasets found in the literature have a number of graphs (|D|) not exceeding 274 860 graphs. The average graph size (|T|) does not exceed 50 edges. The maximum number of labels (|L|) for these datasets is 90 vertices and 4 edges. For the most dense datasets, the average graph size does not exceed 3636 vertices and 206 747 edges.

[7] 1 request and 2 reminders have been sent to authors

Table 2.7: Available Implementations of FSM Algorithms (Complete Search) (March 2016)

Algorithm	**Available versions**	**Last Release**
FSG [Kuramochi 2001]	FSG Original v1.37 (PAFI v1.0.1) [Karypis 2003]	2003
gSpan [Yan 2002b]	gSpan Original v.6 [Yan 2009]	2009
	gSpan Original 64-bit v.6 [Yan 2009]	2009
	gSpan ParSeMis [Philippsen 2011, Henderson 2014]	2011
	gSpan (Kudo) [Nowozin 2013]	2004
	gSpan ParMol [8]	2013
	gSpan (Zhou)[9] [Zhou 2015]	2015
MoFa/MoSS [Borgelt 2002]	MoFa ParMol [Wörlein 2005, Meinl 2007]	2013
	MoSS ParMol [Wörlein 2005, Meinl 2007]	2013
	MoFa/Moss Original (Miner v6.13) [Borgelt 2016]	2015
AcGM [Inokuchi 2002]	AcGM Original [Inokuchi 2014]	-
FFSM [Huan 2003]	FFSM Original v3.0 [Fei 2010]	2010
	FFSM ParMol [Wörlein 2005, Meinl 2007]	2013
Gaston [Nijssen 2004]	Gaston Original v1.1 [Nijssen 2005a]	2005
	Gaston Original RE v1.1 [Nijssen 2005a]	2005
	Gaston ParMol [Wörlein 2005, Meinl 2007]	2013
	Gaston ParSeMis [Philippsen 2011, Henderson 2014]	2011
DMTL [Al Hasan 2005]	DMTL Original v1.0 (g++ 4.8 compiler) [Zaki 2008]	2006

Table 2.8: FSM Implementations with Technical Drawbacks (Complete Search) (March 2016)

Implementation	**Technical Drawbacks**
gSpan ParSeMis	- Quality of Frequent Subgraphs (redundancy) - Error during the execution
gSpan Kudo2004	- Requiring an additional software (MATLAB)
FFSM Original	- Error with Input Files (No answer from authors about this error)
AcGM Original	- No information about Memory Consumption or Runtime (binary code and no response from authors) - The output is only the DFS code of frequent subgraphs
Gaston ParSeMis	- Error during the execution

The number of graphs (|D|) for these datasets is 11. The largest dense dataset contains a maximum of 1178 graphs (|D|) with an average graph size (|T|) not exceeding 360 vertices and 910 edges. Table 2.9 shows the largest, most dense and largest dense datasets characteristics.

Table 2.9: Characteristics of Tested Centralized Graph Transaction Datasets in the Literature [2016]

Dataset Type (Name)	**|D|**	**|T|**	**|L|**
Largest dataset (DS3) [Aridhi 2015]	274860	40-50 (e)	-
Most Dense dataset (US Stock Market) [Wang 2006, Zeng 2006, Zeng 2009]	11	3636 (v) 206747 (e)	-
Largest Dense dataset (DD) [Douar 2014]	1178	284 (v) 716 (e)	82 (v) 1 (e)

Synthetic datasets do not exceed 100 000 graphs (|D|). A dense synthetic dataset contains (generally) a maximum of 400 vertices and 1000 edges.

For evaluating FSM implementations in a distributed environment, we found real datasets that can contain 46 703 496 graphs [Lin 2014] and synthetic datasets that can contain 100 000 000 graphs [Aridhi 2015].

Memory Resources

The maximum size of main memory used in most of the experiments found in the literature does not exceed 4 GB except for (i) gSpan, Gaston, FFSM, FSG and AcGM in [Nijssen 2006] with 10 GB, (ii) gSpan and Takigawa algorithm [Takigawa 2010] with 48 GB and (iii) gSpan and Gaston [Saha 2014] with 128 GB.

Evaluation Metrics

Typically, three common metrics have been used to compare implementations: (i) execution time, (ii) memory consumption and (iii) number of extracted frequent subgraphs. More detailed metrics about subtasks efficiency and the quality of subgraphs (*e.g.,* the execution time of the subtasks [Nijssen 2006], the sub-optimality [Wörlein 2005], the number of duplicate candidates [Gago-Alonso 2010b]) were used as well.

2.3 Experimental Study

In this section, we present the results of our experiments. We provide the description of our experimental setting. We split our study into : (i) intra-algorithm study where various implementations of a given algorithm are compared, (ii) a comparison of results (for each algorithm) with those of the state of the art and (iii) inter-algorithms study where implementations

of several algorithms are compared. We conclude this section by a final selection of the most efficient algorithms and some learned lessons regarding the performance of FSM algorithms.

2.3.1 Experimental Setup

Our experimental settings include: (i) the inputs of implementations (*i.e.,* datasets and minimum support threshold), (ii) the used resources, (iii) the metrics used to evaluate the efficiency of the implementations, and (iv) information about implementations configuration.

Inputs of Implementations

There are two inputs for FSM implementations: the datasets and the minimum support threshold.

Datasets Selecting the datasets which were used the most in the experiments (reported in literature) is an important issue because it would enable us to compare the results with existing studies. To our knowledge, about thirty-one real datasets with four different formats (TXT, SDF, SMILES, XML) were tested with FSM implementations. 78% of them are chemical and biological datasets.

The FSM implementations we collected are useful only with the TXT format, except for ParMol and MoSS Original accepting other formats. For instance, ParMol is able to parse TXT and SDF. MoSS Original parses only chemical formats of data (*e.g.,* SDF, SMILES). We conducted our experiments with twelve available datasets (out of thirty-one) of the two most used formats (TXT, SDF). For all implementations, the default choice was TXT format except for MoFa Original implementation where we used SDF format. SDF datasets were converted to TXT format using ParMol parsers [Meinl 2007].

- Datasets description. We categorized datasets into : (i) their size (*i.e., small*, *medium*, *large*), (ii) their density (*i.e., sparse*, *dense*) and (iii) the size of their graphs (*i.e., small-sized, medium-sized* and *large-sized*). We define these characteristics as follow :

  - Size characteristic : a dataset is considered *medium* if the number of graphs $|D| >= 10\ 000$ and *large* if $|D| >= 100\ 000$.
  - Density characteristic : We consider a graph as *sparse* if the number of edges is close to the number of vertices. We attribute the *dense* characteristic to datasets where the average number of edges ($|T_e|$) is considerably higher than the average number of vertices ($|T_v|$). Formally, the dataset is sparse if ($|T_e| < |T_v| * log(|T_v|)$) [Adamchik].

– Size of graphs characteristic : The graphs in the dataset are *medium-sized* if the average number of vertices $|T_v| >= 100$. They are considered as *large-sized* if the average number of vertices $|T_v| >= 1000$.

We use the term *"large"* in reference to the size of datasets and graphs in the FSM literature related to centralized graph transaction databases. However, typically the term *"large"* refers to a greater volume of data. Table 2.10 displays the characteristics of the twelve datasets where $|P|$ denotes the number of FSM experiments in literature (*i.e.,* papers) performed on the dataset, F is the original format of the dataset, S is the dataset size on disk (in KB), $|D|$ is the number of graphs in the dataset, $|T|$ is the average size of a graph by vertex(v)/edge(e) count, $|L|$ is the number of labels (for vertices and edges) in the dataset, $|M|$ is the maximum size of a graph by vertex/edge count and LT is the last date the dataset was experimented.

The selected datasets include the three most used datasets (PTE, AID2DA99, HIV-CA) in literature, the largest dataset, namely DS3 and the largest dense dataset, namely DD (see Table 2.10). The PTE dataset was used in twenty-two FSM experiments. The HIV/AIDS dataset is used in twenty experiments with two available HIV releases AID2DA99 (October 1999) and AIDS (unknown release). The dataset HIV-CA (all releases) was used in eleven experiments. We found an available HIV-CA release (March 2002) that was used in six experiments. The remaining datasets (shown in Table 2.10) were selected due to their: (i) availability, (ii) format (*i.e.,* TXT or SDF) compatible with the FSM implementations, and (iii) characteristics (*e.g.,* dense, large, medium).

Table 2.10: Available Datasets used in the Literature (March 2016)

Dataset	$\|P\|$	F	S	$\|D\|$	$\|T\|$	$\|L\|$	$\|M\|$	LT
Small, Sparse and Small-Sized Graph **Datasets**								
PTE[10] [Yan 2009, Nijssen 2005a]	22	TXT	169.7	340	27(v)/ 27(e)	66(v)/ 4(e)	214(v)/ 214(e)	[Nadimi-Shahraki 2015]
HIV-CA [Yan 2009]	6	TXT	285.2	422	40(v)/ 42(e)	21(v)/ 4(e)	189(v)/ 196(e)	[Krishna 2011]
Medium, Sparse and Small-Sized Graph **Datasets**								
NCI145 [Thoma 2010]	1	TXT	9 900	19 553	30(v)/ 32(e)	53(v)/ 3(e)	110(v)/ 116(e)	[Douar 2014]
NCI330 [Thoma 2010]	1	TXT	9 700	23 050	25(v)/ 27(e)	57(v)/ 3(e)	120(v)/ 132(e)	[Douar 2014]
CAN2DA99 [Nicklaus 2000]	4	SDF	266000(SDF)/ 14500(TXT)	32 553	26(v)/ 28(e)	66(v)/ 3(e)	229(v)/ 236(e)	[Gago-Alonso 2008]
AIDS [Thoma 2010]	1	TXT	26 200	56 213	28(v)/ 30(e)	62(v)/ 4(e)	222(v)/ 247(e)	[Douar 2014]
AID2DA99 [Nicklaus 2000]	4	SDF	111000(SDF)/ 18500(TXT)	42 682	26(v)/ 28(e)	62(v)/ 3(e)	222(v)/ 247(e)	[Gago-Alonso 2008]
Large, Sparse and Small-Sized Graph **Datasets**								
NCI250 [Nicklaus 2000]	1	SDF	960000(SDF)/ 89600(TXT)	250 251	21(v)/ 23(e)	82(v)/ 3(e)	252(v)/ 276(e)	[Nijssen 2004]
DS3 [11]	1	TXT	101 700	273 324	22(v)/ 24(e)	83(v)/ 3(e)	99(v)/ 99(e)	[Aridhi 2015]
Small, **Dense and Small/Medium-Sized Graph Datasets**								
PS[12]	1	TXT	2 975	90	67(v)/ 256(e)	21(v)/ 3(e)	76(v)/ 320(e)	[Saha 2014]
DD [Thoma 2010]	1	TXT	13 100	1 178	284(v)/ 716(e)	82(v)/ 1(e)	5748(v)/ 14267(e)	[Douar 2014]
Small, **Dense and Large-Sized Graph Datasets**								
PI[13]	1	TXT	3 500	3	2154(v)/ 81607(e)	2154(v)/ 1(e)	2154(v)/ 136264(e)	[Al Hasan 2010]

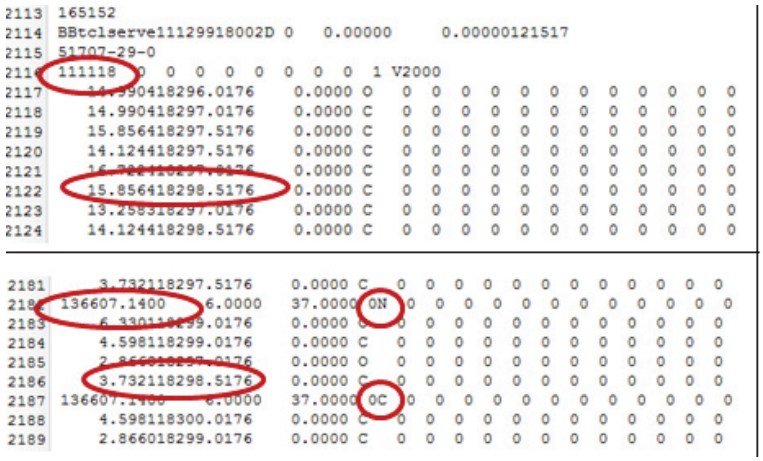


Figure 2.1: Examples of SDF file errors - AID2DA99 dataset

- Dataset modification. In some cases, we made some changes to the datasets: (i) correction of the parsing errors (NCI250 dataset) with potential graph elimination (AID2DA99, CAN2DA99 datasets). In fact, we removed 7 graphs from AID2DA99 dataset and 4 graphs from CAN2DA99 dataset due to data format[14] errors (*e.g.,* no space between two values such as '15.856418298.5176', see Figure 2.1), (ii) conversion from SDF to TXT format (*e.g.,* AID2DA99), (iii) grouping a set of files into one dataset file (AIDS, NCI145, NCI330 datasets), (iv) converting string vertex labels to integer ones (DS3 dataset). The FSM implementations (except FSG Original) work with integer labeled TXT datasets. For this, we modified the string vertices labels of DS3 dataset to integers. We named this modified dataset *DS3M*. We used the available codes of ParMol software[15] with small modifications to perform these tasks.

Minimum Support Threshold (MST). Different implementations of FSM algorithms convert differently the minimum support threshold (relative value) to the internal minimum frequency (absolute value). In fact, the conversion is done by carrying out one of the following options: (i) *Truncation* of the support value (denoted by **L**), (ii) *Truncation+1* (denoted by **H**), and (iii) *Rounding* (denoted by **L/H**).

Table 2.11 shows the input type of minimum threshold used by each FSM solution. The input type is either a support value (denoted by S) or a frequency value (denoted by F). Some implementations allow both input types. For an implementation where the support value (float) is used,

[14]Please refer to: `http://c4.cabrillo.edu/404/ctfile.pdf` for a basic SDF file format

[15]ParMol [Meinl 2007]. 2006. `github.com/yangyi0318/MyParMol/tree/master/ParMol` [Accessed 2019-05-30]

the corresponding conversion strategy (denoted by C) is mentioned. For example, gSpan Original v.6 allows a support value as input and its used conversion strategy is the truncation of the support to a frequency value.

Later on in this chapter, we compared the implementations of the same strategy (L or H).

Table 2.11: Algorithms' strategy of Minimum Support/Frequency Input

Algorithm Implementation	**S**	**F**	**C**
gSpan Original v.6	x		Truncation (L)
gSpan-64bit Original v.6	x		
gSpan (Zhou)	x		
ParMol (Gaston, gSpan, FFSM, MoFa, MoSS)	x	x	Truncation+1 (H)
MoFa Original v6.13	x	x	
Gaston Original v1.1		x	
Gaston Original RE v1.1		x	-
DMTL Original v1.0		x	
FSG Original (PAFI v1.0.1)	x		Rounding (L/H)

Used Resources

All our experiments were performed using a machine with 4 GB of RAM memory and a Quad core processor except for the experiment with a large-sized graph dense dataset PI (see Table 2.12). For experimenting with the PI dataset we used a different machine with 7 GB of memory and a Quad core processor.

We used Linux OS for deploying all FSM solutions. The Windows OS was used only to estimate the effect of varying the OS on the performance results (see Section 2.3.6).

Table 2.12: Machine Characteristics

Cases	**Default**	**Large-sized graph dense dataset**
Processor	Intel Core i3 Quad Core	
	2.40GHz	3.2GHz
RAM	4 GB	7 GB
Hard Disk	192.8 GB	226 GB
OS	*Default* : Ubuntu (14.04) : All Software	

Evaluation Metrics

We use the three common metrics as in the literature (see Section 2.2.6): (i) the execution time, (ii) the memory consumption, and (iii) the number

of returned frequent subgraphs. The solutions will be compared with each other by considering one of the three metrics.

Implementation Settings

We describe the configurations of implementations in terms of their framework, appropriate input/output and the used arguments to run them.

Framework Setting The available FSM solutions are either implemented in java language, including ParMol and MoFa Original solutions, or in C/C++ including the others implementations (gSpan Original versions, gSpan (Zhou), FSG Original, DMTL Original and Gaston Original versions). The configurations set for these solutions are the following : for java solutions, the JVM used version is 1.8.0_65-b17 and java heap space is set to 3.8GB. For C++ solutions, the gcc version used is 5.3 (see Table 2.13).

We ran each solution **three times** for **each support value**. The results that are reported in this chapter are the mean of the three executions. Some of the solutions (gSpan Original, gSpan (Zhou), ParMol) propose optional multi-threading execution. We used single thread in our experiments.

Table 2.13: Framework characteristics

Java solutions	Java Heap Space : 3.8GB
	JVM version : 1.8.0_65-b17
C & C++ solutions	gcc version :5.3

Output Configuration All the implementations provide information about the execution time, the number of frequent subgraphs, and some of them provide memory consumption. In the following, we describe the configurations of the implementations in terms of these outputs. It is worth noting that any modification we added to the FSM implementations has no effect[16] on performance results.

- Memory Consumption. Some FSM solutions are open source and some others are binary codes (see Table 2.14). We added memory consumption parameter for some open code FSM implementations (Gaston Original (v1.1 and RE v1.1), gSpan (Zhou) and DMTL Original).

  The binary code solutions (see Table 2.14) do not return information about the memory consumption. For this, we tried to deduce the limit of memory consumption by testing the lowest support threshold values that could be reached by the solution. We verified that the failure at low support values is due to a lack of memory (by resorting to a machine with 128 GB of memory).

  For ParMol implementations, we set the '*memoryStatistics*' argument to true, which enables the calculation of memory consumption. It

[16] We tested the effect of our modifications on performance

Table 2.14: Code Accessibility

Open Source	**Binary Code**
- Gaston Original versions - MoFa Original - gSpan (Zhou) - DMTL Original - ParMol framework	- gSpan Original versions - FSG Original

is worth noting that activating this argument can change the performance of the implementation (see Section 2.3.6).

- Execution Time. The execution time is composed of parsing time and the time to extract frequent subgraphs. It is worth noting that FSG Original is the only implementation which does not provide information about parsing time. Thus, in this case, we estimated the parsing time by using an external time calculation function (see Table 2.15).

Table 2.15: Our Estimated Parsing Time of the FSG Algorithm

Dataset	**Parsing Time (sec)**
HIV-CA	0.5
PTE	0.3
AID2DA99	11
CAN2DA99	8
AIDS	15
NCI145	5
NCI330	5
NCI250	52
DS3	57
DD	7
PS	13

For ParMol, we set the argument '*debug*' to 1, to display the subtasks runtime. We used the sum of the runtimes of the substasks in ParMol solutions.

- Number of Frequent Subgraphs. ParMol algorithms and MoFa Original are set by default to return only closed frequent subgraphs. We set off this option. MoFa Original v6.13 proceeds to a special modification of edge labels (conversion of found Kekule representations[17] into aromatic bonds[18] [Borgelt 2002]). We ran two versions of this software which return different numbers of frequent subgraphs : *(a)* with Kekule Representation conversion and *(b)* without conversion.

[17] Alternating between labels 1 and 2 in a chemical ring.
[18] relabeling edges by label 4.

Theoretically, complete search FSM algorithms return all frequent subgraphs that are above a specified minimum support threshold. However, in practice, the available FSM solutions of complete search algorithms produce a lower number of graphs compared to the complete set. According to the authors we contacted of ParMol and gSpan Original, this happens because of other internal thresholds and rounding effects defined in the implementation.

Input Configurations By default, ParMol is not set to parse TXT format. However, there is a TXT parser (LineGraphParser) in the ParMol package[19]. We used it for our TXT datasets. MoFa Original v6.13 parses chemical datasets (*e.g.,* SDF, SLN). Therefore, we tested it only with the three available SDF datasets (see Table 2.10).

Fixed Parameters For all implementations, we set three parameters : the input file, the minimum support threshold and the output file. Additional parameters were used for ParMol, namely '*memoryStatistics*' (memory consumption) and '*debug*' set to 1 (subtasks runtime).

Abbreviations of implementations (see Table 2.16) will be used further in experimentation results (Table 2.18 - Table 2.58).

Table 2.16: Abbreviations of implementations in Tables

Implementation	**Abbreviation**
gSpan Original	**SO**
gSpan-64bit Original	**SO64**
gSpan ParMol	**SP**
gSpan (Keren Zhou, 2015)	**SK**
Gaston Original	**GO**
Gaston Original RE	**GR**
Gaston ParMol	**GP**
DMTL Original	**D**
FSG Original	**F**
FFSM ParMol	**FF**
MoFa ParMol	**MFP**
MoSS ParMol	**MSP**
MoFa Original (with Kekule Representation Conversion)	**MOa**
MoFa Original (without Kekule Representation Conversion)	**MOb**
all ParMol implementations	**P**

In Tables, **Comp** will denote the qualitative comparison between two implementations, **Diff** will denote a quantitative interval corresponding to the difference (*e.g.,* runtime) between two implementations at the lowest

[19]ParMol [Meinl 2007]. 2006. `github.com/yangyi0318/MyParMol/tree/master/ParMol` [Accessed 2019-05-30]

and the highest support value. The symbol ≈ will indicate that the two implementations have approximately identical values. The symbol **(F)** will indicate that the versions have fluctuations in performance (*i.e.,* one version can be better than another in a run and be worse in another run).

2.3.2 Intra-Algorithm Performance Study

In this section, we compare different implementations of one algorithm in order to use the best implementation(s) in a further comparison with the other algorithms. There are three algorithms with more than one implementation, namely gSpan, Gaston and MoFa/MoSS (see Table 2.7). Only gSpan and Gaston implementations are evaluated in this Section. MoFa/ MoSS implementations will be evaluated with all the other algorithms (see Section 2.3.4).

gSpan Implementations

We tested four implementations of gSpan: Two original implementations provided by authors of gSpan [Yan 2002b] (gSpan Original v.6, gSpan 64-bit Original v.6) and third-party implementations (gSpan ParMol, gSpan (Zhou)).

Number of Frequent Subgraphs. GSpan (Zhou) was able to run with only small sparse & small-sized graph datasets (*e.g.,* HIV-CA or PTE). It was not able to run with larger datasets (*e.g.,* AID2DA99, CAN2DA99) or dense datasets (*e.g.,* DD). In addition, gSpan (Zhou) generated significantly fewer frequent subgraphs than, the two other solutions (ParMol, Original) (see Table 2.17).

Table 2.17: Number of Frequent Subgraphs by gSpan (L strategy) - HIV-CA

Min Sup	**SP (L)**	**SO/SO64**	**SK**
4%	-	6825311	-
5%	905299	905298	723603
6%	293406	293404	250518
7%	65260	65259	60183
8%	28559	28558	26304
9%	17512	17511	15945
10%	15973	15972	14486
15%	4476	4476	4152
20%	937	936	915
25%	248	248	239
30%	124	124	120
40%	60	60	56
45%	39	39	35
50%	32	32	29
60%	19	19	16

The two versions of gSpan Original (v.6 and 64bit v.6) generated the same number of frequent subgraphs except for the NCI330 dataset (for 6% and 8% minimum support threshold, there was a difference respectively of 15 and 4 graphs, see Table 2.18).

Table 2.18: gSpan Original *vs.* gSpan Original 64bit : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**
NCI330		
5%	SO = SO64	-
6%, 8%	SO > SO64	15, 4
9% - 90%	SO = SO64	-
The rest of datasets		
SO = SO64		

Typically, gSpan ParMol (L) and gSpan Original (v.6, 64-bit v.6) generated the same number of frequent subgraphs. Sometimes, it can produce one or two graphs in more or less than gSpan Original (v.6, 64-bit v.6) (*e.g.,* HIV-CA dataset, see Table 2.17). Additionally, in some exceptional cases, such as for PTE dataset, with low support threshold 1.5% and 2%, gSpan Original generated 53 and 49 (respectively) more graphs than the gSpan ParMol version. Table 2.19 shows the difference (denoted by *Diff*) between gSpan ParMol and gSpan Original v.6 in the number of frequent subgraphs.

The two values of *Diff* correspond to the difference of the number of frequent subgraphs between the two solutions, for the lowest and highest support, respectively (denoted by *Support Interval*).

It is worth noting also that gSpan algorithm implemented by Original authors and in ParMol, can compute the frequent subgraphs differently. For example, for NCI330 dataset with 6% MST, the two implementations of gSpan generated 4 subgraphs with different frequency[20] values. However, the frequency values are close. For example, the frequency values for a selected frequent subgraph out of the 4 are 4990 and 5107 for SO and SP, respectively.

Memory Consumption. Table 2.20 shows that gSpan (Zhou) required considerably more memory (denoted by *Memory*) than gSpan ParMol with lesser number of frequent subgraphs (denoted by *Number of FS*). Also, it was not able to reach the same low support thresholds as gSpan Original v.6, due to high memory consumption (see Table 2.21).

[20]The frequency is the number of occurrences of the subgraph, it is the absolute value, while the support is the relative value.

Table 2.19: gSpan Original *vs.* gSpan ParMol : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Large Datasets**		
HIV-CA			**AID2DA99, CAN2DA99**			**NCI250**		
5% - 20%	SO > SP	1 - 1	SO = SP			2% - 90% \{3%}	SO = SP	-
30% - 90%	SO = SP	-	**AIDS**			3%	SO > SP	931
PTE			1.5% - 2%	SO < SP	2 - 1	**DS3M**		
1.5% - 3%	SO < SP	53 - 1	3% - 90%	SO = SP	-	5%, 20%	SO < SP	1, 1
4% - 90%	SO = SP	-	**NCI330**			3% - 90% \{5%, 20%}	SO = SP	-
Dense Datasets			4%, 6%	SO > SP	1142 , 15	**Dense Datasets**		
DD			5% - 90% \{4%,6%}	SO = SP	-	**PS**		
4% - 90%	SO = SP	-	**NCI145**			80% - 90%	SO = SP	-
			2% - 90% \{3%}	SO = SP	-			
			3%	SO < SP	1			

Table 2.20: Examples of Memory Consumption of two gSpan versions (L strategy)

Implementation	**Memory (GB)**	**Number of FS**
1.5% (PTE)		
SP (L)	1.1	721 249
SK	48	698 934
5% (HIV-CA)		
SP (L)	2.01	905 299
SK	87	723 603

For low support threshold, gSpan-64bit Original v.6 required more memory than gSpan (Zhou), and significantly more than gSpan Original v.6 and gSpan ParMol. For example, gSpan Original-64bit could run with a threshold greater or equal to 8% for the HIV-CA dataset, while gSpan Original could run with 4% successfully (see Table 2.21). The lowest minimum support threshold we tested in Table 2.21 is 1.5%.

GSpan-64bit Original and gSpan (Zhou) could not run with very low support threshold values (see Table 2.21), unlike gSpan ParMol and gSpan v.6 Original.

Table 2.21: Minimal Support threshold value reached by gSpan versions (L strategy)

Dataset	**Small**		**Dense**	**Medium**		**Large**
	PTE	*HIV-CA*	*DD*	*CAN2DA99*	*NCI330*	*NCI250*
Version	**Min Support Threshold**					
SP	1.5%	5%	4%	2%	4%	2%
SO	1.5%	4%	1.5%	1.5%	3.5%	2%
SK	2.5%	7%	-	-	-	-
SO64	3%	8%	20%	3%	5%	4%

GSpan Original v.6 is the only implementation among gSpan versions that was able to reach the lowest minimum support threshold for all datasets (*e.g.,* 4% for the HIV-CA dataset and 1.5% for the DD dataset, see Table 2.21).

Runtime. Our experiments show that gSpan (Zhou) is the fastest algorithm for high support threshold values (see Figures 2.2 & 2.3). However, this version could not be used in the context of dense datasets (*e.g.,* dataset DD) or datasets that are not small in size[21] (*e.g.,* AID2DA99, CAN2DA99).

[21]The size of datasets is in terms of the number of graphs

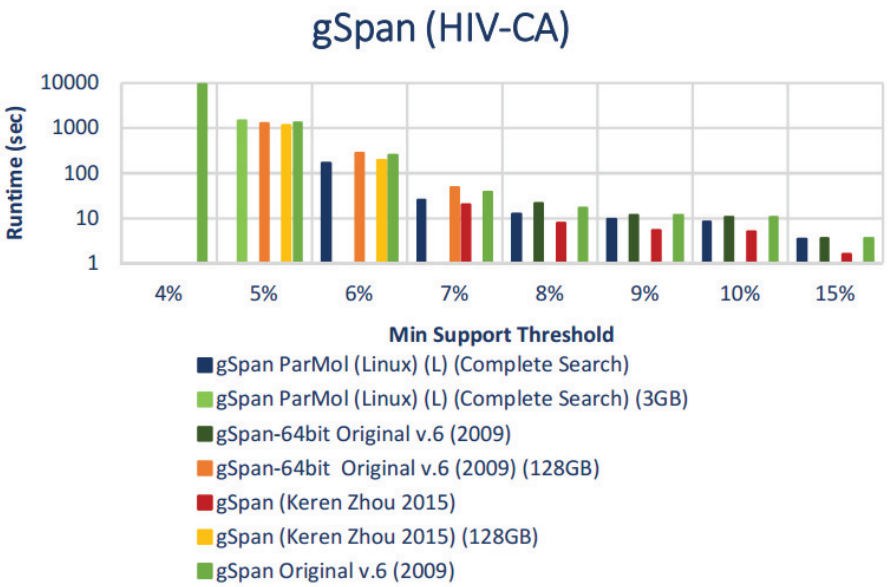


Figure 2.2: gSpan Runtime (Low Support Threshold) - HIV-CA

In addition, this version generated significantly fewer frequent subgraphs than the other versions (see Table 2.17).

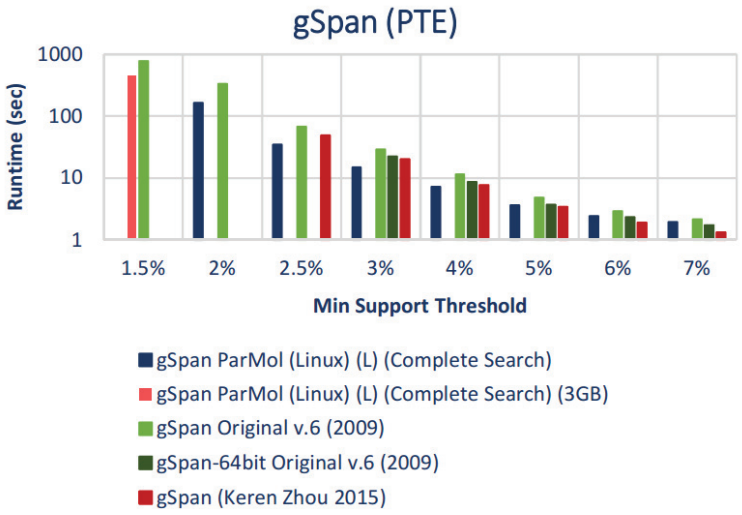


Figure 2.3: gSpan Runtime (Low Support Threshold) - PTE

Tables 2.23, 2.22 and 2.24 show the runtime comparison between the other gSpan implementations (gSpan Original versions and gSpan ParMol). The difference between execution times (denoted by *Diff*) is mentioned in seconds. The two values of *Diff* correspond to the runtime difference of

the lowest and the highest support value (denoted by *Support Interval*). For example, with the minimum support value 2%, gSpan Original v.6 consumes about 241 seconds more than gSpan Original v.6 64-bit for the AID2DA99 dataset and 5 seconds more for 90% (see Table 2.23).

Typically, our experiments also show that gSpan-64bit Original v.6 is faster than gSpan Original v.6 for all the tested datasets (see Table 2.23). However, for low support threshold values (*e.g.,* 8% - 15% for HIV-CA, see Table 2.23), gSpan-64bit Original v.6 can become slower than gSpan Original v.6 due to a higher memory consumption.

Furthermore, our experiments reveal that gSpan ParMol (L) is faster than gSpan Original v.6 for small and medium datasets. For example, gSpan ParMol (L) consumed about 850 seconds less than gSpan Original for AID2DA99 and MST 1.5% (see Table 2.22). However, for some cases of small and medium sparse and small-sized graph datasets, gSpan ParMol (L) can be slower if the support threshold is very low (*e.g.,* 2% for NCI145, see Table 2.22). For large sparse small-sized graph and small dense datasets, gSpan ParMol (L) is slower than gSpan Original v.6 for low and medium support values (*e.g.,* DD dataset, see Table 2.22).

Table 2.22: gSpan Implementations Runtime Comparison (gSpan ParMol *vs.* gSpan Original)

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5%	SO < SP	109	1.5% - 90%	SO > SP	850 - 3.4
6% - 10%	SO > SP	91 - 2.4	**AIDS**		
15%	$SO \approx SP$	-	1.5% - 90%	SO > SP	4636 - 4
20% - 80%	SO < SP	0.4 - 0.27	**CAN2DA99**		
PTE			2% - 80%	SO > SP	837 - 2.54
1.5% - 7%	SO > SP	321 - 0.08	**NCI145**		
8% - 90%	SO < SP	0.23 - 0.34	2%	SO < SP	169
Large Datasets			3% - 90%	SO > SP	1558 - 1.8
NCI250			**NCI330**		
2% - 70%	SO < SP	2089 - 1.5	4% - 5%	SO < SP	292 - 16
80% - 90%	SO > SP	3 - 5	6% - 90%	SO > SP	6.6 - 1.5
Dense Datasets			**Dense Datasets**		
DD			**PS**		
4% - 90%	SO < SP	5882 - 11	80%	SO > SP	13
			90%	SO < SP	0.7

Table 2.23: gSpan Implementations Runtime Comparison (gSpan Original versions)

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**			**Large Datasets**		
HIV-CA			**AID2DA99**			**DS3**		
8%	SO $<$ SO64	4	2% - 90%	SO $>$ SO64	241 - 5	5% - 90%	SO $>$ SO64	123 - 27
9% - 15%	SO $\approx$ *SO64*	-	**AIDS**			**NCI250**		
20% - 90%	SO $>$ SO64	0.1 - 0.083	2% - 90%	SO $>$ SO64	1008 - 6.7	4% - 90%	SO $>$ SO64	200 - 24
PTE			**CAN2DA99**			**Dense Datasets**		
2.94%	SO $<$ SO64	4	3% - 80%	SO $>$ SO64	187 - 4.1	**DD**		
3% - 90%	SO $>$ SO64	7 - 0.004	**NCI145**			20% - 30%	*SO* $\approx$ *SO64*	1
			5% - 90%	SO $>$ SO64	251 - 2.7	40% - 90%	SO $>$ SO64	2
			NCI330			**PS**		
			5% - 90%	SO $>$ SO64	45 - 2.5	80% - 90%	SO $>$ SO64	3 - 0.04

For small dense and large sparse small-sized graph datasets, gSpan-64bit Original v.6 is faster than gSpan ParMol for low[22] and medium support threshold values. For example, gSpan-64bit Original v.6 consumed about 1020 seconds lesser than gSpan ParMol for NCI250 dataset and MST 4% (see Table 2.24).

For medium sparse and small sparse datasets, gSpan-64bit Original is slower than gSpan ParMol for low support threshold values except for the NCI330 dataset (see Table 2.24). However, it has a competitive performance compared to gSpan ParMol for high support threshold values.

Summary of gSpan Implementations

- Of all gSpan solutions, gSpan Original v.6 is the most efficient one in terms of memory consumption for very low support threshold values. However, gSpan ParMol fails to achieve the search for some low threshold values (*e.g.,* HIV-CA 4%, see Table 2.21) and gSpan-64bit Original fails earlier (*e.g.,* HIV-CA 8%). The failures are mainly due to memory consumption. However, gSpan Original v.6 is able to complete the execution successfully (*e.g.,* HIV-CA 4%, see Table 2.21).

- GSpan-64bit Original v.6 can be used in a context where execution time is critical and the support threshold values are not low.

- Instead of gSpan Original v.6, the open source implementation gSpan ParMol can be used for better runtime performance if the dataset is small or medium and sparse, also the support values should be not too low.

[22]The low support threshold values that are reacheable by the FSM solution. For some low values, the solution fails to terminate.

Table 2.24: gSpan Implementations Runtime Comparison (gSpan ParMol *vs.* gSpan Original 64bit)

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
8% - 10%	SO64 > SP	8 - 2	2%	SO64 > SP	90	5% - 9%	SO64 > SP	125 - 13
15%	$SO64 \approx SP$	-	3% - 4%	SO64 (F) SP	16/38 - 1	10% - 50%	SO64 (F) SP	9 - 0.4
20% - 90%	SO64 < SP	0.5 - 0.35	5% - 9%	SO64 < SP	25 - 4	60% - 90%	SO64 < SP	0.03 - 0.9
PTE			10% - 90%	SO64 (F) SP	4 - 1.5	**NCI330**		
3% - 4%	SO64 > SP	6 - 1.45	**AIDS**			5% - 8%	SO64 < SP	61 - 5.5
5% - 6%	$SO64 \approx SP$	-	2% - 60%	SO64 > SP	712 - 0.1	9% - 60%	SO64 > SP	1.25 - 0.3
7% - 90%	SO64 < SP	0.3 - 0.34	70%	$SO64 \approx SP$	-	70% - 90%	SO64 < SP	0.4 - 1
Dense Datasets			80% - 90%	SO64 < SP	3 - 2.1	**Large Datasets**		
DD			**CAN2DA99**			**NCI250**		
20% - 90%	SO64 < SP	64 - 14	3% - 5%	SO64 > SP	43 - 6	4% - 90%	SO64 < SP	1020 - 18
PS			6% - 40%	SO64 (F) SP	6 - 0.7			
80%	SO64 > SP	10	60% - 80%	SO64 < SP	0.34 - 1.5			
90%	SO64 < SP	0.8						

Gaston Implementations

There are three implementations of Gaston: two (Gaston Original v1.1, RE v1.1) are from original authors [Nijssen 2004] and the other one (Gaston ParMol) is from a third-party implementer [Wörlein 2005].

Table 2.25: Gaston Original *vs.* Gaston Original RE : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**
Dense Datasets		
DD		
2% - 20%	GO > GR	1359 - 4
30% - 90%	GO = GR	-
PS		
60% - 80%	GO > GR	17013 - 45
90%	GO = GR	-
Rest of Datasets		
GO = GR		

Number of Frequent Subgraphs. Typically, Gaston Original versions (v1.1, RE v1.1) generated the same number of frequent subgraphs. However, there could be some exceptions such as the ones we found for the DD (under 20%, see Table 2.25) and PS (under 80%, see Table 2.25) datasets. For example, for the 2% MST of DD dataset, Gaston Original v1.1 produced 1359 frequent subgraphs more than Gaston v1.1 RE (see Table 2.25). In Table 2.25, *Diff* denotes the difference, in terms of frequent subgraphs, produced by the two implementations for the lowest and the highest support values.

Gaston Original (v1.1, RE v1.1) (H) and Gaston ParMol (H) produced a different number of frequent subgraphs for all datasets. This is shown in Table 2.26 where *Diff* denotes the difference, in terms of frequent subgraphs, produced by the two implementations for the lowest and the highest support values.

For example, with PTE dataset, for 1.5% MST, Gaston Original versions (v1.1, RE v1.1) (H) generated 57946 frequent cyclic graphs, 282724 frequent trees and 2268 frequent paths. However, Gaston ParMol (H) generated 57951 frequent cyclic graphs, 284294 frequent trees and 2234 frequent paths. It is worth noting that Gaston Original versions (v1.1, RE v1.1) do not include frequent subgraphs with single vertex (0 edges). However, Gaston ParMol does include these subgraphs. This could justify only the difference in the number of frequent paths.

Table 2.26: Gaston Implementations: Number of Frequent Subgraphs Comparison (Gaston ParMol *vs.* Gaston Original) - (L/H strategy)

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99/CAN2DA99**[23]			**NCI330**		
6% - 7%	GP < GO	32 - 34	2% - 90%	GP > GO	10 - 3	4% - 90%	GP > GO	90 - 1
8% - 90%	GP > GO	8 - 3	**AIDS**			**Large Datasets**		
PTE			2%	GP < GO	5	**NCI250**		
1.47% - 2.94%	GP < GO	4324 - 10	3% - 90%	GP > GO	8 - 2	60% - 90%	GP > GO	3 - 1
3%	GP = GO	-	**NCI145**			**Dense Datasets**		
4% - 90%	GP > GO	16 - 1	2% - 90%	GP > GO	1443 - 1	**DD**		
						4% - 5%	GP < GO	49 - 21
						6% - 90%	GP > GO	4 - 18
						PS		
						80% - 90%	GP > GO	7399 - 8

The difference in the number of frequent trees and graphs between Gaston ParMol and Gaston Original (v1.1, RE v1.1) needs to be explained further. Also, these two implementations of Gaston do not have a constantly positive (or negative) difference. For example, for AIDS dataset, Gaston ParMol generated 5 frequent subgraphs less than Gaston Original v1.1 for MST 2% and 8 more frequent subgraphs for MST 3% (see Table 2.26). This variability should also be explained by authors.

Memory Consumption. Typically, Gaston ParMol consumed more memory for all datasets (*e.g.,* AID2DA99, see Figure 2.4) and produced different numbers of frequent subgraphs compared to Gaston Original versions.

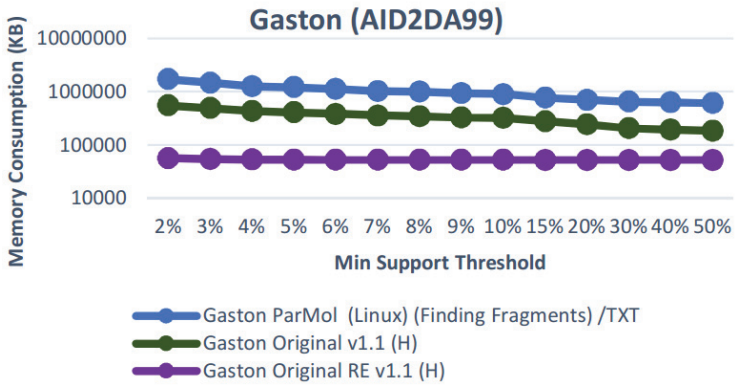


Figure 2.4: Memory Consumption of Gaston Original versions - AID2DA99

Table 2.27 shows some examples of the memory consumption of the three Gaston versions with their number of frequent subgraphs. For example, for AID2DA99 and MST 2%, Gaston ParMol consumed about three times the memory of Gaston Original v1.1 with 9 more frequent subgraphs.

Table 2.27: Examples of Memory Consumption and Number of Frequent Subgraphs of Gaston versions

Implementation	Memory (MB)	Number of FS
3% (PTE)		
GO (H)	7	18 121
GR (H)	4	18 121
GP	25	18 121
2% (AID2DA99)		
GO (H)	554.916	25 197
GR (H)	56.300	25 197
GP	1729.857	25 206
5% (DD)		
GO (H)	251812	795623
GR (H)	50964	795717
GP	876978	795696

However, for small sparse small-sized graph datasets and relatively medium values of support threshold, Gaston ParMol required fewer memory than Gaston Original versions (*e.g.,* above 6% MST for PTE, see Figure 2.5).

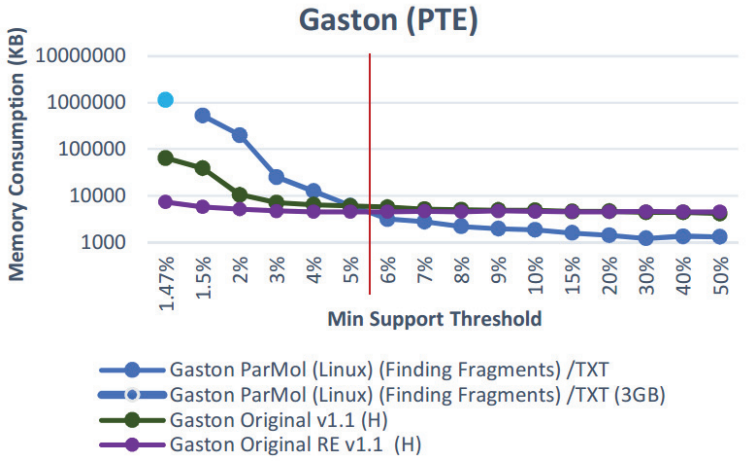


Figure 2.5: Memory Consumption of Gaston - PTE

Gaston Original RE v1.1 was proposed by [Nijssen 2005b] in order to reduce the memory consumption of Gaston Original v1.1. We found that Gaston Original RE, when it is able to run, had in fact a linear memory consumption lower than Gaston Original (*e.g.,* AID2DA99, see Figure 2.4 and PTE, see Figure 2.5) for all the tested datasets except for the small dense DD and PS datasets.

However, for very low support threshold values (*e.g.,* 3% for NCI330, 6% for HIV-CA) or for relatively large sparse small-sized graph datasets (*e.g.,* DS3, NCI250), Gaston Original RE produced an exception and hence the operation was terminated. For the same cases, Gaston Original completed successfully. For example, for HIV-CA dataset, Gaston Original v1.1 RE failed to terminate for 6% MST (see Table 2.28). However, Gaston Original v1.1 reached MST 4% successfully (see Table 2.28).

Runtime. The results show that for all tested datasets, the runtime performance of Gaston Original v1.1 was the best among all Gaston versions. It is worth noting that Gaston Original RE v1.1 required less memory than Gaston Original v1.1 (*e.g.,* AID2DA99, see Figure 2.4), as a trade-off, it was slower (*e.g.,* AID2DA99, see Figure 2.6).

For the small dense datasets (DD and PS), Gaston Original RE required more time and memory than Gaston Original with a different number of frequent subgraphs.

Table 2.28: Limits of Memory Consumption (KB) of Gaston versions for low support threshold

Version/ Dataset	**GO (L)**	**GR (L)**		**GP**
Min Sup	**4%**	**6%**	**7%**	**6%**
HIV-CA	15456	*Segmentation Fault*	3956	183400
Min Sup	**1%**	**3%**	**3.5%**	**4%**
NCI330	238512	*Segmentation Fault*	46072	676437
Min Sup	**2%**	**90%**		**60%**
NCI250	2759732	*Segmentation Fault*		2557084
Min Sup	**2%**	**90%**		**50%**
DS3M	3067400	*Segmentation Fault*		*Out Of Memory*
Min Sup	**1%**	**1.5%**	**2%**	**3.5%**
DD	66944	*Killed*	2744180	2114479

Summary of Gaston Implementations

- Gaston Original v1.1 should be used for applications where runtime is critical.
- Gaston Original RE v1.1 can be used to save memory (despite the required time) for the following cases: (a) support threshold values not too low (*e.g.,* above 6% MST for HIV-CA) and (b) datasets that are not large (*e.g.,* smaller than DS3, NCI250) and not dense (*e.g.,* less dense than DD). If neither (a) nor (b) are verified, then (c) the provided RAM memory should be large enough to handle the mining task. If none of the cases (a) and (b), or (c) are true, then Gaston Original v1.1 has to be used.
- Gaston ParMol consumed the highest amount of memory amongst all Gaston versions (except for small sparse datasets and high support values), yet it is the slowest solution (*e.g.,* for AID2DA99 dataset, see Figure 2.6) and it produced a number of frequent subgraphs different from what Gaston Original versions produced.

2.3.3 Comparison with the State of the Art

In this section, we compare our results regarding the implementations of the six algorithms with the results we found in state of the art. The comparison shows the similarities and differences between the results. This comparison could be seen as an update of the FSM literature with the consideration of implementations releases.

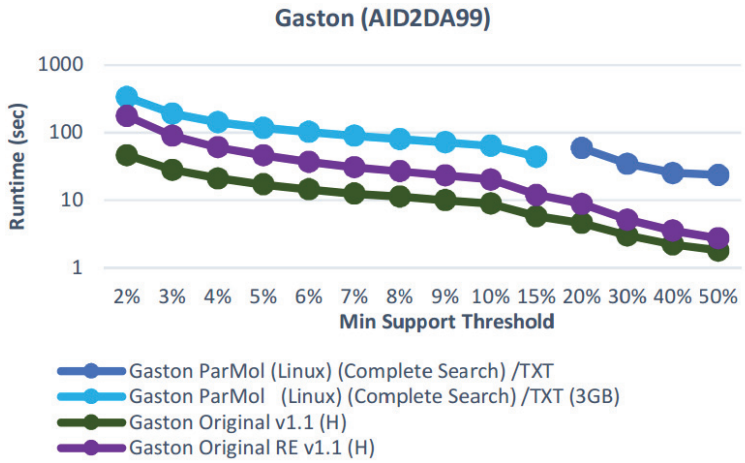


Figure 2.6: Gaston Runtime - AID2DA99

In fact, according to our knowledge, no study focuses on the impact of the evolution of FSM implementation versions (*e.g.,* gSpan Original 2002 *vs.* gSpan Original v.6 2009) on the performance.

According to our understanding, differences of our results with state of the art might occur due to different machine characteristics and different implementation releases (*e.g.,* gSpan v.5, gSpan v.6).

It is worth noting that we eliminated some graphs from the datasets AID2DA99 and CAN2DA99 (see Section 2.3.1). Therefore, our versions of these datasets contain slightly fewer number of graphs (7 and 4 graphs, respectively) than the ones tested in state of the art; we believe this could have an impact on the outcome and so on results. We took these inconveniences into consideration for our comparison.

GSpan Comparison

We present our comparison with gSpan Original versions first then with gSpan ParMol.

GSpan Original. Our experiment with gSpan Original v.6 generated a number of frequent subgraphs that is different (superior) from the result found in [Nijssen 2004, Nijssen 2006, Krishna 2011] for the datasets PTE and HIV-CA (see Table 2.29). The values comparable in Table 2.29 are in bold and non-grey coloured cells (using the same support strategy L or H). For example, in Table 2.29 for MST 3%, gSpan Original in our experiments generated 22785 frequent subgraphs. However, gSpan Original generated 22758 frequent subgraphs in [Nijssen 2004, Nijssen 2006]. Additionnally, Table 2.29 shows the difference in the number of frequent subgraphs for all gSpan versions in our experiments in comparison with gSpan Original results found in [Nijssen 2004, Nijssen 2006]. For some cases, this illustrates

that none of our tested gSpan versions has an equal number of frequent of subgraphs as found in [Nijssen 2006] (*e.g.,* MST 3%, see Table 2.29).

Table 2.29: Number of Frequent Subgraphs of gSpan : Comparison of our results (Left) with the Literature [Nijssen 2004, Nijssen 2006] (Right) - PTE

Our experiments					**Nijssen et *al.***
Min Sup	**SP (L)**	**SP (H)**	**SO**	**SK**	**SO (L/H)**
2% (6.7)	344513	**136981**	344464	338284	**136949**
3% (10.2)	**22786**	18146	**22785**	**22200**	**22758**
4% (13.59)	8776	**5955**	8776	8706	**5935**
5% (17.0)	**3627**	**3627**	**3627**	**3607**	**3608**
6% (20.4)	**2343**	2138	**2343**	**2326**	**2326**
7% (23.8)	1861	**1786**	1861	1845	**1770**
8% (27.19)	**1339**	1240	**1339**	**1323**	**1323**
9% (30.6)	1065	**993**	1065	1049	**977**
10% (34.0)	**860**	**860**	**860**	**844**	**844**
20% (68.0)	**199**	**199**	**199**	**190**	**190**
30% (102.00001)	**75**	75	**75**	**68**	**68**

We found the same frequent subgraphs produced by gSpan Original as reported in [Aridhi 2015] for the DS3 dataset. The number of frequent subgraphs found in [Yan 2003] is approximately[24] the same as our result for the dataset HIV-CA. Considering the runtime performance, gSpan Original v.6 was slightly slower than gSpan Original - reported in [Nijssen 2004, Nijssen 2006, Krishna 2011, Aridhi 2015] for PTE, HIV-CA and DS3 datasets. The result reported in [Yan 2003] for HIV-CA dataset was approximately similar to ours. According to our understanding, the difference regarding runtime could be due to different machine characteristics and to the number of generated frequent subgraphs. Regarding the impact of the differences between the gSpan versions (2002-2009), Xifeng Yan - the contacted author of gSpan Original explained the following: *"They are the same, except the new one supports more labels and it is running on a 64 bit system"... "The new version supports multi-threads, and more labels. Therefore, it consumes more memory (50%-100%)..."*

GSpan ParMol. Our experiment with gSpan ParMol (for PTE dataset) produced the same number of frequent subgraphs as in [Gago-Alonso 2010a]. For AID2DA99 dataset, our experiment is considerably faster (see Figure 2.7), consumed slightly more memory (see Figure 2.8) and produced approximately[25] the same number of duplicates (see Figure 2.9) compared to gSpan ParMol result found in [Gago-Alonso 2008, Gago-Alonso 2010b, Gago-Alonso 2010a].

[24]The results are given in a graphical form, we could not deduce a more precise conclusion.

[25]Results are represented in a graphical way

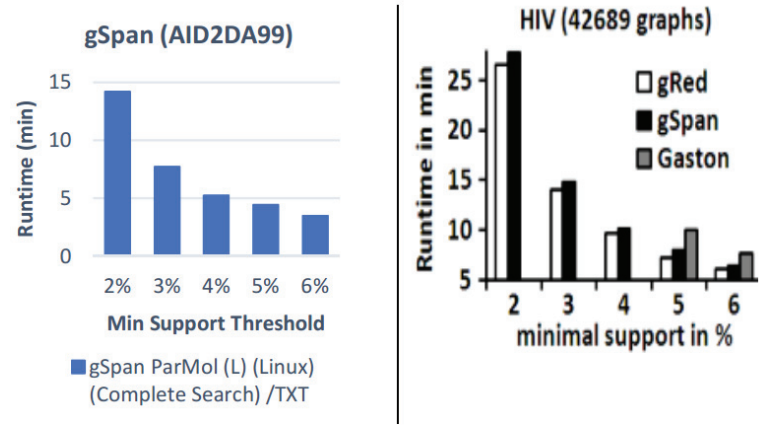


Figure 2.7: GSpan ParMol Runtime : Comparison of our results (Left) with the Literature [Gago-Alonso 2008, Gago-Alonso 2010a, Gago-Alonso 2010b] (Right) - AID2DA99

The difference in runtime could not be completely understood. Although, initially we assumed that the runtime performance is different due to different machine specifications. However, we found that even using a machine with the same resource specification, did not alleviate this difference.

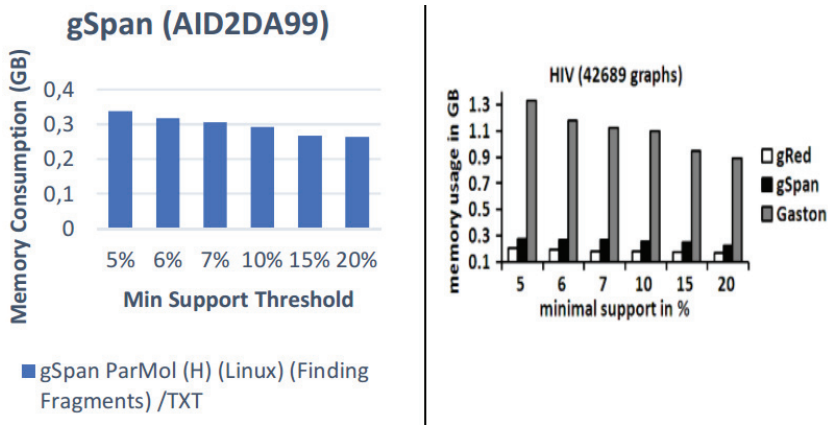


Figure 2.8: gSpan ParMol Memory Consumption : Comparison of our results (Left) with the Literature [Gago-Alonso 2008] (Right) - AID2DA99

Regarding the different versions of ParMol, Thorsten Meinl, one of the contacted authors of ParMol, stated what follows : *"We released several versions ... the changes had only minor effects on runtime and memory consumptions since those are mostly determined by the algorithm and not by the implementation".*

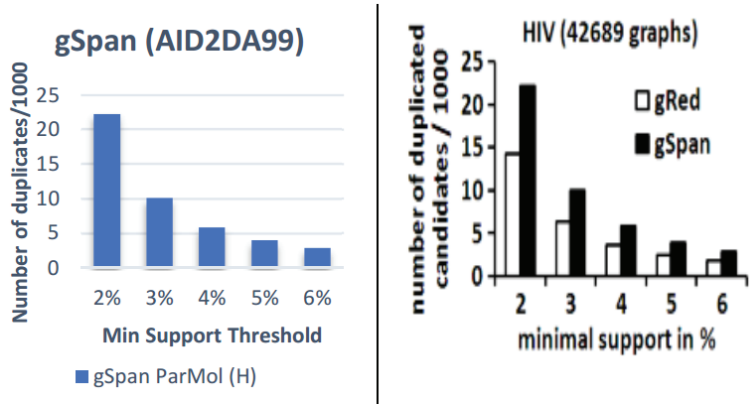


Figure 2.9: Number of Duplicates with gSpan ParMol : Comparison of our results (Left) with the Literature [Gago-Alonso 2008, Gago-Alonso 2010b, Gago-Alonso 2010a] (Right) - AID2DA99

Gaston Comparison

We present our comparison with Gaston Original versions first then with Gaston ParMol.

Gaston Original. In our experiment, Gaston Original versions (v1.1, RE v1.1) generated the same number of frequent subgraphs as in [Nijssen 2004, Krishna 2011] (for PTE and HIV-CA respectively). It generated a fewer (one less) number of frequent subgraphs than the result in [Aridhi 2015] (for 30% MST, DS3 dataset).

Table 2.30: Memory Consumption (MB) of Gaston Original : Comparison of our results (Left) with the Literature [Nijssen 2004, Nijssen 2006] (Right) - PTE

Our experiments					**Nijssen et *al.***	
Min Sup	**GO (L)**	**GO (H)**	**GR (L)**	**GR (H)**	**GO (L/H)**	**GR (L/H)**
2%	38.786	**10.421**	5.669	**5.096**	**9.1**	**1.5**
3%	**7.158**	7.062	**4.688**	4.618	**4.4**	**1.3**
4%	6.588	**6.354**	4.518	**4.518**	**3.4**	**1.3**
5%	**5.946**	**5.946**	**4.518**	**4.518**	**3.0**	**1.3**
6%	**5.688**	**5.598**	**4.558**	**4.520**	**2.7**	**1.3**
7%	5.237	**5.088**	4.516	**4.576**	**2.1**	**1.3**
8%	**5.042**	4.945	**4.552**	4.510	**1.9**	**1.3**

Gaston Original (v1.1, RE v1.1) consumed more memory than the version found in [Nijssen 2004, Nijssen 2006] (PTE dataset, see Table 2.30).

The values comparable in Table 2.30 are in bold and non-grey coloured cells (using the same support strategy L or H). For example, for MST set

to 2%, Gaston Original in our experiments consumed about 10.421 MB. However, Gaston Original consumed 9.1 MB in [Nijssen 2004, Nijssen 2006]. Also, for MST set to 3%, Gaston Original in our experiments consumed about 7.158 MB compared to 4.4 MB in [Nijssen 2004, Nijssen 2006].

Our runtime performance of Gaston Original was better than the result in [Nijssen 2004, Nijssen 2006] (PTE dataset, see Table 2.31) and in [Krishna 2011] (HIV-CA dataset). However, Gaston Original v1.1 has competitive runtime as in [Aridhi 2015] (for the DS3 dataset).

Table 2.31: Gaston Original Runtime : Comparison of our results (Left) with the Literature [Nijssen 2004, Nijssen 2006] (Right) - PTE

Our experiments					**Nijssen et *al.***	
Min Sup	**GO (L)**	**GO (H)**	**GR (L)**	**GR (H)**	**GO (L/H)**	**GR (L/H)**
2%	6.6275	**2.2836**	24.0183	**9.9545**	**7.9**	**39.6**
3%	**0.4545**	0.3635	**2.0841**	1.6018	**1.7**	**8.5**
4%	0.1932	**0.1509**	0.8405	**0.6583**	**0.6**	**2.7**
5%	**0.0959**	**0.0959**	**0.3836**	**0.3836**	**0.4**	**1.6**
6%	**0.0684**	**0.0599**	**0.2463**	**0.2258**	**0.3**	**1.0**
7%	0.0529	**0.0501**	0.1915	**0.1797**	**0.3**	**0.8**
8%	**0.0426**	0.0415	**0.1432**	0.1213	**0.2**	**0.6**

The values comparable in Table 2.31 are in bold and non-grey coloured cells (using the same support strategy L or H). For example, for MST set to 2%, Gaston Original in our experiments required about 2.28 seconds and Gaston Original RE required about 9.95 seconds. However, in the literature [Nijssen 2004, Nijssen 2006] Gaston Original required about 7.9 seconds and Gaston Original RE required about 39.6 seconds.

However, our results attest that Gaston Original v1.1 is much faster (see Table 2.31) and requires much more main memory (see Table 2.30) than Gaston Original RE v1.1 as it was reported in [Nijssen 2004, Nijssen 2006].

It is worth noting that we used different resource specification including a more powerful processor[26] than the one used in [Nijssen 2004, Nijssen 2006] and different from the ones used in [Krishna 2011, Aridhi 2015].

Gaston ParMol. In our experiment, Gaston ParMol generated a number of frequent subgraphs which is different from the number reported in [Gago-Alonso 2010a] (for AID2DA99). For example, Gaston ParMol generated 18121 frequent subgraphs in our experiment for AID2DA99 and MST 3%. However, it generated 18146 frequent subgraphs in [Gago-Alonso 2010a] (see Table 2.32). According to our results, Gaston ParMol is faster than the one tested in [Gago-Alonso 2010a, Gago-Alonso 2008, Gago-Alonso 2010b] (AID2DA99, PTE).

[26] [Nijssen 2006] used a single processor of a 2GHz Pentium, see Table 2.12 for our processor characteristics

Table 2.32: Number of Frequent Subgraphs with Gaston ParMol : Comparison of our results (Left) with the Literature [Gago-Alonso 2010a] (Right) - AID2DA99

Our experiments		Gago-Alonso et *al.*
Min Sup	**GP**	**GP**
3%	18121	18146
4%	5951	5955
5%	3625	3627
30%	75	75
40%	62	62
50%	37	37

Figure 2.10 shows the runtime difference. For example, for PTE dataset and MST set to 2%, Gaston ParMol consumed about 0.21 minutes. However, Gaston ParMol consumed about 0.4 minutes in literature [Gago-Alonso 2008, Gago-Alonso 2010a, Gago-Alonso 2010b]. Gaston ParMol consumed less memory, compared to what was reported in [Gago-Alonso 2008] (AID2DA99 dataset). In Figure 2.11, for AID2DA99 dataset and MST set to 5%, Gaston ParMol consumed about 1.2 GB in our experiment. It consumed a bit more than 1.3 GB in [Gago-Alonso 2008].

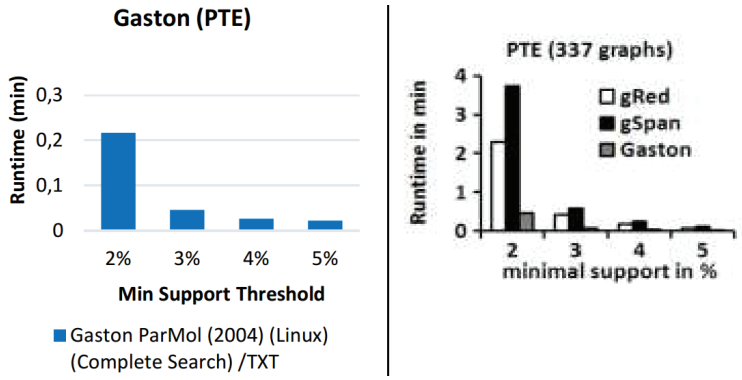


Figure 2.10: Gaston ParMol Runtime: Comparison of our results (Left) with the Literature [Gago-Alonso 2008, Gago-Alonso 2010a] (Right) - PTE

FSG Comparison

In our experiment, FSG Original v1.37 generated different number of frequent subgraphs compared to the version tested by Kuramochi et *al.* [Kuramochi 2002] for some threshold values (*e.g.* 2%, 7.5% for PTE, see Table 2.33). However, the experiments reported in [Nijssen 2003, Krishna 2011, Aridhi 2015] produced the same number of subgraphs as in our experiments (for PTE, HIV-CA and DS3 datasets, respectively).

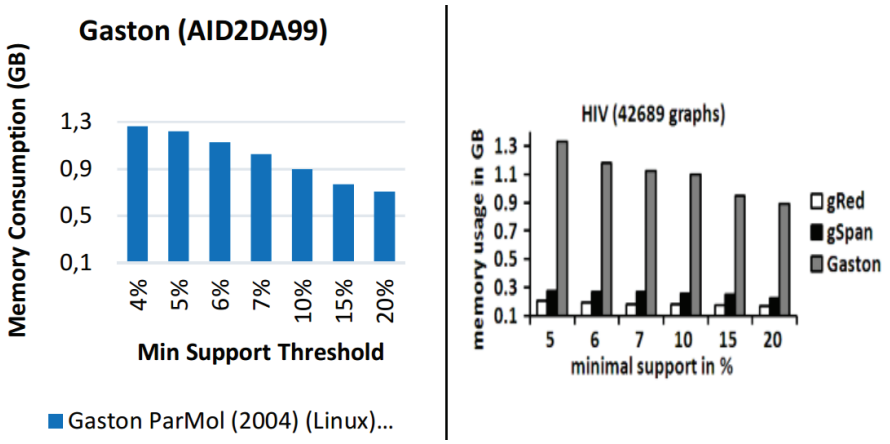


Figure 2.11: Gaston ParMol Memory Consumption (GB) : Comparison of our results (Left) with the Literature [Gago-Alonso 2008] (Right) - AID2DA99

The runtime performance in our experiments with FSG Original was close to the performance[27] reported in [Krishna 2011, Aridhi 2015] (for HIV-CA and DS3 datasets). However, it showed a better performance (*i.e.,* two or three times) than the FSG evaluated[28] in [Nijssen 2003] (for PTE dataset). For example, for PTE and MST set to 2%, FSG Original v1.37 required about 128.5 seconds. It required about 307 seconds for the same support and dataset in [Nijssen 2003] (see Table 2.34).

[27]It is worth noting that the processor of [Krishna 2011] and [Aridhi 2015] are different from our

[28]It is worth noting that our processor was more powerful than in [Nijssen 2003]

Table 2.33: Number of Frequent Subgraphs with FSG : Comparison of our results (Left) with the Literature [Kuramochi 2002] (Right) - PTE

Min Sup	**Our experiments**	**Kuramochi et *al.***
	F(L/H)	**F(L/H)**
2%	*136949*	*136927*
3%	22758	22758
4%	5935	5935
5%	3608	3608
6%	2326	2326
7%	1770	1770
7.5%	*1459*	*1590*
8%	1323	1323
9%	977	977
10%	844	844

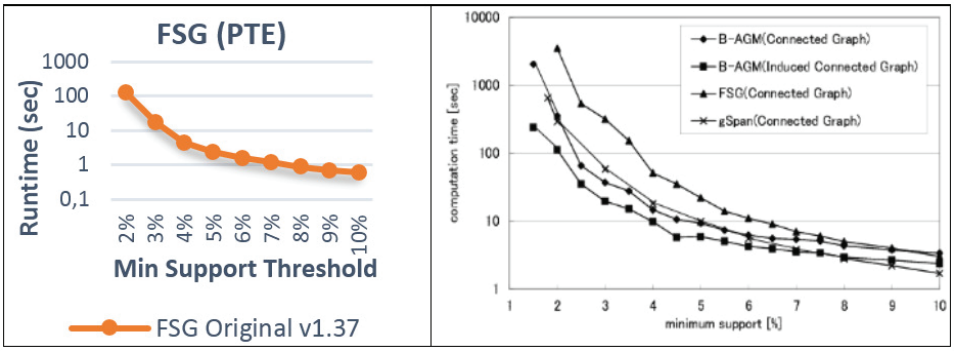


Figure 2.12: FSG Original Runtime : Comparison of our results (Left) with the Literature [Inokuchi 2005] (Right) - PTE

FSG has considerably better runtime (up to 50 times less, see Figure 2.12) than the experiment reported in [Inokuchi 2005, Yan 2003] (for PTE and HIV-CA datasets, respectively). It is worth noting that we used a more powerful processor than the ones used in [Inokuchi 2005, Yan 2003]. However, the difference in FSG results cannot be only related to the processor. In fact, gSpan in our experiments did not have such a huge difference compared to the literature results (less than 2 times slower [Inokuchi 2005, Yan 2003]). We relate the difference of the FSG results to FSG version evolution. Since our experiments rely on binary release of FSG, we could not compare memory consumption with state of the art.

Table 2.34: FSG Runtime (sec) : Comparison of our results (Left) with the Literature [Nijssen 2003] (Right : Nijssen et *al*) - PTE

Min Sup	**Our experiments**	**Nijssen et *al.***
	F(L/H)	**F(L/H)**
2%	128.5333	307.4
3%	18	43.9
4%	4.4	11.0
5%	2.5	6.3
6%	1.6	4.0
7%	1.2	2.9
8%	0.9	2.4
9%	0.7	1.8
10%	0.6	1.6
20%	0.2	0.6
30%	0.1	0.3

DMTL Comparison

We found only one *available real* dataset tested with DMTL in the literature (the dense dataset PI [Al Hasan 2009b]). In [Al Hasan 2009b], the basic version of DMTL crashes in few minutes with a 2 GB of RAM and MST set to 50%. For the same dataset and support value, we left DMTL running for days, it did not complete. We then aborted the execution.

FFSM Comparison

The comparison of our results with the ones found in the literature [Gago-Alonso 2010a] shows that the number of duplicates generated by FFSM ParMol is the same for the PTE dataset and a slightly more for the AID2DA99 dataset. We mentioned in Section 2.3.1 that we removed 7 graphs from AID2DA99 due to file errors. This raises a question - if we have *less* graphs and labels in our modified AID2DA99 dataset than the one in [Gago-Alonso 2010a], what makes the number of found duplicates in our result *more* than the one in [Gago-Alonso 2010a] ?

The number of frequent subgraphs is reported only in a graphical form in the literature [Skonieczny 2009]. Therefore, it was not possible to derive a precise conclusion.

Our experiment result shows that FFSM ParMol is considerably faster[29] than the result reported in [Gago-Alonso 2010a] for the PTE dataset (see Figure 2.13) with the same number of subgraphs duplicates. For example, for PTE and MST set to 2%, FFSM ParMol required slightly less than 1 minute in our experiment. However, it required slightly less than 4.8 minutes.

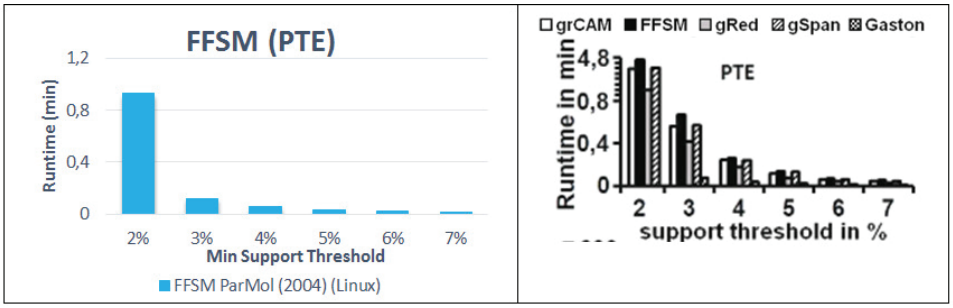


Figure 2.13: FFSM ParMol Runtime : Comparison of our results (Left) with the Literature [Gago-Alonso 2010a] (Right) - FFSM ParMol - PTE

We found competitive runtime with the result reported in [Skonieczny 2009] with approximately the same number of frequent subgraphs (graphical estimation). However, since no information about the

[29]The processing power of our resource is better than the one in [Gago-Alonso 2010a] (Intel Core 2 Duo 2.2 GHz processor)

system specification was provided in [Skonieczny 2009], we could not derive a conclusion regarding the runtime closeness. No information was reported about FFSM ParMol memory consumption in the literature.

MoFa Comparison

We were unable to compare our experiment results of MoFa ParMol with the ones found in the literature. The reasons are: (i) unavailability of the dataset reported in [Wörlein 2005], (ii) lack of sufficient details about the experiment (no information about machine characteristics was provided in [Skonieczny 2009]), and (iii) the ambiguity[30] about the used implementation (MoFa or MoSS) [Gago-Alonso 2010a].

Also, we were not able to compare our results regarding MoFa Original with state of the art mainly because of the unavailability of HIV-CM dataset and the lack of efficiency results[31] in state of the art [Borgelt 2002].

2.3.4 An Inter-Algorithms Performance Study

In this section, we compare the performance between different implementations of algorithms. It is important to notice that some algorithms were tested with the (H) strategy and some others with the (L) strategy (see Table 2.11). We considered the used strategy in our comparative study.

In this chapter, we report results of some tested datasets through our experiments. Additionally, we present a summarized comparison between some competitive implementations for all datasets and support threshold values. It is worth noting that all conclusions in this chapter are based on all experimental results[32] and not only the results shown in this chapter.

Results are presented according to the number of frequent subgraphs, runtime and memory consumption.

Number of Frequent Subgraphs

Table 2.36 shows the number of frequent subgraphs for implementations with the (H) strategy for the PTE dataset.

Gaston ParMol. The Gaston ParMol generated a number of frequent subgraphs which is different from the other implementations, for the low support threshold values (*e.g.,* PTE, see Table 2.36).

Table 2.35 shows the difference in the number of frequent subgraphs between Gaston ParMol and gSpan ParMol for all tested datasets. For example, for NCI145 and MST values ranging from 2% to 7%, Gaston ParMol generated between 1435 and 1 more frequent subgraphs than gSpan ParMol.

[30]The contacted authors [Gago-Alonso 2010a] could not remember if they used MoFa or MoSS in their experiments

[31]Experiments were mainly focusing on the quality of results [Borgelt 2002].

[32]For all results, see `https://liris.cnrs.fr/rihab.ayed/DFSM.pdf`

Table 2.35: Gaston ParMol *vs.* gSpan ParMol: Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
5% - 7%	GP < SP	151 - 42	2%	GP > SP	1	2% - 7% \{3%}	GP > SP	1435 - 1
8% - 90%	GP = SP	-	3% - 90%	GP = SP	-	3%	GP < SP	365
PTE			**CAN2DA99**			8% - 90%	GP = SP	-
1.5% - 5%	GP < SP	1575 - 2	2% - 3%	GP > SP	2 - 1	**Large Datasets**		
6% - 90%	GP = SP	-	4% - 90%	GP = SP	-	**NCI250**		
Dense Datasets			**AIDS**			50% - 90%	GP = SP	-
DD			2%	GP < SP	15	**DS3M**		
4% - 90%	GP = SP	-	3% - 90%	GP = SP	-	50% - 90%	GP = SP	-
PS			**NCI330**					
80% - 90%	GP = SP	-	4% - 7%	GP > SP	84 - 1			
			8% - 90%	GP = SP	-			

Table 2.36: Number of Frequent Subgraphs (H strategy) - PTE

Min Sup	SP (H)	GP (H)	GO (H)	F	FF (H)	D (H)	MFP (H)	MSP (H)
2%	136981	136513	136949	136949	136981	136949	136981	-
4%	5955	5951	5935	5935	5955	5935	5955	-
5%	3627	3625	3608	3608	3627	3608	3627	-
7%	1786	1786	1770	1770	1786	1770	1786	654
9%	993	993	977	977	993	977	993	464
10%	860	860	844	844	860	844	860	390
20%	199	199	190	190	199	190	199	120
25%	126	126	117	117	126	117	126	76
40%	62	62	58	58	62	58	62	36
50%	37	37	34	58	37	34	37	26

GSpan (Zhou). It produced a number of frequent subgraphs considerably different from gSpan versions (see Section 2.3.2). It was also different from Gaston Original for low and medium support threshold values. For example, for HIV-CA and MST set to 5%, gSpan (Zhou) generated 181687 more frequent subgraphs than Gaston Original (see Table 2.37).

Table 2.37: $gSpan^Z$ *vs.* Gaston Original : Number of Frequent Subgraphs Comparison

Support	Comp	Diff
Small Datasets		
HIV-CA		
5% - 20%	SK < GO	181687 - 17
30% - 60%	SK = GO	-
PTE		
1.5% - 5%	SK < GO	22279 - 1
6% - 50%	SK = GO	-

*GSpan ParMol, FFSM and MoFa ParMol.*Three ParMol implementations (gSpan, FFSM, MoFa) produced the same number of frequent subgraphs for all tested datasets (*e.g.,* PTE, see Table 2.36).

Table 2.38 shows the number of frequent subgraphs between implementations with the (L) strategy for the PTE dataset.

GSpan Original. As mentioned in Section 2.3.2, the number of subgraphs produced by gSpan ParMol and gSpan Original v.6 were almost the same except for some low support thresholds. For those low support thresholds, the difference varied between 1142 and 1 frequent subgraphs (see Table 2.38). Typically, the number of frequent subgraphs produced by Gaston Original was different from gSpan Original for all datasets (see Table 2.39). This difference could be partially justified for some support threshold values due to the fact that Gaston Original does not include frequent subgraphs with one vertex, unlike gSpan Original (*e.g.,* [3%, 50%] for PTE).

Table 2.38: Number of Frequent Subgraphs by FSM solutions - (L strategy) - PTE

Min Sup	**SP (L)**	**SO**	**SK**	**GO (L)**	**F**	**D (L)**
1.5%	721249	721196	698934	721213	721213	-
3%	22786	22785	22200	22758	22758	22758
5%	3627	3627	3607	3608	3608	3608
6%	2343	2343	2326	2326	2326	2326
8%	1339	1339	1323	1323	1323	1323
10%	860	860	844	844	844	844
15%	437	437	424	424	424	424
20%	199	199	190	190	190	190
25%	126	126	117	117	117	117
30%	75	75	68	68	68	68
40%	62	62	58	58	58	58
50%	37	37	34	34	58	34

However, other differences (*e.g.,* 1.5% for PTE) cannot be rationalized by the same fact. Besides the difference in the number of frequent subgraphs between Gaston Original and gSpan Original, there is also a difference in the counting of subgraphs.

For example, for AIDS with MST set to 10%, they produced 16 frequent subgraphs - out of 510 - that are the same but have different frequency values (*e.g.,* 21 759 and 21 761 are the frequency for one frequent subgraph by gSpan Original and Gaston Original, respectively).

Table 2.39: Gaston Original *vs.* gSpan Original: Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Dense Datasets**		
HIV-CA			**DD**		
4% - 80%	SO > GO	8 - 3	2% - 5%	SO < GO	1434 - 21
PTE			6% - 90%	SO > GO	4 - 18
1.5% - 2%	SO < GO	17 - 15	2%	SO < GR	75
3% - 90%	SO > GO	27 - 1	3% - 90%	SO > GR	133 - 18
Medium Datasets			**PS**		
AID2DA99/CAN2DA99 /AIDS/NCI145[33]			60%	SO < GO	25814977
			70%	SO > GO	176572
2% - 90%	SO > GO	9 - 3	80%	SO < GO	7399
NCI330			90%	SO > GO	8
3% - 90%	SO > GO	8 - 1	60%	SO < GR	25797964
Large Datasets			70%	SO > GR	176572
NCI250/DS3/DS3M			80%	SO < GR	7354
2% - 90%	SO > GO	8 - 1	90%	SO > GR	8

Gaston Original and FSG Original. Typically, the Gaston Original versions and FSG produced the same number of frequent subgraphs, with some exceptions. For example, for DD dataset with MST between 7% and 9%, FSG Original produced between 24 and 8 subgraphs less than Gaston Original (see Table 2.40). Also, for AIDS dataset and MST between 1.5% and 2%, FSG Original produced 1 more frequent subgraph than Gaston Original (see Table 2.40).

Besides this difference in the number of frequent subgraphs, there was also a difference in the counting of subgraphs. Noticeably, Gaston Original and FSG Original compute differently the frequency of subgraphs. For example, for AIDS with MST set to 2%, the two implementations generated 27 frequent subgraphs - out of 17 694 - that are the same but with different frequency values (*e.g.,* a frequency by Gaston Original equal to 13558 and by FSG Original equal to 13553 for one frequent subgraph).

DMTL. It produced significantly a fewer number of frequent subgraphs than the others for the NCI330 and NCI145 datasets (see Table 2.41).

For example, for NCI145 dataset and MST between 2% and 80%, DMTL produced between 449691 and 4 frequent subgraphs less than Gaston Original. For the other datasets, DMTL produced the same number as Gaston Original versions (see Table 2.41).

MoSS ParMol. The MoSS ParMol produced a number of frequent subgraphs which is considerably different from all implementations for PTE (see Table 2.38), PS and HIV-CA datasets (see Table 2.42).

MoFa Original. MoFa Original with case b (MOb) produced the same number of frequent subgraphs as MoFa ParMol for the 3 SDF used datasets (*e.g.,* AID2DA99, see Table 2.43). However, MoFa Original with case a (MOa) produced significantly a different number of frequent subgraphs. This is due to the edge relabeling strategy of chemical aromatic bonds [Borgelt 2002].

Table 2.40: FSG Original *vs.* Gaston Original : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Dense Datasets**			**Large Datasets**		
HIV-CA, PTE			**DD**			**NCI250**		
F = GO			7% - 9%	F < GO	24 - 8	2%	F < GO	1
Medium Datasets			30% - 90%	F = GO	-	3% - 90%	F = GO	-
AID2DA99, CAN2DA99, NCI330, NCI145			**PS**			**DS3M**		
F = GO			80% - 90%	F < GO	1733033 - 7415	1% - 2%	F < GO	16 - 1
AIDS						4% - 90%	F = GO	-
1.5% - 2%	F > GO	1 - 1						
3% - 90%	F = GO	-						

Table 2.41: DMTL Original *vs.* Gaston Original (or FSG Original) : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
5% - 80%	D = GO	-	40% - 90%	D = GO	-	2% - 80%	D < GO	449691 - 4
PTE			**CAN2DA99**			90%	D = GO	-
1.5% - 90%	D = GO	-	20% - 90%	D = GO	-	**NCI330**		
Dense Datasets			**AIDS**			4% - 60%	D < GO	49081 - 1
DD			80% - 90%	D = GO	-	70% - 90%	D = GO	-
3% - 30%	D < GO	274 - 1	**Dense Datasets**					
40% - 90%	D= GO	-	**PS**					
			70% - 90%	D = F	-			

Table 2.42: MoSS ParMol *vs.* MoFa ParMol : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI330**		
8% - 90%	MSP < MFP	23525 - 1	20% - 90%	MSP = MFP	-	20% - 90%	MSP = MFP	-
PTE			**CAN2DA99**			**Large Datasets**		
7% - 50%	MSP < MFP	1132 - 11	20% - 80%	MSP = MFP	-	**NCI250**		
70% - 90%	MSP = MFP	-	**AIDS**			70% - 90%	MSP = MFP	-
Dense Datasets			10% - 90%	MSP = MFP	-	**DS3M**		
DD			**NCI145**			80% - 90%	MSP = MFP	-
10% - 90%	MSP = MFP	-	20% - 90%	MSP = MFP	-			
PS								
90%	MSP < MFP	8						

Table 2.43: Number of Frequent Subgraphs by MoFa Implementations - (H strategy) - AID2DA99

Min Sup	MFP	MSP	MOa	MOb
2%	25205	-	9741	25205
3%	11531	-	4395	11531
4%	6670	-	2566	6670
5%	4442	-	1763	4442
6%	3162	-	1224	3162
8%	1869	-	695	1869
9%	1484	-	590	1484
10%	1185	-	484	1185
20%	326	326	146	326
30%	133	133	75	133
40%	71	71	37	71
50%	45	45	33	45
70%	19	19	11	19
90%	3	3	2	3

Summary of the number of frequent subgraphs

According to our results, the 13 FSM implementations can be classified according to their similarity in the number of frequent subgraphs. Our classification is not rigid as it tolerates some exceptions (with slight differences). We argument further this classification (see Figure 2.14).

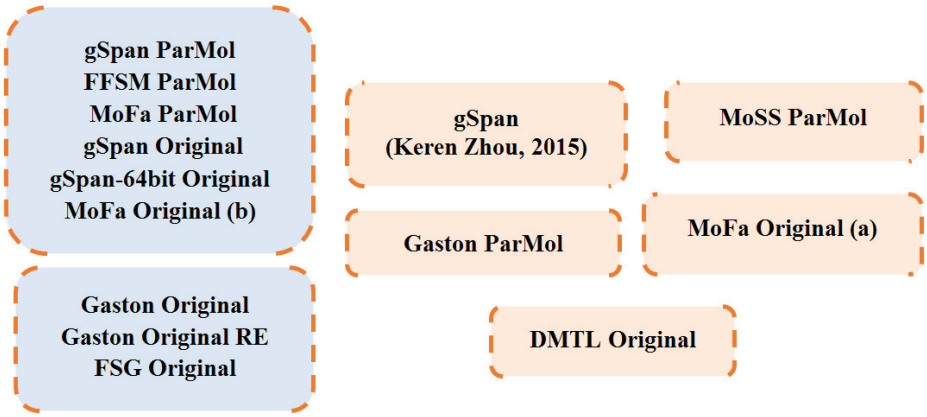


Figure 2.14: Classification of FSM Implementation according to the Number of Frequent Subgraphs

- GSpan (Zhou) produced a number of frequent subgraphs which was generally different from other implementations for low and medium

support thresholds (see Table 2.37). This difference is not neglectful (*e.g.,* 22279 frequent subgraphs, see Table 2.37). This is why we do not compare it with other implementations.

- Gaston Original generated a number of frequent subgraphs which is always different from gSpan Original for all datasets and support thresholds (see Table 2.39). The difference was slight (between 27 to 1 subgraphs, see Table 2.39) except for small dense datasets. Since the difference was reported for all datasets, we classified these two implementations separately.

- DMTL generated a number of frequent subgraphs equal to Gaston Original with some exceptions. The difference of DMTL with Gaston was a huge number (NCI330 and NCI145, see Table 2.41). The same applies for MoSS ParMol in comparison with MoFa ParMol (see Table 2.42). For this, we classified DMTL apart from Gaston and MoSS ParMol apart from MoFa ParMol.

- Also, FSG Original generated the same number of frequent subgraphs as Gaston Original with some exceptions. The difference was produced with low support threshold or small dense datasets (see Table 2.40). However, typically the difference was slight (*e.g.,* 1 to 24 subgraphs, see Table 2.40) except for the small dense dataset PS. We tolerate these slight exceptions and classified FSG with Gaston Original.

- GSpan Original versions, gSpan ParMol, FFSM ParMol, MoFa ParMol and MoFa Original (b) can be classified together, if we tolerate the slight difference of produced frequent subgraphs at low support thresholds between gSpan Original and the three ParMol implementations (see Table 2.38).

- We found it intriguing that Gaston ParMol generated a different number of frequent subgraphs from the original version of Gaston for all tested cases (see Section 2.3.2). Also, it is interesting to notice that it had a number of frequent subgraphs different from the other implementations of the same framework (ParMol) for low support thresholds (see Table 2.35). For this, we classified it apart.

In our experiments, we found that implementations do not produce the same number of subgraphs. In state of the art, we found that a different[34] number of frequent subgraphs was reported only in [Aridhi 2015], for Gaston, FSG and gSpan Original versions. However, for the rest of literature (*e.g.,* [Nijssen 2004, Nijssen 2006, Krishna 2011]), gSpan, Gaston, FFSM and FSG Original implementations are supposed to produce the same number of frequent subgraphs. Also, ParMol implementations (gSpan, Gaston,

[34]Authors tried to explain this difference for their tests [Aridhi 2014] (French paper).

FFSM, MoFa) produced the same number of frequent subgraphs (reported in [Gago-Alonso 2008, Gago-Alonso 2010b, Gago-Alonso 2010a]).

Further explanations should be provided about these observations. As an end-user study, we do not explain the difference of results between implementations.

Runtime

Figure 2.15 shows the runtime with the (H) strategy for PTE dataset and Figure 2.16 with (L) strategy.

DMTL. Our experiment shows that DMTL Original was significantly slower than all the other implementations (*e.g.,* PTE, see Figures 2.15, 2.16) for the same number of frequent subgraphs.

Gaston Original. Among all, for all the used datasets (*e.g.,* PTE, see Figure 2.15), Gaston Original v1.1 performed the best regarding runtime, and Gaston Original RE the second.

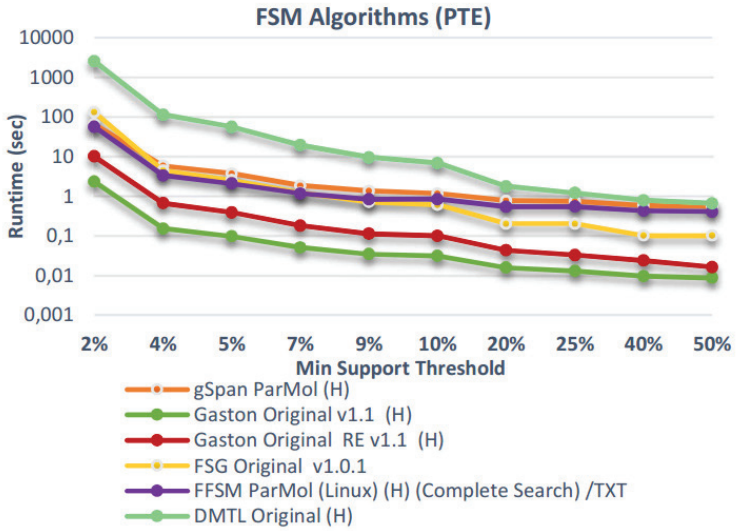


Figure 2.15: FSM Algorithm Runtime (PTE) - (H strategy)

GSpan Original and Gaston Original. GSpan Original may require a significant runtime for parsing a dataset (*e.g.,* for DS3 dataset, it consumed 28 seconds), while Gaston is faster in parsing (*e.g.,* for DS3 dataset, less than 0.2 seconds). Furthermore, gSpan was slower than Gaston for extracting frequent subgraphs (*e.g.,* PTE, see Figure 2.16).

We observed a comparative performance between gSpan versions, FFSM ParMol and FSG Original in terms of runtime (see Figures 2.16, 2.17). Thus, these three FSM algorithms are investigated futher.

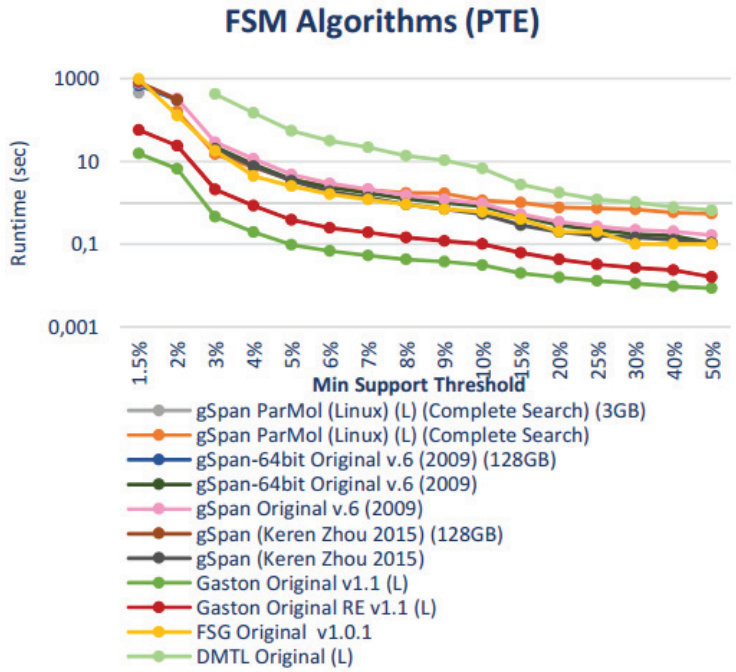


Figure 2.16: FSM Algorithm Runtime (PTE) - (L strategy)

We conducted the following comparison: (i) FSG Original with gSpan versions and (ii) FFSM ParMol with gSpan versions.

FSG Original and gSpan versions. We compared FSG Original with the fastest versions of gSpan (gSpan ParMol or gSpan Original).

Table 2.44 displays the results for all the datasets. The FSG Original was faster than gSpan versions for low support threshold and medium sparse or large sparse datasets. For small sparse datasets, it was slower than gSpan versions for low support threshold and slightly faster or close to gSpan versions for high support threshold. For small dense datasets, it was slower than gSpan versions.

It is worth noting that the number of subgraphs produced by FSG was slightly lesser than the result produced by gSpan versions for all tested datasets (between 36 and 1, see Table 2.45). Even though the comparison is biased due to different number of frequent subgraphs, it is interesting to compare the performance that was not highly dependent of this number. For example, for DD and MST set to 7%, FSG Original required 14 234 seconds more than gSpan Original (see Table 2.44) for a number of frequent subgraphs lesser (20) than gSpan Original (see Table 2.45).

FFSM ParMol and gSpan ParMol. Table 2.46 shows a runtime comparison between FFSM ParMol and gSpan ParMol. In Table 2.46, 'F' stands for fluctuation. FFSM ParMol is slower than gSpan ParMol for medium

(*e.g.,* 10%) and high support threshold values (*e.g.,* 50%) for medium sized datasets. For low support threshold, it could be slower or faster depending on the dataset (*e.g.,* AID2DA99, AIDS). FFSM ParMol is faster than gSpan ParMol for small dense datasets (see Table 2.46). However, for large sparse datasets, FFSM ParMol is slower. For small sparse datasets, it was slightly faster or almost equal to gSpan ParMol except for very low support threshold values where it could be slower (*e.g.,* 5% for HIV-CA, Table 2.46).

MoFa ParMol. Our results show that MoFa ParMol was the slowest among gSpan, FFSM and Gaston ParMol, for all the tested datasets (*e.g.,* AID2DA99, see Figure 2.17).

MoSS ParMol. It was the slowest among ParMol implementations (*e.g.,* AID2DA99, see Figure 2.17) for the same number of frequent subgraphs or for cases of lesser frequent subgraphs (see Table 2.42).

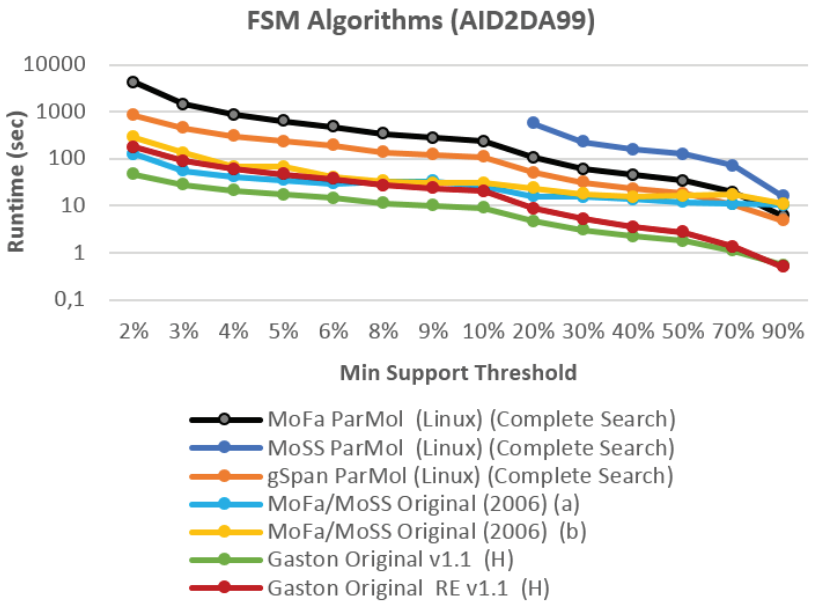


Figure 2.17: FSM Algorithm Runtime (AID2DA99) - (H strategy)

MoFa Original (b). It was the fastest implementation amongst all ParMol implementations for the two medium sparse datasets AID2DA99 and CAN2DA99.

Also, we observed that it had close runtime to the one reported for Gaston Original v1.1 with low support threshold values (*e.g.,* AID2DA99, see Figure 2.17). However, for the large sparse dataset NCI250, it was slower than ParMol implementations (gSpan, Gaston, MoFa) with the high reached support values (90%).

Table 2.44: FSG Original *vs.* gSpan versions (L/H strategy): Runtime Comparison

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5% - 15%	F > SP	256 - 0.4	1.5% - 5%	F < SP	510 - 24
20% - 80%	$F \approx SP$	0.1	6% - 90%	F > SP	14 - 7
PTE			**CAN2DA99**		
1.5% - 3%	F > SP	526 - 3.3	2% - 6%	F < SP	398 - 16
4% - 60%	F < SP	1 - 0.1	7% - 80%	F > SP	15 - 4.5
Large Datasets			**AIDS**		
NCI250			1.5% - 4%	F < SP	3239 - 12.5
2% - 20%	F < SO	4371 - 1	5% - 90%	F > SP	2.6 - 10
30% - 90%	F > SO	1.4 - 18	**NCI145**		
Dense Datasets			2% - 6%	F < SP	5263 - 8
DD			7% - 90%	F > SP	22 - 4
7% - 90%	F > SO	14234 - 6	**NCI330**		
PS			4% - 90%	F > SP	85 - 3
80% - 90%	F > SO	0.5 - 12.9			

Table 2.45: FSG Original *vs.* gSpan versions (L strategy): Number of Frequent Subgraphs

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5% - 90%	F < SP	9 - 2	1.5% - 90%	F < SP	8 - 1
PTE			**CAN2DA99**		
1.5% - 90%	F < SP	36 - 1	2% - 80%	F < SP	8 - 3
Large Datasets			**AIDS**		
NCI250			1.5% - 90%	F < SP	9 - 2
2% - 90%	F < SO	8 - 1	**NCI330**		
Dense Datasets			4% - 90%	F < SP	6 - 1
DD			**NCI145**		
7% - 90%	F < SO	20 - 18	2% - 90%	F < SP	8 - 1
PS					
80% - 90%	F < SO	16 - 13			

Table 2.46: gSpan *vs.* FFSM ParMol (L strategy): Runtime Comparison

Support	Comp	Diff (sec)	Support	Comp	Diff (sec)
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5%	FF > SP	756	2%	FF (F) SP	-
6% - 50%	FF < SP	13.7 - 0.06	3% - 8%	FF < SP	23 - 3
60% - 80%	FF > SP	0.05 - 0.02	9% - 90%	FF > SP	33 - 2
PTE			**CAN2DA99**		
2% - 40%	FF < SP	10 - 0.14	2% - 80%	FF > SP	166 - 3.5
50%	$FF \approx SP$	-	**AIDS**		
60% - 90%	FF < SP	0.06 - 0.1	2% - 70%	FF > SP	1238 - 5
Large Datasets			80% - 90%	FF < SP	0.1 - 0.05
NCI250			**NCI145**		
30% - 90%	FF > SP	1882 - 87	2% - 9%	FF < SP	2357 - 11
DS3M			10% - 90%	FF > SP	2 - 0.5
40% - 90%	FF > SP	2463 - 214	**NCI330**		
Dense Datasets			4% - 6%	FF < SP	403 - 5
DD			7% - 90%	FF > SP	3.9 - 0.02
4% - 90%	FF < SP	4204 - 12			
PS					
80% - 90%	FF < SP	3 - 0.4			

Summary of the runtime performance

- Gaston Original versions (v1.1, RE v1.1) were the fastest implementations independently of the dataset or support threshold values.

Memory Consumption

Figure 2.18 shows the memory consumption of implementations with the (H) strategy for the PTE dataset and Figure 2.19 with the (L) strategy.

GSpan (Zhou). For low support threshold values, gSpan (Zhou) consumed the highest amount of memory among all implementations (*e.g.,* PTE, see Figure 2.19) with a much lesser number of frequent subgraphs (see Table 2.37).

DMTL. For any support threshold value, the largest memory was consumed by DMTL implementation (see Figures 2.18 and 2.19).

For further comparison, we considered the implementations that were found to be competitive. This is the case for gSpan ParMol, Gaston Original versions and FFSM ParMol (see Figure 2.18). We did not consider further analysis of Gaston ParMol because of its intriguing difference regarding frequent subgraphs (see Section 2.3.4).

GSpan ParMol and Gaston Original. It is worth noting that gSpan ParMol and Gaston Original produced a different number of frequent subgraphs. Table 2.47 shows the comparison between gSpan ParMol and Gaston Orig-

inal versions with respect to their memory consumption.

For small sparse datasets, gSpan ParMol generated more frequent subgraphs (up to 36) than Gaston Original. For this case, gSpan ParMol consumed more memory for low support threshold and lesser memory for high support threshold than the two Gaston Original versions.

For medium sparse datasets, gSpan ParMol generated up to 9 more frequent subgraphs (except for NCI330) than Gaston Original versions. For this case, gSpan ParMol consumed more memory than Gaston Original RE. However, it required less memory than Gaston Original with low and medium support threshold. For some cases with low support threshold, it consumed more memory than Gaston Original (*e.g.,* 3% for NCI145 dataset, see Table 2.47). Furthermore, for large sparse datasets, gSpan ParMol generated more frequent subgraphs (up to 9) than Gaston Original. For this case, gSpan ParMol consumed less memory than Gaston Original for low support threshold. However, it consumed more memory for high support threshold.

For small dense datasets, gSpan ParMol consumed more memory than Gaston Original independently if the number of frequent subgraphs is less (*e.g.,* 49 for DD and MST 4%) or more (*e.g.,* 18 for DD and MST 90%). However, it consumed more memory than Gaston Original RE for low support threshold and less memory for high support threshold. For small dense datasets, gSpan ParMol produced more frequent subgraphs than Gaston Original RE.

GSpan ParMol and FFSM ParMol. We compared memory consumption of FFSM ParMol and gSpan ParMol (see Table 2.48).

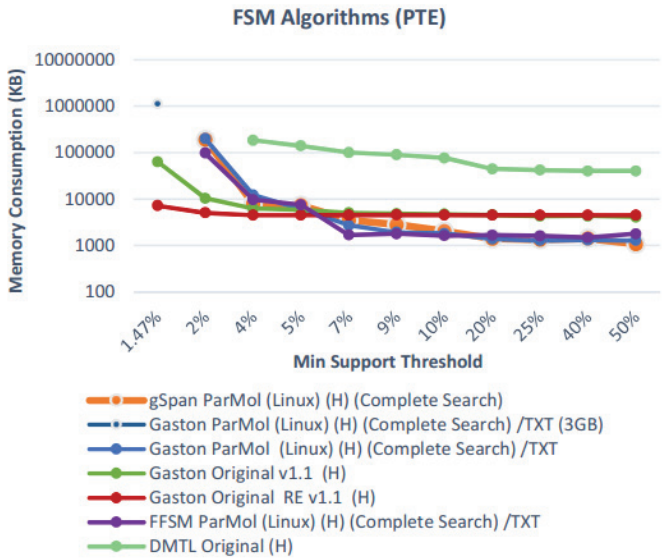


Figure 2.18: FSM Algorithm Memory Consumption (PTE) - (H strategy)

In Table 2.48, 'F' stands for fluctuation[35] of the performance. For small sparse datasets, FFSM ParMol consumed less memory than gSpan ParMol for low support threshold values. However, its consumption of memory was close to gSpan ParMol for high support threshold values.

For medium sparse and small dense datasets, FFSM ParMol consumed more memory than gSpan ParMol. Additionally, for large sparse datasets, it consumed significantly more memory than gSpan ParMol.

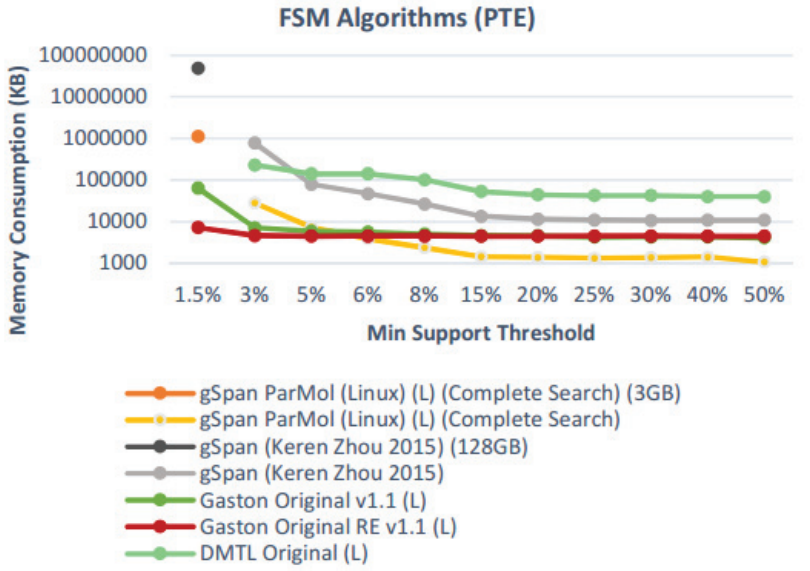


Figure 2.19: FSM Algorithms Memory Consumption (PTE) - (L strategy)

Original versions of FSG, gSpan and Gaston. The FSG Original and gSpan Original are provided as binary codes with no information about memory consumption. Therefore, we tried to deduce their respective limits regarding memory consumption by testing the lowest support threshold values. FSG Original was not able to run with low support threshold for some datasets (*e.g.,* DD dataset, see Table 2.49). We conclude that FSG used more memory than gSpan Original, for low support threshold values.

We could not conclude about the memory consumption limit of gSpan Original compared to Gaston Original. However, it is worth noting that for some datasets (*e.g.,* NCI330) and with low support thresholds, gSpan Original took a huge time without completing the mining (*e.g.,* 6 days for NCI330 with MST set to 2%).

Gaston Original completed it in a more reasonable time and with a lower support (*e.g.,* 9 hours for NCI330 with MST set to 1%).

[35]None of solutions performed consistently for more than two successive support threshold values

Table 2.47: gSpan ParMol *vs.*. Gaston Original versions (L strategy): Memory consumption comparison

Support Interval	**Comp**	**Diff (MB)**	**Support Interval**	**Comp**	**Diff (MB)**	**Support Interval**	**Comp**	**Diff (MB)**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
5% - 10%	SP > GO	2031 - 17	1.5% - 90%	SP > GR	468 - 84	2% - 90%	SP > GR	1702 - 53
20% - 80%	SP < GO	3.2 - 2.8	1.5% - 15%	SP < GO	66 - 10	2% - 3%	SP > GO	1324 - 76
7% - 20%	SP > GR	105 - 0.7	20% - 90%	SP > GO	19 - 39	4% - 50%	SP < GO	84 - 10
25% - 80%	SP < GR	1.8 - 1.7	**CAN2DA99**			60% - 90%	SP > GO	15 - 6
PTE			2% - 80%	SP > GR	369 - 100	**NCI330**		
1.5% - 5%	SP > GO	1066 - 1.2	2% - 20%	SP < GO	121 - 1.9	4% - 90%	SP > GR	392 - 24
6% - 90%	SP < GO	2.5 - 1.7	40% - 80%	SP > GO	36 - 40	4% - 5%	SP > GO	212 - 15
1.5% - 5%	SP > GR	1114 - 2.8	**AIDS**			6% - 20%	SP < GO	0.7 - 7.7
6% - 90%	SP < GR	1 - 3.1	1.5% - 90%	SP > GR	480 - 116	30% - 90%	SP > GO	7.5 - 6.7
Dense Datasets			1.5% - 70%	SP < GO	429 - 68	**Large Datasets**		
DD			80% - 90%	SP > GO	51 - 82	**DS3M**		
4% - 10%	SP > GR	1020 - 78	**Dense Datasets**			2% - 10%	SP < GO	908 - 14
20% - 90%	SP < GR	24 - 55	**PS**			20% - 90%	SP > GO	196 - 419
4% - 90%	SP > GO	1348 - 38	80%	SP > GO	26	**NCI250**		
			90%	SP < GO	1.8	2% - 10%	SP < GO	885 - 44
			80%	SP > GR	25	20% - 90%	SP > GO	178 - 386
			90%	SP < GR	2.9			

Table 2.48: gSpan ParMol *vs.* FFSM ParMol (L/H strategy): Memory consumption comparison

Support Interval	**Comp**	**Diff (MB)**	**Support Interval**	**Comp**	**Diff (MB)**
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5% - 10%	FF < SP	502 - 7	2% - 90%	FF > SP	512 - 44
15% - 80%	FF (F) SP	1.9 - 0.4	**CAN2DA99**		
PTE			2% - 80%	FF > SP	589 - 84
2% - 3%	FF < SP	93 - 5	**AIDS**		
4% - 90%	FF (F) SP	1.8 - 0.1	2% - 90%	FF > SP	1101 - 141
Dense Datasets			**NCI145**		
DD			2% - 90%	FF > SP	211 - 40
4% - 80%	FF > SP	568 - 15	**NCI330**		
90%	FF < SP	6	4% - 90%	FF > SP	123 - 18
PS			**Large Datasets**		
80%	FF < SP	12	**NCI250**		
90%	$FF \approx SP$	-	30% - 90%	FF > SP	1675 - 1377
			DS3M		
			40% - 90%	FF > SP	1592 - 1674

MoFa/MoSS versions. According to our results for MoFa/MoSS solutions, we observed that MoFa ParMol and FFSM ParMol consumed the same amount of memory (see Figure 2.20). MoSS ParMol consumed more memory with a number of subgraphs potentially lesser than all the other implementations (see Figure 2.20). Additionally, MoFa Original (b) consumed about twice (or one half) the amount of memory[36] required by MoFa ParMol for the medium sparse (AID2DA99, CAN2DA99) and large sparse (NCI250) datasets (see Table 2.50). For example, for MST set to 2%, MoFa ParMol consumed about 817 MB. However, MoFa Original (b) consumed between 1400 and 1500 MB.

Table 2.49: Minimal Support threshold value reached by FSM Algorithms-(L strategy)

Algorithm	**HIV-CA**	**DD**
SP	5%	4%
SO	4%	1%
F	5%	7%
GO	4%	1%

[36]We estimated the memory consumption of MoFa Original by the JVM it required

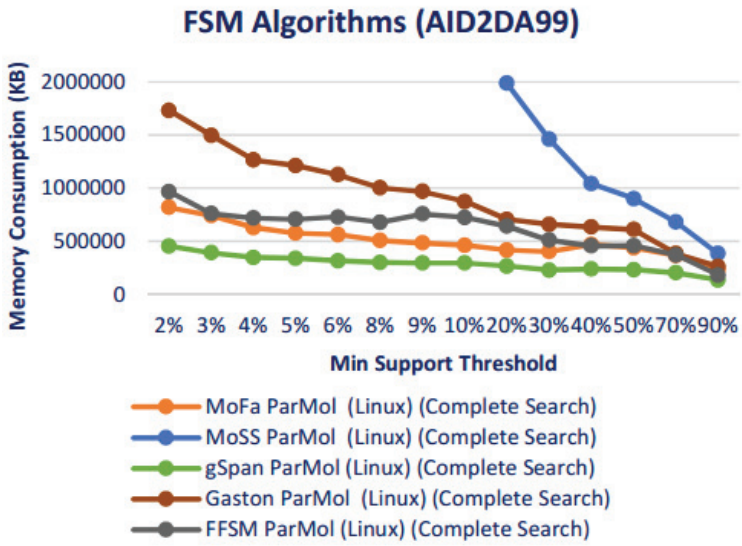


Figure 2.20: FSM Algorithms Memory Consumption (AID2DA99) - (H strategy)

Summary of memory performance

- According to our analysis, gSpan ParMol is more suitable than Gaston Original versions, for memory bound systems, in the following cases: (i) for large sparse datasets and low support threshold, (ii) for small sparse datasets and high support threshold values. It is worth noting that in the cases (i, ii) gSpan ParMol produced more frequent subgraphs than Gaston Original versions.

- DMTL is the worst memory consumer of tested FSM implementations (see Figures 2.18 and 2.19).

- Based on our study, we conclude that for memory bound systems, FFSM ParMol can be used instead of gSpan ParMol if the dataset is small sparse and the support threshold values are low. However, it is better to use Gaston Original versions for this case.

- According to our analysis, for memory bound systems, gSpan Original or Gaston Original is more suitable to use than FSG Original or gSpan ParMol for low support threshold values (*e.g.,* DD, HIV-CA, see Table 2.49).

- MoFa ParMol was the slowest among Gaston, FFSM and gSpan ParMol implementations and it consumed an amount of memory close to FFSM ParMol.

Table 2.50: Memory Consumption (MB) of two MoFa implementations (AID2DA99) - (H strategy)

Min Sup	**MFP**	**MOb**
AID2DA99		
2%	817	]1400 - 1500]
5%	576	]1100 - 1200]
10%	462	]1000 - 1100]
50%	437	]700 - 800]
90%	230	]500 - 600]
NCI250		
80%	1862	> 3500
90%	1611	]2400 - 2500]

In the following, we present results of FSM implementations with the lowest support thresholds reached and with the small & large-sized graph dense dataset PI (see Table 2.10). These experiments show the limits of complete search FSM implementations.

Bottleneck experiments

According to our experiments, Gaston Original is the implementation that was able to reach the lowest support threshold values. Table 2.51 shows the limits of the Gaston Original implementation with eleven datasets (see Table 2.10) and very low support threshold values. We used the same machine (see Table 2.12) as in the previous experiments.

Table 2.51: Bottleneck Experiment of Complete Search FSM Algorithms (Gaston)

Dataset	**Success of Mining Limit**				
	S	N	M	R	O
PTE	1%	48732156	42	637.5	21900
HIV-CA	4%	6825303	48	15.4	3400
AID2DA99	1%	107693	20	623.7	19
CAN2DA99	1%	176292	21	586.2	33
AIDS	1%	335483	27	1038.9	9
NCI145	1%	235740772	44	470.05	103000
NCI330	1%	268761360	42	238.5	192000
DD	1%	159820929	15	66.9	20100
PS	60%	63641199	28	5.9	13500
NCI250	1%	70405	21	3033.4	0.00003
DS3	1%	83310	21	3429.5	15.7

We emphasized on Gaston Original because it is the most efficient solution. The notations we used in the Table 2.12 are, S : reached support threshold, N : number of frequent subgraphs, M : Max size (vertices) of frequent subgraphs, R: consumed RAM memory (MB) and O : output file size (MB). For all tested datasets, Gaston Original was able to run with MST set to 1%, except for HIV-CA and PS datasets. In fact, the implementation was not able to run under MST set to 4% for HIV-CA due to lack of disk space. For PS with MST set to 50%, it spent 8 days running without completing the experiment, yet an output file of 3 GB was created. Similarly, in another experiment, it spent 4 days running without completing for the same support and with a maximum size of frequent subgraphs (28 vertices).

Table 2.52 displays the scalability of FSM implementations with a small & large-sized graph dense dataset (PI). Only Gaston Original RE and DMTL were able to process[37] the PI dataset without generating an error. We did not experiment DMTL any further for mining frequent subgraphs since it performs only a complete search, rather we experimented Gaston RE which is able to reduce the mining set.

Table 2.52: Execution of Implementations with Very Dense Datasets - PI

Algorithms versions	**P**	**SO**	**SO64**	**GO**	**GR**	**F**	**D**
Dataset Processing	-	-	-	-	+	-	+

Table 2.53 shows the limits of Gaston Original RE in mining PI dataset, where *MSF* denotes the maximum size (vertices) of frequent subgraphs, *Min Sup* : the minimum support threshold, *R* : runtime, *RM* : the used RAM memory, *DM* : the used disk memory, and *NF* : the number of frequent subgraphs.

Table 2.53: Mining Performance of Gaston RE with Very Dense Datasets - PI (Incomplete Search)

MSF	**Min Sup**	**R (sec)**	**RM (GB)**	**DM (MB)**	**NF**
2	70% - 100%	1.017	2.21	0.0079	256
	50%	-	*Segmentation fault*	0.0041	-
3	100%	1.578	2.21	0.0852	1928
	70%	-	*Aborted*	0.0011	-
5	100%	873.745	2.84	12	1578086
10	100%	-	*Killed*	3300	-

Gaston RE was able to find frequent subgraphs of maximum size 5 with

[37]No abortion in the beginning of the execution

MST set to 100%. For this case, it consumed about 2.84 GB of RAM (memory) and generated 1578086 frequent subgraphs (see Table 2.53). However, it was not able to complete the mining with the same support for maximum size 10. The same applies for the cases of MST set to 70% and max size 3 or MST set to 50% and max size 2. These findings confirm the results of [Saha 2014, Al Hasan 2009b] about the limits of complete search algorithms with dense datasets.

2.3.5 Discussion

According to our observations, the sources of results ambiguities in state of the art (see Section 2.2.3) are as follows: different styles of implementating an FSM algorithm (*e.g.,* ParMol or Original), the dataset characteristics (*e.g.,* small, large, sparse, dense), and the support threshold values (*e.g.,* [2%, 50%], [10%, 90%]). For example, the third-party implementation of Gaston (Gaston ParMol) was the highest memory consumer among gSpan ParMol and FFSM ParMol. However, the Original version of Gaston was the least memory consumer among gSpan ParMol and FFSM ParMol. These observations illustrate that an FSM algorithm (*e.g.,* Gaston) when implemented differently, can affect considerably its efficiency.

The experimental study we conducted allowed to alleviate some of the ambiguities and specify some cases of FSM implementations performance. According to our results, eight implementations among thirteen (see Table 2.54) are not adequately efficient due to : (i) their high memory and/or time consumption, (ii) a number of frequent subgraphs different from the other implementations, (iii) their inability to handle relatively large datasets or run for low support thresholds.

Table 2.54: FSM Algorithms with performance drawbacks

Solution Characteristics	**Solution Characteristics**
DMTL	*Gaston ParMol*
(-) Considerable time and memory consumption (-) Number of frequent subgraphs considerably different for some datasets (+) Able to process small dense large-sized graph datasets	(-) Number of frequent subgraphs different from Gaston Original and ParMol solutions (-) Large memory consumption
MoFa ParMol	*MoSS ParMol*
(-) Considerable time and memory consumption (+) Extra mining options for biochemical data	(-) Small number of frequent subgraphs (-) Large memory consumption
gSpan-64bit Original v.6	*gSpan^Z*
(-) Slower than Gaston Original for high support (-) For low support values, it consumed considerable memory compared to gSpan Original, gSpan ParMol and Gaston Original	(-) Number of frequent subgraphs considerably different from others (-) Large memory consumption (-) Dedicated for small sparse datasets
MoFa Original	*FFSM ParMol*
(-) Bad memory consumption (+) Close runtime to Gaston Original v1.1 with low support values and medium sparse datasets (+) Extra options for mining biochemical data	(-) Worse memory consumption than gSpan ParMol and Gaston Original (-) Slower than Gaston Original versions

We selected five implementations out of thirteen as efficient, including Gaston Original, gSpan ParMol, gSpan Original, FSG Original and Gaston Original RE.

The first four implementations (see Table 2.55) were selected based on the following criteria : (i) they consumed the least amount of memory, among all the thirteen FSM implementations, (ii) they are relatively fast (Gaston Original is the fastest), (iii) they are able to complete the mining with relatively large datasets or for low support threshold values.

Gaston Original RE was chosen (see Table 2.55) due to its ability to process small large-sized graph dense datasets unlike the four others and also because of its good performance with medium sparse datasets.

Table 2.55: FSM Algorithms with performance advantages

Solution	**Performance Characteristics**
Gaston Original	(+) Second/third in memory consumption and the fastest (+) Able to run with relatively large sparse datasets or very low support values
gSpan ParMol	(+) Third/fourth best memory consumption for medium sparse, large sparse datasets or for low support (+) Third fastest for small sparse/medium sparse datasets and not low support threshold values (-) Unable to run for very low support values reached by gSpan Original
gSpan Original	(+) First/second best memory consumption for low support threshold (+) Third fastest for small dense datasets, or for high support values and large sparse datasets (+) Able to run for some very low support threshold values or for relatively large sparse datasets compared to other implementations (*e.g.,* gSpan ParMol, FSG) (-) Compared to Gaston Original, unable to finish in a reasonable time for some very low support threshold values
FSG Original	(+) Able to run for low support threshold or relatively large datasets (+) Third fastest for medium sparse, large sparse datasets and for low support values (-) Requiring more memory than gSpan Original and gSpan ParMol for low support values
Gaston Original RE	(+) Second in Runtime (+) First/second in memory consumption with medium sparse datasets (+) Able to process small large-sized graph dense datasets (*e.g.,* PI) (-) Not to be used with large datasets or very low support threshold values

Then we reduce the set of four selected implementations (Gaston Origi-

nal, gSpan ParMol, gSpan Original and FSG Original) to two usable implementations (Gaston Original and gSpan Original) for two general cases : (i) applications that need to save memory, and (ii) applications where runtime is critical. Both Gaston Original and gSpan Original are suitable for the former and Gaston Original is suitable for the latter.

During our experiments, we realized that the size of a dataset and the minimum support influenced the performance of the tested FSM solutions. Therefore, we changed some other parameters of the experimental environment in order to observe their impact on the performance. In the next section, we discuss our results.

2.3.6 Impacts of the Environment variations on the results

We discuss in the following the impact of changing the variables of experimental environment on the performance of FSM implementations. The environment variables include the dataset size, the operating system, a potential IDE, the used input arguments of implementations, the input data format and the labelling strategy of data.

Dataset variation

We studied the impact of tested datasets[38] on the *runtime*, *memory* and the *number of frequent subgraphs*. We considered two variables for datasets : *size* and *density*. These parameters are the same as defined in Section 2.3.1. We discuss our results of experiments with Gaston Original, gSpan ParMol and gSpan Original.

Runtime. According to our observation, experiments with *small sparse datasets* required typically the lowest runtime among all datasets (*e.g.,* PTE, see Figure 2.21). However, the experiments with low support threshold over the small sparse datasets (*e.g.,* HIV-CA) required more or the same time than the other datasets (*e.g.,* DS3) to complete the execution.

- *Medium datasets* had similar runtime performance (*e.g.,* NCI330 and CAN2DA99, see Figure 2.21). However, for experiments with low support threshold, two medium datasets (NCI330 and NCI145) were considerably slower than the other medium datasets (*e.g.,* AID2DA99).

- Typically, *large datasets* (*e.g.,* DS3) required more time than small and medium datasets (*e.g.,* CAN2DA99), except with very low support threshold values.

- *Small Dense datasets* (*e.g.,* DD) required similar amount of time compared to medium sparse datasets (*e.g.,* CAN2DA99) with high support thresholds (*e.g.,* 30%, see Figure 2.21). However, with low sup-

[38]The used datasets were defined in Section 2.3.1

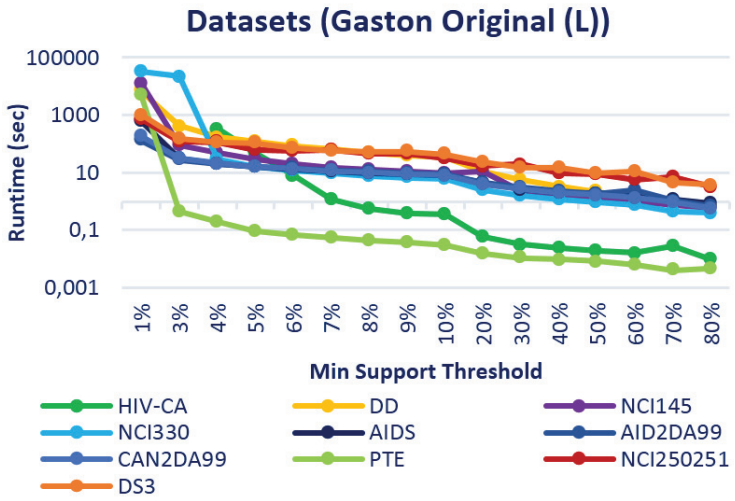


Figure 2.21: Dataset Variation Effect on the Runtime - Gaston Original (L)

port threshold (*e.g.,* 3%), they required more time than large sparse datasets (*e.g.* DS3).

Memory Consumption. Our experiments with Gaston Original and *small sparse datasets* consumed the lowest amount of memory (*e.g.,* PTE, see Figure 2.22). Then, follows small dense (*e.g.,* DD), medium sparse (*e.g.* NCI330) and large sparse (*e.g.* DS3) datasets, respectively (see Figure 2.22).

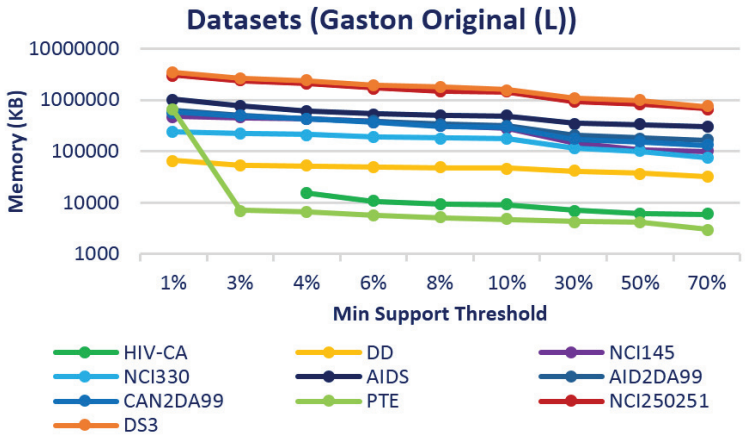


Figure 2.22: Dataset Variation Effect on the Memory - Gaston Original (L)

Our experiments with gSpan ParMol and *small dense datasets* consumed less memory than medium sparse datasets with high support values. However, with low support values, they consumed more memory than medium sparse datasets.

The consumption of memory for all datasets was typically linear (see Figure 2.22). However, we observed some exceptions for small sparse datasets and low support threshold values where there was an important increase of memory (*e.g.,* PTE with Gaston or gSpan ParMol, HIV-CA with gSpan ParMol). The amount of memory consumption for this case (*e.g.,* 1% PTE, see Figure 2.22) was approximately the same or greater than the amount of memory consumed by medium and large datasets.

Number of Frequent Subgraphs. Different sizes of datasets produced typically close number of frequent subgraphs (*e.g.,* DS3 and CAN2DA99, see Figure 2.23). However, our experiment with the *small dense* dataset DD produced a considerably larger number of subgraphs than the other datasets (*e.g.,* DS3). Additionally, with low support values (lower than 6% MST), experiments with small sparse (HIV-CA, PTE) and some medium sparse datasets (NCI330, NCI145) produced significantly a larger number than the one produced by all the other datasets (see Figure 2.23).

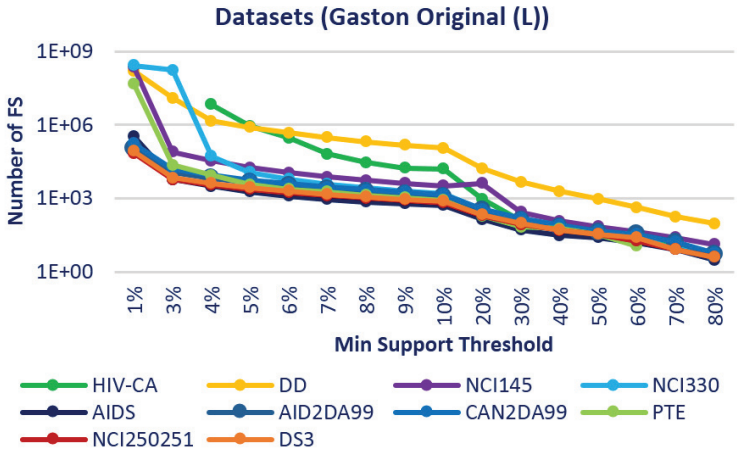


Figure 2.23: Dataset Variation Effect on the Number of FS - Gaston Original (L Strategy)

OS and IDE variation

We used the ParMol framework to test this effect. We conducted experiments using two IDEs : Eclipse with two versions (Mars 4.5.1, Neon 4.6) and Netbeans 8.2. We also experimented ParMol on a terminal. We used the JDK version 1.8_77. The same machine was used as in the previous experiments (see Table 2.12, Section 2.3.1).

In figures 2.24 and 2.25, *Eclipse N*, *Eclipse M*, *Netbeans* and *Terminal* + stand for the use of IDE Eclipse Neon, Eclipse Mars, Netbeans and the Terminal, respectively.

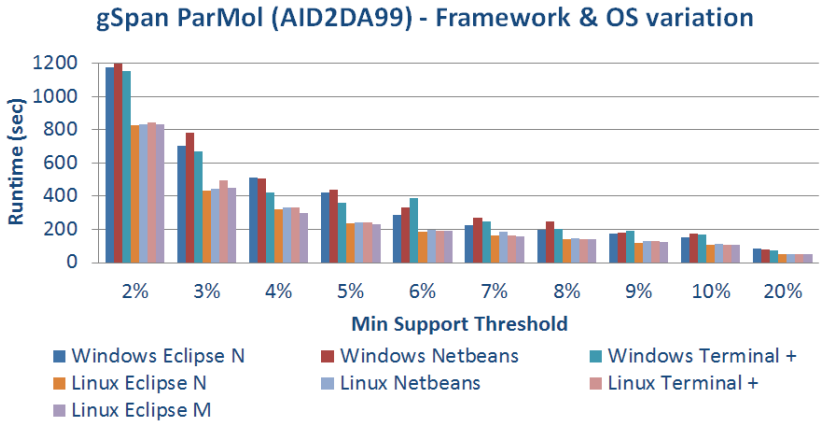


Figure 2.24: gSpan ParMol Runtime performance by OS and IDE - AID2DA99 dataset

The results show that using the same OS (Windows or Linux) and different IDEs (*e.g.,* Eclipse or Netbeans) did not affect the runtime (see Figure 2.24) or memory consumption performance (see Figure 2.25).

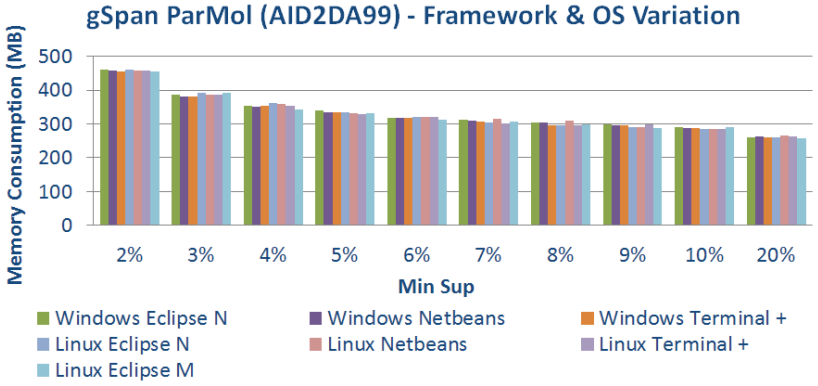


Figure 2.25: gSpan ParMol Memory Consumption performance by OS and IDE - AID2DA99 dataset

However, changing the OS (Linux to Windows) did have an impact on the runtime performance. This is due to the use of the parameter *'memoryStatistics'* in ParMol that calculates the memory consumption. With this parameter, Windows OS had worse runtime performance than Linux (see Figure 2.24) and the same memory consumption (see Figure 2.25). In case this argument (*memoryStatistics*) is set to false, we found no impact of OS variation on the performance (see Figure 2.26, the Windows and Linux *Terminal* - result).

Parameter variation

We performed experiments with ParMol to study the impact of changing its parameters on performance. Among ParMol parameters, *'memoryStatistics'* is the one that had an impact on runtime performance (see Figure 2.26).

In Figure 2.26, *Terminal +* stands for experiments performed on a terminal with the parameter *'memoryStatistics'* set to true and *Terminal -* stands for experiments with the parameter *'memoryStatistics'* set to false. We tested the impact of this parameter combined with two different OS (Linux and Windows).

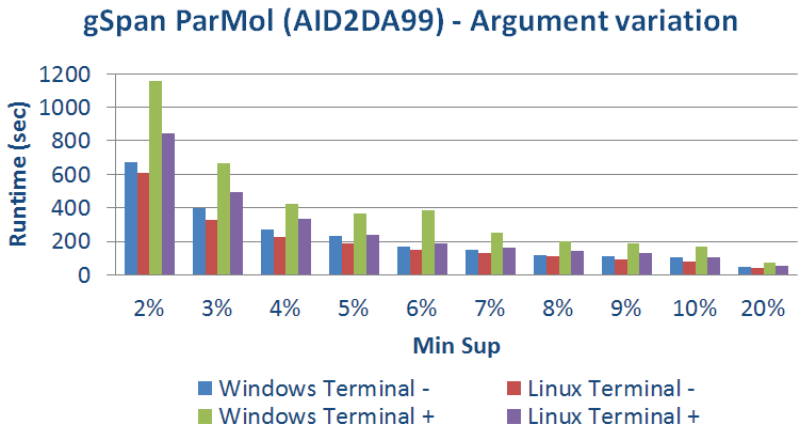


Figure 2.26: gSpan ParMol Runtime performance by argument variation and OS - AID2DA99 dataset

File Format variation

We performed experiments over datasets serialized in different formats including TXT and SDF. No noticeable change in runtime performance or memory consumption was observed.

Labeling strategy variation

In our experiments, we modified the DS3 dataset[39] that contains vertices labeled with integers and strings (*e.g.,* '1', '1u', '2f', '36'). The modification resulted in a dataset labeled with integers only which we named DS3M. Only FSG Original is able to parse string labeled TXT datasets. Hence, this experiment was performed with FSG Original.

The labeling strategy did not affect the performance of FSG Original regarding the number of frequent subgraphs (see Table 2.56) and the runtime (see Table 2.57).

[39]Please refer to Section 2.3.1 for DS3 characteristics.

Table 2.56: FSG Original - Number of Frequent Subgraphs - DS3 *vs.* DS3M

Min Sup	**F**	
	DS3	**DS3M**
1%	80722	80722
3%	6534	6534
5%	2651	2651
7%	1414	1414
10%	725	725
30%	93	93
50%	35	35
80%	4	4

We specify in the following some additional options of Complete search FSM implementations. These options include the incomplete search of frequent subgraphs and multi-threading.

Table 2.57: FSG Original Runtime - DS3 *vs.* DS3M

Min Sup	**F**	
	DS3	**DS3M**
1%	13657.7	13666.2
3%	1463.1	1459.4
5%	891.8	891.8
7%	678	678.6
10%	505.7	505.7
30%	201.633	201.6
50%	130.266	130.2
80%	70.8	70.4

2.3.7 Other Options of tested FSM Implementations

It is possible to perform incomplete search (see Section 1.3.3) using the complete search FSM available implementations (see Table 2.58). This optional setting is important because the search space of complete FSM mining is rich but it is exponential [Ranu 2009, Aggarwal 2010]. There is a need to reduce the set by eliminating the redundancy of subgraph isomorphism [Ullmann 1976].

The proposed settings include the following : (i) specifying the maximum and minimum size of frequent subgraphs to return (gSpan ParMol, Gaston Original), (ii) specifying the minimum and maximum support threshold (gSpan ParMol), (iii) returning only supergraphs (*i.e.,* closed or maximal subgraphs) (gSpan ParMol, FSG Original) (see Table 2.58).

Table 2.58: Optional settings for FSM Implementations

FSM solution	P	SO,SO64	GO	GR	F	D
Incomplete Search Options						
Min and max support threshold	x					
Min of frequent subgraphs size	x	x			x	
Max of frequent subgraphs size	x		x	x	x	
Closed frequent subgraphs	x					
Maximal frequent subgraphs					x	
Trees	x		x	x		
Paths	x		x	x		
Maximum number of subgraph isomorphisms		x				
Input Options						
String labeled TXT datasets					x	
SDF datasets and other chemical formats	x					
Output Options						
Dataset statistics	x				x	
TXT format	x	x	x	x	x	
DFS code format		x		x		x
TID List	x	x		x	x	
PC List					x	
Other Options						
Multi-threading	x	x				
Significant/Discriminative patterns		x				
Weighted graphs		x				

Also, other options are available. Multi-threading is afforded by gSpan Original and ParMol to perform the mining faster (see Table 2.58). Only FSG Original parses string labeled TXT datasets and gSpan ParMol can parse chemical formats of datasets (*e.g.,* SDF). Implementations return frequent subgraphs in TXT format or as DFS codes.

2.4 Conclusion

We reviewed state of the art FSM algorithms in *centralized graph transaction* databases and we selected a subset of algorithms according to some predefined criteria (*e.g.,* availability of implementations). We conducted an experimental study with the selected algorithms using the datasets found in the literature. In our study, we considered algorithms that have working implementation. In fact, our objective is to provide a synthesis of existing FSM solutions for end users. We tested all the implementations of an algorithm (in case more than one implementation was found for a given algorithm). We analyzed the behavior of the FSM solutions according to

the following parameters: (i) execution time, (ii) memory consumption and (iii) the number of frequent subgraphs. Also, we analyzed them by varying two input parameters: datasets and minimum support threshold. We categorized the datasets according to their size (*small*, *medium* and *large*) and density (*sparse*, *dense*) and the size of graphs (*small, medium, large-sized graphs*) ; the minimum support threshold is categorized into : *low* or *high*.

For our experiments, only already tested datasets in the literature are used. This choice would allow the comparison of our results with state of the art findings. This is useful in order to have a summarized synthesis about FSM solutions in the whole literature. We used only real-world datasets because synthetic datasets are generated randomly and thus their features are not easy to compare [Wörlein 2005]. We considered additional variations (including operating system, labeling strategy and the format of datasets) in our study to evaluate their impact on the results of experiments with FSM implementations. We compared our results with the results reported in literature.

This work presented a comprehensive and preliminary study of complete search FSM implementations in centralized graph transaction databases for end-users. We studied all the algorithms found in literature and outlined their merits and demerits. Additionally, we presented the results of an experimental study with the selected and available FSM implementations. We investigated the difference between the algorithms in a quantitative manner, instead of an abstract way (*e.g.,* gSpan is slower than FFSM in general). Our study unearthed the differences and similarities between different implementations of one single algorithm and between different implementations of algorithms based on the literature datasets. Also, we experimented the FSM solutions regarding different datasets and different thresholds. Such a comparison could assist in a preliminary way the end-users in making decision regarding the selection of an implementation for their specific context of use. We tested only datasets and FSM implementations of the literature. This is to allow relating our results with the ones of state of the art.

The added value of this work to existing studies concerns mainly the enumeration of all complete search implementations and a justified selection of tested implementations. Also, we studied the cases of performance of implementations according to the input variation. Finally, this study is an update to the literature because it is performed on the last versions of the implementations. Several research directions are lined up including analysis and explanations that should be linked to this work regarding the difference between the results (number of frequent subgraphs, runtime and memory). Also, this study has been performed only on the literature datasets, we will conduct another study with the most efficient implementations over generic, larger and more diverse datasets. In the following chapter, we will describe the application context of FSM algorithms which is Relational Aggregated Search in graph databases.

Chapter 3

Aggregated Search in RDF Graph Databases

Contents

3.1 Introduction 94
3.2 Aggregated Search in Information Retrieval . . 95
3.2.1 Classical Information Retrieval 95
3.2.2 Federated Search 97
3.2.3 Cross-Vertical Aggregated Search 97
3.2.4 Composite Retrieval 99
3.2.5 Relational Aggregated Search 99
3.2.6 A Comparison of Aggregation-Oriented Information Retrieval Approaches 100
3.2.7 Relational AS and Semantic Graphs 101
3.3 Semantic Graph-based Search 102
3.3.1 Graph search 102
3.3.2 RDF graph : A Knowledge representation model . 102
3.3.3 SPARQL queries 103
3.4 Used techniques in RDF Federated Search dedicated to Aggregation 105
3.4.1 Aggregated Query Decomposition Strategies 106
3.4.2 RDF Graph Partitioning 111
3.5 Query/Fragment Matching 112
3.6 Relational Aggregated IR Scenario : What is done ? 113
3.6.1 Aggregation of fragments in Exact graph search . . 115
3.6.2 Aggregation of fragments in Inexact graph search . 115
3.6.3 Entity-oriented search and Structured result of aggregation 116
3.6.4 Assembling fragments from hetergeneous sources . 116
3.7 Conclusion 116

3.1 Introduction

We define in this chapter the notions related to aggregated search, mainly relational aggregated search. We relate, in this chapter, federated approaches to aggregation approaches and RDF graphs. Federated approaches are interesting for us since they investigated strategies for query decomposition and graph partitioning for query optimization. Graph and semantic data structures have a rich and ready-to-use relations usable for aggregation. Figure 3.1 shows the interactions between the approaches. Please note that comparison of the state-of-the art (in Tables or Figures) in this chapter are marked by the end date of our literature research (*e.g.,* November 2018). This helps future studies to update easier the state-of-the art (*e.g.,* studies after November 2018).

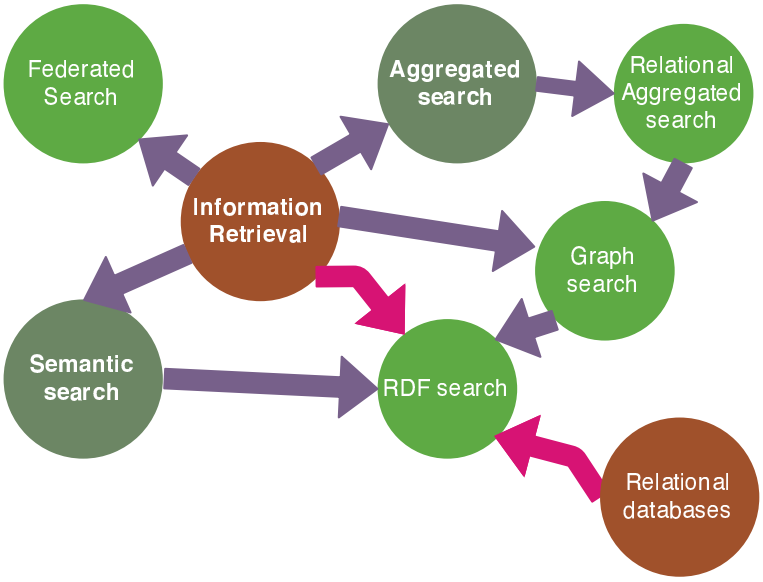


Figure 3.1: An associative map about our concerned fields[1]

This chapter is organized as follows : Section 2 defines information retrieval (IR) process in a classical way and the existing recent categories of IR including relational aggregated search. Section 3 defines the semantic graph search. Section 4 defines query decomposing and graph partitioning strategies used in Federated search and useful in relational aggregated search. Section 5 defines the matching strategies. Section 6 enumerates and describes the studies that contributed in defining relational aggregated IR systems.

[1]Draw.io Tool : `www.draw.io`

3.2 Aggregated Search in Information Retrieval

In this section, we define the information retrieval field, its subcategories including the scope of our work (*i.e.,* the relational aggregated search). We compare the different subcategories.

3.2.1 Classical Information Retrieval

The goal of Information Retrieval (IR) is to conceive systems that are able to find a set of documents constituting potential answers to a user information need. An Information Retrieval System (IRS) searches for relevant documents for a user query [Salton 1986, Moreau 2006, El Charif 2006].

Architecture of an IRS

An IRS consists of three main modules [Belkin 1992] (see Figure 3.2) : (i) Query Formulation (and reformulation), (ii) Indexing and (iii) Document/Query Matching. The querying module consists in the query formulation process. The indexing module corresponds to the representation of the document and query contents. The matching module refers mainly to a matching function between documents and a query. Also, it relates to the evaluation of the relevance of documents compared to the query.

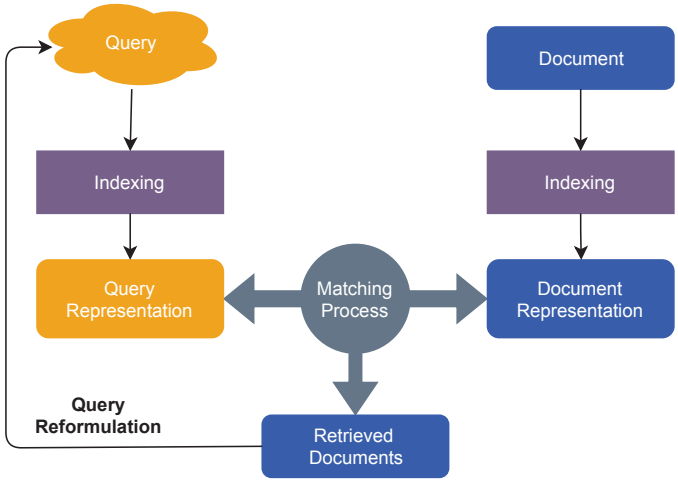


Figure 3.2: The Architecture of an IRS[2][Belkin 1992]

IRS Components

We define in this section the components of an IRS :

[2]Draw.io Tool : `www.draw.io`

Document. A document is a focus element of an IRS. A document can be a text, a part of text, a webpage, an image, a video, etc. A document is any stored data unit that could constitute a response to a user query [Manning 2008].

Corpus. A corpus is typically a structured grouping of integral, documented texts that are potentially labelled. The texts in a corpus should be homogeneous (*e.g.,* sharing the same field) [Rastier 2005].

Query. A query is a possible representation of a user's information need.[3] A query is a set of descriptors and could be expressed by [Van Rijsbergen 1979] : natural language, a bag of words, a boolean expression, graphical form, a formal query language.

Document Relevance. A document is considered relevant to a query by an IRS, if its *system relevance* is high. A system relevance is a score that measures the difference between (i) the similarity between the document and the query and (ii) the relevance probability of documents to the query [Manning 2008]. The system relevance is different from the *user relevance*. The user relevance is mainly concerned with the consistency of the result compared to the query.

A key IRS Process

Indexing Process. Document and query representations (see Figure 3.2) are built using a set of rules and notations transforming the document (query) to a concise and structured representation. This transformation process is called Indexing. The result of indexation is typically a list of representative terms (*i.e.,* descriptors) of a document (query) [Nassr 2002]. The list of representative terms of documents constitutes the indexing language [Nassr 2002].

IRS Evaluation

There are two notions of system evaluation : efficiency and effectiveness [Brini 2007, Sauvagnat 2005].

- *Efficiency.* It is concerned with the time and space. A system is considered better if its time performance between the query formulation and the system response is fast and the used space for search is small.
- *Effectiveness.* It is concerned with the user intellectual effort, the result representation, the quality of the corpus according to the query and the precision/recall. The precision/recall [Kent 1955] criteria concerns the ability of the system to retrieve interesting documents and eliminate the least interesting ones [Zargayouna 2005].

[3] An information need is a mental expression of a user. There are three types of cognitive need [Ingwersen 1996] : verification need, deepening need about a subject (directed need) and a fuzzy need about a subject.

We define, in the following, the categories of recent IR approaches mainly Federated search, aggregated search and composite retrieval.

3.2.2 Federated Search

Federated search consists in searching multiple information sources (called *resources*) [Callan 2002, Lalmas 2011] and aggregating the results returned by the sources. The query is split into subqueries which are sent to sources. The join order of subqueries results should be optimized. The sources do not have access to the whole query [Nassopoulos 2016].

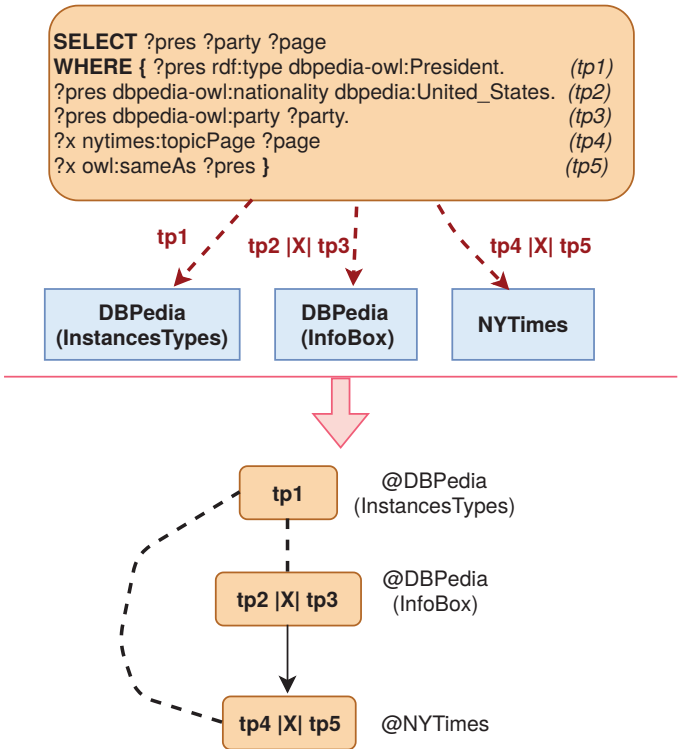


Figure 3.3: Federated search scenario [simplified] [Nassopoulos 2016]

3.2.3 Cross-Vertical Aggregated Search

Aggregated search [Murdock 2008, Lalmas 2011] refers to the tasks of : *(i)* searching, *(ii)* assembling (or aggregating) of information from heterogeneous sources on the Web - called verticals[4] - and *(iii)* presenting these information to the user in one interface. The concept of Aggregated search has been introduced explicitly by Google in 2007 [GooglePress 2007,

[4]Verticals could concern different fields (*e.g.,* News, Travel, Shopping), type of documents(*e.g.,* blog) or different type of media (*e.g.* Image, Video)

Lalmas 2011]. Figure 3.4 illustrates an example of retrieved results from the Google Search engine using verticals (*e.g.,* News, Video) that are relevant to the user query.

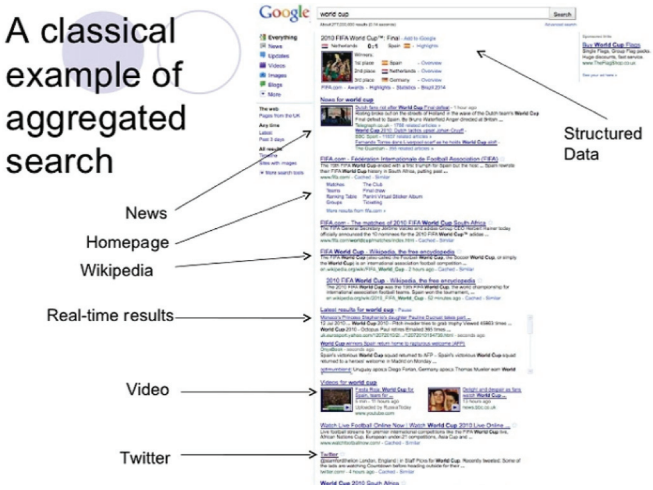


Figure 3.4: An example of cross-vertical Aggregated Search Results (Google) [Lalmas 2011]

Two main processes in the Aggregated Information Retrieval System are different from the classical IR systems (see Section 3.2.1) : *(i)* Matching process and *(ii)* Ranking of Results.

Matching Process

The matching process of classical Information Retrieval consists in extracting relevant *documents* from a corpus while considering a query. However, the matching process of aggregated search consists in extracting relevant *information* from relevant documents using relevant verticals [Lalmas 2011].

Ranking Results

Aggregated Search compares documents of different types (*e.g.,* images with web pages). Due to the heterogeneity of the characteristics of documents, ranking algorithms - used for homogeneous documents ranking - in classical information retrieval systems could not be used.

The aggregated set of documents of Cross-vertical aggregated search system conveys diversity of results. However, the retrieved documents are returned as-is in a list, (i) no aggregation between relevant fragments of these documents is performed [Lalmas 2011, Sushmita 2012, Kopliku 2014] and (ii) no organization of results in a semantic way is provided. The search task from the returned results is still heavy for the user [Kopliku 2014].

Relational aggregated search and composite retrieval try to provide effective solutions for these issues. We define them in the following.

3.2.4 Composite Retrieval

Cross-vertical aggregated search aggregates information from *heterogeneous* sources (*verticals*) and returns results in a single interface. Composite retrieval [Bota 2014] aggregates *heterogeneous* results of verticals in objects (called *bundles*) that are semantically coherent [Bota 2015]. Composite retrieval is dedicated to complex queries. The system should return several bundles answering the query where each bundle could cover a subtopic of the query.

3.2.5 Relational Aggregated Search

Relational aggregated search [Kopliku 2011] consists in aggregating homogeneous or heterogeneous fragments of information using their relationships and returning them in a single result. These fragments of information (also called nuggets [Kopliku 2014]) are constructed by decomposing the information set. Further, relationships are mined or defined between these fragments. The relationships are the key elements for having a relevant, synthesized and rich result. Relations can be extracted : (i) explicitly from the documents or (ii) implicitly found in external relational sources such as knowledge graphs, semantic resources (*e.g.,* ontologies), relational databases or web semantic data [Kopliku 2014]. Also, Information Extraction techniques can be used to extract new relations from the Web [Kopliku 2014].

Object-level or Entity-oriented Search

The Google search engine proposed in 2012 an associative concept named *Google knowledge graph*. The project is oriented to one concept (or entity) queries. Queries containing more than one concept are not considered. The knowledge graph is used for mining associations of facts or data, that are related to the user query. An aggregated *information sheet* (see Figure 3.5) is presented to the user containing attributes about an entity and other associated entities.[5]

In the following, we summarize the difference between the previously defined IR approaches.

[5]PremiumSEOsolutions 2012, Google Knowledge Graph : New search technique
www.premiumseosolutions.com.au/blog/seo-news/google-knowledge-graph-new-searchtechnique/

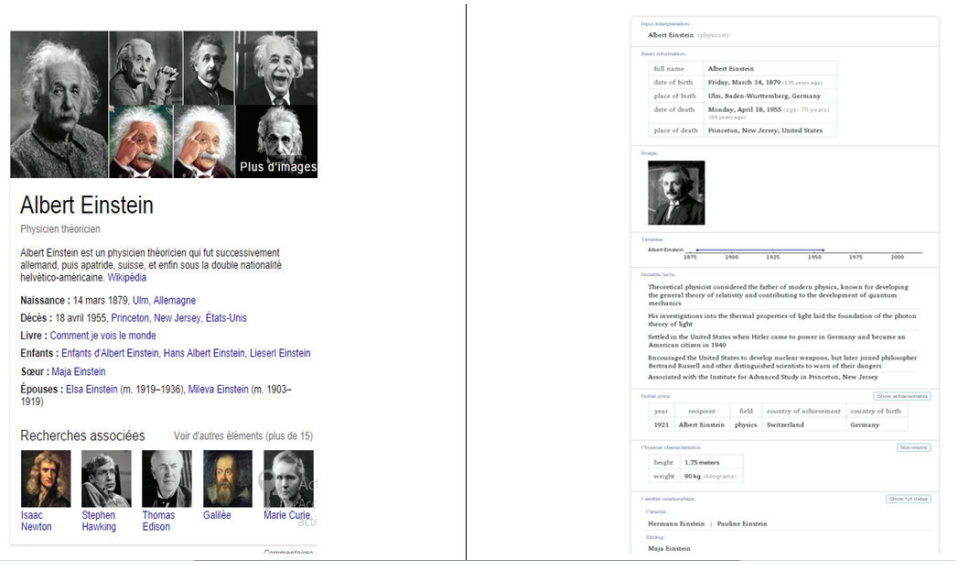


Figure 3.5: Examples of Relational aggregated search from two engines (Left : Google, Right : Wolfram Alpha) for "Albert Einstein" query

3.2.6 A Comparison of Aggregation-Oriented Information Retrieval Approaches

Table 3.1 illustrates the difference between the IR approaches in terms of query decomposition and result aggregation. The compared approaches are : Federated search [Callan 2002], Meta-search [Callan 2002], Cross-vertical [Murdock 2008], Relational aggregated search [Kopliku 2011] and Composite Retrieval [Bota 2014]. Heterogeneous document types in Table 3.1 convey different type of data (*e.g.,* image, video, text). Homogeneous documents have the same type of data (*e.g.,* RDF data, HTML tables).

The heterogeneous documents can be represented by homogeneous data (*e.g.,* an image and a text can be represented by text data) [Achsas 2016, Achsas 2018] (see Section 3.6.4).

Relational Aggregated Search *vs.* Federated Search

Federated search shares the same characteristics with Relational aggregated search (see Table 3.1), the difference is that federated search performs basic aggregation based on multi-source results, while relational aggregated search performs aggregation based on discovered relationships between returned results.

Table 3.1: A comparison of IR Approaches for aggregation (July 2018)

IR Approach	**Decomposes Complex Queries**	**Documents Type**	**Aggregates**	
			Sources	**Documents' Fragments**
Federated Search [Callan 2002]	Yes	*Homogeneous*	Yes	Yes (**One result**)
MetaSearch [Callan 2002]	No	*Homogeneous*	Yes	No (Documents are ordered as-is in **One interface**)
Cross-Vertical Aggregated Search [Murdock 2008]	No	*Heterogeneous*	Yes	
Relational Aggregated Search [Kopliku 2011]	Yes	*Homogeneous/ Heterogeneous*	Yes	Yes (**One result**)
Composite Retrieval [Bota 2014]	Yes (Theoretically)	*Heterogeneous*	Yes	No (Documents are assembled as-is in a result [Bota 2014]) **(Several results)**

3.2.7 Relational AS and Semantic Graphs

One of the subprocesses of Relational AS is the retrieval of fragments. This subprocess is also called *nugget retrieval* [Kopliku 2014] (also named *focused retrieval* in semi-structured IR). The nuggets of information and their relationships could be intuitively represented by a graph structure. Besides, the graph structure is massively used nowadays for representing data in information retrieval systems of several domains (*e.g.,* Biology, Social networks, business processes [Leymann 2008]).

The relational aggregated graph search uses *exact/approximate* graph matching and aggregates the subgraphs to form an *exact* response to the query (see Figure 3.6).

We are interested in the following in defining semantic graph-based search.



Figure 3.6: Aggregation of subgraphs from two graphs *g1* and *g3* responding to the query q [Le 2012]

3.3 Semantic Graph-based Search

We briefly define the classical graph search, followed by the components of the semantic search : RDF graphs and SPARQL queries.

3.3.1 Graph search

Traditional information retrieval in graph databases has a significant difficulty in extracting the result. The candidates of this result are all subgraphs of a graph that could be relevant to the user query. The number of candidates to compare could be very large. Besides, the subgraph-isomorphism task is expensive (NP-complete problem [Cook 1971]). An efficient *graph matching* between the query and the data is performed by reducing : *(i)* the search space (*i.e.,* number of candidates) and *(ii)* the number of relevance checking of candidate according to the query.

The graph matching could retrieve graphs that respond *exactly* to the query. In this case, graph isomorphism is performed. Also, the retrieved graphs can respond *approximatively* to the query (*e.g.,* a fragment of the query). In this case, subgraph isomorphism is performed and a measure of similarity degree between graphs is used.

3.3.2 RDF graph : A Knowledge representation model

Based on Cognitive Science, an assertion postulates that humans construct models in the associative memory in order to organize knowledge [Quillan 1966]. Semantic networks [Quillan 1966] and their formalism [Fouqueré 1994] have been proposed as an external representation of a human set of knowledge about a specific domain [Habrant 1999]. A semantic network is a graph oriented structure without cycles. It encodes objects with their properties. The nodes of the graph are labeled by concepts and the arcs are labeled by (typically binary) relations between concepts.

A derivation of semantic networks used for the representation of the data of the Web is : *RDF graphs*. The RDF (Resource Description Framework) formalism represents data by identifying their resources, properties and defining relationships between them. A relationship between two resources forms a triple (also called a statement). The first resource is named *subject* and the second is named *object*. The relation between an subject

and an object is directed, and is named a *predicate*. Subjects and predicates are represented by *URIs (Uniform Resource Identifier*, see Table 3.2). IRI (Internationalized Resource Identifier)[6] is a generalization of URI (Uniform Resource Identifier) where URI does not allow using non-ASCII characters. Subjects can also be represented by a *blank node* which means that the resource does not have an IRI and is local in the RDF dataset. The objects can be URIs, blank nodes or literals. A literal is a string value that can have a datatype. A triple is represented as:

$$<subject> <predicate> <object>$$

Table 3.2 represents an example of RDF triple *where DIP_Presents_the_Upsetter is an album and its producer is Lee_Perry.*

Table 3.2: An example of triple RDF

Subject	<http://dbpedia.org/resource/**DIP_Presents_the_Upsetter**>
Property	<http://dbpedia.org/property/**producer**>
Object	<http://dbpedia.org/resource/**Lee_Perry**>

The set of triples forms a RDF graph which is a directed, labeled graph, and where the nodes represent resources and the edges represent the predicates.

Figure 3.7 shows an example of a RDF graph specifying for entities (*e.g.,* Peter), their relationships (*e.g.,* knows) with other entities (*e.g.,* Simon) and their attributes (*e.g.,* age 42).

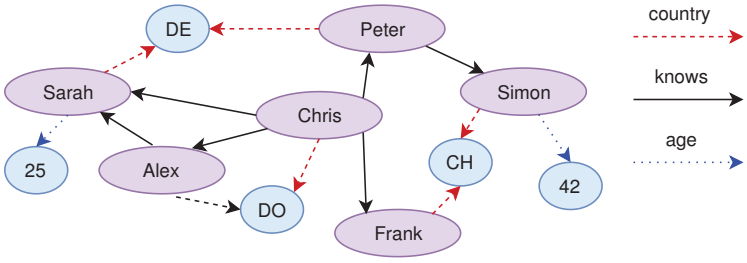


Figure 3.7: A RDF graph example [Przyjaciel-Zablocki 2011]

3.3.3 SPARQL queries

SPARQL[7] is a standard query language for RDF data. We define in the following the structure of a SPARQL query.

[6]Resource Description Framework (RDF) `www.w3.org/RDF/`
[7]SPARQL Query Language for RDF `www.w3.org/TR/rdf-sparql-query/`

Four Forms of SPARQL Query

The SPARQL query language proposes four forms of results : (i) *SELECT* query returns all (or a subset of) the variables in the query pattern, (ii) *CONSTRUCT* returns a constructed RDF graph by replacing variables in triple patterns, (iii) *ASK* returns a boolean value indicating if there are matches for the query pattern or not, and (iv) *DESCRIBE* returns an RDF graph describing the retrieved resources.

We defined the first clause of a SPARQL query according to the aim of query (*i.e.,* returning variables instances, describing variables, constructing a graph from variables, verifying the existence of graph patterns). We define in the following the components of the WHERE clause.

Graph Patterns of SPARQL Query

The second clause WHERE of SPARQL defines the graph pattern matching type [W3C 2013].

Graph Pattern. Let the query q [Fletcher 2008] be

SELECT ?a ?t WHERE { *?a authored ?d ?d type ?t .* }

A triple pattern (*e.g., ?a authored ?d*) has the RDF triple form, however, the subject, predicate and object can be variables.[8] A graph pattern of a SPARQL query represents a set of triple patterns that would be matched to subgraphs of the graph data [W3C 2013].

The WHERE clause specifies the *basic graph pattern(**BGP**)*[W3C 2013]. For example, in query q, the basic graph pattern contains two triple patterns *?a authored ?d and ?d type ?t.*

Keywords (or operators) in the WHERE clause define the graph pattern matching type [Cunha 2015, Abbas 2017]. We briefly define these operators.

- *AND* : All graph patterns should match for the final result. The patterns are combined by conjunction and forms a group graph pattern.
- *OPTIONAL* : Additional patterns can be added to the result.
- *UNION* : Offers alternative graph patterns. One or more of the matched patterns are retrieved.
- *MINUS* : The matches of a graph pattern are removed from another graph pattern.
- *FILTER* : Limits the number of results to be returned. Regular expressions can be used to express conditions of filtering.

[8]The variables in SPARQL queries are prefixed by the symbol "?" or "$"

Query Modifiers

Some query modifiers are proposed like in SQL : After the *WHERE clause*, ORDER BY modifier used to order results, OFFSET modifier used to get a subset of solutions starting from one of them, LIMIT modifier to restrict the number of solutions. In *SELECT clause*, DISTINCT modifier used for returning unique solutions and REDUCED used to reduce the number of redundant solutions.

We define, in the following, the main used techniques in Federated search for RDF graph databases that are useful for aggregating systems.

3.4 Used techniques in RDF Federated Search dedicated to Aggregation

Different strategies have been proposed in order to have an optimized and effective search in RDF graphs. [Hammoud 2015] categorizes these studies in four quadrants (see Figure 3.8) where SPARQL query and RDF database are either decomposed/partitioned or not.

We present, in the following, the main strategies for query and database representation : (i) query decomposition and (ii) graph partitioning. We refer to these strategies as conceptual indexing techniques. RDF technical indexing strategies (*e.g.,* RDF indexing structures, RDF storing) are not addressed.

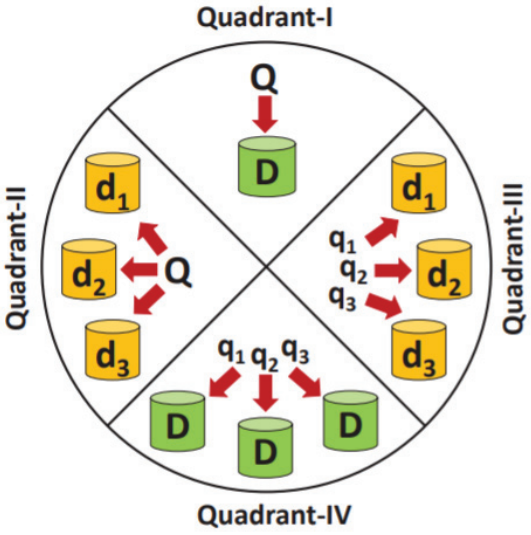


Figure 3.8: Four quadrants of RDF systems [Hammoud 2015]

Table 3.3: An example of SPARQL-like Query and its BGP decomposed

Query : *Find a nobel prize winner named Roald who visited Tunisia and is a researcher supervising PhD students*

SELECT ?X	
WHERE	
{ ?X ***Type*** Person	<– **SQ1 [Persons]**
?X ***First_Name*** Roald	<– **SQ1**
?X ***Won*** Nobel_Prize	<– **SQ2 [Nobel Prizes]**
?X ***Visited*** Tunisia	<– **SQ3 [Travels]**
?X ***Supervised*** ?Y	<– **SQ4 [Universities]**
?Y ***SubType*** PhD_Student }	<– **SQ4**

3.4.1 Aggregated Query Decomposition Strategies

An aggregated query is a query that could be decomposed into parts. Those parts (*e.g.,* sub-queries SQ1 to SQ4, see Table 3.3) are submitted to several sources of information. The results to these sub-queries are aggregated in order to return one result. The need for splitting queries is due to several issues such as : (i) A complex query sent as a whole to data sources, may return no result. Splitting the query according to the sources content can help responding to the expected information need, (ii) A need to reduce the number of sources to which the query is sent, (iii) A need to reduce the complexity of matching between query/graphs (iv) Parallelism to speed up the search (*e.g.,* two subqueries are sent in parallel to the same source).

Studies about query decomposition in RDF search have been mainly elaborated by Federated Search Community.

In the following, we present our selected list of studies that represent a decomposing strategy (see Table 3.4). It is worth noting that this is not an exhaustive list of all studies, we only selected the most representative strategies. A comparison table of other query federation/processing systems is proposed in [Rakhmawati 2013, Rakhmawati 2017]. Another comparison of RDF systems -including query decomposing subprocess- is proposed in [Oguz 2015, Abdelaziz 2017, Horrocks 2017, Yasin 2018]. We notice also that studies (*e.g.,* Trinity.RDF [Zeng 2013]) proposing the decomposition of queries into triple patterns (*i.e.,* one triple per subquery) and studies performing query optimization with *query (execution) plan* (*e.g.,* join order [Schätzle 2016b]) are not in the scope of our following comparison. RDF query processing systems that decompose queries are presented.

DARQ (Distributed Jena ARQ) [Quilitz 2008]. A service description

contains a description of the data available from the source and limitations on access patterns. According to the list of predicates in a service description, a query is rewritten, and subqueries are created [Rakhmawati 2013]. It is worth noting that no up-to-date solution of DARQ exists, since it is no longer developed.

FedX (Federation layer extending Sesame) [Schwarte 2011]. It uses techniques to optimize query processing. A user query is split into subqueries and sent to data sources. The result of the subqueries are returned in an aggregated way. In order to reduce the number of requests sent to the sources and to minimize intermediate results, authors propose applying *join order* optimization and *grouping subqueries*. The triple patterns in the query having the same relevant sources are grouped in a subquery. The relevant sources associated with triple patterns are guessed by sending -in an initial phase- *SPARQL ASK queries* to data sources. The result of these ASK queries is stored in a cache [Rakhmawati 2013].

Defender [Montoya 2012] is used in the query engine ANAPSID. The query decomposer creates unitary sub-queries. Then, it merges subqueries that (i) shares exactly one variable and (ii) can be sent to the same source (*i.e.,* endpoint), in the same subquery using a greedy algorithm. The last optimize the size of intermediate results and reduce the number of requests to be sent to sources.

SHAPE [Lee 2013]. The queries are decomposed according to the graph database partitioning. The database is partitioned into blocks of a vertex and its neighbors (named vertex block). A query is partitioned into subqueries that avoid inter-communication between sources. The approach checks gradually if the query can be decomposed in the minimum number of subqueries that could be processed in intra-partition way. First, all vertices in the query are set as candidates. For each vertex candidate, the largest subgraph (under a k-hop value) in the query and executable in an partition, is found. The aggregation of intermediate results of subqueries is done by Hadoop jobs.

SemStore [Wu 2014a]. It identifies all the rooted vertices in the query graph and construct a rooted subgraph pattern (representing a subquery) for each rooted vertex. If two subqueries share the same source, then they are merged together to form one subquery. This is in order to further reduce the intermediate results.

OLinda [Cunha 2015]. The query is decomposed into subqueries, one for each of the local datasets (*e.g.,* DBLP, DBPedia and Kisti). The subqueries (QDBLP, QDBPedia, QKisti) represent a rewritten part of the query using a matching between the domain ontology and the local ontology of each dataset. Authors do not propose a solution for aggregating the intermediate results.

DREAM [Hammoud 2015, Hasan 2016]. DREAM proposes to decompose SPARQL queries and to not partition RDF graphs. The database is

maintained as a whole in each machine. A SPARQL query is first represented by a directed graph. The *query planner* partitions the query into subgraphs by considering some conditions. The query planner selects the vertices with a degree[9] strictly superior to 1. These vertices are called *join vertices*. Then, it locates the candidates of subqueries which are : (i) the *exclusive* basic subgraphs : Subgraphs in the query with one join vertex and (ii) the *shared* basic subgraphs : Subgraphs with two join vertices. Some rules are applied to assign these subgraphs to a join vertex. These rules include : (i) priority is given to subgraphs that are directly connected to a join vertex. (ii) A shared subgraph can be assigned to one or two join vertices. (iii) A join vertex must have been assigned at least two basic subgraphs. Further, a subgraph is selected for each join vertex according to the lowest cost of the query graph plan. The lowest cost is estimated by collecting statistics for each query plan (using RDF-3X). The statistics use join estimation techniques or mining frequent join paths and return mainly the numbers of visited and generated triples. After the selection of the query plan, the subqueries are executed and the aggregation of intermediate results are performed by one machine (*i.e.,* master machine).

STAR [Yang 2016]. It represents queries as graphs, and decomposes them in star subqueries. Heuristics are used to reduce the number of subqueries and so the number of joins.

Most of the query processing systems use (i) vertices and/or (ii) the sources information in order to decompose the query. Edges (*i.e.,* predicates) in a SPARQL query were used for source selection only (*e.g.,* Defender [Montoya 2012]). In addition, the partition-aware query decomposing studies (*e.g.,* SemStore, see Table 3.4) do not use semantic relationships between triple patterns. Semantic associations between triple patterns in the query are expressed only by common vertices, common paths or neighbourhood. OLinda [Cunha 2015] (see Table 3.4) used metadata of datasets to rewrite the query. Metadata of RDF datasets should be further exploited in order to decompose the query more semantically.

[9]The sum of the in and out edges

Table 3.4: Query federation approaches in RDF search (November 2018)

Approach	**Decomposing Strategy**	**Database Config.**	**Advantages and Drawbacks**
DARQ [Quilitz 2008]	* Builds sub-queries according to the information in the **service descriptions** * A subquery can be answered by one or several sources	Different sources (Datasets)	(-) Need to have a service description (-) Decomposing per source only (-) No longer developed
FedX [Schwarte 2011]	* Subqueries can be grouped according to their **exclusive relevance** to sources * Relevant sources for a triple pattern are known by resorting to **ASK** query	Different sources (Datasets)	(+) No need to use a source description catalog
Defender [Montoya 2012]	* It splits the initial query into unitary triple patterns. It merges together triple patterns sharing **one variable** and that **can be executed** on the same endpoint * Each subquery is sent to a source * It uses a greedy algorithm to optimize the size of intermediate results and the number of requests to be sent to sources	Different sources (Datasets)	(-) For one source, if the variables in the subquery are dependent, no decomposition intra-source could be further done
SHAPE [Lee 2013]	* The query is decomposed according to **vertex-based** database partitions	Partitioned dataset into clusters	(+) No inter-communication between sources (+) Replication of data between partitions

SemStore [Wu 2014a]	* A partition-aware query decomposition * Detection of **rooted vertices** in the query to be decomposed into subqueries	Partitioned dataset into clusters	(+) Partition-aware query decomposition
DREAM [Hammoud 2015]	* Generation of subqueries for each **join vertex** * An estimated minimal cost (calculated using **database statistics**) for each candidate query plan	A dataset replicated in different clusters	(+) Data communication is minimized by using one master machine for aggregation (-) The query decomposition strategy requires **on-line** calculation (*i.e.,* calculating the cost of query plan from the dataset) (-) **One-dataset** approach : Each subquery is sent to a machine containing the same dataset
oLinda [Cunha 2015]	* Subqueries represent a part of query rewritten corresponding to a **local ontology** of each dataset	Different sources (Datasets) and their ontologies	(+) Decomposition of SPARQL queries with operators FILTER and OPTIONAL (+) A subquery uses the same ontology as the targeted dataset (-) A subquery for each dataset, no further decomposition for complex local query
STAR [Yang 2016]	* Queries are decomposed in **star-shaped subqueries** * Finds top-k matches of the query	A dataset	(+) It performs inexact matching of query and graph

3.4.2 RDF Graph Partitioning

Large RDF datasets are typically partitioned into cluster tables (for centralized systems) and cluster machines (named also workers, slave machines or nodes in distributed systems). There are several ambitions leading to partitioning RDF graphs and graph databases in general, *e.g.,*: *(i)* Sending the subquery to a minimum number of sources, *(ii)* handling big datasets storage, *(iii)* optimizing runtime by sending subqueries in parallel to different partitions, *(iv)* minimizing the number of joins and the number of intermediate results by not partitioning a subgraph that would contain data that is frequently queried together. One of the main issues involving partitioning is the potential communication overhead between nodes of data. In what follows, we define the categories of partitioning. A table comparing partitioning systems of RDF data is provided in [Peng 2016, Abdelaziz 2017].

Random Partitioning

Horizontal Partitioning. This category partitions RDF dataset with a number of triples T by requiring a number of partitions N. Each partition contains a subset of triples and sized as T/N [Akhter 2018].

Structure-based Partitioning

Hash-Partitioning. It consists of partitioning data by subject or predicate and it requires a number of partitions as input. This category assigns triples to partitions using a hash value computed based on subjects (or predicates) modulo the total number of required partitions. All the triples with the same subject are assigned to one partition. Due to modulo operation, partitioning imbalance is a potential drawback for this category. Virtuoso (v.6.1.5 Edition20) is an example of predicate-oriented RDF systems [Hammoud 2015]. This category uses fine-grained structural information and does not consider the whole RDF graph structure. For this, partitioning based on larger structures in graphs is proposed.

Graph-based Partitioning. It uses the structure of the graph to partition data. For example, the METIS system puts vertices that are close to each other in the same partition (except the vertices at the boundary of a partition) [Hammoud 2015]. Another example is partitioning the dataset by rooted-subgraphs [Wu 2014a].

Semantic-based Partitioning

Hierarchical Partitioning This category uses IRI's path to determine the hierarchy of resources. This strategy is based on the assumption that resources with the same hierarchy prefix are often queried together. If the number of

prefixes is equal or greater than the number of required partitions, then hash-based partitioning on prefixes is used [Akhter 2018].

Workload-aware Partitioning

It exploits the query workload in order to partition datasets. It collocates together fragments of the data that are frequently accessed by frequent query patterns. Many queries are evaluated without communication using this category of partitioning. Examples of this category are mentioned in [Abdelaziz 2017].

Partitioning Interactions

The partitioning of data affects the query performance. In the literature, the partitioning approach could be chosen according to the shape of queries (*e.g.,* star-shaped, chained queries) [Hammoud 2015]. The partitioning is also impacted by the connectivity of graphs. The more the graphs are connected, the harder is to partition them. Some strategies are included in order to reduce the connectivity of graphs (*e.g.,* eliminating the secondary properties such as rdf:type) [Hammoud 2015].

3.5 Query/Fragment Matching

We categorize studies into : (i) Studies (*e.g.,* [Echbarthi 2017]) which do not perform the query decomposition -in Federated or relational aggregated search- consider inexact matching (Case 1, see Figure 3.9) of query/RDF graph.

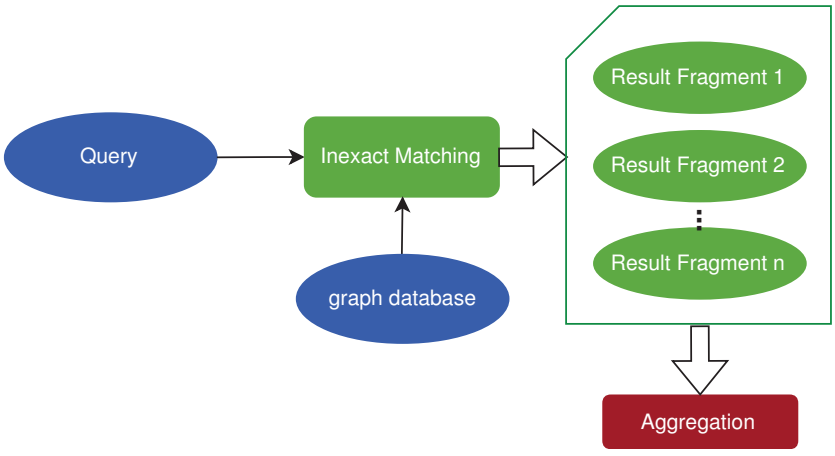


Figure 3.9: Result Aggregation with no query decomposing (Case 1)

They use similarity measures between the query and fragments of graphs, and (ii) Studies performing query decomposition and which may perform exact or inexact matching, and aggregate the results of subqueries (Case 2, see Figure 3.10). In our scope, we consider only the studies of the second category (*i.e.,* Case 2).

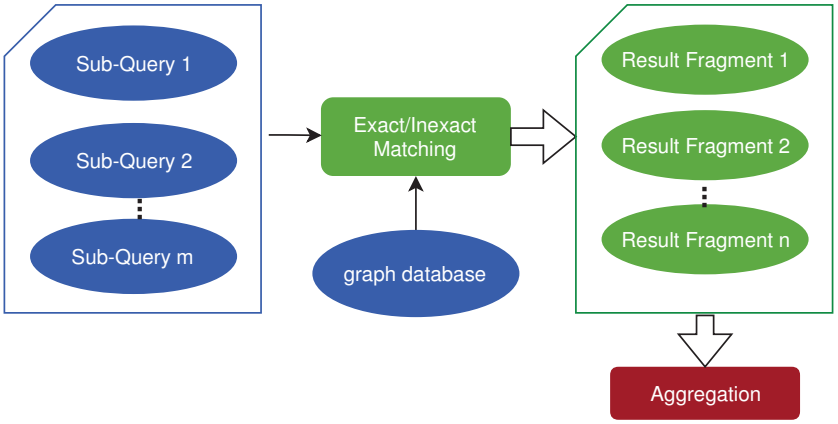


Figure 3.10: Result Aggregation with query decomposing (Case 2)

3.6 Relational Aggregated IR Scenario : What is done ?

We describe in this section the approaches that considered aggregation using **relations** between data. All the studies and their main contributions are shown in Figure 3.11. In general, approaches perform exact aggregation which means that only join operations are performed between fragments. For aggregating in federated search, some join methods (*i.e.,* bound join, nested loop join, hash join, symmetric join and multiple hash join) are detailed in [Oguz 2015].

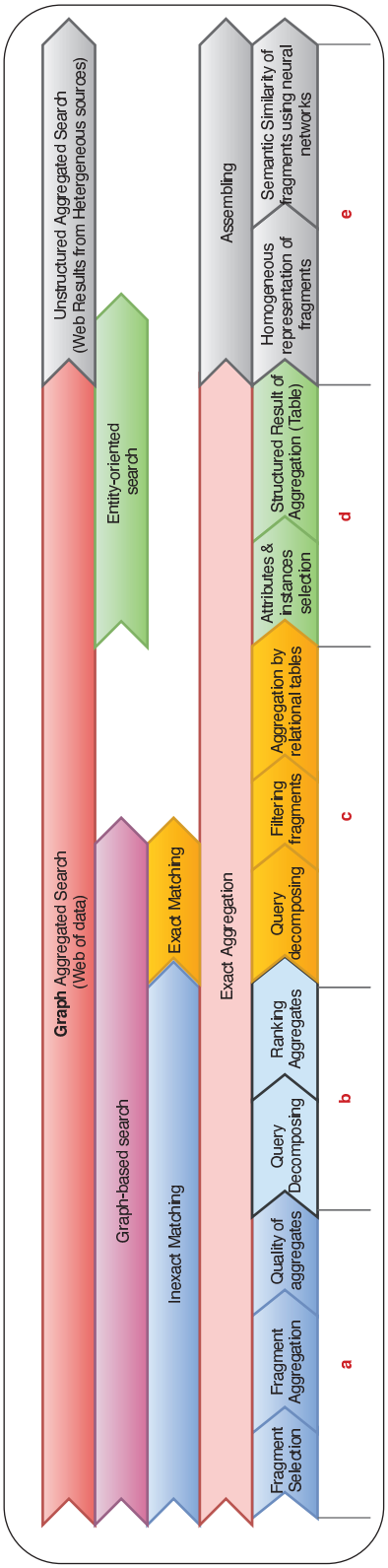


Figure 3.11: Approaches related to Relational Aggregated Search (July 2018)

3.6.1 Aggregation of fragments in Exact graph search

The studies (*colored in Yellow, c*, see Figure 3.11) [Le 2012, Elghazel 2011] propose a methodology for answering aggregated queries in graph databases. [Le 2012] represents graph database in a relational infrastructure. The technique is as follows: First, the common edges between a query and the graph database are extracted using SQL tables. The query is decomposed in two parts : (i) labeled nodes (*i.e.,* constant) subquery, (ii) anonymous (*i.e.,* variable) nodes subquery. The exact matching between data graphs and a query graph is performed as follows: The labeled nodes/arcs are searched in relational tables. Then, the unlabeled nodes related to labeled arcs and nodes are searched. The query is updated and search is complete if all unlabeled nodes have been instantiated.

3.6.2 Aggregation of fragments in Inexact graph search

The study (*colored in Blue, a*, see Figure 3.11) *LaSas [Echbarthi 2017]* proposes a query processing algorithm in aggregated inexact search. The relevant fragments are selected from a dataset using a similarity function. The similarity function (considering nodes label and graph structure similarity) compares the query and the fragments in the graph database. The approach does not require query decomposition strategy. The part of the query that is answered by a fragment is discarded and the query is updated for a new fragment selection. Aggregation consists in a set of the union of obtained fragments. Further, a refinement step is performed that aims at connecting unconnected fragments and pruning irrelevant nodes.

The study [Hsu 2016] (*colored in Blue, b*, see Figure 3.11) proposes a solution for aggregating fragments of *personal process description* graphs (*i.e.,* how-to-do process) for the case when there is no single graph that can anwser the whole query. The graph dataset is decomposed into atomic fragments. The query is decomposed into two categories : (i) *constant* subqueries where each subquery contains two constant nodes and an edge, and (ii) *anonymous* subqueries where each subquery has either *(a)* an anonymous node or *(b)* two constant nodes and a path. The anonymous subqueries are searched based on constant subqueries results. The graph similarity between fragments and subqueries is based on nodes similarity and structure similarity. For reducing the cost of node similarity, the dataset has an inverted index, where words of the node labels of the graph are stored and clustered in word sets, the graphs corresponding to the word sets are stored also. The set of result fragments are then ranked in order to aggregate them. A ranking score of an aggregate is defined based on the product of similarity scores of its fragments and a factor considering the aggregation of fragments from the same process better than the one from different processes.

3.6.3 Entity-oriented search and Structured result of aggregation

Studies (*colored in Green, d*, see Figure 3.11) such as [Kopliku 2011] propose an approach using *HTML tables* in web pages. It detects and orders relevant attributes for an instance or class query. In the same sense, [Krichen 2012] proposes an approach which generates, for each class, a group of homogeneous entities and their attributes with associated values that are collected from a source (*i.e.,* DBPedia) of *the Web of data*. Further, [Abbes 2013] extracts attributes of an entity from *many sources* of the web of data (11 datasets) and evaluates the approach by user-judgments.

3.6.4 Assembling fragments from hetergeneous sources

The study (*colored in Grey, e*, see Figure 3.11) [Achsas 2016] focuses on assembling hetergeneous results (*i.e.,* texts, multimedia). The study concerns Web results. Authors use an homogeneous representation (*i.e.,* text) for the hetergeneous results. For textual documents, they extract paragraphs from the relevant results. Regarding multimedia results, they are represented by their textual description. Then, two processes based on neural networks (*i.e.,* word2vec and stacked autoencoders) are applied on these fragments and their learned semantic similarity is extracted. Further, a clustering algorithm (*i.e.,* DBSCAN) is used to cluster the fragments. Finally, clusters of fragments (*e.g.,* paragraphs, images) are returned. This work helps in returning a set of hetergeneous fragments together using their similarity. A user judgment evaluation is performed on the homogeneity of each cluster. 64.5% of users are satisfied by aggregated results in each cluster. We think that this study is concerned with both Relational AS and Composite Retrieval.

Studies in [Echbarthi 2017, Kopliku 2011, Krichen 2012] considered a framework of aggregation where : (i) there is no need for query decomposition and (ii) there is no distributed sources. These studies focused on retrieval of relevant fragments and their aggregation in a basic framework (*i.e.,* one dataset and one query to send). Studies [Elghazel 2011, Le 2012] uses the relational model as a structure for storing graphs to optimize graph aggregation. [Abbes 2013] focused on entity queries. [Achsas 2016] considered web results and textual representations. For our case, we would consider the web of data and graphs.

3.7 Conclusion

In this chapter, we listed the recent Information Retrieval approaches dedicated to aggregation of fragments from different sources. Also, we described the used techniques for aggregation either in Federated search or relational

aggregated search. From the state of the art of these techniques we observed that the metadata of the web of data was weakly used to discover and then exploit relationships between fragments. In the following chapter, we propose an approach using semantic relationships discovered from the metadata of the web of data, in order to optimize aggregated query evaluation.

Chapter 4

Predicate-based Clustering Approach in Relational Aggregated Search

Contents

4.1 Introduction 118
4.2 Clustering Approach for Aggregated Search . . 118
4.2.1 Semantic Clustering 119
4.2.2 Structural Clustering 126
4.3 Partition-Aware Query Decomposition 126
4.4 Architecture 130
4.5 Conclusion 131

4.1 Introduction

Our approach aims at optimizing the access to information sources in the context of RDF aggregated search. This optimization can be done by : (i) Database reorganization (offline) by clustering fragments that could contribute to answer the same subquery (*e.g.,* the subgraphs with predicates 'firstname' and 'lastname' should be put in the same cluster), and (ii) Query decomposition approach (online) used to send a part of the query only to sources that would provide answers (efficiently). We present in this chapter the two approaches (*i.e.,* Data clustering and query decomposition) of our indexing process.

The remainder of this chapter is organized as follows : Section 1 presents our clustering approaches: semantic and structure oriented. Section 2 defines the query decomposition strategy which exploits the clustering approaches. Section 3 summarizes our approach and describes the whole process (*i.e.,* Aggregated Search system).

4.2 Clustering Approach for Aggregated Search

We propose clustering RDF graphs in order to put together data sources that would be better targeted by a query. The objective of our approach is to be

able to perform as less as possible of external joins. We use two clustering approaches : (i) a semantic one to capture the intuitive associations of data graph independently of the graph structure, and (ii) a structural one using co-occurrence of the data in the graph database.

The source selection for a query in federated search is generally based on a predicate list [Oguz 2015]. This is due to the fact that predicates are often constants in a SPARQL query. Variables are typically subjects or objects. Besides, RDF predicates are fewer to enumerate than resources. We use a unitary data (*i.e.,* predicate) of RDF graphs because we are interested about the impact of associations between these unitary data on the performance of aggregated search systems. Based on these ascertainments, we use the predicates as a key for clustering RDF data and decomposing SPARQL queries.

4.2.1 Semantic Clustering

The clustering of RDF graphs should take into consideration the intuitive relatedness (*e.g.,* First_name and Social_security_number, see Figure 4.1) of predicates (in the mind of the user). For this, we investigated the literature dealing with semantic relatedness/proximity of predicates in RDF graphs.

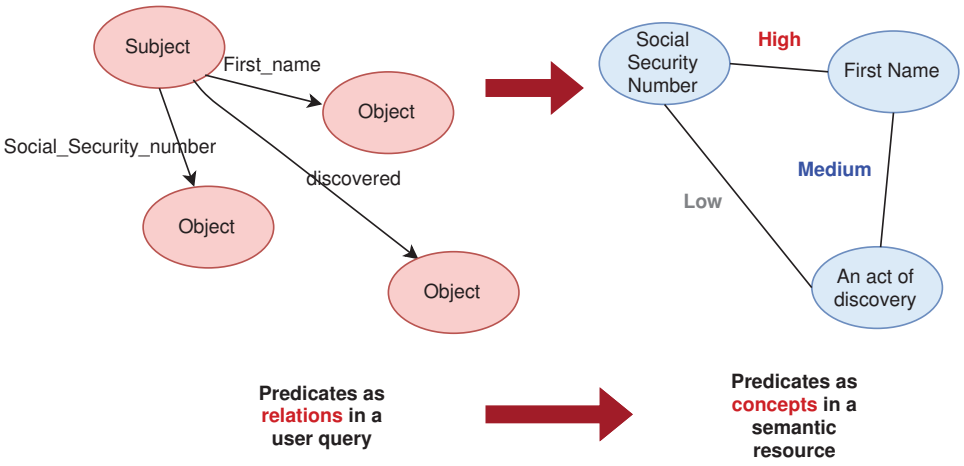


Figure 4.1: From Resources Relatedness to Predicates Relatedness[1]

Semantic Relatedness/Proximity of Predicates

Measures of semantic proximity/distance in RDF graphs have been discussed mainly in Ontology alignment [David 2008, Harispe 2013], web recommendation systems [Leal 2013] or inexact RDF search [Zheng 2016]. The proposed measures focus on comparing concepts or instances in an ontology. The predicates are used in order to compare instances/concepts. To

our knowledge, no semantic measure has been proposed for comparing only predicates in RDF graphs.

We provide a way to semantically compare predicates by resorting to the use of metadata. We use the description of predicates where the concepts in the descriptions are compared. Predicates (edges) are projected as a set of concepts (nodes) in a Knowledge Representation resource (*e.g.,* ontology, thesaurus). An example is shown in Figure 4.1. To compare descriptions, we resort to semantic text similarity approaches [Resnik 1999, Han 2013].

Semantic Text Similarity (STS). STS approaches encompass three main categories [Han 2013] :

- The use of vector space model[2] [Meadow 1992]. A text is modeled as a *"bag of words" vector*. The cosine similarity is computed between the two vectors corresponding to two texts.
- The words (or expressions) in the two sentences (or short texts) are collected. Each *pair of words* from the two texts are aligned. This category of techniques maximizes the summation of the similarities of the pairs [Mihalcea 2006].
- The third category uses measures of machine learning models. Lexical, semantic and syntactic features are computed for the texts using resources. Theses features are then classified [Šarić 2012].

Our approach does not require a specific STS approach. We choose two STS approaches affording available implementations. These two tools we selected are ADW [Pilehvar 2013] and UMBC [Han 2013]. They use the second and third STS categories. Other STS approaches can be used further. In the following, we describe briefly the two solutions.

ADW(Align Disambiguate and Walk) [Pilehvar 2013]. Given two texts T1 and T2, ADW tool maximizes the similarity between the senses (*i.e.,* word meanings) [Miller 1995] of words in these two texts. For this, authors use a *semantic alignment* method where for each word in text T1 is represented by a sense that has a maximal similarity with any sense of the words in text T2. The similarity between senses is calculated based on the semantic signature. The semantic signature is a probability vector of a text representing the similarity of senses generated from random walks over the Knowledge Representation resource (*i.e.,* WordNet). Three methods are used for comparing these signatures (vectors) : Cosine Similarity, and two other methods (*i.e.,* Weighted Overlap and Top-k Jaccard) that use sense ranking instead of probability in order to avoid biased values due to the difference of text sizes and WordNet unbalanced structure.

[2]From information retrieval area

UMBC [Han 2013]. UMBC tool proposes an approach (out of three) that combines (i) lexical similarity features (*i.e.,* LSA word, POS tagging) based on a Corpus and (ii) semantic similarity based on WordNet Knowledge.

Authors used a Web corpus (February 2007) from the Stanford WebBase project that contains 100 million web pages from 50 000 websites. A corpus of three billion english words has been obtained. POS (Part of Speech) tagging and lemmatizing was performed. Word co-occurrences in the corpus were counted in a moving window of a fixed size (*e.g.,* size <=1 or size<=4). A co-occurrence matrix of 29 000 words is constructed where the words are POS tagged.

The UMBC *hybrid similarity* measure between two words uses : (i) the LSA similarity which applies a cosine similarity of two word vectors after performing an SVD (Singular Value Decomposition) transformation (*i.e.,* it reduces the 29 000 word vectors to 300 dimensions which correspond to the 300 largest singular values from the matrix), (ii) it also increases the similarity if some specified relations between words exist in WordNet corpus.

For two texts T1 and T2, the UMBC tool applies the Stanford POS tagger and lemmatization. For each word in text T1, an alignment function is applied to find all words in T2 that maximize the similarity function. The final score between the two texts is computed using a similarity function that sums the word similarities for each text divided by their size.

We define the semantic relatedness as follow :

$$sim(p_i, p_j) = STS(desc(p_i), desc(p_j)) \tag{4.1}$$

Where p_i and p_j are the predicates to compare and $desc(p_x)$ is the description of a predicate p_x

Clustering Algorithm based on Semantic Proximity

After applying the selected STS solutions on predicate descriptions, a basic data clustering algorithm (*e.g.,* k-means) is used to create partitions. There are different categories [Wong 2015, Sajana 2016] of clustering algorithms (*e.g.,* Partitional clustering, Hierarchical clustering, density-based, grid-based, correlation clustering, spectral clustering, gravitational clustering). We set one restriction about the clustering solution to use, is that it should propose an affinity matrix (*i.e.,* precomputed distance/similarity between objects) as input. The affinity matrix represents the predicates relatedness values.

Knowledge Resources

We briefly describe the main types of knowledge resources including ontologies that are exploited in our semantic clustering.

Taxonomy. A taxonomy is a hierarchical organization of controlled vocabulary terms [Zargayouna 2005].

Thesaurus. A thesaurus affords more metadata than a taxonomy. A thesaurus contains controlled vocabulary of terms that are related by three types of relationships : Hierarchical, equivalence and associative [Roussey 2011, TopQuadrant 2013].

Ontology. Thesaurus (*e.g.,* UMLS[3]) use generic relationships (*e.g.,* broader, related and "use for"). However, an ontology contains relationships, attributes (or properties) and classes that are defined by the constructor of the ontology [Gruber 1995].

WordNet[4] is a linguistic/terminological ontology [Roussey 2011] since it focuses on defining terms and their linguistic relationships (*e.g.* synonym, homonym). DBPedia Ontology[5] is a cross-domain ontology that provides the classes and properties (*i.e.,* relationships and attributes) used in the DBpedia dataset.

We use in our approach two knowledge resources in two levels (see Figure 4.2): (1) Metadata level : an ontology related to a RDF database (*e.g.,* DBPedia Ontology) and (2) Meta-Metadata level : a Linguistic/Terminological Ontology [Roussey 2011] (*e.g.,* WordNet) for the semantic relatedness of predicates metadata (*i.e.,* description).

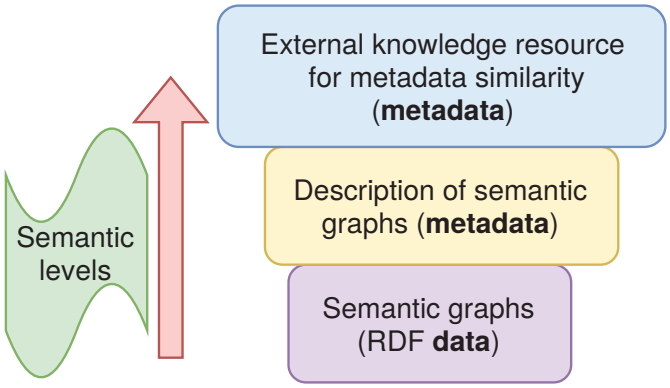


Figure 4.2: Use of metadata in our semantic clustering approach[6]

Semantic Clustering: Assumptions

We define some hypothesis (H) underlying our approach.

H.1 : Predicates are annotated by their descriptions (*i.e., rdfs:comment*). If no description is provided, we use the label (*i.e.,*

[3]UMLS Homepage : `https://www.nlm.nih.gov/research/umls/`
[4]WordNet Homepage : `https://wordnet.princeton.edu/`
[5]DBPedia Ontology `http://dbpedia.org/ontology/`
[6]Draw.io Tool : `www.draw.io`

rdfs:label).

H.2. Predicates with no descriptive metadata are also considered using their local name in URI (*e.g.,* ArtistId => Artist Id).

H.3. Predicates that are semantically related should be put in the same cluster. Similar predicates can be clustered together (*e.g.,* work_in, has_job) if inexact matching will be performed. Otherwise, they can be found in different clusters.

H.4. No semantic technique is used for resources relatedness or similarity. Only predicates are considered.

H.5. Clusters are constructed based only on frequent predicates. Infrequent predicates are added to pre-existing clusters (see Section 4.2.1).

Some general assumptions (A) that should be coherent with our approach results. We list them as follows:

A.1 : Predicates which characterize the same entity should be put together in the same cluster.

A.2 : Predicates which characterize semantically related entities (*e.g.,* Student and University) should be put together in the same cluster.

A.3 : Similar predicates related to (generally) different entities should not be put in the same cluster.

Clustering the predicates should take into consideration not only their relatedness but also their frequency in the dataset. In the following, we present our clustering strategy for the different cases of frequent/infrequent, weakly related, unrelated or non-described predicates.

Frequent and Infrequent Predicates

We discriminate the predicates (see Figure 4.3) in order to : (i) reduce the number of predicates to compare (offline during indexing), and (ii) reduce the number of clusters to target (online during query evaluation). For this, we categorize predicates into frequent and infrequent. Infrequent predicates will generate a low number of intermediate results. For this reason, we choose to cluster on the basis of frequent predicates. We set a minimum frequency threshold value (σ) for predicates frequency.

Frequent Predicates. Frequent predicates (F_p) constitute the core of clustering. They are compared together and according to their semantic relatedness, clusters are created. Frequent predicates that have no relatedness ($sim(p_i, p_x) = 0, \forall p_x \in F_p$) with any other frequent predicate are added to a 'special cases' cluster.

Infrequent Predicates. Infrequent predicates (I_p) are compared with frequent predicates and added to existing clusters (*i.e.,* no additional clusters are generated). We choose to cluster them according to their maximum average relatedness with top-N frequent predicates per cluster. We define the average relatedness of an infrequent predicate (p_i) with a cluster (c_j) as

follow :

$$Sim(p_i, c_j) = avg(sim(p_i, p_x)), \forall p_x \in TF_p(N)) \quad (4.2)$$

The top-N relatedness ($TF_p(N)$) with frequent predicates is calculated based on the most influencing frequent predicates in a cluster (*e.g.,* highly related frequent predicates with others).

Each infrequent predicate is stored in one cluster. This is because duplicating the infrequent predicates in several clusters is useless since the number of joins of triples will not be important in this case. Two cases are considered according to the value of maximum relatedness of the predicate (P_i).

- Case 1 : $max(Sim(p_i, c_j)) > \theta$) The infrequent predicate is stored in the cluster having maximum of relatedness.
- Case 2 : $max(Sim(p_i, c_j)) < \theta$) The infrequent predicate is stored in a 'special cases' cluster. In fact, an infrequent predicate weakly related to predicates (*e.g.,* $\theta = 0.15$) should not be clustered with them.

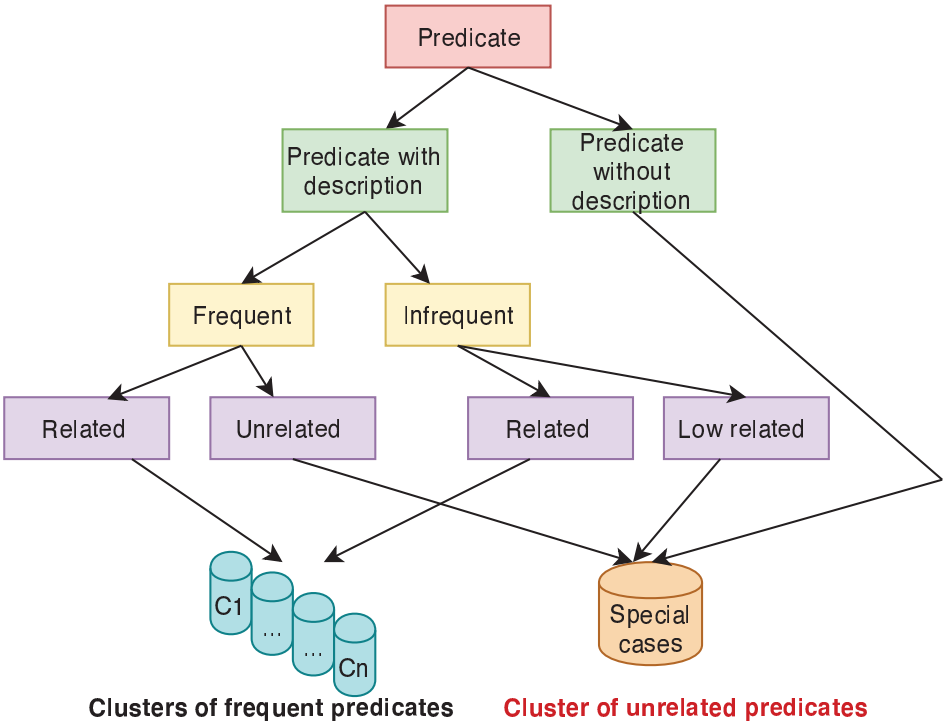


Figure 4.3: Our Clustering Process of Predicates

Table 4.1: ADW Results of 5 most related predicates to "artist"

URI & Description	**Score**
< *http* : *//dbpedia.org/ontology/***lyrics** > "Creator of the text of a MusicalWork, eg Musical, Opera or Song"	0.47
< *http* : *//dbpedia.org/ontology/***producer** > "The producer of the creative work."	0.45
< *http* : *//dbpedia.org/property/***workInstitutions** > "work institutions"	0.42
< *http* : *//dbpedia.org/property/***workInstitution** > "work institution"	0.42
< *http* : *//dbpedia.org/ontology/***developer** > "Developer of a Work (Artwork, Book, Software) or Building (Hotel, Skyscraper)"	0.35

'Special Cases' Cluster

The 'special cases' cluster (see Figure 4.3) contains predicates that are:

- Frequent but not related to any other frequent predicate
- Infrequent and the maximum relatedness with frequent predicates is below a threshold

Example

We took an example of a predicate from DBPedia. We calculated the semantic proximity of the predicate "artist" using the ADW and UMBC tools.

(URI : < *http* : *//dbpedia.org/ontology/artist* >,
Description : *"The performer or creator of the musical work"*)

For ADW tool, the 5 most related predicates to *"artist"* are *"lyrics"*, *"producer"*, *"workInstitutions"*, *"workInstitution"* and *"Developer"*, mentioned in Table 4.1. For UMBC tool, the 5 most related predicates to *"artist"* are *"lyrics"*, *"music"*, *"creators"*, *"creator"* and *"producer"* (see Table 4.2). However, the most related predicates generated by ADW were also generated by UMBC as highly related (*i.e.,* around 0.4).

From the unrelated predicates to "artist" using ADW, there are, *e.g.,*:
< *http* : *//dbpedia.org/property/***name** > "name",
< *http* : *//dbpedia.org/property/***released** > "released",
< *http* : *//dbpedia.org/property/***instruments** > "instruments",
< *http* : *//dbpedia.org/property/***fusiongenres** > "fusiongenres",
< *http* : *//dbpedia.org/property/***bgcolor** > "bgcolor"
< *http* : *//dbpedia.org/property/***subgenres** > "subgenres".
And from the unrelated predicates using UMBC tool, there are :

Table 4.2: UMBC Results of 5 most related predicates to "artist"

URI & Description	**Score**
$< http://dbpedia.org/ontology/\mathbf{lyrics} >$ "Creator of the text of a MusicalWork, eg Musical, Opera or Song"	0.74
$< http://dbpedia.org/property/\mathbf{music} >$ "music"	0.68
$< http://dbpedia.org/property/\mathbf{creators} >$ "creators"	0.68
$< http://dbpedia.org/property/\mathbf{creator} >$ "creator"	..
$< http://dbpedia.org/ontology/\mathbf{producer} >$ "The producer of the creative work."	0.61

$< http://dbpedia.org/property/\mathbf{cover} >$ "Cover"
$< http://dbpedia.org/property/\mathbf{type} >$ "type"
$< http://dbpedia.org/property/\mathbf{length} >$ "Length"
$< http://dbpedia.org/property/\mathbf{reference} >$ "reference"
$< http://dbpedia.org/property/\mathbf{fusiongenres} >$ "fusiongenres"
$< http://dbpedia.org/property/\mathbf{bgcolor} >$ "bgcolor"
$< http://dbpedia.org/property/\mathbf{subgenres} >$ "subgenres"

4.2.2 Structural Clustering

Detecting graph patterns is a task used in many graph applications (*e.g.,* graph indexing, search, similarity, classification, clustering). We propose to use frequent patterns (see Chapter 1) consisting of k-predicates from the dataset. The k-predicate frequent subgraph is a frequent subgraph in the database with k edges. FSM algorithms use general graph input. We convert RDF graphs to general graphs. Then, we apply Gaston original solution (see Chapter 2) to extract k-predicate frequent subgraphs with a medium or high minimum support threshold.

4.3 Partition-Aware Query Decomposition

In our work, we assume that we do not have access to the query logs. We use the source description to decompose a query. The key of the decomposition is the relatedness of predicates. A subquery contains strongly related predicates.

SPARQL query is decomposed also according to the graph patterns keywords (*i.e.,* FILTER, LIMIT, UNION, OPTIONAL, MINUS, see Section

3.3.3) and the case where the predicate is a variable. We define these cases and illustrate them with examples of query Graph Patterns (GP).

FILTER Clause

We specify three cases (named F.1, F.2 and F.3) of "FILTER" as follows :

- **F.1 :** The FILTER clause is concerned with condition(s) about one variable resource (*i.e.,* subject or object) : The FILTER clause is sent with triple patterns relative to the resource variables. GP Example :

$$?x\ P1\ ?y.\ \textbf{\textit{FILTER}}\ (?x\ STR("e")\ \&\&\ ?y < "1988")$$

- **F.2 :** The FILTER clause contains one condition, and it is concerned with one predicate variable : The triple patterns containing the predicate variable are sent to the clusters containing the predicate specified by the FILTER clause. If there is a negation operator in the FILTER clause, then triple patterns containing the predicate variable are sent to all sources. GP Examples :

$$?x\ ?p\ ?t.\ \textbf{\textit{FILTER}}\ (?p = \textbf{\textit{P1}})$$

$$?x\ ?p\ ?t.\ \textbf{\textit{FILTER}}\ (?p\ != \textbf{\textit{P1}})$$

- **F.3 :** The FILTER clause contains several conditions with different variables (*i.e.,* predicates and resources) : (i) Filtering values of variables are concerned with the same triple pattern for resources and the same source for predicates : The FILTER clause is sent together with its triple patterns, (ii) Otherwise, FILTER clauses are created where each FILTER is concerned with one source. The clauses are sent together with their triple patterns to the sources. The master machine performs the union (or intersection) of all FILTER results. Finally, it processes the real FILTER clause on the results. An example of GP :

$$?x\ ?p\ ?t.\ \textbf{\textit{FILTER}}\ ((?p = \textbf{\textit{P1}}\ \&\&\ ?t < "1988")\ ||\ ?p\ !=\textbf{\textit{P2}})$$

OPTIONAL Clause

Clauses "OPTIONAL" are considered according to two cases (named O.1 and O.2) as follows. Triple patterns preceding an OPTIONAL clause should be all executed before the OPTIONAL one.[7] For OPTIONAL clauses containing predicate variables, please refer to the cases of FILTER clause or no FILTER clause.

[7]Apache JENA : SPARQL Tutorial - Optional Information `https://jena.apache.org/tutorials/sparql_optionals.html`

- **O.1 :** The OPTIONAL clause contains one triple pattern or several triple patterns that has (have) predicates in the same source. The query contains triple patterns before the OPTIONAL clause having predicates all from the same source. The predicates in the OPTIONAL clause have the same source as those of triple patterns preceding it. The query is sent as is.

  Example :

  $$?x\ \textbf{\textit{P1}}\ ?y\ \textbf{\textit{OPTIONAL}}\ \{\ ?x\ \textbf{\textit{P2}}\ "u"\ \}$$

- **O.2 :** The OPTIONAL clause contains one triple pattern or several triple patterns that has (have) predicates in the same source. We name the triple patterns preceding the OPTIONAL clause as X. The predicates in X are not in the same source (*e.g.,* P1, P2) and the predicates in the OPTIONAL clause are in the same source as some triple patterns in X (*e.g.,* P2 and P3 are in the same source), the OPTIONAL clause cannot be merged with these triple patterns in the same query. The OPTIONAL clause is sent alone and is executed after X.

  Example :

  $$?x\ \textbf{\textit{P1}}\ ?y.\ ?z\ \textbf{\textit{P2}}\ ?x\ \textbf{\textit{OPTIONAL}}\ \{\ ?x\ \textbf{\textit{P3}}\ "u"\ \}$$

- **O.3 :** The OPTIONAL clause contains N triple patterns containing predicates of several sources. Subqueries of triple patterns of the same source are created. The master machine applies the join between triple patterns. The OPTIONAL operator is processed on the joined results. Example:

  $$?x\ \textbf{\textit{P1}}\ ?y\ \textbf{\textit{OPTIONAL}}\ \{\ ?x\ \textbf{\textit{P2}}\ "u".\ ?y\ \textbf{\textit{P1}}\ ?t\}$$

UNION/MINUS Clauses

The following four cases (named U.1, U.2, U.3, U.4) apply for the UNION clause. The MINUS clause also uses the same four cases. For UNION clauses containing predicate variables, please refer to the cases of the FILTER clause or no FILTER clause.

- **U.1 :** The UNION is performed between two triple patterns T1 and T2 (*i.e.,* T1 UNION T2). Two subcases are considered : (i) The predicates in T1 and T2 are in the same source : The subquery T1 UNION T2 is sent to that source. (ii) The predicates in T1 and T2 are in different sources : T1 and T2 are sent separately to their sources. The UNION of results is performed by the master machine. Examples:

?x ***P1*** *?y UNION* {*?y* ***P1*** *?x*}
?x ***P1*** *?y UNION* {*?x* ***P2*** *?y*}

- **U.2 :** The UNION is performed between two sets of triple patterns S1 and S2 and contains other keywords (*e.g.,* OPTIONAL, FILTER). Two sub-cases are considered : (i) All the predicates in S1 and S2 are in the same source : S1 UNION S2 is sent to that source. (ii) Predicates are from different sources : *(a)* if S1 and S2 do not contain OPTIONAL clause and if S1 and S2 have a common core (*i.e.,* triple patterns that are the same in S1 and S2), and the uncommon triple patterns are from the same source (*e.g.,* P2 and P3), a UNION subquery is performed between the uncommon triple patterns of S1 and S2. The master machine performs then the join of uncommon and common triple patterns. *(b)* Otherwise, S1 and S2 are two subqueries that will be further decomposed according to the cases. The UNION is performed by the master machine. Example:

{*?x* ***P1*** *?y. ?z* ***P2*** *?t. OPTIONAL* {*?x* ***P3*** *?z*}**}**
UNION **{***?y* ***P1*** *?x. ?z* ***P3*** *?t. OPTIONAL* {*?y* ***P4*** *?z*}**}**

- **U.3 :** There are several unrelated UNION clauses in the query: The two first cases (*i.e.,* U.1 and U.2) are considered for each UNION clause. Example:

{*?x* ***P1*** *?y. ?z* ***P2*** *?t.***}**
UNION **{***?y* ***P1*** *?x. ?z* ***P3*** *?t.* **}**
?t ***P4*** *?v*
UNION **{** *?e* ***P4*** *?v. ?e* ***P5*** *"G"* **}**

- **U.4 :** There are several related UNION clauses (*e.g.,* S1 UNION S2 UNION S3) in the query. Two subcases are considered : (i) all predicates in UNION clauses are from the same source : The query *S1 UNION S2 UNION S3* is sent to that source. (ii) Predicates are from different sources : S1, S2 and S3 are considered as subqueries that are further decomposed according to the cases. The UNIONs between S1, S2 and S3 is performed by the master machine on the partial results of the subqueries. Example:

{*?x* ***P1*** *?y. ?z* ***P2*** *?t.***}**
UNION **{***?y* ***P1*** *?x. ?z* ***P3*** *?t.* **}**
UNION **{** *?y* ***P3*** *?x. ?t* ***P5*** *?z* **}**

Predicate variables with no FILTER Clause

Predicate variables in the query with no additional information about their potential values are sent to all sources.

Examples:

?z P2 ***?x.*** ***?x*** *?p* ***?t.*** ***?t*** *P1 "u".*

?x ?p ***?t.*** *Uri1 P1* ***?t.***

?z *P2 ?x.* ***?z*** *?p* ***?t.*** ***?z*** *P1 "u".*

?z *?p* ***?t.*** *Uri1 P1* ***?t.***

?x *P1 ?u. ?t* ***?x*** *?z*

It is worth noting that triple patterns that would be sent to the same source are aggregated in the same subquery having one subquery per source, expect for the cases of UNION, MINUS and OPTIONAL. For these last cases, several subqueries can be sent to the same source. The LIMIT keyword is applied by the master machine if the query is decomposed into several subqueries. In this section, we listed the cases of query decomposition according to the SPARQL keywords and to our clustering. We specified the tasks that would be performed by slave machines and the ones that would be performed by the master machine.

We now summarize our clustering approach by specifying the whole architecture of the underlying system.

4.4 Architecture

The general process is shown in Figure 4.4: predicates are extracted from a RDF dataset (*step 1*) as well as their respective metadata using an Ontology of the dataset (*step 2a*). Further, the semantic relatedness between predicate descriptions is computed using a STS solution (*step 3a*).

A basic data clustering algorithm is applied and RDF data is partitioned according to the clusters of predicates (*step 4*). A parallel partitioning is performed for structural clustering, where RDF graphs are converted into general graphs (*step 2b*) and frequent k-predicate subgraphs are stored (*step 3b*).

A SPARQL query is decomposed into fragments (*step 5*) according to clusters of predicates. Subqueries are sent to a specific partition (*step 6*) and exact (or inexact) matching is performed (*step 7*). Results are aggregated (step 8) by a master machine that performs joins of intermediate results.

[8]Draw.io Tool : `www.draw.io`

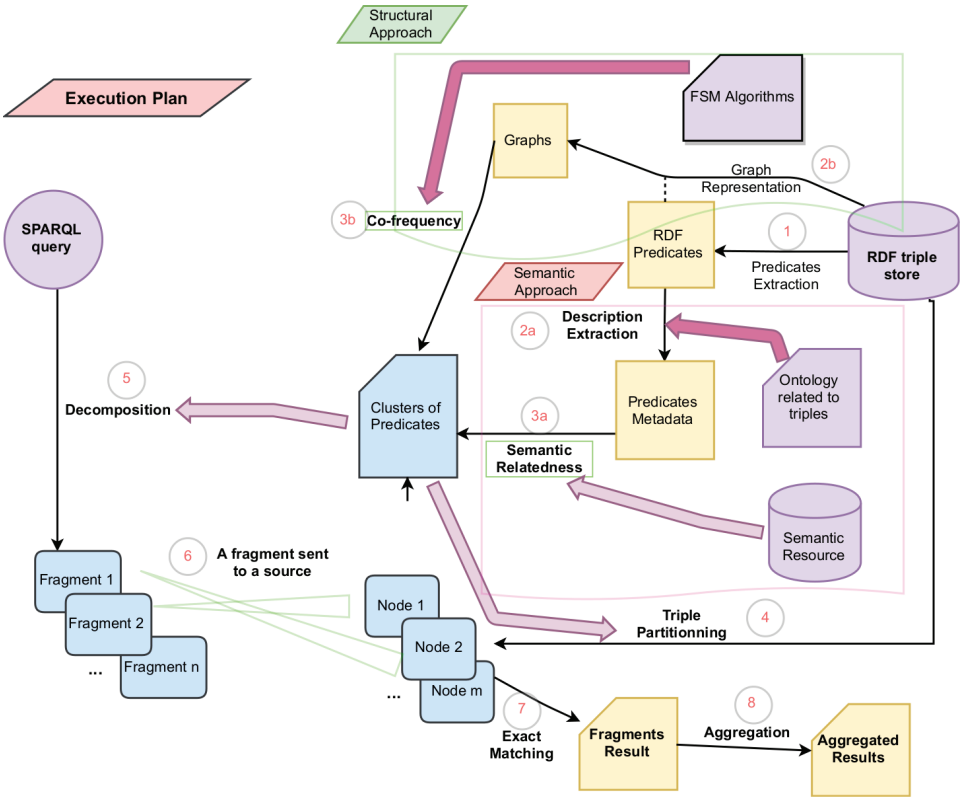


Figure 4.4: Our Aggregated Information Retrieval System[8]

4.5 Conclusion

In this chapter, we described our two clustering approaches for RDF data in an aggregation-oriented search system. We presented our semantic clustering approach which exploits the semantics of RDF in order to assemble semantically related predicates. Semantic clustering focuses on relatedness of predicates rather than on graph structure. Also, we presented our structural clustering using FSM algorithms (presented in Chapter 1 and 2).

We finally specified the query decomposition strategy by considering SPARQL keywords and in accordance with the semantic/structural partitioning. In the next chapter, we conduct experiments to evaluate their impact (if any) on query optimization.

Chapter 5

Experimental Study of Predicate-based Clustering Approach in RA Search

Contents

5.1 Introduction 132
5.2 Definition of Experimental Notions 133
5.2.1 RDF Benchmarks 133
5.2.2 Triple Stores 138
5.2.3 Query Processing Engines and SPARQL servers . . 139
5.2.4 Query Shapes 139
5.3 Experimental Setup 140
5.3.1 DBPSB Benchmark Dataset 140
5.3.2 RDF Framework 141
5.3.3 Machines Configuration 141
5.3.4 Clustering Configuration 141
5.3.5 DBPSB Queries Characteristics 143
5.3.6 Evaluation Metrics of our Aggregated Search System 145
5.4 Experimental Results 146
5.4.1 Predicate Relatedness Results 146
5.4.2 Query Decomposing Results 146
5.5 Conclusion 150

5.1 Introduction

We experiment in this chapter the impact of clustering RDF graphs on aggregated search system using semantics and structural relatedness of predicates. Also, we experiment our query decomposition approach and its effect on aggregated queries.

The remainder of this chapter is as follows : The Section 1 defines main notions related to experiments (*i.e.,* Query features and shapes, RDF Benchmarks). The Section 2 describes the experimental setup composed of the chosen benchmark, metrics and clustering characteristics. Section 3 presents the preliminary results of our experiment.

5.2 Definition of Experimental Notions

We present in this section definitions and literature studies related to experimental configuration. Mainly, we justify our choice for RDF dataset and query set to experiment.

5.2.1 RDF Benchmarks

In order to test our approach, we choose to use an existing benchmark in order to make our results more easily comparable with future approaches. Two types of RDF Benchmarks have been proposed in Literature : Centralized and Distributed. We compare in this section some RDF Benchmarks of Literature and justify our choice of a Benchmark.

Centralized RDF Benchmarks

Centralized RDF benchmarks [W3C 2018] aim at evaluating query engines devoted to access data stored in a single repository. Those benchmarks are used in order to evaluate the triple stores performance. Benchmarks (*e.g.,* SRBench [Zhang 2012]) dedicated to data streaming systems are not included in the following.

Synthetic Centralized Benchmarks They generate triples by using a specific and fixed ontology.[1] Examples of this category of benchmark are LUBM [Guo 2005], BSBM [Bizer 2013], SP2Bench [Schmidt 2008].

Realworld-like Centralized Benchmarks. Some benchmarks propose to generate sized samples of real-world datasets and samples of queries from query logs. Examples of these benchmarks are POSB [Atemezing 2018], BioBenchmark [Wu 2014b], DBPSB [Morsey 2011].

Domain-specific Benchmarks There are some benchmarks dedicated to specific domains. Examples of these benchmarks are SNB/SIB Social Network Intelligence Benchmark, LDBC Social Network Benchmark [Boncz 2013, Erling 2015] for social networks, BioBenchmark [Wu 2014b] for biological data, Last.fm Benchmark [Przyjaciel-Zablocki 2013] for Last.fm data, BSBM (Berlin SPARQL Benchmark) [Bizer 2013] for e-commerce (*i.e.,* products, sellers, consumers and products reviews of consumers), SP2B/SP2Bench (SPARQL Performance Benchmark) [Schmidt 2008] for generating arbitrarly large DBLP-like data, LUBM (Lehigh University BenchMark) [Guo 2005] for generating OWL data from University ontology (*i.e.,* Univ-Bench Ontology), POSB (Publications Office SPARQL Benchmark) [Atemezing 2018] using the EU publications Office (PO) data, WatDiv/WSTB (Waterloo SPARQL Diversity Test Suite) [Aluç 2014] generating synthetic sized dataset and queries from User/Product dataset.

[1] `https://cedar.liris.cnrs.fr/papers/PRIM1-TripleGenerator.pdf`

We are interested by the benchmarks that propose cross-domain and real-world datasets. We describe the most relevant ones in the following.

- **IGUANA (Integrated Suite for Benchmarking SPARQL)** [Conrads 2017]. It is an unified benchmark execution platform for SPARQL queries. It allows the execution of state-of-the-art benchmarks (synthetic or real-world) and the comparison of triple stores based on performance of load times, parallel query execution by choosing the number and type of workers and optional different data changes (*e.g.,* adding triples at runtime). As an example, authors integrated two SPARQL benchmarks generators : FEASIBLE [Saleem 2015] and DBPSB [Morsey 2011]. They evaluated state-of-the-art triple stores using four dataset loads from DBpedia and Semantic Web Dog Food (SWDF) datasets. IGUANA is also able to execute federated (distributed) SPARQL benchmarks. IGUANA supports both query templates and query sets as query input.

- **FEASIBLE (A Featured-Based SPARQL Benchmark Generation Framework)** [Saleem 2015]. It is a query generator framework. Queries sets are generated based on query logs of RDF datasets (*e.g.,* DBPedia). FEASIBLE considers SPARQL query forms : SELECT, ASK, DESCRIBE and CONSTRUCT. Also, it considers the query result sizes, execution times, triple patterns and join selectivities, and number of join vertices.

- **DBPSB (DBPedia SPARQL Benchmark)** [Morsey 2011]. DBPSB is now a deprecated project. IGUANA framework includes the DBPSB benchmark and some of its functionalities. The benchmark proposes a real-world like dataset that simulate the distribution of the DBPedia data [Kim 2015]. The query generator in DBPSB produces a set of query templates. Further, the query generator instantiates these templates with RDF terms from the dataset. Using query templates instead of query sets DBPSB considers SPARQL clauses : UNION, OPTIONAL, FILTER, LANG, REGEX, STR, and DISTINCT. And it produces samples of SELECT queries. Other query features are not considered for evaluation [Saleem 2015].

We summarize in Table 5.1 the characteristics of these three SPARQL/RDF benchmarks with their advantages and drawbacks.

Table 5.1: Centralized RDF benchmarks (June 2018)

Benchmark	Database	Queries	Metrics & Variables	Pros and Cons
IGUANA [Conrads 2017, Conrads 2015]	Centralized or federated (Datasets are taken from state-of-the art benchmarks)	Queries from state-of-the-art benchmarks	Triple stores performance : Load times, parallel query execution and data updates	(+) A unified benchmark for the execution of all the state of the art benchmarks (+) It provides metrics not provided by classical benchmarks and a possibility of user update of data (+) Usable with centralized or federated benchmarks (-) Two benchmarks need to be used : IGUANA and a benchmark providing dataset and queries

FEASIBLE [Saleem 2015]	No	Generated queries from query logs related to a RDF dataset (*e.g.*, DBPedia, SWDF)	* Triple Stores performance : Queries per second (QpS) and query mixes per hour (QMpH) * Query features : Query clauses : UNION, DISTINCT, ORDER BY, REGEX, LIMIT, OFFSET, OPTIONAL, FILTER, GROUP BY, query result sizes, execution times, triple patterns and join selectivities, number of join vertices * Query Forms : SELECT, ASK, CONSTRUCT, DESCRIBE	(+) Use of different query features and query forms for evaluation (+) Queries can be generated providing a user query log (-) Benchmark providing only queries
DBPSB [Morsey 2011]	Centralized datasets (a sized sample of DBPedia)	Query generation using query logs (SELECT queries)	* Triple Stores overall performance : query mixes per hour (QMpH) * Triple Stores query-based performance : Queries per Second (QpS) * Query features : UNION, OPTIONAL, FILTER, LANG, REGEX, STR, and DISTINCT	(+) Cross-domain real world like dataset and queries (-) Only SELECT queries (-) Basic query features (*i.e.*, SPARQL clauses)

Dataset Characteristics of Benchmarks

Table 5.2: RDF datasets characteristics in Benchmarks from [Kim 2015] (modified)

Benchmark	Predicates	Triples (millions)
LUBM [Guo 2005]	18	**1335**
SNB [Boncz 2013]	44	387
SP2Bench [Schmidt 2008]	77	**1399**
WatDiv [Aluç 2014]	85	< 1
DBPSB [Morsey 2011]	**39 675**	183

The DBPSB benchmark proposes a number of predicates larger than other benchmarks (*e.g.,* LUBM and SP2Bench, see Table 5.2). This is due to the fact that DBPSB uses DBPedia that is cross-domain dataset [Kim 2015]. It is worth noting that the largest RDF dataset (2015) in literature [Hammoud 2015] consists of 13.6 billions of triples (disk size 2.5 TB).

To summarize, we choose to use the datasets and queries of the DBPSB benchmark for the following reasons :

(i) DBPedia is a real cross-domain dataset that is widely known. Other benchmarks afford domain-specific datasets (*e.g.,* e-commerce for BSBM, university for LUBM, social networks for SNB)

(ii) The number of the predicates (*i.e.,* 39 675 predicates) is much superior to other benchmarks (*e.g.,* LUBM with 18 predicates)

IGUANA benchmark is also used for DBPSB query templates instantiation and performance metrics. We will present in the following, how the DBPSB benchmark generates datasets and queries.

Characteristics of DBPSB Benchmark There are two methods (*i.e., raw* or *seed*) of data generation proposed in DBPSB [Morsey 2011]. The experimented seed method in [Morsey 2011] shows better degree of similarity between the whole dataset and the sample, rather than the rand method.

- **Cross-domain data** DBPedia is a cross-domain dataset, that extracts RDF structured data from the Wikipedia project [Cunha 2015, Abbas 2017]. DBpedia v3.6 contains 289,016 classes. 275 of these classes are in the DBPedia ontology. DBPedia v3.6 contains also 42,016 properties. 1335 of these properties are DBPedia-specific [Morsey 2011].

- **Data generation of DBPSB (Seed method)** The dataset is a sample of DBPedia with a scale factor x% (*i.e.,* 10%, 50% or 100%). A seed method is used in DBPSB where x% of classes is selected. For each class, the x% of instances is selected. For each instance, a statement-generation process named *concise bound description (CBD)* is used to retrieve statement with new resources. These new resources

should not be subjects in the initial graph. This process is repeated until the x% of the DBPedia dataset is reached (in terms of the number of triples).

- **Query set** A three-month query log of DBpedia SPARQL is used. The query log contains 31.5 million queries. Queries that are frequently (above 10) submitted are selected. Query redundancy is solved by using sequential variable names (*e.g.,* var0, var1). The set of queries contains 35,965 queries. The queries are clustered according to string similarities.[2] A graph clustering approach of the BorderFlow algorithm is used to cluster queries. 12 272 query clusters are generated. Clusters of size above 5 are considered. Clusters are ranked according to the frequency of the queries they contain. For the highly ranked cluster, the queries with the highest frequency in a cluster and respecting some selected features are chosen. 25 queries are chosen.

Distributed RDF Benchmarks

Federated (distributed) RDF benchmarks such as FedBench [Schmidt 2011], LargeRDFBench [Saleem 2018], QALD-4 [Unger 2014] can be used. Using datasets from different fields of these benchmarks is possible in our approach. However, these datasets should have common semantics (*e.g.,* common predicates).

In this work, we are interested by centralized benchmarks in our experiments.

5.2.2 Triple Stores

RDF triples can be stored in either centralized or distributed systems. The triple stores can use relational-based or graph-based stores [Hammoud 2015].

Centralized RDF systems

The main property of centralized RDF systems is that they do not use any communication between nodes, avoiding any overhead [Hammoud 2015]. Apache Jena TDB [Wilkinson 2003], Sesame [Broekstra 2002] and Oracle [Chong 2005] are some of the centralized RDF triple stores.

Distributed RDF systems

With distributed systems, RDF triples are partitioned into clustered machines using partitioning algorithms (see Section 3.4.2, Chapter 4). The

[2]LIMES framework, the Levenshtein string similarity measure and a threshold of 0.9 are used

drawback of these systems is the intermediate data shuffling and communication between nodes [Hammoud 2015].

5.2.3 Query Processing Engines and SPARQL servers

SPARQL Engines have been proposed to execute queries on RDF datasets. The engines contain implementation of data management techniques. Engines can either run queries using APIs provided with RDF frameworks or using a Web access interface named a SPARQL Endpoint [Van Herwegen 2015, Taelman 2018]. A SPARQL Endpoint[3] allows receiving and processing SPARQL protocol requests (*e.g.,* GET, POST of a query) using a HTTP network. The datasets inquired in the SPARQL endpoints are generally decentralized datasets. However, local datasets can be exposed as a SPARQL end-point accessible over HTTP by using SPARQL servers (*e.g.,* Fuseki server, Sesame server) provided by RDF APIs (*e.g.,* Apache Jena, Sesame). APIs provide also SPARQL engines (*e.g.,* Jena ARQ) that are usable in a programming environment. Federated query engines (*e.g.,* Jena DARQ [Quilitz 2008], ANAPSID [Acosta 2011]) are also proposed to process federated queries (*e.g.,* query decomposing) and to access distributed endpoints.

5.2.4 Query Shapes

Basic graph patterns (see Section 3.3.3) of SPARQL queries can have different shapes and these shapes impact the query performance [Aluç 2014, Schätzle 2016a]. The position of variables (as subject or object) in the triple patterns defines the query shapes [Schätzle 2016a].

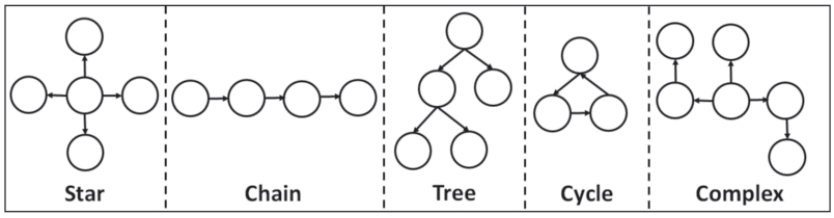


Figure 5.1: SPARQL BGP Query shapes [Wylot 2018]

Star Queries

The queries are of diameter[4] equal to one. The star query allows joins of subject to subject. The join variable in the query is the subject of all triple patterns.

[3] `https://www.w3.org/TR/sparql11-protocol/`

[4] The diameter of a SPARQL BGP is the longest path of triple patterns ignoring edge direction [Schätzle 2016a].

Path Queries

Also named Linear, chained or chain queries. A path query contains triple patterns connected like a chain. It allows joins of a subject to object (or object to subject).

Tree Queries

The tree query allows joins of subject to subject (or subject to object).

Cycle Queries

The cycle query contains cycles and it allows three types of join : Subject to subject, subject to object and object to object.

Complex Queries

The complex query is a combination of different shapes. For example, snowflake-shaped query is a combination of star-shaped queries connected by short paths [Schätzle 2016a].

We present in the following our experimental configuration for testing our clustering approach in RDF Aggregated search system.

5.3 Experimental Setup

We present in this section the statistics of the used dataset, used RDF framework, machines configuration and characteristics of queries that are part of the benchmark DBPSB (see Section 5.2.1). We also define the clustering characteristics. Finally, we define the used metrics.

5.3.1 DBPSB Benchmark Dataset

We choose to use a sample of 10% of DBPedia provided by the benchmark.[5] We loaded the dataset after correcting a parsing error. The error concerns a space in an URI *".../brake horsepower"* that we fixed by *".../brakeHorsepower"* using the *sed*[6] command in the terminal.

Table 5.3 illustrates the characteristics of this dataset in terms of $|T|$: Number of triples, $|P|$: Number of properties, $|DP|$: Number of properties with description (comment or label), $|LP|$: Number of properties with label, $|AP|$: Number properties with added description, $|FP_{1010}|$ and $|FP_{100}|$: Number of frequent predicates with minimum frequency threshold set to 1010 and 100, respectively.

the average of frequency approximatively equal to 1010

[5] https://github.com/dice-group/IGUANA/wiki/How-to-execute-DBPSB

[6] Sed - Cut. `https://doc.ubuntu-fr.org/sed`

Table 5.3: DBPedia10 (10% of DBPedia) Characteristics

$\|T\|$	$\|P\|$	$\|DP\|$	$\|AP\|$	$\|LP\|$	$\|MLP\|$	$\|FP_{1010}\|$	$\|FP_{100}\|$
14 274 115	14 130	11 100	3 030	10 955	652	1 218	3 132

Most of the predicates are described by labels (see Table 5.3). Only 145 predicates are described by a definition (*i.e.,* comment). Also, about 22% (*i.e.,* 3 030) of predicates do not have a description, we added them one by using their URI's local name. 147 out of these 3030 predicates are frequent (σ=100). We modified 652 predicates by removing special characters (*e.g.,* combatant_id => combatant id).

5.3.2 RDF Framework

We used the Apache Jena framework (v3.0.0) which is deployed with the java language. For java programming, we used the Eclipse Framework (Oxygen v.2) and JDK1.8.0_151. For triple stores, we used Jena TDB (v3.8.0) and Fuseki server (v3.8.0). We was inspired by the code in RDF Jena examples[7] for querying and storing RDF data. Practically for our case, three commands are possible for querying RDF data according to the input : *(i) s-query* of Fuseki server if the database is an endpoint (service), *(ii) arq* if the database is provided in a file or *(iii) tdbquery* requiring the triple store path as an input. We use the last one.

The java heap space was set to 2GB. We used fuseki server to load the dataset in the triple store TDB.

5.3.3 Machines Configuration

We set two type of machines : (i) A master machine which is supposed to perform the clustering, send partitions to each slave machine, decompose the query, get the intermediate results and aggregate them and (ii) Slave machines that stores partitions of the dataset, queries them and return sub-results to master machine. We created script files (generated by java code) to execute the following steps : *(i)* machine configuration (*e.g.,* JRE, Fuseki server), *(ii)* store data partitions and load them on triple stores of each machine and *(iii)* send subqueries and execute them.

5.3.4 Clustering Configuration

We present in this section the steps leading to clustering predicates and partitioning data.

[7] castagna jena-examples : `https://github.com/castagna/jena-examples`

Table 5.4: DBPedia10 Predicate Metadata Ontologies

Ontology link	Number of Properties	Number of Triples	Useful Metadata
DBPedia			
dbpedia/property	13 057	6 298 536	rdfs:label
dbpedia/ontology	1055	2 681 739	rdfs:label rdfs:comment
W3C			
www.w3.org	7	4 452 466	rdfs:label rdfs:comment
FOAF			
xmlns.com/foaf	9	780 367	rdfs:label rdfs:comment
Others			
georss.org	1	51 111	-
purl.org/dc/elements/1.1/	1	9896	rdfs:label rdfs:comment

Extraction of predicates and their descriptions

We extracted all predicates having a comment or a label using DBPedia metadata. Table 5.4 illustrates the used ontologies affording descriptions (*i.e.,* label or comment) and their characteristics. The predicates and their descriptions are stored in a relational database.

Predicates (URIs) and their description are stored. We attributed Ids to predicates.

Relatedness Computation

We used two tools UMBC and ADW (see Section 4.2.1). We integrated both ADW and UMBC solutions in our java code. ADW provides a JAR file (ADW.feb2015.jar) that we used to calculate similarity. UMBC provides a web service that we used. WordNet v3.0 was used. Proximity values are from 0 to 1. We stored similarity values and their respective predicates (Ids) in a relational database.

Clustering Method

We create a matrix of relatedness for performing clustering of frequent predicates (See Section 4.2.1). Input variables of the clustering were defined previously (see Section 4.2.1, Chapter 4). For the top-N frequent predicates per cluster ($TF_p(N)$) - used to cluster the infrequent ones - we define the top-N as the most influencing frequent predicates. An *influencing* frequent

predicate is a frequent predicate highly (a minimum relatedness threshold is set, *e.g.,* 0.8) related to many other frequent predicates in the database. The top-N influencing predicates per cluster are compared with infrequent ones. In this experiment, we set N to 1 and the minimum relatedness threshold (β) for influencing frequent predicates to 0.5. The minimum frequency of predicates (σ) is set to 1010. We choose this value since it represents the average of predicate frequency in the RDF database (*i.e.,* 10% of DBPedia). We set three configurations for the number of clusters ($|C|$) to 10, 40 and 100. The minimum relatedness threshold (θ) for infrequent predicates to 0.1. We used the library scikit-learn (v0.20.2) which contains tools for data mining and machine learning to perform clustering. We added Python IDE to Eclipse (PyDev 7.0.3 for Eclipse). We used Spectral Clustering[8] (see Section 4.2.1) from this library as a clustering model. We used this model since it uses an affinity matrix (*i.e.,* input values are not vectors in a space but precomputed similarity values). Other basic models (*e.g.,* k-means) in the library accept only a vector matrix. We set the random state of the algorithm to 0.

After clustering, infrequent predicates are attributed to the formed clusters according to their relatedness. Also, frequent predicates that are unrelated to any other frequent predicates (see Section 4.2.1) and infrequent predicates having low relatedness (see Section 4.2.1) are attributed to a 'special cases' cluster. Clusters (Ids) and their predicates are stored in the database.

5.3.5 DBPSB Queries Characteristics

DBPSB benchmark affords two sets of query templates (*i.e.,* QuerySet2011, QuerySet2012). The proposed queries in 2011 are more complex queries than the ones proposed in 2012 (see Appendix Tables A.2, A.3).

For each query template in the benchmark, there is a successive query template that generally contain the same predicates with additional variations in the variables number and BGP clauses (adding a FILTER, LIMIT). Some queries contain variables that are predicate and resource in the same time (*e.g.,* Q22 in DBSPB Query2012). Some queries have a FROM clause. Since, we work on our own database, we delete the "FROM" clause in these queries.

QuerySet2011

The query set contains 50 query templates.

Predicate Number. 36 queries out of 50 contain at least two constant and distinct predicates. 29 queries out of 50 contain at least 3 constant and

[8]Scikit Learn Spectral Clustering : `https://scikit-learn.org/stable/modules/clustering.html#spectral-clustering`

distinct predicates. There are at most 12 distinct predicates in a query (*i.e.,* Q13, Q14). We can notice that performing query decomposition for this query set is an interesting task.

Predicate Variability. 11 queries out of 50 contain variable predicates. 6 of these queries have filtering values in the query which helps targeting sources. The rest of the queries have unknown predicates that should target sources according to the resources in the query.

Predicate Redundancy. 19 queries out of 50 contain redundant predicates. Most of these queries (*i.e.,* 17 out of 19) have low redundancy (*i.e.,* 1 or 2). The redundancy of predicates does not exceed 4 in a query (*e.g.,* Q9, Q10 with 4 redundant predicates and 5 distinct predicates).

Resource Number. There are at most 12 variable resources in a query (*e.g.,* Q40, see Appendix Table A.2).

Query Shapes. We illustrate in Table 5.5 the query shapes (see Section 5.2.4) of the query set. 33 queries out of 50 are star or path queries.

Table 5.5: Query Shapes of QuerySet2011

Query Shapes	**Total**
Path Queries	13
Star Queries	18
Tree Queries	8
Complex Queries	6
Star/Tree Queries	3
Star/Path Queries	2

Query Keywords. The query set includes some keywords (*i.e.,* UNION, FILTER, OPTIONAL, LIMIT) in queries. 18 queries out of 50 have UNION clauses and 16 queries out of 50 have OPTIONAL clauses (see Table 5.6). There are at most 2 UNIONs in a query. There are at most 8 OPTIONALs in a query (*e.g.,* Q13, see Appendix Table A.2). Different query shapes include these keywords.

Table 5.6: Query Keywords of QuerySet2011

Query Keywords	**Total**
UNION	18
OPTIONAL	16

QuerySet2012

The query set contains 40 query templates.

Predicate Number. 26 queries out of 40 contain at least two constant and distinct predicates. 14 queries out of 40 contain at least three constant

and distinct predicates. There are at most 8 distinct predicates in a query (*i.e.,* Q13, Q14, see Appendix Table A.3). We can notice that performing query decomposition for this query set is an interesting task.

Predicate Variability. 3 queries out of 40 contain variable predicates. All these queries have filtering values in the query which helps targeting sources.

Predicate Redundancy. 6 queries out of 40 contain redundant predicates (see Appendix Table A.4). All these queries have a redundancy equal to 1 (*i.e.,* One predicate is redundant once in the query).

Resource Number. There are at most 5 variable resources in a query (*e.g.,* Q24, see Appendix Table A.3).

Query Shapes. We illustrate in Table 5.6 the query shapes (see Section 5.2.4) of the query set. 33 queries out of 40 are star or path queries.

Table 5.7: Query Shapes of QuerySet2012

Query Shapes	**Total**
Path Queries	17
Star Queries	16
Tree Queries	2
Complex Queries	3
Path/Complex Queries	2

We can notice that the 2012 query set in terms of query shapes, is less balanced than the one of 2011.

Query Keywords. The query set includes some keywords (*i.e.,* UNION, FILTER, OPTIONAL, LIMIT) in queries. 7 queries out of 40 have UNION clauses and 8 queries out of 40 have OPTIONAL clauses (see Table 5.8). Most of the queries contain 1 or 2 UNIONs. There are two queries containing 8 UNIONs (*i.e.,* Q13, Q14, see Appendix Table A.3). Queries contain at most 2 OPTIONAL clauses (*e.g.,* Q24, see Appendix Table A.3). Different query shapes include these keywords.

Table 5.8: Query Keywords of QuerySet2012

Query Keywords	**Total**
UNION	7
OPTIONAL	8

Query sets 2011 and 2012 have different characteristics. The QuerySet2011 contains more complex queries than the QuerySet2012. However, the QuerySet2012 have more UNION clauses per query than QuerySet2011.

5.3.6 Evaluation Metrics of our Aggregated Search System

The potential metrics to test the effect of our clustering on aggregated search are : (i) Runtime : the time spent by tasks devoted to query processing and

aggregation of results, (ii) Quality of IR results (iii) Number of requests sent to sources, (iv) Size of intermediate results and number of joins, (v) Number of local vs. external joins, (vi) Partition quality : The representativity of clustered subgraphs compared to the structure of graphs in the dataset. Comparison between partition quality of semantic and structural approaches. (vii) Decomposing quality : Comparing results according to query characteristics (*e.g.,* query shape). In this report, we will focus on the third metric.

Basic System for Comparison

We aim to compare these variations for comparison :

(i) Centralized dataset (no clustering approach) : **NC**

(ii) No query decomposing approach : **NQD**

(iii) Systems performing query decomposing and data partitioning (See chapter 4) : **SDP**

(iv) Unitary subqueries (one predicate) : **OPQ**

(v) Randomly n-predicates subqueries compared to our approach with also n-predicates subqueries : **RNQ**

In this report, we focus on the third and fourth comparisons.

5.4 Experimental Results

We present in the following the results about the impact of the semantic clustering mainly on the query decomposing.

5.4.1 Predicate Relatedness Results

We present first the results of the two tested tools (*i.e.,* ADW and UMBC). We focus mainly on results of frequent predicates. In table 5.9, UMBC tool using both Wordnet and a corpus (see Section 4.2.1) showed a low relatedness between frequent predicates with a 70.79% of similarities are equal to 0 and a 24.03% of similarities are lower than 0.15. ADW tool using only Wordnet showed better results where the number of unrelated frequent predicates is equal to 27.66%. ADW tool showed a majority of relatedness (*i.e.,* 71.13%) that is inferior to 0.15.

5.4.2 Query Decomposing Results

In this section, we present the query decomposing results of the Queries 2012 and 2011 sets (see Section 5.3.5) using our semantic clustering approach. The queries that would lead to one predicate (unitary queries) or to redundant cases of decomposing (*e.g.,* Q37 a subcase of Q31) are not presented.

Table 5.9: Predicate relatedness statistics

Number of similarities : frequent predicates		
	UMBC	**ADW**
$sim = 1$	0.06%	0.04%
$sim >= 0.5$	0.83%	0.11%
$0.25 =< sim < 0.5$	1.65%	0.24%
$0.15 < sim < 0.25$	2.70%	0.85%
$0 < sim <= 0.15$	24.03%	**71.13%**
$0 < sim < 0.05$	7.22%	14,63%
$sim = 0$	**70.79%**	27.66%
TS	741 153	

Table 5.10 shows the predicate relatedness in each query of the Queries2012 set (For Queries 2011 see Appendix A.6) where the number of clusters is set to 40 or to 10. For each query, the clusters of predicates are presented (*e.g.,* Q9 has 3 clusters of predicates : {*birthDate, deathDate*}, {*birthPlace*} *and* {*name*}).

Table 5.10: Semantic relatedness in Queries2012 set

Queries	**Related Predicates**	
	$\|C\| = 40$	$\|C\| = 10$
Q5	type, label, homepage	type, label, page
	page	thumbnail, homepage
	thumbnail	
Q9	birthDate, deathDate	
	birthPlace	
	name	
Q13	writer, creator, starring, director, producer	writer, creator, director, producer
	executiveProducer	executiveProducer, starring, series
	series	guest
	guest	
Q23	subject, comment	
	name	
Q25	influenced	influenced
	page	page, label
	label	
Q29	comment, depiction, homepage	comment, depiction
		homepage
Q31	type, label	
Q33	label, comment	
	abstract, owner	
	redirect	

Clusters in blue represent the predicates that are unrelated to any other frequent predicate in the database (*e.g., name, guest*). Clusters in gray represent the predicates that are not found in the database (*e.g., redirect*).

Comparison of Our approach with Unitary decomposing

Tables 5.11 and 5.12 show the number of resulting subqueries according to our semantic approach, where **TP** is the total number of predicates per query and $|C|$ corresponds to the number of clusters. If the query contains variable and constant predicates, it is noted as c & v **var***(f)* where c corresponds to the number of constants, v to the number of variables and f to the number of constant predicates in FILTER clauses corresponding to the variable predicates.

Table 5.11: Queries 2012 and the number of subqueries

Queries	**TP**	$\|C\| = 100$	$\|C\| = 40$	$\|C\| = 10$
Q5	5	5	4	3
Q9	4	3	3	3
Q13	8	7	3	3
Q23	3	3	3	3
Q25	3	3	3	2
Q29	3	3	1	2
Q31	2	2	1	1
Q33	1 var (7)	5	4	4

We can compare our approach with unitary decomposing. 81% (*i.e.,* 18 out of 22) of queries in the two query sets (*i.e.,* 2011 and 2012) are decomposed with less number of subqueries than unitary decomposition, where the number of clusters is equal to 40 or 10 (see Tables 5.11 and 5.12). For some cases, OPTIONAL and UNION operators do not allow a more optimized subquery number. We can notice that the number of external joins is reduced compared to a unitary decomposing. Also, the number of requested data is reduced compared to a non-partitioned system.

Comparison of Our approach with Predicate source description partitioning

We compare our approach with an approach that partition data according to the predicate source (SDP). Comparison is based on the number of targeted sources. The total number of sources for the second approach (SDP) is equal to 9 corresponding to the predicate metadata sources (including 4

sub-sources of W3C : OWL[9], RDFS[10], RDF[11] and SKOS[12], see Section 5.3.4). In our approach, we consider the number of sources set to 10.

Table 5.12: Queries 2011 and the number of subqueries

Queries	**TP**	$\|C\| = 100$	$\|C\| = 40$	$\|C\| = 10$
Q9	5	5	4	3
Q11	3	3	2	2
Q13	12		12	8
Q15	3	3	2	3
Q17	8	8	8	7
Q19	7 & 1 var (3)	10	8	4
Q21	2 & 1 var (0)	3	3	3
Q23	3	3	2	2
Q29	2	2	1	1
Q31	1 & 1 var (0)	2	2	2
Q33	2	1	1	1
Q39	12	8	8	8
Q43	2	2	1	2
Q47	3	3	3	3
Q49	5	4	4	3

In 58% of queries, our approach targets less sources than the SDP approach for the two query sets 2011 and 2012 (see Tables 5.13 and 5.14).

Table 5.13: Decomposing approaches of Queries2012 set : Targeted sources

Queries	**Our approach**	**SDP approach**
Q5	2	4
Q9	3	3
Q13	**3**	**1**
Q23	2	3
Q25, Q29	2	2
Q31, Q37	1	2
Q33	**10**	**9**
Q39	1	3

In a context of aggregation where different sources and different predicates could exist in the user query, an information retrieval system should

[9]OWL `http://www.w3.org/2002/07/owl#`
[10]RDFS `http://www.w3.org/2000/01/rdf-schema#`
[11]RDF `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
[12]SKOS `http://www.w3.org/2004/02/skos/core#`

consider our semantic clustering approach for reducing the number of targeted sources to ask.

Table 5.14: Decomposing approaches of Queries2011 set : Targeted sources

Queries	**Our approach**	**SDP approach**
Q9, Q15	3	4
Q11, Q23	2	3
Q13, Q39	4	6
Q15, Q49	**3**	**2**
Q17	4	5
Q21, Q47	**2**	**1**
Q29, Q33	1	2
Q43	2	2

5.5 Conclusion

We presented in this chapter, the main notions related to experiments, our setup configuration. Preliminary results were reported about our semantic clustering and its effect on query decomposing.

General Conclusion and Perspectives

Aggregated Search and Semantic Web can be considered as two key fields in Information Retrieval and for distributed sources. Literature relative to a subcategory of aggregated search (AS) : Relational AS does not exceed ten studies. However, contributions in federated search and graph search are strongly related to relational AS.

Our PhD was doing part of the CAIR[13] project which proposed to define the main processes of an AS system. Lining with the same objective, we proposed to define an indexing approach for Relational AS.

We studied for this, frequent subgraph mining in order to store fragments of information that could optimize aggregation. We noticed that the FSM literature lack of some important information about algorithms performance such as that recent algorithms are tested with the least recent ones and no study is proposed about the effect of the variability of input on the performance of FSM solutions. We elaborated an experimental study of FSM solutions and we selected four most efficient ones that can be used according to the case of use.

In order to optimize aggregation operations and sources access, we proposed to cluster information and to decompose queries in order to send subqueries to targeted sources. Sources should contain related information essentially in terms of semantics. For this, we studied the semantic proximity approaches for RDF graphs and proposed to use a unit of graphs (*i.e.,* predicates) that is mostly known in queries. We used the description of predicates in order to guess their semantic relatedness. Our theory is based on the idea that if two predicates are related (*e.g.,* studied, Project), they should be also found related in the query of the user. And so they should be found together in sources for better aggregation.

We partitioned RDF graphs based on this semantic relatedness or on structure relatedness (using Frequent subgraph mining algorithms). We present the cases of SPARQL query decomposing according to the query clauses and the clustering approach (semantical or structural).

We presented an experimental study of the two approaches of clustering. Our experiments use an existing benchmark (*i.e.,* DBPSB) that proposes a cross-domain dataset. We inspected by this study the effect of clustering RDF graphs (mainly semantically) on optimizing the access to sources and reducing the number of joins.

Our future works concern extracting frequent subgraphs of the dataset (DBSPB) for structural clustering and proposing a complete solution of the

[13]CAIR home page: `www.irit.fr/CAIR/fr/`

query decomposing process. Also, the scenario of aggregated search should be completed by using an aggregation algorithm (in the master machine). Finally, we should study the impact of clustering approaches on the constructed Aggregated Search System by comparing the semantic results with structural ones.

Appendix A

Appendix

A.1 CAIR Project

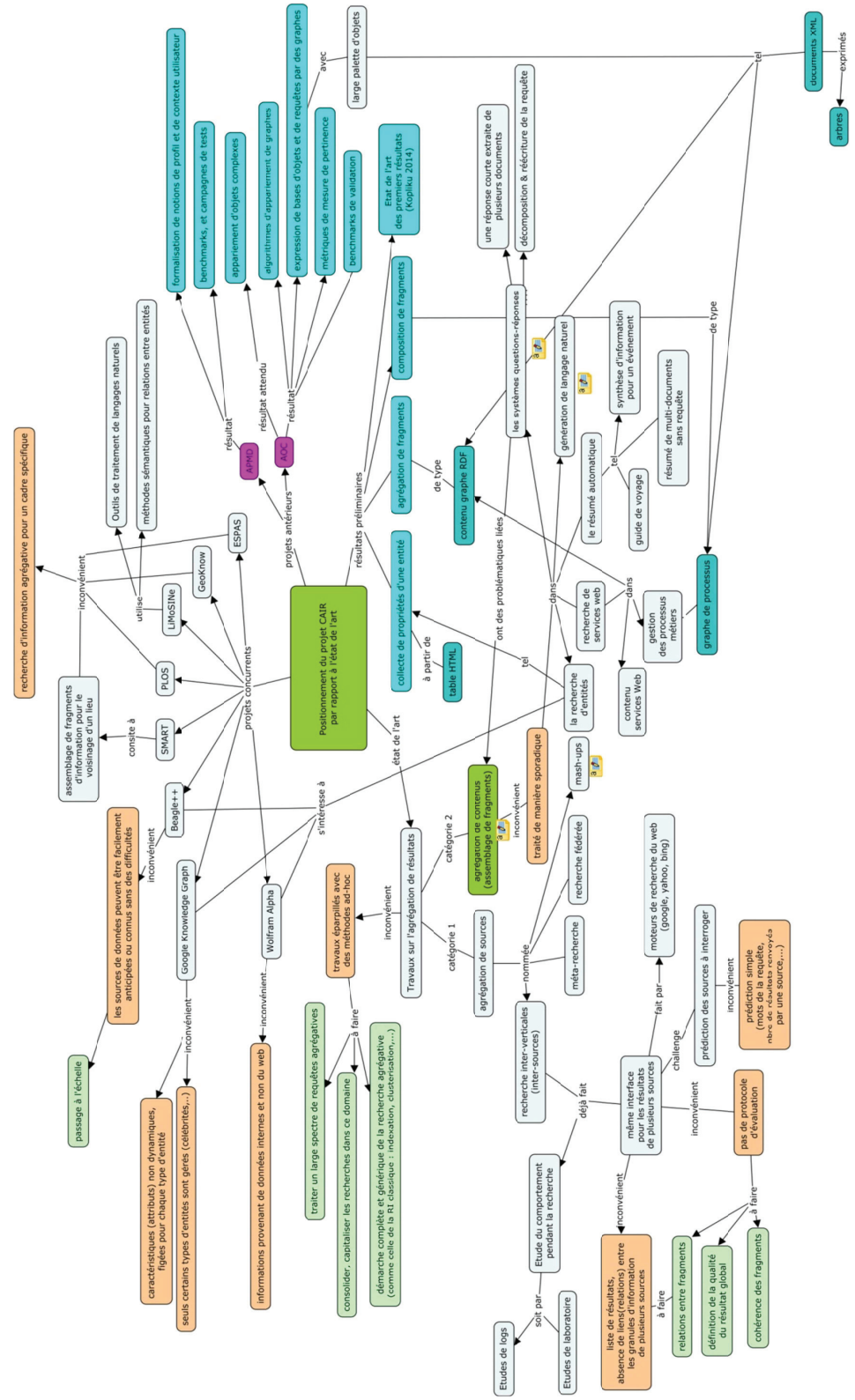


Figure A.1: A Conceptual Map of Aggregated Search Literature described in CAIR project [French Version]

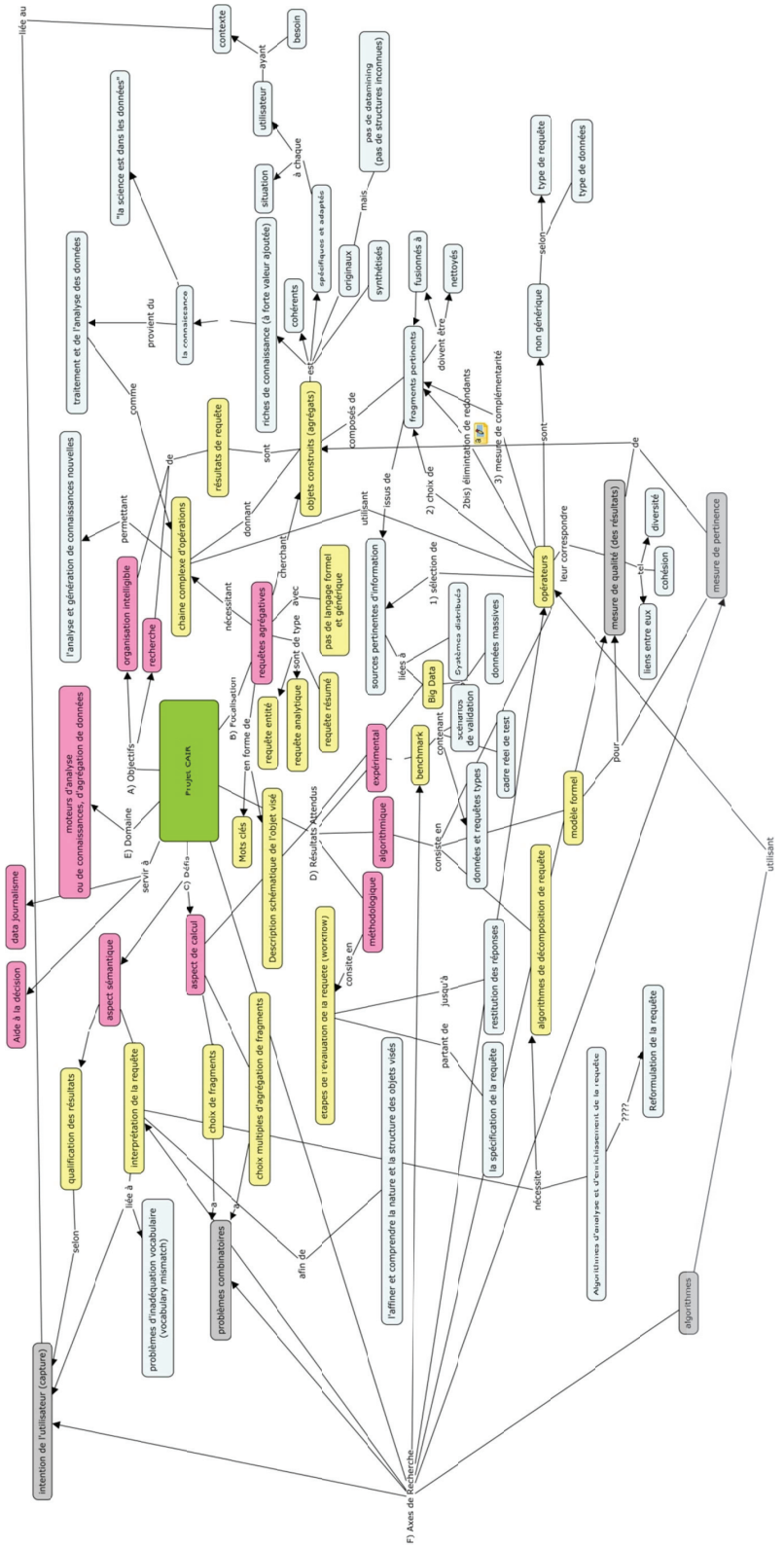


Figure A.2: The axis of study in the CAIR project synthesized in a Conceptual Map [French Version]

A.2 FSM Keywords

Table A.1: FSM Keywords

Graph-based Data Mining (GBDM) / Frequent Subgraph Mining(FSM)/ Pattern Growth Approach/ Frequent Pattern Mining (FPM) / Frequent Subgraph Discovery/ Partial Oder Graph (POG)/ Apriori Approach / graph transaction / support / Graph Frequency / Downward Closure Property / Anti-monotonicity property / Graph Isomorphism detection / Embedding / subgraph isomorphism (SI) / search space structure / DFS code tree / exact matching / error tolerant matching (approximate graph matching)/ inexact matching / semantic variations/ graph topology/ fuzzy matching / frequent conserved subgraph patterns discovery / node or vertex disjoint subgraph homeomorphism / exact search /inexact search/ Structural restrictiveness of exact patterns / frequent approximate patterns / frequent approximate subgraph (FAS) / frequent approximate subgraph mining (FASM)/ Smoothed support/ Completeness of the search / Graph Representation / Search Strategy / Adjacency matrix / Adjacency list / Incidence Matrix/ Trie / Hash table / Canonical Representation (Canonical Labeling, Canonical Labeling strategy) / canonical code/ Lexicographical ordering / Lexicographic code / DFS code / DFS tree / minimum DFS code / Enumeration Tree (Lexicographic Tree) / Depth-First search(DFS) / Breadth-First search (BFS) / Vertical support counting / Horizontal support counting / Graph Partial Order / Random search strategy / Candidate Generation / Level-wise join / Level-wise / Extension/ right most path extension / maximum source node extension / Refinement / Node Refinement / Cyclic Closing Refinement/ Embedding list / Transaction List / Dynamic Graph (or Streaming Graph) / Static Graph / Scalable subgraph mining / sampling based FSM methods / Pruning / exact support / expected-support / Inductive Logic Programming (ILP)/ Embedded Pattern / induced Pattern/ Canonical Spanning Tree (CST) / Canonical Adjacency Matrix (CAM) / Constrained Frequent Pattern Mining / Constrained Graph Pattern Mining/ Constraint -Based frequent pattern mining/ Constraints / Arbitrary Constraints / connectivity constraints/ Compressed Representations of Frequent Patterns / Motifs / Fragments / Frequent Pattern Compression / Frequent Closed Patterns / Maximal Pattern Mining / sub-patterns/ Super-patterns/ Closed Pattern Mining / Informative Patterns / Optimal Patterns / Significant Patterns / Jump Patterns / Representative Orthogonal Graph Patterns / Representative Set of Graph Patterns / top-k frequent patterns/ Informative Patterns / Discriminative Patterns / Non-redundant patterns / Significant patterns/ Representative Patterns/ Frequent Representative Patterns / Topological Representative Subgraphs/ Feature Selection /Summarizing frequent graph patterns/ Summarization algorithms/ Sampling algorithms/ Uniform Sampling of Frequent Patterns/ Informative Subgraphs/ Interesting graph patterns/ Interestingness measure / Frequent Coherent Subraphs / Frequent Quasi-cliques / Weighted Frequent Subgraph Mining / Weight-based Constraints / High-Significance / Low-redundancy / Duplicate Removal / One Single large graph database / Graph transaction database / Transaction setting / Large collection of graphs / Database structure / locally frequent subgraphs / locally infrequent subgraphs / visited subgraphs / unvisited subgraphs / Branch and bound approach / Disk -based Graph Databases / Main Memory based Graph Databases / Synthetic Dataset / Real-world dataset / Graph Generator / Uncertain Graphs/ Exact Graphs / Attributed graphs / Treewidth / Frequent Outerplanar Subgraph Mining Problem (FOSM) / Simple Graph / Multi-Graph/ Maximum Common Subgraphs (MCS)/ Maximum Common Edge Subgraphs (MCES) / Maximum Common Connected Edge Subgraph / Contrast Subgraph Patterns / Interactive Mining

A.3 DBSPB Query Characteristics

- *Keyword(m) (e.g., FILTER(3))* : There keyword is repeated m times in the query.
- *FILTER(nC)* : There is n FILTER keywords in the query where the FILTER contains more than one condition.
- *x (+1 R)* : There are *x+1* predicates including one that is redondant. Redundancy is often due to UNION clauses.
- *y (+1 P)* : There are *y+1* variables including a variable that is a predicate.
- *y (+1 Pr)* : There are *y+1* variables including a variable that is a predicate and a resource in the same time.
- *x (+b var : a) or (var : a)* : There x constant predicates in the query with b variables predicates. The variable predicates are filtered in the query using a constant predicates.
- *x (2 C)* : There are x constant predicates in the query including two of them that are auto-cyclic (*i.e.,* the subject is also the object).

Table A.2: DBSPB Queries2011 Characteristics

Id	**Shape**	**Predicates**	**Variables**	**BGP Keywords**
Q3	Path	0 (+var : 2)	2	FILTER(1C)
Q4	Path	0 (+var : 2)	3	FILTER(1C), LIMIT
Q35	Paths	0 (+var : 5)	2 (+1 P)	FILTER(2C), UNION(1)
Q36	Paths	0 (+var : 6)	3 (+1 P)	FILTER(2C), UNION(1), LIMIT
Q1, Q41	Path	1	1	-
Q2, Q42	Path	1	2	LIMIT
Q37	Path	1 (+1 R)	1	UNION(1), FILTER(1C)
Q38	Path	1 (+1 R)	2	UNION(1), FILTER(1C), LIMIT
Q7	Path/Star	1 (+1 R) (+var : 0)	4 (+1 P)	UNION(1)
Q8	Path/Star	1 (+1 R) (+var : 0)	5 (+1 P)	UNION(1), LIMIT
Q31	Complex	1 (+1 R) (+var : 0)	5 (+1 Pr)	FILTER(2C), OPTIONAL(2)

Q32	Complex	1 (+1 R) (+var : 0)	4 (+1 Pr)	FILTER(2), OPTIONAL(2), LIMIT
Q29 Q43	Star	2	2	FILTER(1) -
Q30 Q34 Q44	Star	2	3	FILTER(1), LIMIT FILTER(1), LIMIT LIMIT
Q33	Star	2 (+2 R)	2	UNION(2), FILTER(1)
Q22	Complex	2 (+var : 0)	3 (+1 Pr)	OPTIONAL(6), UNION(2), LIMIT
Q45	Star	3	2	-
Q15	Star	3 (+1 R)	2	FILTER(2), UNION(1)
Q21	Paths	3	3	OPTIONAL(6), UNION(2)
Q11 Q12 Q46	Star	3	3	FILTER(1) FILTER(1), LIMIT LIMIT
Q25	Paths	3	3	FILTER(1), UNION(2)
Q16	Star	3 (+1 R)	3	UNION(1), LIMIT
Q47	Complex	3 (+1 R)	3	-
Q23	Tree	3 (+1 R)	3	FILTER(1)
Q26	Path	3	4	FILTER(1), UNION(2), LIMIT
Q27	Star	3 (+1 R)	4	OPTIONAL(2), FILTER(2)
Q24	Tree	3 (+1 R)	4	FILTER(1), LIMIT
Q48	Tree	3 (+1 R)	4	LIMIT
Q28	Star	3 (+1 R)	5	OPTIONAL(2), FILTER(2), LIMIT
Q5	Star	5	4	OPTIONAL(1)
Q6 Q49	Star	5	5	OPTIONAL(1), LIMIT -
Q50	Star	5	6	LIMIT
Q9	Star/Tree	5 (+4 R)	4	UNION(1)
Q10	Star/Tree	5 (+4 R)	6	UNION(1), LIMIT
Q20	Tree	7 (+var : 3)	8 (+1 P)	OPTIONAL(1), FILTER(2), LIMIT
Q19	Tree	8 (+var : 3)	8 (+ 1 P)	OPTIONAL(1), FILTER(4)
Q17	Tree	9	5	UNION(2), OPTIONAL(2)

Q18	Tree	9	6	UNION(2), OPTIONAL(2)
Q39	Tree	10 (+2 R)	10	UNION(1), OPTIONAL(4), FILTER(1C)
Q40	Tree/Star	10 (+2 R)	12	UNION(1), OPTIONAL(4), FILTER(1C), LIMIT
Q13	Complex	12 (2 C)	10	OPTIONAL(8)
Q14	Complex	12 (2 C)	11	OPTIONAL(8), LIMIT

Table A.3: DBSPB Queries2012 Characteristics

Id	**Shape**	**Predicates**	**Variables**	**BGP Keywords**
Q1 Q3 Q15 Q27 Q35	Path	1	1	- - FILTER(1) FILTER(1) -
Q2 Q4 Q16 Q28 Q36	Path	1	2	- - FILTER(1), LIMIT FILTER(1), LIMIT LIMIT
Q11	Path	0 (+var : 1)	1 (+1 P)	-
Q33	Complex	0 (+var : 5)	3	FILTER(2C), UNION(1)
Q34	Paths	0 (+var : 5)	4	FILTER(2C), UNION(1), LIMIT
Q12	Path	1	2	FILTER(1), LIMIT
Q31 Q37	Star	2	2	OPTIONAL(1), FILTER(1) -
Q19	Star	2	2	FILTER(1)
Q7	Path/ Complex	2	2	UNION(2)
Q21	Complex	2	2	OPTIONAL(1), FILTER(2)
Q32 Q38	Star	2	3	OPTIONAL(1), FILTER(1), LIMIT LIMIT
Q22	Complex	2	3	OPTIONAL(1), FILTER(2), LIMIT
Q20	Star	2	3	FILTER(1), LIMIT
Q8	Path/ Complex	2	3	UNION(2), LIMIT
Q17	Tree	2 (+1 R)	3	FILTER(1)
Q18	Tree	2 (+1 R)	4	FILTER(1), LIMIT
Q39	Star	3	2	-
Q25 Q40	Star	3	3	FILTER(1) LIMIT
Q29	Paths	3	3	FILTER(1), UNION(2)
Q26	Star	3	4	FILTER(1), LIMIT
Q30	Path	3	4	FILTER(1), UNION(2), LIMIT

Q23	Star	3 (+1 R)	4	OPTIONAL(2), FILTER(2)
Q24	Star	3 (+1 R)	5	OPTIONAL(2), FILTER(2), LIMIT
Q9	Star	4	4	FILTER(1)
Q10	Star	4	5	FILTER(1), LIMIT
Q5	Star	5	4	OPTIONAL(1)
Q6	Star	5	5	OPTIONAL(1), LIMIT
Q13	Path	8 (+1 R)	1	UNION(8)
Q14	Path	8 (+1 R)	2	UNION(8), LIMIT

A.4 DBSPB Query 2012 Decomposing with semantic Clustering

Table A.4: DBSPB Queries2012 Decomposed
(Number of clusters : 10)

Query	*Subqueries*
Q5 SELECT ?var4 ?var8 ?var10 WHERE { ?var5 dbpedia-owl:thumbnail ?var4 . ?var5 rdf:type dbpedia-owl:Person . ?var5 rdfs:label ?v . ?var5 foaf:page ?var8 . OPTIONAL { ?var5 foaf:homepage ?var10 } }	**Q5S1** SELECT * WHERE { ?var5 dbpedia-owl:thumbnail ?var4 . } **Q5S2** SELECT * WHERE { ?var5 rdf:type dbpedia-owl:Person . ?var5 rdfs:label ?v . ?var5 foaf:page ?var8 . } **Q5S3** (*OPTIONAL is run by the master machine*) SELECT * WHERE { ?var5 foaf:homepage ?var10 . }

Q9	**Q9S1**
SELECT ?var8 ?var6 ?var10 ?var4	SELECT *
WHERE { ?var4	WHERE { ?var4 dbpedia2:birthPlace ?v . }
dbpedia2:birthPlace ?v .	**Q9S2**
?var4 dbo:birthDate ?var6 .	SELECT *
?var4 foaf:name ?var8 .	WHERE { ?var4 dbo:birthDate ?var6 .
?var4 dbo:deathDate ?var10	?var4 dbo:deathDate ?var10
FILTER (?var6 < '1900-01-01'^^xsd:date) . }	FILTER (?var6 < '1900-01-01'^^xsd:date) . }
	Q9S3
	SELECT *
	WHERE { ?var4 foaf:name ?var8 . }

Q13 SELECT DISTINCT ?var1 WHERE { { ?v dbpp:writer ?var1 . } UNION { ?v dbpp:executiveProducer ?var1 . } UNION { ?v dbpp:creator ?var1 . } UNION { ?v dbpp:starring ?var1 . } UNION { ?v dbpp:executiveProducer ?var1 . } UNION { ?v dbpp:guest ?var1 . } UNION { ?v dbpp:director ?var1 . } UNION { ?v dbpp:producer ?var1 . } UNION { ?v dbpp:series ?var1 . } }	*The master machine will perform an UNION between subqueries* **Q13S1** SELECT * WHERE { { ?v dbpp:writer ?var1 . } UNION { ?v dbpp:creator ?var1 . } UNION { ?v dbpp:director ?var1 . } UNION { ?v dbpp:producer ?var1 . } **Q13S2** SELECT * WHERE { ?v dbpp:executiveProducer ?var1 . UNION { ?v dbpp:executiveProducer ?var1 . } UNION { ?v dbpp:starring ?var1 . } UNION { ?v dbpp:series ?var1 . } **Q13S3** SELECT * WHERE { ?v dbpp:guest ?var1 . }

Q23 SELECT ?var6 ?var8 ?var10 ?var4 WHERE { ?var4 skos:subject ?v . ?var4 foaf:name ?var6 . OPTIONAL { ?var4 rdfs:comment ?var8 . FILTER (LANG(?var8) = 'en') . } OPTIONAL { ?var4 rdfs:comment ?var10 . FILTER (LANG(?var10) = 'de') . } }	*Two OPTIONAL are run by the master machine* **Q23S1** SELECT * WHERE { ?var4 skos:subject ?v . } **Q23S2** SELECT * WHERE { ?var4 rdfs:comment ?var8 . FILTER (LANG(?var8) = 'en') . } **Q23S3** SELECT * WHERE { ?var4 rdfs:comment ?var10 . FILTER (LANG(?var10) = 'de') . } **Q23S4** SELECT ?var6 ?var4 WHERE { ?var4 foaf:name ?var6 . }
Q25 SELECT distinct ?var4 ?var6 WHERE { ?var3 dbowl:influenced ?v . ?var3 foaf:page ?var4 . ?var3 rdfs:label ?var6 FILTER(lang(?var6)='en') }	**Q25S1** SELECT * WHERE { ?var3 foaf:page ?var4 . ?var3 rdfs:label ?var6 FILTER(lang(?var6)='en') } **Q25S2** SELECT * WHERE { ?var3 dbowl:influenced ?v . }

Q29 SELECT * WHERE { { ?v rdfs:comment ?var0. FILTER (lang(?var0) = 'en') } UNION { ?v foaf:depiction ?var1 } UNION { ?v foaf:homepage ?var2 } }	*UNION will be performed by the master machine* **Q29S1** SELECT * WHERE {{ ?v rdfs:comment ?var0. FILTER (lang(?var0) = 'en') } UNION { ?v foaf:depiction ?var1 } } **Q29S2** SELECT * WHERE { ?v foaf:homepage ?var2 }
Q31 SELECT DISTINCT ?var2 ?var4 WHERE { ?var2 rdf:type ?v OPTIONAL { ?var2 rdfs:label ?var4 . FILTER(lang(?var4) = 'en') . } }	**Q31S1** SELECT DISTINCT ?var2 ?var4 WHERE { ?var2 rdf:type ?v OPTIONAL { ?var2 rdfs:label ?var4 . FILTER(lang(?var4) = 'en') . } }

Q33 SELECT ?var3 ?var4 ?var5 WHERE { { ?v ?var3 ?var4. FILTER ((STR(?var3) ='rdfs:label' && lang(?var4) = 'en') || (STR(?var3) = 'dbo:abstract' && lang(?var4) = 'en') || (STR(?var3) = 'rdfs:comment' && lang(?var4) = 'en') || (STR(?var3) ! = 'dbo:abstract' && STR(?var3) ! = 'rdfs:comment' && STR(?var3) != 'rdfs:label')) } UNION { ?var5 ?var3 ?v FILTER (STR(?var3) = 'dbo:owner' || STR(?var3) = 'dbp:redirect') } }	*The UNION will be performed by the master machine* **Q33S1** SELECT * WHERE { { ?v ?var3 ?var4. FILTER ((STR(?var3) = 'rdfs:label' && lang(?var4) = 'en') || (STR(?var3) = 'rdfs:comment' && lang(?var4) = 'en') || (STR(?var3) != 'dbo:abstract' && STR(?var3) != 'rdfs:comment' && STR(?var3) != 'rdfs:label')) } **Q33S2** SELECT * WHERE { { ?v ?var3 ?var4. FILTER ((STR(?var3) = 'dbo:abstract' && lang(?var4) = 'en') || (STR(?var3) != 'dbo:abstract' && STR(?var3) != 'rdfs:comment' && STR(?var3) != 'rdfs:label')) } **Q33S3** SELECT * WHERE { { ?v ?var3 ?var4. FILTER ((STR(?var3) != 'dbo:abstract' && STR(?var3) != 'rdfs:comment' && STR(?var3) != 'rdfs:label')) } **Q33S4** SELECT * WHERE { ?var5 ?var3 ?v FILTER (STR(?var3) = 'dbo:owner' || STR(?var3) = 'dbp:redirect') }

Q37 SELECT * WHERE { ?var0 <http://www.w3.org/2000/01/rdf-schema#label> %%v%% ; <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?var1 . }	**Q37S1** SELECT * WHERE { ?var0 <http://www.w3.org/2000/01/rdf-schema#label> %%v%% ; <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?var1 . }
Q39 SELECT ?var4 WHERE { ?var2 rdf:type dbpedia-owl:Person . ?var2 rdfs:label %%v%% . ?var2 foaf:page ?var4 . }	**Q39S1** SELECT ?var4 WHERE { ?var2 rdf:type dbpedia-owl:Person . ?var2 rdfs:label %%v%% . ?var2 foaf:page ?var4 . }

A.5 DBSPB Query 2011 Decomposing with semantic Clustering

Table A.5: DBSPB Queries2011 Decomposed
(Number of clusters : 10)

Query	*Subqueries*
Q9 SELECT DISTINCT ?var3 ?var4 ?var5 WHERE { { ?var3 dbpp:series %%var1%% ; foaf:name ?var4 ; rdfs:comment ?var5 ; rdf:type %%var0%% . } UNION { ?var3 dbpp:series ?var8 . ?var8 dbpp:redirect %%var1%% . ?var3 foaf:name ?var4 ; rdfs:comment ?var5 ; rdf:type %%var0%% . } }	*UNION is run by the master machine* **Q9S1** *%%var1%% is replaced by ?var8 in the UNION clause* SELECT * WHERE { ?var3 dbpp:series %%var1%% ; rdfs:comment ?var5 ; rdf:type %%var0%% . } **Q9S2** SELECT * WHERE { ?var3 foaf:name ?var4 } **Q9S3** SELECT * WHERE { ?var8 dbpp:redirect %%var1%% . }

Q11

```
SELECT DISTINCT ?var3 ?var5 ?var7
WHERE { ?var3 rdf:type
<http://dbpedia.org/class/yago/Company108058098>
.
?var3 dbpedia2:numEmployees ?var5
FILTER ( xsd:integer(?var5) >= %%var%% ) .
?var3 foaf:homepage ?var7 . }
```

Q11S1

```
SELECT *
WHERE { ?var3 rdf:type
<http://dbpedia.org/class/yago/Company108058098>
}
```

Q11S2

```
SELECT *
WHERE { ?var3 dbpedia2:numEmployees ?var5.
?var3 foaf:homepage ?var7
FILTER ( xsd:integer(?var5) >= ?v) }
```

Q13 SELECT distinct ?var0 ?var1 ?var2 ?var3 ?var5 ?var6 ?var7 ?var10 WHERE { ?var0 rdfs:comment ?var1. ?var0 foaf:page %%var%% OPTIONAL { ?var0 skos:subject ?var6 } OPTIONAL { ?var0 dbpedia2:industry ?var5 } OPTIONAL { ?var0 dbpedia2:location ?var2 } OPTIONAL{ ?var0 dbpedia2:locationCountry ?var3 } OPTIONAL { ?var0 dbpedia2:locationCity ?var9; dbpedia2:manufacturer ?var0 } OPTIONAL { ?var0 dbpedia2:products ?var11; dbpedia2:model ?var0 } OPTIONAL { ?var0 <http://www.georss.org/georss/point> ?var10 } OPTIONAL { ?var0 rdf:type ?var7 } }	*OPTIONAL is run by the master machine* **Q13S1** SELECT * WHERE { ?var0 rdfs:comment ?var1. ?var0 foaf:page %%var%% OPTIONAL { ?var0 skos:subject ?var6 } OPTIONAL { ?var0 <http://www.georss.org/georss/point>?var10 } OPTIONAL { ?var0 rdf:type ?var7 } } **Q13S2** SELECT * WHERE { ?var0 dbpedia2:industry ?var5 } **Q13S3** SELECT * WHERE { ?var0 dbpedia2:location ?var2 } **Q13S4** SELECT * WHERE { ?var0 dbpedia2:locationCountry ?var3 } **Q13S5** SELECT * WHERE { ?var0 dbpedia2:locationCity ?var9 } **Q13S6** SELECT * WHERE { ?var0 dbpedia2:manufacturer ?var0 } **Q13S7** SELECT * { ?var0 dbpedia2:products ?var11 } **Q13S8** SELECT * WHERE { ?var0 dbpedia2:model ?var0 }

Q15	**Q15S1**
SELECT ?var2 ?var4	SELECT *
WHERE { { ?var2 rdf:type %%var1%%.	WHERE { ?var2 rdf:type ?v }
?var2 dbpedia2:population ?var4.	**Q15S2**
FILTER (xsd:integer(?var4) > %%var0%%) }	SELECT *
UNION { ?var2 rdf:type %%var1%%.	WHERE { ?var2 dbpedia2:population ?var4.
?var2 dbpedia2:populationUrban ?var4.	FILTER (xsd:integer(?var4) > ?v) }
FILTER (xsd:integer(?var4) > %%var0%%) } }	**Q15S3**
	SELECT *
	WHERE { ?var2 dbpedia2:populationUrban ?var4
	FILTER (xsd:integer(?var4) > ?v) }

Q17 SELECT * WHERE { ?var2 a <http://dbpedia.org/ontology/Settlement>; rdfs:label %%var%% . ?var6 a <http://dbpedia.org/ontology/Airport>. { ?var6 <http://dbpedia.org/ontology/city> ?var2 UNION { ?var6 <http://dbpedia.org/ontology/location> ?var2 } { ?var6 <http://dbpedia.org/property/iata> ?var5. } UNION { ?var6 <http://dbpedia.org/ontology/iataLocationIdentifier> ?var5. } OPTIONAL { ?var6 foaf:homepage ?var6_home. } OPTIONAL { ?var6 <http://dbpedia.org/property/nativename> ?var6_name. } }	*OPTIONAL and UNION are run by the master machine* **Q17S1** SELECT * WHERE { ?var2 a <http://dbpedia.org/ontology/Settlement>. ?var6 a <http://dbpedia.org/ontology/Airport> } **Q17S2** SELECT * WHERE { ?var2 rdfs:label ?v } **Q17S3** SELECT * WHERE { ?var6 <http://dbpedia.org/ontology/city> ?var2 UNION { ?var6 <http://dbpedia.org/ontology/location> ?var2 } } **Q17S4** SELECT * WHERE { ?var6 foaf:homepage ?var6_home }

Q17S5
SELECT *
WHERE { ?var6 <http://dbpedia.org/property/iata>
?var5 }
Q17S6
SELECT *
WHERE { ?var6 <http://dbpedia.org/ontology/
iataLocationIdentifier> ?var5 }
Q17S7
SELECT *
WHERE { ?var6 <http://dbpedia.org/property/
nativename> ?var6_name }

Q19 SELECT DISTINCT ?var7 { ?var3 foaf:page ?var7. ?var3 rdf:type <http://dbpedia.org/ontology/SoccerPlayer> . ?var3 dbpedia2:position ?var16 . ?var3 <http://dbpedia.org/property/clubs> ?var8 . ?var8 <http://dbpedia.org/ontology/capacity> ?var9 . ?var3 <http://dbpedia.org/ontology/birthPlace> ?var31 . ?var31 ?var33 ?var34. OPTIONAL { ?var3 <http://dbpedia.org/ontology/number> ?var35. } Filter (?var33 = <http://dbpedia.org/property/populationEstimate> || ?var33 = <http://dbpedia.org/property/populationCensus> || ?var33 = <http://dbpedia.org/property/statPop>) Filter (xsd:integer(?var34) > % %var1%%) . Filter (xsd:integer(?var9) < %%var0%%) . Filter (?var16 = 'Goalkeeper'@en || ?var16 = <http://dbpedia.org/resource/Goalkeeper_%28association_football%29> || ?var16 = <http://dbpedia.org/resource/Goalkeeper_%28football%29>) }	*A UNION is performed by the master machine between Q19S3 and Q19S4* **Q19S1** SELECT * WHERE { ?var3 foaf:page ?var7. ?var3 rdf:type <http://dbpedia.org/ontology/SoccerPlayer> . ?var3 dbpedia2:position ?var16 . ?var8 <http://dbpedia.org/ontology/capacity> ?var9 . ?var3 <http://dbpedia.org/ontology/birthPlace> ?var31 Filter (?var16 = 'Goalkeeper'@en || ?var16 = <http://dbpedia.org/resource/Goalkeeper_%28association_football%29> || ?var16 = <http://dbpedia.org/resource/Goalkeeper_%28football%29>) } **Q19S2** *OPTIONAL is run by the master machine* SELECT * { ?var3 <http://dbpedia.org/ontology/number> ?var35 }

Q19S3
SELECT *
WHERE { ?var3 <http://dbpedia.org/property/clubs>
?var8.
?var31 ?var33 ?var34.
Filter (?var33 = <http://dbpedia.org/property/
populationEstimate>
|| ?var33 = <http://dbpedia.org/property/
populationCensus>)
Filter (xsd:integer(?var34) > ?v) }
Q19S4
SELECT *
WHERE { ?var31 ?var33 ?var34.
Filter (?var33 = <http://dbpedia.org/property/
statPop>)
Filter (xsd:integer(?var34) > ?v) }

Q21 SELECT distinct ?var3 ?var4 ?var2 WHERE { { %%var%% dbpedia2:subsid ?var3 OPTIONAL { ?var2 %%var%% dbpedia2:parent } OPTIONAL { %%var%% dbpedia2:divisions ?var4 } } UNION { ?var2 %%var%% dbpedia2:parent } OPTIONAL { %%var%% dbpedia2:subsid ?var3 } OPTIONAL { %%var%% dbpedia2:divisions ?var4 } } UNION { %%var%% dbpedia2:divisions ?var4 } OPTIONAL { %%var%% dbpedia2:subsid ?var3 } OPTIONAL { ?var2 %%var%% dbpedia2:parent } } }	*OPTIONAL and UNION are run by the master machine* **Q21S1** SELECT * WHERE { %%var%% dbpedia2:subsid ?var3 } **Q21S2** SELECT * WHERE { ?var2 %%var%% dbpedia2:parent } **Q21S3** SELECT * WHERE { %%var%% dbpedia2:divisions ?var4 }
Q23 SELECT DISTINCT ?var5 WHERE { ?var2 rdf:type dbpedia-owl:Person . ?var2 dbpedia-owl:nationality ?var4 . ?var4 rdfs:label ?var5 . ?var2 rdfs:label %%var%% . FILTER (lang(?var5) = 'en') }	**Q23S1** SELECT * WHERE { ?var2 rdf:type dbpedia-owl:Person . ?var4 rdfs:label ?var5 . ?var2 rdfs:label %%var%% . FILTER (lang(?var5) = 'en') } **Q23S2** SELECT * WHERE { ?var2 dbpedia-owl:nationality ?var4 }
Q29 SELECT DISTINCT ?var2 ?var3 WHERE { ?var2 rdf:type %%var%% ; rdfs:label ?var3 . FILTER regex(str(?var3), 'pes', 'i') }	**Q29S1** SELECT DISTINCT ?var2 ?var3 WHERE { ?var2 rdf:type %%var%% ; rdfs:label ?var3 . FILTER regex(str(?var3), 'pes', 'i') }

Q31 SELECT DISTINCT ?var7 ?var4 ?var6 ?var5 WHERE { %%var%% ?var4 ?var5 . OPTIONAL { ?var5 rdfs:label ?var6 } . FILTER(langMatches(lang(?var6),'EN') || (! langMatches(lang(?var6),'*'))) . FILTER(langMatches(lang(?var5),'EN') || (! langMatches(lang(?var5),'*'))) . OPTIONAL { ?var4 rdfs:label ?var7 } }	**Q31S1** *The query Q31 is sent to the source containing the predicate 'label'* SELECT DISTINCT ?var7 ?var4 ?var6 ?var5 WHERE { %%var%% ?var4 ?var5 . OPTIONAL { ?var5 rdfs:label ?var6 } . FILTER(langMatches(lang(?var6),'EN') || (! langMatches(lang(?var6),'*'))) . FILTER(langMatches(lang(?var5),'EN') || (! langMatches(lang(?var5),'*'))) . OPTIONAL { ?var4 rdfs:label ?var7 } } **Q31S2** *The subquery is sent to the rest of sources* SELECT * WHERE { ?v ?var4 ?var5 FILTER(langMatches(lang(?var5),'EN')|| (! langMatches(lang(?var5),'*'))) }

Q33	**Q33S1**
SELECT DISTINCT ?var2 ?var3	SELECT DISTINCT ?var2 ?var3
{ { ?var2 <http://www.w3.org/2004/02/skos/core# subject> %%var%%. }	{ { ?var2 <http://www.w3.org/2004/02/skos/core# subject> %%var%%. }
UNION { ?var2 <http://www.w3.org/2004/02/skos/core# subject> <http://dbpedia.org/resource/Category: Prefectures_in_France>. }	UNION { ?var2 <http://www.w3.org/2004/02/skos/core# subject> <http://dbpedia.org/resource/Category :Prefectures_in_France>. }
UNION { ?var2 <http://www.w3.org/2004/02/skos/core# subject> <http://dbpedia.org/resource/Category:German_ state_capitals>.	UNION { ?var2 <http://www.w3.org/2004/02/skos/core# subject> <http://dbpedia.org/resource/Category :German_ state_capitals>.
?var2 <http://www.w3.org/2000/01/rdf-schema#label> ?var3.	?var2 <http://www.w3.org/2000/01/rdf-schema#label> ?var3.
FILTER (lang(?var3)='fr') }	FILTER (lang(?var3)='fr') }

Q39 SELECT * WHERE { ?var6 a <http://dbpedia.org/ontology/PopulatedPlace>; <http://dbpedia.org/ontology/abstract> ?var1; rdfs:label ?var2; geo:lat ?var3; geo:long ?var4. { ?var6 rdfs:label %%var%%. } UNION { ?var5 <http://dbpedia.org/property/redirect> ?var6; rdfs:label %%var%%. } OPTIONAL { ?var6 foaf:depiction ?var8 } OPTIONAL { ?var6 foaf:homepage ?var10 } OPTIONAL { ?var6 <http://dbpedia.org/ontology/populationTotal> ?var12 } OPTIONAL { ?var6 <http://dbpedia.org/ontology/thumbnail> ?var14 } FILTER (langMatches(lang(?var1), 'de') && langMatches(lang(?var2), 'de')) }	*OPTIONAL is run by the master machine* **Q39S1** SELECT * WHERE { ?var6 a <http://dbpedia.org/ontology/PopulatedPlace> } **Q39S2** SELECT * WHERE { ?var6 <http://dbpedia.org/ontology/abstract> ?var1. FILTER (langMatches(lang(?var1), 'de')) } **Q39S3** SELECT * WHERE { ?var6 geo:lat ?var3. ?var6 geo:long ?var4. } **Q39S4** *?var6 is replaced by ?var5 for the UNION clause* SELECT * WHERE { ?var6 rdfs:label ?var2. FILTER (langMatches(lang(?var2), 'de')) }

	Q39S5 *UNION is run by the master machine* SELECT * WHERE { ?var5 <http://dbpedia.org/property/redirect> ?var6 } **Q39S5** SELECT * WHERE { ?var6 foaf:depiction ?var8 } **Q39S6** SELECT * WHERE { ?var6 foaf:homepage ?var10 } **Q39S7** SELECT * WHERE { ?var6 <http://dbpedia.org/ontology/populationTotal> ?var12 } **Q39S8** SELECT * WHERE { ?var6 <http://dbpedia.org/ontology/thumbnail> ?var14 }
Q43 SELECT ?var2 WHERE { ?var3 <http://xmlns.com/foaf/0.1/homepage> ?var2 . ?var3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> %%var%% . }	**Q43S1** SELECT * WHERE { ?var3 <http://xmlns.com/foaf/0.1/homepage> ?var2 . } **Q43S2** SELECT ?var2 WHERE { ?var3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> %%var%% . }

Q47 SELECT * where { ?var1 a <http://dbpedia.org/ontology/Organisation> . ?var2 <http://dbpedia.org/ontology/foundationPlace> %%var0%% . ?var4 <http://dbpedia.org/ontology/developer> ?var2 . ?var4 a %%var1%% . }	**Q47S1** SELECT * WHERE { ?var1 a <http://dbpedia.org/ontology/Organisation> . ?var4 a %%var1%% . } **Q47S2** SELECT * WHERE { ?var2 <http://dbpedia.org/ontology/foundationPlace> %%var0%% . } **Q47S3** SELECT * WHERE { ?var4 <http://dbpedia.org/ontology/developer> ?var2 . }
Q49 SELECT ?var0 ?var1 ?var2 ?var3 where { ?var6 rdf:type %%var%%. ?var6 dbpprop:name ?var0. ?var6 dbpprop:pages ?var1. ?var6 dbpprop:isbn ?var2. ?var6 dbpprop:author ?var3. }	**Q49S1** SELECT * where { ?var6 rdf:type ?v. ?var6 dbpprop:pages ?var1. } **Q49S2** SELECT * where { ?var6 dbpprop:name ?var0. ?var6 dbpprop:isbn ?var2. } **Q49S3** SELECT * where { ?var6 dbpprop:author ?var3. }

A.6 Semantic relatedness of predicates

Table A.6: Semantic relatedness in Queries2011 set

Queries	**Related Predicates**	
	$\|C\| = 40$	$\|C\| = 10$
Q9	series	series, comment, type
	name	name
	comment, type	redirect
	redirect	
Q11	type, homepage	type
	numEmployees	homepage, numEmployees
Q13	comment, subject, location, type	comment, page, subject, model, point, type
	industry, manufacturer	industry, location, manufacturer
	page	
	locationCountry	locationCountry, locationCity
	locationCity	
	products	products
	model	
	point	
Q15	type	type
	population, populationUrban	population
		populationUrban
Q17	label, location, homepage	city, location, homepage
	iata, nativename	iata, nativename
	iataLocationIdentifier	iataLocationIdentifier
	city	label
Q19	capacity, birthPlace, number	page, type, position, capacity, birthPlace, number
	populationEstimate, populationCensus	clubs, populationEstimate, populationCensus
	statPop	statPop
	page	
	type	
	position	
	clubs	
Q21	subsid	
	divisions	
Q23	type, label	
	nationality	
Q29	type, label	

Q33	subject, label	
Q39	abstract, thumbnail	abstract, homepage,
	label, depiction, homepage	populationTotal, thumbnail
	lat, long	lat, long
	populationTotal	label, depiction
	redirect	redirect
Q43	homepage, type	homepage
		type
Q47	foundationPlace	
	developer	
Q49	name, isbn	name, isbn
	type	type, pages
	pages	author
	author	

Bibliography

[Abbas 2017] Abdullah Abbas. *Static Analysis of Semantic Web Queries with ShEx Schema Constraints*. PhD thesis, Université Grenoble-Alpes, 2017.

[Abbes 2013] Rafik Abbes, Arlind Kopliku, Karen Pinel-Sauvagnat, Nathalie Hernandez and Mohand Boughanem. *Apport du Web et du Web de Données pour la recherche d'attributs*. In Conférence en Recherche d'Information et Applications-CORIA 2013, page 0, 2013.

[Abdelaziz 2017] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat and Panos Kalnis. *A survey and experimental comparison of distributed SPARQL engines for very large RDF data*. Proceedings of the VLDB Endowment, vol. 10, no. 13, pages 2049–2060, 2017.

[Abdelhamid 2016] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat and Fuad Jamour. *Scalemine: Scalable parallel frequent subgraph mining in a single large graph*. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 61. IEEE Press, 2016.

[Achsas 2016] Sanae Achsas and El Habib Nfaoui. *Intelligent layer for relational aggregated search based on deep neural networks*. In Computer Systems and Applications (AICCSA), 2016 IEEE/ACS 13th International Conference of, pages 1–2. IEEE, 2016.

[Achsas 2018] Sanae Achsas and El Habib Nfaoui. *Improving relational aggregated search from big data sources using stacked autoencoders*. Cognitive Systems Research, vol. 51, pages 61–71, 2018.

[Acosta-Mendoza 2012] Niusvel Acosta-Mendoza, Andrés Gago-Alonso and José E. Medina-Pagola. *Frequent approximate subgraphs as features for graph-based image classification*. Knowledge-Based Systems, vol. 27, pages 381–392, March 2012.

[Acosta-Mendoza 2015] Niusvel Acosta-Mendoza, Jesús Ariel Carrasco-Ochoa, Andrés Gago-Alonso, José Fco Martínez-Trinidad and José Eladio Medina-Pagola. *Representative Frequent Approximate Subgraph Mining in Multi-Graph Collections*. 2015.

[Acosta 2011] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo and Edna Ruckhaus. *ANAPSID: an adaptive query processing engine for SPARQL endpoints*. In International Semantic Web Conference, pages 18–34. Springer, 2011.

[Adamchik] Victor Adamchik. *Graph Theory*. `www.cs.cmu.edu/~adamchik/21-127/lectures/graphs_1_print.pdf`.

[Aggarwal 2010] Charu C. Aggarwal, Haixun Wang and others. Managing and mining graph data, volume 40. Springer, 2010.

[Aggarwal 2014] Charu C Aggarwal and Jiawei Han. Frequent pattern mining. Springer, 2014.

[Agrawal 1993] Rakesh Agrawal, Tomasz Imieliński and Arun Swami. *Mining Association Rules Between Sets of Items in Large Databases*. In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD '93, pages 207–216, New York, NY, USA, 1993. ACM.

[Agrawal 1994] Rakesh Agrawal, Ramakrishnan Srikant*et al.* *Fast algorithms for mining association rules*. In Proc. 20th int. conf. very large data bases, VLDB, volume 1215, pages 487–499, 1994.

[Akhter 2018] Adnan Akhter, Axel-Cyrille Ngomo Ngonga and Muhammad Saleem. *An Empirical Evaluation of RDF Graph Partitioning Techniques*. In European Knowledge Acquisition Workshop, pages 3–18. Springer, 2018.

[Al Hasan 2005] Mohammad Al Hasan, Vineet Chaoji, Saeed Salem, Nagender Parimi and Mohammed J Zaki. *DMTL: A Generic Data Mining Template Library*. 2005.

[Al Hasan 2007] M. Al Hasan, V. Chaoji, S. Salem, J. Besson and M. J. Zaki. *ORIGAMI: Mining Representative Orthogonal Graph Patterns*. In Seventh IEEE International Conference on Data Mining (ICDM 2007), pages 153–162, October 2007.

[Al Hasan 2009a] M. Al Hasan and M. Zaki. *Musk: Uniform Sampling of k Maximal Patterns*. In Proceedings of the 2009 SIAM International Conference on Data Mining, Proceedings, pages 650–661. Society for Industrial and Applied Mathematics, 2009.

[Al Hasan 2009b] Mohammad Al Hasan and Mohammed J. Zaki. *Output Space Sampling for Graph Patterns*. Proc. VLDB Endow., vol. 2, no. 1, pages 730–741, August 2009.

[Al Hasan 2010] Mohammad Al Hasan. *Mining Interesting Subgraphs by Output Space Sampling*. SIGKDD Explor. Newsl., vol. 12, no. 1, pages 73–74, November 2010.

[Aluç 2014] Güneş Aluç, Olaf Hartig, M Tamer Özsu and Khuzaima Daudjee. *Diversified stress testing of RDF data management systems*. In International Semantic Web Conference, pages 197–212. Springer, 2014.

[Aridhi 2014] Sabeur Aridhi, Laurent d'Orazio, Mondher Maddouri and Engelbert Mephu Nguifo. *Un partitionnement basé sur la densité de graphe pour approcher la fouille distribuée de sous-graphes fréquents*. Technique et Science Informatiques, vol. 33, no. 9-10, pages 711–737, 2014.

[Aridhi 2015] Sabeur Aridhi, Laurent d'Orazio, Mondher Maddouri and Engelbert Mephu Nguifo. *Density-based data partitioning strategy to approximate large-scale subgraph mining*. Information Systems, vol. 48, pages 213–223, March 2015.

[Arimura 2007] Hiroki Arimura, Takeaki Uno and Shinichi Shimozono. *Time and Space Efficient Discovery of Maximal Geometric Graphs*. In Vincent Corruble, Masayuki Takeda and Einoshin Suzuki, editors, Discovery Science, number 4755 de Lecture Notes in Computer Science, pages 42–55. Springer Berlin Heidelberg, October 2007. DOI: 10.1007/978-3-540-75488-6_6.

[Atemezing 2018] Ghislain Auguste Atemezing and Florence Amardeilh. *Benchmarking Commercial RDF Stores with Publications Office Dataset*. In European Semantic Web Conference, pages 379–394. Springer, 2018.

[Belkin 1992] Nicholas J Belkin and W Bruce Croft. *Information filtering and information retrieval: Two sides of the same coin?* Communications of the ACM, vol. 35, no. 12, pages 29–38, 1992.

[Bizer 2013] Christian Bizer and Andreas Schultz. *Berlin SPARQL Benchmark (BSBM)*. `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/`, 2013. [Online ; accessed 2018-11-06].

[Boncz 2013] Peter Boncz, Minh-Duc Pham, Orri Erling, Ivan Mikhailov and Yrjana Rankka. *Social Network Intelligence BenchMark*. `https://www.w3.org/wiki/Social_Network_Intelligence_BenchMark`, 2013. [Online ; accessed 2018-11-06].

[Borgelt 2002] C. Borgelt and M. R. Berthold. *Mining molecular fragments: finding relevant substructures of molecules*. In 2002 IEEE International Conference on Data Mining, 2002. ICDM 2003. Proceedings, pages 51–58, 2002.

[Borgelt 2016] Christian Borgelt. *MoSS - Molecular Substructure Miner*. `http://www.borgelt.net/moss.html`, 2016. [Online; accessed 2016-05-30].

[Bota 2014] Horatiu Bota, Ke Zhou, Joemon M Jose and Mounia Lalmas. *Composite retrieval of heterogeneous web search*. In Proceedings of the 23rd international conference on World wide web, pages 119–130. ACM, 2014.

[Bota 2015] Horaţiu Bota, Ke Zhou and Joemon J Jose. *Exploring composite retrieval from the users' perspective*. In European Conference on Information Retrieval, pages 13–24. Springer, 2015.

[Brini 2007] Asma-Hedia Brini, Mohand Boughanem and Didier Dubois. *Un modèle de réseau possibiliste pour la recherche d'information*. Information-Interaction-Intelligence, Cépaduès Editions, vol. 7, no. 1, pages 31–54, 2007.

[Broekstra 2002] Jeen Broekstra, Arjohn Kampman and Frank Van Harmelen. *Sesame: A generic architecture for storing and querying rdf and rdf schema*. In International semantic web conference, pages 54–68. Springer, 2002.

[Callan 2002] Jamie Callan. *Distributed information retrieval*. In Advances in information retrieval, pages 127–150. Springer, 2002.

[Chen 2008] Chen Chen, Cindy Xide Lin, Xifeng Yan and Jiawei Han. *On Effective Presentation of Graph Patterns: A Structural Representative Approach*. In Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM '08, pages 299–308, New York, NY, USA, 2008. ACM.

[Chong 2005] Eugene Inseok Chong, Souripriya Das, George Eadon and Jagannathan Srinivasan. *An efficient SQL-based RDF querying scheme*. In Proceedings of the 31st international conference on Very large data bases, pages 1216–1227. VLDB Endowment, 2005.

[Cohen 2004] Moti Cohen and Ehud Gudes. *Diagonally Subgraphs Pattern Mining*. In Proceedings of the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD '04, pages 51–58, New York, NY, USA, 2004. ACM.

[Conrads 2015] Felix Conrads. *IGUANA AN INTEGRATED SUITE FOR BENCHMARKING SPARQL*. `http://iguana-benchmark.eu/download.html`, 2015. [Online ; accessed 2018-11-06].

[Conrads 2017] Felix Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey and Axel-Cyrille Ngonga Ngomo. *IGUANA: A generic framework for benchmarking the read-write performance of triple stores*. In International Semantic Web Conference, pages 48–65. Springer, 2017.

[Cook 1971] Stephen A. Cook. *The Complexity of Theorem-proving Procedures*. In Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[Cook 1994] Diane J. Cook and Lawrence B. Holder. *Substructure Discovery Using Minimum Description Length and Background Knowledge*. J. Artif. Int. Res., vol. 1, no. 1, pages 231–255, 1994.

[Cunha 2015] Danusa RB Cunha and Bernadette Farias Lóscio. *An Approach for Query Decomposition on Federated SPARQL Query Systems*. Journal of Information and Data Management, vol. 6, no. 2, page 106, 2015.

[David 2008] Jérôme David and Jérôme Euzenat. *Comparison between ontology distances (preliminary results)*. In International Semantic Web Conference, pages 245–260. Springer, 2008.

[de Sousa Gomide 2011] Rodrigo de Sousa Gomide, Cristina Dutra de Aguiar Ciferri, Ricardo Rodrigues Ciferri and Marina Teresa Pires Vieira. *ADI-Minebio: A Graph Mining Algorithm for Biomedical Data*. Journal of Information and Data Management, vol. 2, no. 3, page 433, September 2011.

[Dehaspe 1998] Luc Dehaspe, Hannu Toivonen and Ross Donald King. *Finding Frequent Substructures in Chemical Compounds*. pages 30–36. AAAI Press, 1998.

[Desrosiers 2007] C. Desrosiers, Ph Galinier, P. Hansen and A. Hertz. *Sygma: Reducing symmetry in graph mining*. Technical report, Les Cahiers du GERAD, 2007.

[Dinari 2014] Hamed Dinari and Hassan Naderi. *A Survey of Frequent Subgraphs and Subtree Mining Methods*. vol. 14, pages 1694–2108, 05 2014.

[Douar 2014] Brahim Douar, Michel Liquiere, Chiraz Latiri and Yahya Slimani. *LC-mine: a framework for frequent subgraph mining with local consistency techniques*. Knowledge and Information Systems, vol. 44, no. 1, pages 1–25, July 2014.

[Echbarthi 2017] Ghizlane Echbarthi and Hamamache Kheddouchi. *Lasas: an aggregated search based graph matching approach*. In The 29th International Conference on Software Engineering and Knowledge Engineering, 2017.

[El Charif 2006] Rana El Charif. *Analyse des paramètres de pondération dans le cadre de collections volumineuses, rapport de DEA d'Informatique*. 2006.

[Elghazel 2011] Haytham Elghazel and Mohand-Said Hacid. *Aggregated search in graph databases: preliminary results*. In International Workshop on Graph-Based Representations in Pattern Recognition, pages 92–101. Springer, 2011.

[Elseidy 2014] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos and Panos Kalnis. *GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph.* Proc. VLDB Endow., vol. 7, no. 7, pages 517–528, March 2014.

[Erling 2015] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham and Peter Boncz. *The LDBC social network benchmark: Interactive workload.* In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 619–630. ACM, 2015.

[Fei 2010] Hongliang Fei. *Fast Frequent Subgraph Mining (FFSM).* `https://sourceforge.net/projects/ffsm/`, 2010. [Online; accessed 2016-05-30].

[Fletcher 2008] George HL Fletcher. *An algebra for basic graph patterns.* In workshop on Logic in Databases, Rome, Italy, 2008.

[Fouqueré 1994] Christophe Fouqueré. *Représentations graphiques et intelligence artificielle.* 1994.

[Fredkin 1960] Edward Fredkin. *Trie Memory.* Commun. ACM, vol. 3, no. 9, pages 490–499, September 1960.

[Gago-Alonso 2008] Andrés Gago-Alonso, José Eladio Medina Pagola, Jesús Ariel Carrasco-Ochoa and José Fco Martínez-Trinidad. *Mining Frequent Connected Subgraphs Reducing the Number of Candidates.* In Walter Daelemans, Bart Goethals and Katharina Morik, editors, Machine Learning and Knowledge Discovery in Databases, number 5211 de Lecture Notes in Computer Science, pages 365–376. Springer Berlin Heidelberg, September 2008. DOI: 10.1007/978-3-540-87479-9_42.

[Gago-Alonso 2010a] Andrés Gago-Alonso and Jesús A. Carrasco-Ochoa. *Full Duplicate Candidate Pruning for Frequent Connected Subgraph Mining.* Integrated Computer-Aided Engineering, vol. 17, no. 3, pages 211–225, 2010.

[Gago-Alonso 2010b] Andrés Gago-Alonso, Abel Puentes-Luberta, Jesús A. Carrasco-Ochoa, José E. Medina-Pagola and José Fco Martínez-Trinidad. *A New Algorithm for Mining Frequent Connected Subgraphs based on Adjacency Matrices.* Intelligent Data Analysis, vol. 14, no. 3, pages 385–403, August 2010.

[Gao 2012] Z. Gao, L. Shang and Y. Jian. *Frequent subgraph mining based on the automorphism mapping.* In 2012 2nd International Conference on Computer Science and Network Technology (ICCSNT), pages 1518–1522, 2012.

[Gholami 2012] Maryam Gholami and Afshin Salajegheh. *A survey on algorithms of mining frequent subgraphs.* International Journal of Engineering Inventions, vol. 1, no. 5, pages 60–63, 2012.

[Gibbons 1985] Alan Gibbons. Algorithmic Graph Theory. Cambridge University Press, June 1985.

[GooglePress 2007] GooglePress. *Google Begins Move to Universal Search.* `http://googlepress.blogspot.com/2007/05/google-begins-move-to-universal-search_16.html`, 2007. [Online; accessed 2018-11-06].

[Gruber 1995] Thomas R Gruber. *Toward principles for the design of ontologies used for knowledge sharing?* International journal of human-computer studies, vol. 43, no. 5-6, pages 907–928, 1995.

[Gudes 2006] E. Gudes, S. E. Shimony and N. Vanetik. *Discovering Frequent Graph Patterns Using Disjoint Paths.* IEEE Transactions on Knowledge and Data Engineering, vol. 18, no. 11, pages 1441–1456, November 2006.

[Guo 2005] Yuanbo Guo, Zhengxiang Pan and Jeff Heflin. *LUBM: A benchmark for OWL knowledge base systems.* Web Semantics: Science, Services and Agents on the World Wide Web, vol. 3, no. 2-3, pages 158–182, 2005.

[Haase 2010] Peter Haase, Tobias Mathäß and Michael Ziller. *An evaluation of approaches to federated query processing over linked data.* In Proceedings of the 6th International Conference on Semantic Systems, page 5. ACM, 2010.

[Habrant 1999] J Habrant, A Corbel, J J Girardot and J Savoy. *utilisation des réseaux sémantiques pour la navigation dans l'hypertexte.* In Conférence Interfaces Homme Machine, 1999.

[Hammoud 2015] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti and Sherif Sakr. *DREAM: distributed RDF engine with adaptive query planner and minimal communication.* Proceedings of the VLDB Endowment, vol. 8, no. 6, pages 654–665, 2015.

[Han 2000] Jiawei Han, Jian Pei and Yiwen Yin. *Mining frequent patterns without candidate generation.* In ACM sigmod record, volume 29, pages 1–12. ACM, 2000.

[Han 2007] Shuguo Han, Wee Keong Ng and Yang Yu. *FSP: Frequent Substructure Pattern mining.* In 2007 6th International Conference on Information, Communications Signal Processing, pages 1–5, 2007.

[Han 2013] Lushan Han, Abhay L Kashyap, Tim Finin, James Mayfield and Jonathan Weese. *UMBC_EBIQUITY-CORE: semantic textual similarity systems.* In Second Joint Conference on Lexical and Computational Semantics (* SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity, volume 1, pages 44–52, 2013.

[Harispe 2013] Sébastien Harispe, Sylvie Ranwez, Stefan Janaqi and Jacky Montmain. *Semantic measures for the comparison of units of language, concepts or instances from text and knowledge base analysis.* arXiv preprint arXiv:1310.1285, 2013.

[Hasan 2016] Aisha Hasan, Mohammad Hammoud, Reza Nouri and Sherif Sakr. *DREAM in action: a distributed and adaptive RDF system on the cloud.* In Proceedings of the 25th International Conference Companion on World Wide Web, pages 191–194. International World Wide Web Conferences Steering Committee, 2016.

[Henderson 2014] Tim Henderson. *ParSeMiS.* `https://github.com/timtadh/parsemis`, 2014. [Online; accessed 2016-05-30].

[Hong 2003] Mingsheng Hong, Haofeng Zhou, Wei Wang and Baile Shi. *An Efficient Algorithm of Frequent Connected Subgraph Extraction.* In Kyu-Young

Whang, Jongwoo Jeon, Kyuseok Shim and Jaideep Srivastava, editors, Advances in Knowledge Discovery and Data Mining, number 2637 de Lecture Notes in Computer Science, pages 40–51. Springer Berlin Heidelberg, April 2003. DOI: 10.1007/3-540-36175-8_5.

[Horrocks 2017] Ian Horrocks and Boris Motik. *Query answering in distributed RDF databases*. PhD thesis, University of Oxford, 2017.

[Hsu 2016] Jing Ouyang Hsu, Hye-young Paik, Liming Zhan and Anne HH Ngu. *Aggregated Search over Personal Process Description Graph*. In International Conference on Database and Expert Systems Applications, pages 254–262. Springer, 2016.

[Hu 2005] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han and Xianghong Jasmine Zhou. *Mining coherent dense subgraphs across massive biological networks for functional discovery*. Bioinformatics, vol. 21, no. suppl 1, pages i213–i221, June 2005.

[Huan 2003] Jun Huan, Wei Wang and Jan Prins. *Efficient mining of frequent subgraphs in the presence of isomorphism*. In Data Mining, 2003. ICDM 2003. Third IEEE International Conference on, pages 549–552. IEEE, 2003.

[Huan 2004] Jun Huan, Wei Wang, Jan Prins and Jiong Yang. *SPIN: Mining Maximal Frequent Subgraphs from Graph Databases*. In Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '04, pages 581–586, New York, NY, USA, 2004. ACM.

[Ingwersen 1996] Peter Ingwersen. *Cognitive perspectives of information retrieval interaction: elements of a cognitive IR theory*. Journal of documentation, vol. 52, no. 1, pages 3–50, 1996.

[Inokuchi 2000] Akihiro Inokuchi, Takashi Washio and Hiroshi Motoda. *An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data*. pages 13–23, 2000.

[Inokuchi 2002] Akihiro Inokuchi, Takshi Washio, Kunio Nishimura and Hiroshi Motoda. *A Fast Algorithm for Mining Frequent Connected Subgraphs*. Technical report, IBM, 2002.

[Inokuchi 2003] Akihiro Inokuchi, Takashi Washio and Hiroshi Motoda. *Complete Mining of Frequent Patterns from Graphs: Mining Graph Data*. Machine Learning, vol. 50, no. 3, pages 321–354, March 2003.

[Inokuchi 2005] Akihiro Inokuchi, Takashi Washio and Hiroshi Motoda. *A general framework for mining frequent subgraphs from labeled graphs*. Fundamenta Informaticae, vol. 66, no. 1-2, pages 53–82, 2005.

[Inokuchi 2014] A. Inokuchi. *AcGM. Kwansei Gakuin University*. `http://ist.ksc.kwansei.ac.jp/~inokuchi/acgm.zip`, 2014. [Online; accessed 2016-05-30].

[Isom 2015] Jed Isom. *Simple Data Mining*. `http://simpledatamining.blogspot.com/2015/03/graph-pattern-mining-gspan-introduction.html`, 2015. [Online ; accessed 2016-05-19].

[Jia 2011] Yi Jia, Jintao Zhang and Jun Huan. *An efficient graph-mining method for complicated and noisy data with real-world applications*. Knowledge and Information Systems, vol. 28, no. 2, pages 423–447, February 2011.

[Jiang 2013] Chuntao Jiang, Frans Coenen and Michele Zito. *A survey of frequent subgraph mining algorithms*. The Knowledge Engineering Review, vol. 28, no. 01, pages 75–105, March 2013.

[Jin 2005] R. Jin, C. Wang, D. Polshakov, S. Parthasarathy and G. Agrawal. *Discovering Frequent Topological Structures from Graph Datasets*. In Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05, pages 606–611, New York, NY, USA, 2005. ACM.

[Karypis 2003] George Karypis. *PAFI Software Package for Finding Frequent Patterns in Diverse Datasets. Karypis Lab.* `http://glaros.dtc.umn.edu/gkhome/project/dm/software?q=pafi/overview`, 2003. [Online; accessed 2016-05-30].

[Ke 2008] Yiping Ke and James Cheng. *Efficient Correlation Search from Graph Databases*. IEEE Trans. Knowl. Data Eng., vol. 20, no. 12, pages 1601–1615, 2008.

[Kent 1955] Allen Kent, Madeline M Berry, Fred U Luehrs Jr and James W Perry. *Machine literature searching VIII. Operational criteria for designing information retrieval systems*. American documentation, vol. 6, no. 2, pages 93–101, 1955.

[Keyvanpour 2012] Mohammad Reza Keyvanpour and Fereshteh Azizani. *Classification and Analysis of Frequent Subgraphs Mining Algorithms*. ResearchGate, vol. 7, no. 1, pages 220–227, January 2012.

[Kim 2015] Kisung Kim, Bongki Moon and Hyoung-Joo Kim. *R3F: RDF triple filtering method for efficient SPARQL query processing*. World Wide Web, vol. 18, no. 2, pages 317–357, 2015.

[Kopliku 2011] Arlind Kopliku. *Approaches to implement and evaluate aggregated search*. PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2011.

[Kopliku 2014] Arlind Kopliku, Karen Pinel-Sauvagnat and Mohand Boughanem. *Aggregated search: A new information retrieval paradigm*. ACM Computing Surveys (CSUR), vol. 46, no. 3, page 41, 2014.

[Kramer 2001] Stefan Kramer, Luc De Raedt and Christoph Helma. *Molecular Feature Mining in HIV Data*. In Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, pages 136–143, New York, NY, USA, 2001. ACM.

[Krichen 2012] Ines Krichen, Arlind Kopliku, Karen Pinel-Sauvagnat and Mohand Boughanem. *Une approche de recherche d'attributs pertinents pour l'agrégation d'information*. Document numérique, vol. 15, no. 1, pages 9–32, 2012.

[Krishna 2011] Varun Krishna, NNR Ranga Suri and G Athithan. *A comparative survey of algorithms for frequent subgraph discovery.* CURRENT SCIENCE, vol. 100, no. 2, page 190, 2011.

[Kuramochi 2001] Michihiro Kuramochi and George Karypis. *Frequent Subgraph Discovery.* In Proceedings of the 2001 IEEE International Conference on Data Mining, ICDM '01, pages 313–320, Washington, DC, USA, 2001. IEEE Computer Society.

[Kuramochi 2002] M. Kuramochi and G. Karypis. *Discovering frequent geometric subgraphs.* In 2002 IEEE International Conference on Data Mining, 2002. ICDM 2003. Proceedings, pages 258–265, 2002.

[Kuramochi 2005] Michihiro Kuramochi and George Karypis. *Finding Frequent Patterns in a Large Sparse Graph.* Data Min. Knowl. Discov., vol. 11, no. 3, pages 243–271, November 2005.

[Lakshmi 2012] K Lakshmi*et al. Frequent Subgraph Mining Algorithms–A Survey And Framework For Classification.* 2012.

[Lalmas 2011] Mounia Lalmas. *Aggregated Search.* In Massimo Melucci and Ricardo Baeza-Yates, editors, Advanced Topics in Information Retrieval, number 33 de The Information Retrieval Series, pages 109–123. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-20946-8_5.

[Le 2012] Thanh-Huy Le, Haytham Elghazel and Mohand-Saíd Hacid. *A Relational-based Approach for Aggregated Search in Graph Databases.* In Proceedings of the 17th International Conference on Database Systems for Advanced Applications - Volume Part I, DASFAA'12, pages 33–47, Berlin, Heidelberg, 2012. Springer-Verlag.

[Leal 2013] José Paulo Leal. *Using proximity to compute semantic relatedness in rdf graphs.* Computer Science and Information Systems, vol. 10, no. 4, pages 1727–1746, 2013.

[Lee 2012] Gangin Lee and Unil Yun. An efficient approach for mining frequent subgraphs with support affinities, pages 525–532. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Lee 2013] Kisung Lee and Ling Liu. *Scaling queries over big RDF graphs with semantic hash partitioning.* Proceedings of the VLDB Endowment, vol. 6, no. 14, pages 1894–1905, 2013.

[Leymann 2008] Frank Leymann and Dipl-Inf Zhilei Ma. *An Graph-based Approach for Querying Structural Information of Business Processes.* 2008.

[Li 2009] Y. Li, Q. Lin, G. Zhong, D. Duan, Y. Jin and W. Bi. *A Directed Labeled Graph Frequent Pattern Mining Algorithm Based on Minimum Code.* In Third International Conference on Multimedia and Ubiquitous Engineering, 2009. MUE '09, pages 353–359, June 2009.

[Li 2010] Yuhua Li, Quan Lin, Ruixuan Li and Dongsheng Duan. *TGP: mining top-K frequent closed graph pattern without minimum support.* In International Conference on Advanced Data Mining and Applications, pages 537–548. Springer, 2010.

[Lin 2014] Wenqing Lin, Xiaokui Xiao and Gabriel Ghinita. *Large-scale frequent subgraph mining in MapReduce*. In 2014 IEEE 30th International Conference on Data Engineering, pages 844–855. IEEE, 2014.

[Luo] Yuan Luo. *An Efficient Duplicate Removal Algorithm for Frequent Subgraph Mining*.

[Manning 2008] C. Manning, P. Raghavan and H. Schütze. *Introduction to information retrieval*, 2008.

[McKay 1981] Brendan D McKay*et al.* Practical graph isomorphism. Department of Computer Science, Vanderbilt University Tennessee, US, 1981.

[Meadow 1992] Charles T Meadow, Bert R Boyce and Donald H Kraft. Text information retrieval systems, volume 20. Academic Press San Diego, CA, 1992.

[Meinl 2007] Thorsten Meinl, Marc Wörlein, Olga Urzova, Ingrid Fischer and Michael Philippsen. *The ParMol Package for Frequent Subgraph Mining*. Electronic Communications of the EASST, vol. 1, no. 0, July 2007.

[Mihalcea 2006] Rada Mihalcea, Courtney Corley, Carlo Strapparava*et al. Corpus-based and knowledge-based measures of text semantic similarity*. In AAAI, volume 6, pages 775–780, 2006.

[Miller 1995] George A Miller. *WordNet: a lexical database for English*. Communications of the ACM, vol. 38, no. 11, pages 39–41, 1995.

[Montoya 2012] Gabriela Montoya, Maria-Esther Vidal and Maribel Acosta. *DEFENDER: a DEcomposer for quEries against feDERations of endpoints*. In Extended Semantic Web Conference, pages 480–484. Springer, 2012.

[Moreau 2006] Fabienne Moreau. *Revisiter le couplage traitement automatique des langues et recherche d'information*. Theses, Université Rennes 1, December 2006.

[Morsey 2011] Mohamed Morsey, Jens Lehmann, Sören Auer and Axel-Cyrille Ngonga Ngomo. *DBpedia SPARQL benchmark–performance assessment with real queries on real data*. In International Semantic Web Conference, pages 454–469. Springer, 2011.

[Murdock 2008] Vanessa Murdock and Mounia Lalmas. *Workshop on aggregated search*. In ACM SIGIR Forum, volume 42, pages 80–83. ACM, 2008.

[Nadimi-Shahraki 2015] Mohammad H Nadimi-Shahraki, Maryam Taki and Mohammad Naderi. *IDFP-TREE: An Efficient Tree for interactive mining of frequent subgraph patterns.* Journal of Theoretical and Applied Information Technology, vol. 74, no. 3, 2015.

[Nassopoulos 2016] Georges Nassopoulos, Patricia Serrano-Alvarado, Pascal Molli and Emmanuel Desmontils. *FETA: Federated QuEry TrAcking for Linked Data*. In International Conference on Database and Expert Systems Applications, pages 303–312. Springer, 2016.

[Nassr 2002] Nawel Nassr. *Croisement de langues en recherche d'information: traduction et désambiguïsation de requêtes*. PhD thesis, Toulouse 3, 2002.

[Nguyen 2004] Phu Chien Nguyen, Takashi Washio, Kouzou Ohara and Hiroshi Motoda. *Using a Hash-Based Method for Apriori-Based Graph Mining*. In Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti and Dino Pedreschi, editors, Knowledge Discovery in Databases: PKDD 2004, number 3202 de Lecture Notes in Computer Science, pages 349–361. Springer Berlin Heidelberg, September 2004. DOI: 10.1007/978-3-540-30116-5_33.

[Nicklaus 2000] M. C. Nicklaus. *Downloadable Structure Files of NCI Open Database Compounds. National Cancer Institute*. `https://cactus.nci.nih.gov/download/nci/index.html`, 2000. [Online; accessed 2016-05-30].

[Nijssen 2001] Siegfried Nijssen and Joost Kok. *Faster association rules for multiple relations*. In In International Joint Conference on Artificial Intelligence, pages 891–896. Morgan Kaufmann, 2001.

[Nijssen 2003] S. Nijssen. *Performance Comparison of Graph Mining Algorithms on PTE*. `http://liacs.leidenuniv.nl/~nijssensgr/farmer/results.html`, 2003. [Online; accessed 2016-05-30].

[Nijssen 2004] Siegfried Nijssen and Joost N. Kok. *A Quickstart in Frequent Structure Mining Can Make a Difference*. In Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '04, pages 647–652, New York, NY, USA, 2004. ACM.

[Nijssen 2005a] S. Nijssen. *GASTON - Download*. `http://liacs.leidenuniv.nl/~nijssensgr/gaston/download.html`, 2005. [Online; accessed 2016-05-30].

[Nijssen 2005b] Siegfried Nijssen and Joost N. Kok. *The Gaston Tool for Frequent Subgraph Mining*. Electronic Notes in Theoretical Computer Science, vol. 127, no. 1, pages 77–87, March 2005.

[Nijssen 2006] Siegfried Nijssen and Joost N. Kok. *Frequent subgraph miners: Runtime dont say everything*. In Proceedings of the International Workshop on Mining and Learning with Graphs (MLG 2006, pages 173–180, 2006.

[Nowozin 2008] S. Nowozin and K. Tsuda. *Frequent Subgraph Retrieval in Geometric Graph Databases*. In 2008 Eighth IEEE International Conference on Data Mining, pages 953–958, December 2008.

[Nowozin 2013] Sebastian Nowozin. *gboost Graph Boosting Toolbox for Matlab*. `https://github.com/aspick/gboost-0.1.1`, 2013. [Online; accessed 2016-05-30].

[Oguz 2015] Damla Oguz, Belgin Ergenc, Shaoyi Yin, Oguz Dikenelli and Abdelkader Hameurlain. *Federated query processing on linked data: a qualitative survey and open challenges*. The Knowledge Engineering Review, vol. 30, no. 5, pages 545–563, 2015.

[Patel 2013] Harsh J Patel, Rakesh Prajapati, Mahesh Panchal and M Patel. *A Survey of Graph Pattern Mining Algorithm and Techniques*. International Journal of Application or Innovation in Engineering & Management, vol. 2, no. 1, pages 125–129, 2013.

[Peng 2016] Peng Peng, Lei Zou, M Tamer Özsu, Lei Chen and Dongyan Zhao. *Processing SPARQL queries over distributed RDF graphs*. The VLDB Journal - The International Journal on Very Large Data Bases, vol. 25, no. 2, pages 243–268, 2016.

[Philippsen 2011] Michael Philippsen. *ParSeMiS - the Parallel and Sequential Mining Suite.* `https://www2.cs.fau.de/EN/research/zold/ParSeMiS/index.html`, 2011. [Online; accessed 2016-05-30].

[Pilehvar 2013] Mohammad Taher Pilehvar, David Jurgens and Roberto Navigli. *Align, disambiguate and walk: A unified approach for measuring semantic similarity.* In Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), volume 1, pages 1341–1351, 2013.

[Przyjaciel-Zablocki 2011] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung and Georg Lausen. *Rdfpath: Path query processing on large rdf graphs with mapreduce.* In Extended Semantic Web Conference, pages 50–64. Springer, 2011.

[Przyjaciel-Zablocki 2013] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Daniel Hornung and Georg Lausen. Towards a sparql 1.1 feature benchmark on real-world social network data. RWTH, 2013.

[Quilitz 2008] Bastian Quilitz and Ulf Leser. *Querying distributed RDF data sources with SPARQL.* In European Semantic Web Conference, pages 524–538. Springer, 2008.

[Quillan 1966] M Ross Quillan. *Semantic memory.* Technical report, BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA, 1966.

[Rakhmawati 2013] Nur Aini Rakhmawati, Jürgen Umbrich, Marcel Karnstedt, Ali Hasnain and Michael Hausenblas. *A comparison of federation over SPARQL endpoints frameworks.* In International Conference on Knowledge Engineering and the Semantic Web, pages 132–146. Springer, 2013.

[Rakhmawati 2017] Nur Aini Rakhmawati. *Evaluating and benchmarking the performance of federated SPARQL endpoints and their partitioning using selected metrics and specific query types.* PhD thesis, 2017.

[Ramraj 2015] T. Ramraj and R. Prabhakar. *Frequent Subgraph Mining Algorithms – A Survey.* Procedia Computer Science, vol. 47, pages 197 – 204, 2015. Graph Algorithms, High Performance Implementations and Its Applications (ICGHIA 2014).

[Ranu 2009] S. Ranu and A. K. Singh. *GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases.* In 2009 IEEE 25th International Conference on Data Engineering, pages 844–855, March 2009.

[Rastier 2005] François Rastier. *Enjeux épistémologiques de la linguistique de corpus.* La linguistique de corpus, pages 31–45, 2005.

[Ray 2014] Abhik Ray, Lawrence B. Holder and Sutanay Choudhury. *Frequent Subgraph Discovery in Large Attributed Streaming Graphs.* In Proceedings of the 3rd International Conference on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications - Volume 36, BIGMINE'14, pages 166–181. JMLR.org, 2014.

[Rehman 2014] Saif Ur Rehman, Sohail Asghar, Yan Zhuang and Simon Fong. *Performance Evaluation of Frequent Subgraph Discovery Techniques.* Mathematical Problems in Engineering, Mathematical Problems in Engineering, vol. 2014, 2014, page e869198, August 2014.

[Resnik 1999] Philip Resnik. *Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language*. Journal of artificial intelligence research, vol. 11, pages 95–130, 1999.

[Ribeiro 2010] Pedro Ribeiro and Fernando Silva. *G-tries: an efficient data structure for discovering network motifs*. In Proceedings of the 2010 ACM Symposium on Applied Computing, pages 1559–1566. ACM, 2010.

[Roussey 2011] Catherine Roussey, Francois Pinet, Myoung Ah Kang and Oscar Corcho. *An introduction to ontologies and ontology engineering*. In Ontologies in urban development projects, pages 9–38. Springer, 2011.

[Saha 2014] Tanay Kumar Saha and Mohammad Al Hasan. *FS^3: A Sampling based method for top-k Frequent Subgraph Mining*. arXiv:1409.1152 [cs], September 2014. arXiv: 1409.1152.

[Sajana 2016] T Sajana, CM Sheela Rani and KV Narayana. *A survey on clustering techniques for big data mining*. Indian Journal of Science and Technology, vol. 9, no. 3, 2016.

[Saleem 2015] Muhammad Saleem, Qaiser Mehmood and Axel-Cyrille Ngonga Ngomo. *Feasible: A feature-based sparql benchmark generation framework*. In International Semantic Web Conference, pages 52–69. Springer, 2015.

[Saleem 2018] Muhammad Saleem, Ali Hasnain and Axel-Cyrille Ngonga Ngomo. *Largerdfbench: a billion triples benchmark for sparql endpoint federation*. Journal of Web Semantics, vol. 48, pages 85–125, 2018.

[Salton 1986] Gerard Salton and Michael J McGill. *Introduction to modern information retrieval*. 1986.

[Šarić 2012] Frane Šarić, Goran Glavaš, Mladen Karan, Jan Šnajder and Bojana Dalbelo Bašić. *Takelab: Systems for measuring semantic text similarity*. In Proceedings of the First Joint Conference on Lexical and Computational Semantics-Volume 1: Proceedings of the main conference and the shared task, and Volume 2: Proceedings of the Sixth International Workshop on Semantic Evaluation, pages 441–448. Association for Computational Linguistics, 2012.

[Sauvagnat 2005] Karen Sauvagnat. *Modele flexible pour la Recherche d'Information dans des corpus de documents semi-structur´ es*. PhD thesis, Université Paul Sabatier-Toulouse III, 2005.

[Schätzle 2016a] Alexander Schätzle. *Distributed RDF querying on hadoop*. PhD thesis, PhD thesis, University of Freiburg, 2016.

[Schätzle 2016b] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic and Georg Lausen. *S2RDF: RDF querying with SPARQL on spark*. Proceedings of the VLDB Endowment, vol. 9, no. 10, pages 804–815, 2016.

[Schmidt 2008] Michael Schmidt, Thomas Hornung, Georg Lausen and Christoph Pinkel. *SP2Bench: A SPARQL Performance Benchmark*. CoRR, vol. abs/0806.4627, 2008.

[Schmidt 2011] Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte and Thanh Tran. *Fedbench: A benchmark suite for federated semantic data query processing.* In International Semantic Web Conference, pages 585–600. Springer, 2011.

[Schwarte 2011] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel and Michael Schmidt. *FedX: a federation layer for distributed query processing on linked open data.* In Extended Semantic Web Conference, pages 481–486. Springer, 2011.

[Skonieczny 2009] ukasz Skonieczny. *Mining for Unconnected Frequent Graphs with Direct Subgraph Isomorphism Tests.* In Krzysztof A. Cyran, Stanisław Kozielski, James F. Peters, Urszula Stańczyk and Alicja Wakulicz-Deja, editors, Man-Machine Interactions, number 59 de Advances in Intelligent and Soft Computing, pages 523–531. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-00563-3_55.

[Sushmita 2012] Shanu Sushmita. *Study of result presentation and interaction for aggregated search.* PhD thesis, University of Glasgow, 2012.

[Taelman 2018] Ruben Taelman, Joachim Van Herwegen, Miel Vander Sande and Ruben Verborgh. *Comunica: a modular SPARQL query engine for the web.* In International Semantic Web Conference, pages 239–255. Springer, 2018.

[Takigawa 2010] Ichigaku Takigawa and Hiroshi Mamitsuka. *Efficiently mining δ-tolerance closed frequent subgraphs.* Machine Learning, vol. 82, no. 2, pages 95–121, September 2010.

[Termier 2007] Alexandre Termier, Yoshinori Tamada, Kazuyuki Numata, Seiya Imoto, Takashi Washio and Tomoyuki Higuchi. *DIGDAG, a First Algorithm to Mine Closed Frequent Embedded Sub-DAGs.* In Mining and Learning with Graphs, MLG 2007, Firence, Italy, August 1-3, 2007, Proceedings, 2007.

[Thoma 2010] Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans-Peter Kriegel, Alex Smola, Le Song, Philip S. Yu, Xifeng Yan and Karsten M. Borgwardt. *Discriminative Frequent Subgraph Mining with Optimality Guarantees.* `http://www.dbs.ifi.lmu.de/cms/Publications/Discriminative_Frequent_Subgraph_Mining_with_Optimality_Guarantees`, 2010. [Online; accessed 2016-05-30].

[Thomas 2010] Lini T Thomas, Satyanarayana R Valluri and Kamalakar Karlapalem. *Margin: Maximal frequent subgraph mining.* ACM Transactions on Knowledge Discovery from Data (TKDD), vol. 4, no. 3, page 10, 2010.

[TopQuadrant 2013] TopQuadrant. *Controlled vocabularies, taxonomies, and thesauruses (and ontologies).* Technical report, TopQuadrant, 2013.

[Ullmann 1976] J. R. Ullmann. *An Algorithm for Subgraph Isomorphism.* J. ACM, vol. 23, no. 1, pages 31–42, January 1976.

[Unger 2014] Christina Unger, Corina Forascu, Vanessa Lopez, Axel-Cyrille Ngonga Ngomo, Elena Cabrio, Philipp Cimiano and Sebastian Walter. *Question answering over linked data (QALD-4).* In Working Notes for CLEF 2014 Conference, 2014.

[Van Herwegen 2015] Joachim Van Herwegen, Ruben Verborgh, Erik Mannens and Rik Van de Walle. *Query execution optimization for clients of triple pattern fragments*. In European Semantic Web Conference, pages 302–318. Springer, 2015.

[Van Rijsbergen 1979] CJ Van Rijsbergen. *Information retrieval. dept. of computer science, university of glasgow.* URL: citeseer. ist. psu. edu/vanrijsbergen79information. html, vol. 14, 1979.

[Vanetik 2002] Natalia Vanetik, Ehud Gudes and Solomon Eyal Shimony. *Computing frequent graph patterns from semistructured data*. In Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on, pages 458–465. IEEE, 2002.

[Vijayalakshmi 2011] Nadarajan Vijayalakshmi. *FP-GraphMiner - A Fast Frequent Pattern Mining Algorithm for Network Graphs*. Journal of Graph Algorithms and Applications, vol. 15, no. 6, pages 753–776, 2011.

[W3C 2013] W3C. *SPARQL Query Language for RDF*. `https://www.w3.org/TR/rdf-sparql-query/#convertGraphPattern`, 2013. [Online; accessed 2018-11-06].

[W3C 2018] W3C. *RDF Store Benchmarking*. `https://www.w3.org/wiki/RdfStoreBenchmarking`, 2018. [Online ; accessed 2018-11-06].

[Wang 2004] Chen Wang, Wei Wang, Jian Pei, Yongtai Zhu and Baile Shi. *Scalable Mining of Large Disk-based Graph Databases*. In Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '04, pages 316–325, New York, NY, USA, 2004. ACM.

[Wang 2005] Wei Wang, Chen Wang, Yongtai Zhu, Baile Shi, Jian Pei, Xifeng Yan and Jiawei Han. *GraphMiner: A Structural Pattern-mining System for Large Disk-based Graph Databases and Its Applications*. In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD '05, pages 879–881, New York, NY, USA, 2005. ACM.

[Wang 2006] Jianyong Wang, Zhiping Zeng and Lizhu Zhou. *CLAN: An Algorithm for Mining Closed Cliques from Large Dense Graph Databases*. In 22nd International Conference on Data Engineering (ICDE'06), pages 73–73, April 2006.

[Washio 2003] Takashi Washio and Hiroshi Motoda. *State of the art of graph-based data mining*. Acm Sigkdd Explorations Newsletter, vol. 5, no. 1, pages 59–68, 2003.

[Wilkinson 2003] Kevin Wilkinson, Craig Sayers, Harumi Kuno and Dave Reynolds. *Efficient RDF storage and retrieval in Jena2*. In Proceedings of the First International Conference on Semantic Web and Databases, pages 120–139. Citeseer, 2003.

[Wong 2015] Ka-Chun Wong. *A short survey on data clustering algorithms*. In Soft Computing and Machine Intelligence (ISCMI), 2015 Second International Conference on, pages 64–68. IEEE, 2015.

[Wörlein 2005] Marc Wörlein, Thorsten Meinl, Ingrid Fischer and Michael Philippsen. *A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston.* In Alípio Mário Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho and João Gama, editors, Knowledge Discovery in Databases: PKDD 2005, number 3721 de Lecture Notes in Computer Science, pages 392–403. Springer Berlin Heidelberg, October 2005. DOI: 10.1007/11564126_39.

[Wu 2008] J. Wu and L. Chen. *A Fast Frequent Subgraph Mining Algorithm.* In Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for, pages 82–87, November 2008.

[Wu 2014a] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Hai Jin and Ling Liu. *Semstore: A semantic-preserving distributed rdf triple store.* In Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, pages 509–518. ACM, 2014.

[Wu 2014b] Hongyan Wu, Toyofumi Fujiwara, Yasunori Yamamoto, Jerven Bolleman and Atsuko Yamaguchi. *BioBenchmark Toyama 2012: an evaluation of the performance of triple stores on biological data.* Journal of biomedical semantics, vol. 5, no. 1, page 32, 2014.

[Wylot 2018] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux and Sherif Sakr. *RDF Data Storage and Query Processing Schemes: A Survey.* ACM Computing Surveys (CSUR), vol. 51, no. 4, page 84, 2018.

[Yan 2002a] Xifeng Yan and Jiawei Han. *gSpan : Graph-Based Substructure Pattern Mining.* Technical report, UIUC, UIUCDCS-R-2002-2296, 2002.

[Yan 2002b] Xifeng Yan and Jiawei Han. *gSpan: graph-based substructure pattern mining.* In 2002 IEEE International Conference on Data Mining, 2002. ICDM 2003. Proceedings, pages 721–724, 2002.

[Yan 2003] Xifeng Yan and Jiawei Han. *CloseGraph: Mining Closed Frequent Graph Patterns.* In Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '03, pages 286–295, New York, NY, USA, 2003. ACM.

[Yan 2004] Xifeng Yan, Philip S. Yu and Jiawei Han. *Graph Indexing: A Frequent Structure-based Approach.* In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04, pages 335–346, New York, NY, USA, 2004. ACM.

[Yan 2005] Xifeng Yan, X. Jasmine Zhou and Jiawei Han. *Mining Closed Relational Graphs with Connectivity Constraints.* In Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05, pages 324–333, New York, NY, USA, 2005. ACM.

[Yan 2008] Xifeng Yan, Hong Cheng, Jiawei Han and Philip S. Yu. *Mining Significant Graph Patterns by Leap Search.* In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, pages 433–444, New York, NY, USA, 2008. ACM.

[Yan 2009] Xifeng Yan. *SOFTWARE - gSpan: Frequent Graph Mining Package.* `http://www.cs.ucsb.edu/~xyan/software/gSpan.htm`, 2009. [Online; accessed 2016-05-30].

[Yang 2016] Shengqi Yang, Fangqiu Han, Yinghui Wu and Xifeng Yan. *Fast top-k search in knowledge graphs*. In Data Engineering (ICDE), 2016 IEEE 32nd International Conference on, pages 990–1001. IEEE, 2016.

[Yasin 2018] Muhammad Qasim Yasin, Xiaowang Zhang, Rafiul Haq, Zhiyong Feng and Sofonias Yitagesu. *A Comprehensive Study for Essentiality of Graph Based Distributed SPARQL Query Processing*. In International Conference on Database Systems for Advanced Applications, pages 156–170. Springer, 2018.

[Zaki 2008] Mohammed J. Zaki. *Data Mining Template Library (DMTL)*. `http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software/Software`, 2008. [Online; accessed 2016-05-30].

[Zargayouna 2005] Haïfa Zargayouna. *Indexation sémantique de documents XML*. PhD thesis, Paris 11, 2005.

[Zeng 2006] Zhiping Zeng, Jianyong Wang, Lizhu Zhou and George Karypis. *Coherent Closed Quasi-clique Discovery from Large Dense Graph Databases*. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06, pages 797–802, New York, NY, USA, 2006. ACM.

[Zeng 2009] Zhiping Zeng, Jianyong Wang, Jun Zhang and Lizhu Zhou. *FOGGER: An Algorithm for Graph Generator Discovery*. In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '09, pages 517–528, New York, NY, USA, 2009. ACM.

[Zeng 2013] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao and Zhongyuan Wang. *A distributed graph engine for web scale RDF data*. In Proceedings of the VLDB Endowment, volume 6, pages 265–276. VLDB Endowment, 2013.

[Zhang 2007] S. Zhang, J. Yang and V. Cheedella. *Monkey: Approximate Graph Mining Based on Spanning Trees*. In 2007 IEEE 23rd International Conference on Data Engineering, pages 1247–1249, April 2007.

[Zhang 2008] Shijie Zhang and Jiong Yang. *RAM: Randomized Approximate Graph Mining*. In Bertram Ludäscher and Nikos Mamoulis, editors, Scientific and Statistical Database Management, number 5069 de Lecture Notes in Computer Science, pages 187–203. Springer Berlin Heidelberg, July 2008. DOI: 10.1007/978-3-540-69497-7_14.

[Zhang 2012] Ying Zhang, Pham Minh Duc, Oscar Corcho and Jean-Paul Calbimonte. *SRBench: a streaming RDF/SPARQL benchmark*. In International Semantic Web Conference, pages 641–657. Springer, 2012.

[Zheng 2016] Weiguo Zheng, Lei Zou, Wei Peng, Xifeng Yan, Shaoxu Song and Dongyan Zhao. *Semantic SPARQL similarity search over RDF knowledge graphs*. Proceedings of the VLDB Endowment, vol. 9, no. 11, pages 840–851, 2016.

[Zhou 2015] Keren Zhou. *gspan algorithm in data mining*. `https://github.com/Jokeren/DataMining-gSpan`, 2015. [Online; accessed 2016-05-30].

[Zou 2009] Zhaonian Zou, Jianzhong Li, Hong Gao and Shuo Zhang. *Frequent Subgraph Pattern Mining on Uncertain Graph Data*. In Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM '09, pages 583–592, New York, NY, USA, 2009. ACM.

[Zou 2010] Z. Zou, J. Li, H. Gao and S. Zhang. *Mining Frequent Subgraph Patterns from Uncertain Graph Data*. IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 9, pages 1203–1218, September 2010.

Aggregated Search in Distributed Graph Databases

Abstract: In this research, we are interested in investigating issues related to query evaluation and optimization in the framework of aggregated search. Aggregated search is a new paradigm to access massively distributed information. It aims to produce answers to queries by combining fragments of information from several sources. The queries search for objects (documents) that do not exist as such in the targeted sources, but are built from fragments extracted from the sources. The sources might not be specified in the query expression, they are dynamically discovered at runtime. In our work, we will consider data dependencies to design a framework to optimize query evaluation over distributed data sources.