



UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD DE MÁLAGA

ETS INGENIERÍA INFORMÁTICA

DEPARTAMENTO LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

PROGRAMA DE DOCTORADO DE TECNOLOGÍAS INFORMÁTICAS

# Processing Structured Data Streams

*Procesamiento de flujos de datos estructurados*

TESIS DOCTORAL

**GALA BARQUERO MORENO**

## **Directores:**

Antonio Vallecillo Moreno

Javier Troya Castilla


Málaga, 2020

---



UNIVERSIDAD  
DE MÁLAGA

AUTOR: Gala Barquero Moreno

 <http://orcid.org/0000-0003-2032-7003>

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga



Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional:

<http://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>

Cualquier parte de esta obra se puede reproducir sin autorización pero con el reconocimiento y atribución de los autores.

No se puede hacer uso comercial de la obra y no se puede alterar, transformar o hacer obras derivadas.

Esta Tesis Doctoral está depositada en el Repositorio Institucional de la Universidad de Málaga (RIUMA): [riuma.uma.es](http://riuma.uma.es)



## DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña GALA BARQUERO MORENO

Estudiante del programa de doctorado TECNOLOGÍAS INFORMÁTICAS de la Universidad de Málaga, autor/a de la tesis, presentada para la obtención del título de doctor por la Universidad de Málaga, titulada: PROCESSING STRUCTURED DATA STREAMS

Realizada bajo la tutorización de ANTONIO VALLECILLO MORENO y dirección de ANTONIO VALLECILLO MORENO Y JAVIER TROYA CASTILLA (si tuviera varios directores deberá hacer constar el nombre de todos)

DECLARO QUE:

La tesis presentada es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, conforme al ordenamiento jurídico vigente (Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), modificado por la Ley 2/2019, de 1 de marzo.

Igualmente asumo, ante a la Universidad de Málaga y ante cualquier otra instancia, la responsabilidad que pudiera derivarse en caso de plagio de contenidos en la tesis presentada, conforme al ordenamiento jurídico vigente.

En Málaga, a 09 de NOVIEMBRE de 2020

Fdo.: GALA BARQUERO MORENO

*A quienes creen en mí*







UNIVERSIDAD  
DE MÁLAGA

*Universidad de Málaga*

UNIVERSIDAD  
DE MÁLAGA







Universidad de Málaga  
ETS Ingeniería Informática  
Departamento Lenguajes y ciencias de la computación

El Dr. Antonio Vallecillo Moreno, Catedrático de Universidad en el Departamento de Languages y Ciencias de la Computación de la E.T.S. de Ingeniería Informática de la Universidad de Málaga, y el Dr. Javier Troya Castilla, Profesor Contratado Doctor en el Departamento de Lenguajes y Sistemas Informáticos de la E.T.S. de Ingeniería Informática de la Universidad de Sevilla,

Certifican que Dña. Gala Barquero Moreno, Ingeniera de Telecomunicaciones, ha realizado en el Departamento de Languages y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a la Tesis Doctoral titulada:

*Processing Structured Data Streams*

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Del mismo modo certifican que las publicaciones que avalan dicha Tesis Doctoral no han sido utilizadas en tesis anteriores.

En Málaga, Noviembre de 2020

.....  
Dr. Antonio Vallecillo Moreno

.....  
Dr. Javier Troya Castilla



# Agradecimientos

---

Tras tres años de duro trabajo finalmente ha llegado el momento de escribir estas palabras. Tres años en los que en ciertos momentos una o varias personas han supuesto la clave para llegar al día de hoy. No podía, por tanto, dejar pasar la oportunidad de darles las gracias.

En primer lugar, a mis directores; Antonio Vallecillo y Javier Troya, ya que sin ellos esta tesis no habría sido posible. Gracias Antonio por brindarme la oportunidad de llevarla a cabo. Doctorarme siempre fue una de mis metas desde que acabé la carrera en 2015 y, finalmente, he podido alcanzarla. Gracias también Javi por toda tu dedicación como director de tesis y por la ayuda que siempre me has brindado incluso sin poder estar de forma presencial.

También quiero agradecer a mis compañeros de laboratorio los buenos ratos que he pasado en el 3.3.3. A pesar de que en el último periodo de esta tesis no he pasado mucho tiempo allí podré recordar almuerzos y descansos muy agradables. Sin embargo, quiero hacer un agradecimiento especial a Loli Burgueño, por ser una gran compañera siempre dispuesta a ayudarme, tanto estando cerca como lejos.

Fuera del ámbito académico quiero dar las gracias a Aurora Cámara por sus consejos, que me han ayudado a seguir adelante en estos tres años. Gracias a Cristina Navarro y a mi segunda familia de Comediantes Malagueños. Esta familia ha sido un ingrediente de desconexión muy necesario en momentos de estrés. Además, siempre representarán mi verdadera pasión: el Teatro.

Por último, quiero hacer mención especial a varias personas. Esta tesis está dedicada a quienes creen en mí y estoy segura de que ellas lo han hecho más que yo misma:

Gracias a Alex Lavado. Por estar siempre ahí cuando lo necesitaba, tanto técnica como emocionalmente. Sin tu ayuda esta tesis no sería una realidad.

Gracias a Encarna, mi abuela. Por saber valorar y ser una testigo incansable

de todos los proyectos en los que me involucro sin importar los años que pasen.

Gracias a Pedro y Sara, mi padre y mi hermana. Porque puedo presumir de tener un buen padre dispuesto a darlo todo por sus hijas y una hermana de la que aprendo muchas cosas aun siendo diez años más joven que yo, aunque puede que sea precisamente esto por lo que puedo aprender de ella. Sara, quiero que esto te sirva de inspiración para aprender, esta vez tú de mí, a no rendirte ante ninguna dificultad.

Gracias a Jose Burgos. Por compartir tu vida conmigo, por ser mi gran apoyo en los malos momentos y no dejarme caer y, también, por disfrutar juntos de los buenos. Siempre eres esa voz que me dice ‘Hazlo, tú puedes y yo te apoyo’.

Y finalmente gracias a Vicky, mi madre. Por no dudar de mí jamás y ser un ejemplo de constancia y esfuerzo. Esta tesis es, en primer lugar, dedicada a ti.

*Gala Barquero Moreno*

# Abstract

---

A large amount of data is daily generated from different sources such as social networks, recommendation systems or geolocation systems. Moreover, this information tends to grow exponentially every year. Companies have discovered that the processing of these data may be important in order to obtain useful conclusions that serve for decision-making or the detection and resolution of problems in a more efficient way, for instance, through the study of trends, habits or customs of the population. The information provided by these sources typically consists of a non-structured and continuous data flow, where the relations among data elements conform graph structures. Inevitably, the processing performance of this information progressively decreases as the size of the data increases. For this reason, non-structured information is usually handled taking into account only the most recent data and discarding the rest, since they are considered not relevant when drawing conclusions. However, this approach is not enough in the case of sources that provide graph-structured data, since it is necessary to consider spatial features as well as temporal features. These spatial features refer to the relationships among the data elements. For example, some cases where it is important to consider spatial aspects are marketing techniques, which require information on the location of users and their possible needs, or the detection of diseases, that use data about genetic relationships among subjects or the geographic scope.

It is worth highlighting three main contributions from this dissertation. First, we provide a comparative study of seven of the most common processing platforms to work with huge graphs and the languages that are used to query them. This study measures the performance of the queries in terms of execution time, and the syntax complexity of the languages according to three parameters: number of characters, number of operators and number of internal variables. We elaborate this study in order to choose the most suitable technology to develop our proposal.



Second, we propose three methods to reduce the set of data to be processed by a query when working with large graphs, namely spatial, temporal and random approximations. These methods are based on Approximate Query Processing techniques and consist in discarding the information that is considered not relevant for the query. The reduction of the data is performed online with the processing and considers both spatial and temporal aspects of the data. Since discarding information in the source data may decrease the validity of the results, we also define the transformation error obtain with these methods in terms of accuracy, precision and recall.

Finally, we present a preprocessing algorithm, called SDR algorithm, that is also used to reduce the set of data to be processed, but without compromising the accuracy of the results. It calculates a subgraph from the source graph that contains only the relevant information for a given query. Since this technique is a preprocessing algorithm it is run offline before the actual processing begins. In addition, an incremental version of the algorithm is developed in order to update the subgraph as new information arrives to the system.

# Contents

---

<b>List of Figures</b>	<b>xxi</b>
<b>List of Tables</b>	<b>xxv</b>
<b>Glossary</b>	<b>xxvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivations and Goals . . . . .	3
1.1.1 General Goals . . . . .	4
1.1.2 Specific Goals . . . . .	4
1.2 Contribution . . . . .	5
1.3 Outline . . . . .	5
<b>2 Background</b>	<b>9</b>
2.1 Model-Driven Engineering . . . . .	9
2.1.1 History . . . . .	10
2.1.2 Main concepts . . . . .	11
2.2 Data Streaming Applications . . . . .	16
2.2.1 Complex Event Processing . . . . .	18
2.2.2 Approximate Query Processing . . . . .	21
2.3 Graph-structured information . . . . .	22
2.3.1 Models as graphs . . . . .	23
2.3.2 Graph processing platforms . . . . .	23
<b>3 Comparison and Performance Evaluation of Processing Platforms</b>	<b>25</b>
3.1 A running example . . . . .	27



3.2	Processing platforms . . . . .	29
3.2.1	Neo4j . . . . .	29
3.2.2	JanusGraph . . . . .	30
3.2.3	OrientDB . . . . .	30
3.2.4	TinkerGraph . . . . .	31
3.2.5	Memgraph . . . . .	31
3.2.6	CrateDB . . . . .	31
3.2.7	Apache Spark. GraphFrames . . . . .	32
3.3	Query Languages . . . . .	33
3.3.1	SQL . . . . .	33
3.3.2	Cypher . . . . .	40
3.3.3	Gremlin . . . . .	46
3.3.4	GraphFrames . . . . .	55
3.4	Performance Analysis and Evaluation . . . . .	60
3.4.1	Research Questions . . . . .	60
3.4.2	Case studies . . . . .	61
3.4.3	Experimental Setup . . . . .	64
3.4.4	Results . . . . .	67
3.4.5	Threats to validity . . . . .	86
3.5	Related work . . . . .	88
3.6	Summary . . . . .	90
<b>4</b>	<b>Improving Performance with Online Techniques</b>	<b>91</b>
4.1	A running example . . . . .	93
4.2	Approach . . . . .	95
4.2.1	Main concepts . . . . .	95
4.2.2	Online AQP Techniques . . . . .	96
4.2.3	Measures for accuracy . . . . .	100
4.3	Performance Analysis and Evaluation . . . . .	101
4.3.1	Research Questions . . . . .	102
4.3.2	Experimental Setup . . . . .	102
4.3.3	Results . . . . .	106
4.3.4	Discussion . . . . .	113
4.3.5	Threats to validity . . . . .	115

4.4	Related Work . . . . .	117
4.5	Summary . . . . .	118
<b>5</b>	<b>Improving performance with Offline techniques</b>	<b>121</b>
5.1	A running example . . . . .	123
5.2	Classification of queries . . . . .	124
5.2.1	Simple filter pattern . . . . .	125
5.2.2	Condition pattern . . . . .	125
5.2.3	Negation pattern . . . . .	126
5.2.4	Conjunctive pattern . . . . .	126
5.2.5	Disjunctive pattern . . . . .	127
5.2.6	Aggregation pattern . . . . .	127
5.3	The SDR algorithm . . . . .	128
5.3.1	The main SDR Algorithm . . . . .	128
5.3.2	Incremental SDR Algorithm . . . . .	136
5.4	Performance Analysis and Evaluation . . . . .	139
5.4.1	Research Questions . . . . .	140
5.4.2	Case Studies . . . . .	140
5.4.3	Experimental Setup . . . . .	144
5.4.4	Experiments and data collected . . . . .	145
5.4.5	Functional Correctness . . . . .	150
5.5	Results . . . . .	151
5.5.1	RQ1: Graph size reduction . . . . .	151
5.5.2	RQ2: Performance improvement . . . . .	152
5.5.3	RQ3: Execution time gains with data streams . . . . .	152
5.5.4	SDR algorithm and Indexing techniques . . . . .	156
5.5.5	Threats to Validity . . . . .	158
5.6	Related Work . . . . .	159
5.7	Summary . . . . .	163
<b>6</b>	<b>Conclusions and Future Work</b>	<b>165</b>
6.1	Summary and Conclusions . . . . .	165
6.2	Publications . . . . .	167
6.2.1	Publications Supporting this Dissertation . . . . .	168

6.2.2	Further Publications . . . . .	168
6.3	Future Work . . . . .	169
6.3.1	Online techniques . . . . .	169
6.3.2	Offline techniques . . . . .	170
6.3.3	Mixed techniques . . . . .	171
<b>Bibliography</b>		<b>173</b>
<b>A Results and queries for the comparison of Processing Platforms</b>		<b>189</b>
A.1	Queries for processing platforms . . . . .	189
A.1.1	TwitterFlickr queries with effect . . . . .	190
A.1.2	TwitterFlickr queries without effect . . . . .	196
A.1.3	TrainBenchmark queries with effect . . . . .	204
A.1.4	TrainBenchmark queries without effect . . . . .	213
A.2	Additional charts and tables displaying TrainBenchmark results . .	222
<b>B Results for Online AQP techniques</b>		<b>229</b>
B.1	Results for Batch A . . . . .	229
B.1.1	Q1 - Random approximation . . . . .	231
B.1.2	Q2 - Random approximation . . . . .	232
B.1.3	Q3 - Random approximation . . . . .	233
B.1.4	Q3 - Temporal approximation . . . . .	234
B.1.5	Q4 - Random approximation . . . . .	235
B.1.6	Q4 - Spatial approximation . . . . .	236
B.1.7	Q5 - Spatial approximation . . . . .	237
B.2	Results for Batch B . . . . .	238
B.2.1	Q1 - Random approximation . . . . .	239
B.2.2	Q2 - Random approximation . . . . .	240
B.2.3	Q3 - Random approximation . . . . .	241
B.2.4	Q3 - Temporal approximation . . . . .	241
B.2.5	Q4 - Random approximation . . . . .	243
B.2.6	Q4 - Spatial approximation . . . . .	244
B.2.7	Q5 - Spatial approximation . . . . .	245

<b>C</b>	<b>SDR Algorithm Execution</b>	<b>247</b>
C.1	<i>ProductPopularity</i> with SDR algorithm . . . . .	247
C.2	Traversals with SDR algorithm . . . . .	254
C.2.1	<i>Where Step</i> . . . . .	255
C.2.2	<i>Not Step</i> . . . . .	255
C.2.3	<i>And Step</i> . . . . .	256
C.2.4	<i>Or Step</i> . . . . .	257
C.3	Results for Experiments with SDR algorithm and streams of information . . . . .	259
C.4	Additional charts and tables displaying experiments results . . . .	263
<b>D</b>	<b>Experiments Replicability</b>	<b>267</b>
D.1	Online Techniques experiments . . . . .	267
D.2	Offline techniques experiments . . . . .	269
D.2.1	Configuration and execution . . . . .	269
D.2.2	Obtaining a subgraph . . . . .	270
D.2.3	Running a query over a graph or a subgraph . . . . .	271
D.2.4	Running the incremental SDR algorithm . . . . .	271
<b>E</b>	<b>Resumen</b>	<b>273</b>
E.1	Motivación y objetivos . . . . .	275
E.1.1	Objetivos generales . . . . .	276
E.1.2	Objetivos específicos . . . . .	276
E.2	Contribuciones . . . . .	277
E.3	Comparación y evaluación del rendimiento de las plataformas de procesamiento . . . . .	278
E.3.1	Plataformas de procesamiento y lenguajes de consulta . . .	278
E.3.2	Métodos de medición . . . . .	279
E.3.3	Parámetros de estudio . . . . .	280
E.4	Mejora del rendimiento empleando técnicas en línea con el procesamiento . . . . .	287
E.4.1	Técnicas AQP en línea . . . . .	289
E.4.2	Medidas de precisión . . . . .	291
E.4.3	Evolución del rendimiento y precisión con modelos aproximados	292

E.5	Mejora del rendimiento empleando técnicas de preprocesamiento . . . . .	296
E.5.1	Clasificación de las queries . . . . .	297
E.5.2	Algoritmo SDR . . . . .	298
E.5.3	Algoritmo SDR incremental . . . . .	302
E.5.4	Mejora del rendimiento con algoritmo SDR . . . . .	304
<b>F</b>	<b>Conclusiones y Contribuciones</b>	<b>311</b>

# List of Figures

---

2.1	Four level architecture proposed by the OMG. . . . .	13
2.2	Example of UML with the organization in four levels proposed by the OMG. . . . .	13
2.3	CEP stream diagram. . . . .	19
2.4	Example of CEP system for fire detection. . . . .	19
3.1	Twitter and Flickr joint Metamodel. . . . .	28
3.2	GraphTraversalSource and GraphTraversal in Gremlin console . . .	49
3.3	TrainBenchmark metamodel [111] . . . . .	63
3.4	Execution time results for queries without effect of TwitterFlickr example with single runs . . . . .	68
3.5	Execution time results for queries without effect of TwitterFlickr example with parallel runs . . . . .	69
3.6	Execution time results for queries with effect of TwitterFlickr example with single runs . . . . .	76
3.7	Execution time results for queries with effect of TwitterFlickr example with parallel runs . . . . .	77
4.1	The Amazon Example Metamodel. . . . .	93
4.2	Accuracy, Precision and Recall with Random Approximations. . .	107
4.3	Comparison between Temporal and Random Approximations with uniformly distributed data. . . . .	108
4.4	Comparison between Temporal and Random Approximation with temporal focus on the data. . . . .	110
4.5	Comparison between Spatial and Random Approximations. . . . .	111
4.6	Spatial Approximations with several sources . . . . .	113





## LIST OF FIGURES

---

4.7	Memory consumption for Q3 and Q4. . . . .	114
5.1	Overall view of queries using the SDR algorithm. . . . .	129
5.2	NY Caption Contest metamodel . . . . .	141
5.3	Youtube Videos metamodel . . . . .	143
5.4	Performance results of the SDR algorithm for the Amazon queries. . . . .	153
A.1	Execution time results for queries without effect of TrainBenchmark example with single runs . . . . .	223
A.2	Execution time results for queries without effect of TrainBenchmark example with parallel runs . . . . .	224
A.3	Execution time results for queries with effect of TrainBenchmark example with single runs . . . . .	225
A.4	Execution time results for queries with effect of TrainBenchmark example with parallel runs . . . . .	226
B.1	Q1 Batch A. Accuracy and Precision with Random Approximations. . . . .	231
B.2	Q2 Batch A. Accuracy and Precision with Random Approximations. . . . .	232
B.3	Q3 Batch A. Accuracy and Recall with Random Approximations. . . . .	233
B.4	Q3 Batch A. Accuracy and Recall with Temporal Approximations. . . . .	234
B.5	Q4 Batch A. Accuracy and Recall with Random Approximations. . . . .	235
B.6	Q4 Batch A. Accuracy and Recall with Spatial Approximations. . . . .	236
B.7	Q5 Batch A. Accuracy and Precision with Spatial Approximations. . . . .	237
B.8	Q1 Batch B. Accuracy and Recall with Random Approximations. . . . .	239
B.9	Q2 Batch B. Accuracy and Precision with Random Approximations. . . . .	240
B.10	Q3 Batch B. Accuracy and Recall with Random Approximations. . . . .	241
B.11	Q3 Batch B. Accuracy and Recall with Temporal Approximations. . . . .	242
B.12	Q4 Batch B. Accuracy and Recall with Random Approximations. . . . .	243
B.13	Q4 Batch B. Accuracy and Recall with Spatial Approximations. . . . .	244
B.14	Q5 Batch B. Accuracy and Precision with Spatial Approximations. . . . .	245
C.1	Graph 1: example for Amazon case . . . . .	248
C.2	Performance results for SDR algorithm in Contest example queries. . . . .	263
C.3	Performance results for SDR algorithm in YouTube example queries. . . . .	264
E.1	Metamodelo conjunto de Twitter y Flickr. . . . .	280

E.2	Metamodelo de TrainBenchmark [111] . . . . .	281
E.3	Metamodelo de ejemplo de Amazon. . . . .	290
E.4	Exactitud, Precisión y Exhaustividad con aproximaciones aleatorias. 293	
E.5	Comparación entre aproximaciones temporales y aleatorias con datos uniformemente distribuidos. . . . .	294
E.6	Comparación entre aproximaciones temporales y aleatorias con un foco temporal en los datos. . . . .	295
E.7	Comparación entre aproximaciones espaciales y aleatorias. . . . .	296
E.8	Overall view of queries using the SDR algorithm. . . . .	297
E.9	Metamodelo del NY Caption Contest . . . . .	305
E.10	Metamodelo del ejemplo de Youtube . . . . .	306
E.11	Performance results of the SDR algorithm for the Amazon queries. 307	



# List of Tables

---

3.1	Processing platforms used in the experiments of the performance study . . . . .	29
3.2	TwitterFlickr example with SQL tables . . . . .	34
3.3	Summary of the models used in the experiments. . . . .	65
3.4	Execution time averages (ms) depending on size model . . . . .	71
3.5	Coefficient of variation (%) of TwitterFlickr queries without effect and parallel runs. . . . .	73
3.6	Execution time averages (ms) depending on model size (with effect over the graph) . . . . .	75
3.7	Overhead of execution time average (%) for updating the graph depending on model size . . . . .	78
3.8	Summary of DSL features for TwitterFlickr case study . . . . .	83
4.1	Summary of the models used in the experiments. . . . .	103
5.1	Summary of the models used in the experiments. . . . .	145
5.2	Summary of the queries used in the experiments. . . . .	146
5.3	Elements savings ratio when running the SDR algorithm. . . . .	148
5.4	Execution times (ms) of queries with the complete graph ( $T_g$ ), the subgraph ( $T_{sg}$ ), and the corresponding speedups (S). . . . .	154
5.5	Gain ratio when using the incremental algorithm in the Amazon case study. . . . .	155
5.6	Number of query executions needed to obtain a positive gain for each query. . . . .	157



## LIST OF TABLES

---

A.1	Coefficient of variation (%) of TrainBenchmark queries without effect and parallel runs. . . . .	227
A.2	Summary of DSL features for TrainBenchmark case study . . . . .	228
C.1	Object weights for <i>ProductPopularity</i> query with SDR Algorithm .	255
C.2	Object weights for <i>ProductPopularity</i> query with <i>not</i> step with SDR Algorithm . . . . .	256
C.3	Object weights for subquery example with SDR Algorithm . . . . .	257
C.4	Object weights for <i>PackagePopularity</i> example with SDR Algorithm	258
C.5	Object weights for <i>SimProductsPopularity</i> example with SDR Algorithm . . . . .	258
C.6	Incremental results for Amazon case study (ms). . . . .	260
C.7	Incremental results for Contest case study (ms). . . . .	261
C.8	Incremental results for Youtube case study (ms). . . . .	262
C.9	Ratio Incremental gain results for Contest case study. . . . .	265
C.10	Ratio Incremental gain results for Youtube case study. . . . .	266
E.1	Plataformas de procesamiento utilizadas en los experimentos del estudio del rendimiento . . . . .	279
E.2	Promedios de tiempo de ejecución (ms) según el tamaño de modelo	283
E.3	Promedios de tiempo de ejecución (ms) según el tamaño del modelo (con efecto sobre el grafo) . . . . .	285
E.4	Resumen de las características de cada DSL para el caso de estudio de TwitterFlickr . . . . .	288
E.5	Ratio de la ganancia de elementos cuando se usa el algoritmo SDR.	308
E.6	Gain ratio when using the incremental algorithm in the Amazon case study. . . . .	309

# Glossary

---

ACID	Atomicity, Consistency, Isolation and Durability
AMT	Approximate Model Transformation
APQ	Approximate Query Processing
CASE	Computer Aided Software Engineering
CEP	Complex Event Processing
DSL	Domain Specific Language
EPL	Event Processing Languages
FN	False Negative
FP	False Positive
M2M	Model to Model
M2T	Model to Text
MDE	Model-Driven Engineering
MLM	Multi-Leve Modelling
MOF	Meta-Object Facility
MT	Model Tranformation
OLAP	Online Analytics Process
OLTP	Online Transaction Process
OMG	Object Management Group
RDBMS	Relational Database Management System
RDD	Resilient Distributed Datasets
RQ	Research Question
SDR	Source Dataset Reduction
TN	True Negative
TP	True Positive
UML	Unified Modeling Language



# Chapter 1

## Introduction

---

A large amount of data is currently generated on a daily basis. This information comes from different sources such as social networks, e-commerce pages or web search engines, among others. A specific example of these sources is Twitter: statistics calculate about 6,000 tweets per second [63]. Every year, this information grows exponentially. In fact, data centers registered 6.5 zettabytes of data in 2016, and it is estimated that this value will reach 44 zettabytes by 2021. The drastic increase in the amount of information produced by these sources requires an efficient processing of data flows in real time to make informed decisions and to detect situations of interest that require instantaneous reactions. An example of the importance of being able to efficiently process large amounts of information flows is shown in the analysis carried out by the Spanish bank BBVA on the economic impact of Barcelona's 2012 *Mobile World Congress* [19]. The study required the online analysis of all credit card transactions during two weeks and they concluded the event had an economic impact on the city of more than 320 million Euros at the local level. Another example is the need for real-time analysis of streams of information in social networks or weblogs in order to detect possible terrorist



attacks [97, 130]. Due to their size and complexity, data cannot be processed by traditional tools, such as relational databases and conventional statistics or visualization packages. Therefore, it is necessary to use software and hardware whose processing speed and storage capacity are powerful enough to manage them.

Many proposals that deal with this increasing amount of information are based on the fact that most of the data that need to be processed for decision making is not significantly relevant, particularly with large volumes of data. In this sense, data flow processing systems are becoming popular, such as Apache Spark [70] or Apache Kafka [68]. Besides, Complex Event Processing (CEP) is also a useful technology in this context. CEP is able to process and analyse streams of information represented as a sequence of simple events in order to obtain conclusions from them, represented as complex events [38, 49, 77, 78]. Several CEP engines and Event Processing Languages (EPLs) exist, such as the Esper language [48]. These technologies consider that only the most recent data are relevant to obtain results. In this way, the information processing is not carried out on the complete set of data, but on a subset that is determined by timestamps. Then, all data that are considered too old for not providing information of interest are discarded. This approach is useful with information sources where events are not related among themselves.

However, in reality, information is commonly conformed by data connected among them, so that they form graph structures. As an example, in Twitter, tweets are published by users who, in turn, are followed by other users and they also follow users. Regarding such graph structures, we can distinguish between persistent and transient information. The former refers to data stored in the system in a persistent way (e.g., users, products or shops). Transient information refers to data that are temporarily stored (e.g., tweets, orders, bank transactions) and are discarded after some period of time—i.e., transient information *expires*. The interconnections between these data need to be processed too, which inevitably implies a decrease in systems performance [111]. This means that, in the search for mechanisms to select only part of the information to process so that performance is improved, it is not enough to consider techniques such as those based merely on the data timestamp [38, 44, 77]. On the contrary, it is necessary to design mechanisms to select a subset of the information with respect to several other features, such as connections, status and topology of the network. Surely, in order to improve performance we need

to discard some of the data, i.e., we must approximate our data. Consequently, the accuracy of our results might be compromised. Nevertheless, there are many applications that do not need extremely accurate results, since they manage non-critical information. Examples are recommendation systems on Facebook, Netflix or Amazon. Here, the goal is to find the right balance between the performance of our queries and the accuracy of their results.

Different works in this line can be found in the literature. For instance, in a previous work [121] the concept of *Approximate Model Transformations* (AMT) was introduced to find the right balance between performance and accuracy in the context of model transformations. Sampling techniques were used in a wireless sensor network example to show the effects of selecting certain subsets of elements. However, the information in study was not composed of interconnected data. Other works apply Approximate Query Processing (AQP) [32, 53, 75, 87], which try to get an approximate answer but precise enough in order to improve the performance. However, most of these works do not consider data flows or graph-structured information. For this reason, the processing of streams of data structured as graphs is still a research problem that needs to be addressed.

This chapter is structured as follows. In Section 1.1 we present the goals and the research question that we aim to answer with this work. In Section 1.2 we show the contributions derived from this dissertation and, finally, Section 1.3 exposes the structure of this thesis.

## 1.1 Motivations and Goals

Summarizing what has been explained above, this dissertation aims to address the problem of improving performance when processing large amounts of information composed of interconnected data. In order to improve such performance, we need to reduce the amount of information to process, therefore jeopardizing the accuracy of the results. In order to target such problem, we consider graph-structured information, i.e., information composed of data highly interconnected among them. Besides, information arrival is never-ending, so we refer to “graph-structured data flows” when talking about this type of information. Thus, the main research question addressed in this thesis is:

**RQ** *Can we obtain a good (or the optimal) trade-off between performance and*

*accuracy loss when processing very-large amounts of information?*

Some goals have been derived from this research question, and are exposed in the following.

### 1.1.1 General Goals

In order to respond to this thesis' research question, three main general goals were derived with regards to techniques for discarding information and the errors that may arise.

- First, we aim to design a mechanism that is able to select the relevant data that is needed for a query. For this purpose, the information has to be filtered temporarily and spatially.
- In order to obtain valid results, we pretend to define the types of errors that may arise when selecting only part of the initial information to process and the meaning of such errors.
- Once errors are defined, we want to calculate and study them according to several parameters such as the amount of initial information or the amount of information selected for processing.

### 1.1.2 Specific Goals

Additionally, the following specific goals were also derived from the research question of this thesis:

- In order to develop an efficient method for selecting only part of the information to process, we need to find a processing platform that meets our requirements.
- Besides, we need to find or develop a query language with a simple syntax to perform the queries over the data.
- Once the method is developed we want to test it in different use cases and different model sizes representing the information.

## 1.2 Contribution

Three main contributions have been obtained throughout the research of this thesis. They can be summarized as follows:

1. A comparative study among the most common processing platforms and Domain Specific Languages (DSLs) that are used to handle huge amounts of data. In this study, we take into account the performance of the queries and the complexity of their syntax. Performance is measured in terms of execution time, while syntax complexity is measured in terms of number of characters, operators and internal variables.
2. Three online methods for discarding information that is not relevant for a given query. These methods are based on online Approximate Query Processing (AQP) techniques and select the information according to time and spatial ranges and random parameters. In order to measure the accuracy loss derived from these methods, the definition of transformation error is given in terms of accuracy, recall and precision.
3. An algorithm based on offline AQP techniques that selects a subset of the source information that is relevant according to the patterns that can be found in a given query, called SDR algorithm. Our empirical experiments show that the accuracy of the results is not affected when applying this algorithm, since it considers all data that are important for the processing. For this reason, there is no need to calculate the errors produced from the execution of this algorithm.

## 1.3 Outline

The contributions described before are explained in detail in the rest of this document. The chapters are structured as follows:

### Chapter 2. Background

We present three main concepts that have been used in this thesis. First, the MDE methodology and its main terms (models, metamodels, Domain-Specific

Languages and model transformations) are presented. Second, we expose an overview of Data Streaming Applications where two techniques are highlighted since they are the basis of this thesis (Complex Event Processing and Approximate Query Processing). Finally, we present the most important concepts about graph theory in order to apply them to our approach.

### Chapter 3. Comparison and Performance Evaluation of Processing Platforms

An analysis of the benefits and limitations of seven processing platforms commonly used in Big Data applications is presented in this chapter. The platforms comprise TinkerGraph [118], Neo4j [89], CrateDB [35], Memgraph [83], JanusGraph [64], OrientDB [29] and GraphFrames [106]. These technologies allow to write queries by means of different query languages, namely Gremlin [6], Cypher [88], SQL and the DSL used for GraphFrames [107]. We tested all these technologies in two case studies in order to measure the performance of the queries, in terms of execution time, and the complexity of the language, in terms of number of characters, operators and variables. Furthermore, two types of experiments are included: (i) queries are run over the data in order to obtain useful information without implying any side effect in the source information and (ii) queries are run and their results modify the source graph by adding, removing or updating the existing information. The purpose of this chapter is to choose the most suitable technology to develop our proposal. For this reason, a combination of platform and query language is chosen to implement our approach, according to our requirements.

### Chapter 4. Improving Performance with Online Techniques

We explore three different online AQP techniques in this chapter: random, temporal and spatial approximations. These techniques are analysed under two circumstances: (i) depending on how the data is organized and (ii) depending on what information needs to be obtained from the data. Furthermore, we propose a method that allows to estimate the errors produced when applying approximations. The goal is to find the right balance between performance gain and accuracy loss when approximating. To achieve this, accuracy loss is defined in terms of accuracy, recall and precision. Approximations are tested in a simplified version of the

Amazon ordering service. Finally, the results of the experiments conclude that it is possible to improve the performance with these techniques.

## **Chapter 5. Improving Performance with Offline Techniques**

This chapter proposes an algorithm to improve the performance when querying graph-structured information flows, called Source Dataset Reduction (SDR) algorithm. The solution implemented does not compromise the accuracy of the results. The algorithm obtains a subgraph of the complete dataset with the relevant elements for a query given. Furthermore, a classification of six different patterns that can be found in a query is proposed in order to study how the performance can be improved depending on the type of query. Finally, an incremental version of the algorithm is also presented so that the relevant query subgraph is automatically updated, at a very low cost, when new information arrives or the system data changes. The algorithm is tested in three case studies and results show that it is able to achieve speedups of more than 100x for some types of queries.

## **Chapter 6. Conclusions and Future Work**

In this chapter we summarize all conclusions and the contributions presented in the different chapters. Besides, we report the publications derived from our research as well as outline several possible lines of future work.

## **Appendix A. Queries for processing platforms**

This appendix shows all the queries for the two case studies that appear in Chapter 3. The queries are implemented for seven different processing platforms, that are also presented in that chapter. They comprise two types of implementations: (i) queries without any side effect and (ii) queries that involve an effect over the source data by means of adding, removing or updating existing elements. In addition, some additional tables and figures displaying the experiments results for one case study are placed in this appendix in order to improve the readability of Chapter 3.

## **Appendix B. Results for Online AQP techniques**

This appendix shows all results for the experiments of the approach presented in Chapter 4 and summarizes the conclusions obtained from them. Note that these experiments comprise three online AQP techniques over several queries with two different data distributions (Batch A and B). These techniques are temporal, spatial and random approximations.

## **Appendix C. SDR Algorithm Execution**

This appendix presents additional information about the algorithm presented in Chapter 5. First, an example of the functioning of the SDR algorithm with a specific query is presented. Second, we explain how the algorithm works depending on four different operators that may appear in a query. Third, three tables with execution times in absolute terms are depicted related to the experiments with information streams. Finally, some additional tables and figures displaying the experiments results for two case studies are placed in this appendix in order to improve the readability of Chapter 5.

## **Appendix D. Experiments Replicability**

This appendix explains the experiments' replicability packages conducted in the thesis.

## **Appendix E. Resumen**

This appendix summarizes this dissertation in Spanish.

## **Appendix F. Conclusiones y Contribuciones**

This appendix exposes our conclusions and the contributions of this dissertation in Spanish.

# Chapter 2

## Background

---

This chapter presents the fundamentals and state of the art of three main pillars on which this thesis is founded. First, an introduction to the Model-Driven Engineering methodology is given along with the most common terms used when working with this field of study. Second, we expose what Data Streaming Applications are and the most common techniques used to work with these systems. Finally, an overview of different methods to work with graph-structured information is given at the end of this chapter.

### 2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a methodology that uses software models as first-class entities throughout the software engineering life cycle. Its goal is to increase productivity, maintenance and maximize compatibility among systems [9]. This is achieved by simplifying the process of design, promoting communication among individuals and teams working on the system.



A model in software engineering is a simplified and generalized representation of a real-world system or concept created to facilitate its understanding. Therefore, a model is a simplification of the reality given as a result of an abstraction process. In this regard, abstraction is a key element for success on software engineering since it enables understanding and/or analyzing complex domains of concern which contain a large number of details. Some examples of these domains are programs, software systems and their application domains.

### 2.1.1 History

During the last decades, several abstractions have been carried out in software engineering in order to facilitate programming tasks. These abstractions aim to focus on software design, omitting complexities and leaving them out from the underlying computing environment, such as memory or CPU. For instance, third-generation languages developed in the early 1970s, such as C, raised the level of abstraction so that programmers could omit low-level details related to memory position access. This implied an advantage with respect to assembly languages. In the same line, early operating system platforms, such as OS/360 (published in 1967) or Unix (originally developed in 1969), allowed to avoid the complexities imposed by programming directly with hardware devices [102].

Some years later, in the 1980s, CASE tools (Computer Aided Software Engineering) were considered the first tools to support MDE. They pretended to simplify the software development providing a graphical means. However, these tools had two main disadvantages: (i) they lacked standardization and (ii) they were based on patented modeling languages and inefficient code generators [57].

In the past three decades, the advances in languages and programming platforms have resulted into a high level of abstraction available for the software development. Some examples are object-oriented programming languages, such as C++, Java or C#. These languages offer a higher level of abstraction than general purpose languages, such as Fortran or C. However, they still had a distinct computing-oriented focus. This was a problem because of the rapidly growing complexity of systems, that moved faster than software development technologies can cope with.

Some other problems are still to be faced. One of them is the necessity to process large volumes of information quickly, generated from diverse source data,

that imposes strong requirements to the system, such as memory, processing time, disk access or network latency. This processing has gained importance with the emergence of different technologies, such as those related to Big Data [82].

### 2.1.2 Main concepts

#### Models

At the beginning of this section, we defined the concept of **model** as “*a simplified and generalized representation of a real world system or concept created to facilitate its understanding*”. However, many discussions throughout the history of MDE have proved that it is difficult to find a consensus about the definition of this concept. For this reason, different definitions can be found in the literature. To cite some examples:

- In [127] Warmer and Kleppe define a model as a description of a system, or a part of it, that is written in a well-defined language.
- In [103] Seidewitz defined a model as a set of statements about some system under study, where *statement* is some expression (that can be true or false) about the system.
- The Object Management Group (OMG) gives different definitions:
  - In [92] a model is a representation of a part of the functionality, structure and/or behavior of a system.
  - In [93] it is defined as the description or specification of a system and its environment defined for a concrete purpose.
  - In [94] a model captures a view of a physical system, with a concrete purpose. The purpose determines which aspects of the physical system are relevant (or not) to be included in the model and at the right level of abstraction.

Despite the a lack of consensus about what a model is, all definitions seem to support the common idea that a model is a simplification of a system. Therefore, we can conclude that the strength of a model lies on its abstraction feature.

Furthermore, the total level of automation possible is elevated thanks to the combination of the abstraction and the executable semantics.

Nevertheless, there is no consensus yet on which features models should have. For example, Stachowiak [109] states that a model has three main features: (i) mapping, since it is a representation of an original system, (ii) reduction, since not all the properties of the subject are mapped, and (iii) pragmatic, since it replaces the original with respect to some purpose and it has to be usable enough. However, Bran Selic, in his presentation entitled *Abstraction Patterns in Model-Based Engineering* in the MODPROD 2011 [26], described six features for a model to be useful: (i) purposefulness, (ii) abstraction, (iii) understandability, (iv) accuracy, (v) predictiveness and (v) inexpensive. As well as these features, Selic proposes four main functions that models should carry out: (i) they should understand the system, (ii) they have to serve as a communication mechanism, (iii) they should validate the system and its design and (iv) they should guide the implementation.

In conclusion, from a software engineering perspective, models are used to better understand the useful characteristics of a real system and its environment. A key difference between a software engineer and other engineers is that the medium in which models are built is very different. Software engineers share the same medium which is the computer, while for other engineers it could be buildings, bridges or aeroplanes. This unique feature of software allows automatic transformations to be defined capable of generating implementations from higher level models. This is something which is much more expensive in other disciplines. Consequently, the purpose of MDE could be summarized as the construction of systems in the most automated possible manner. This is achieved using models and model transformations. The latter are explained below.

### Metamodels

In order to better understanding what a model is, it is also necessary to define the concept of metamodel. A **metamodel** is a model that describes another model, specifying the concepts of language as well as the relationships among them, the structural rules that restrict the possible elements in the valid models and those combinations among elements with regards to the domain semantic rules. Therefore, a model is described in the language defined by its metamodel, which yields into a

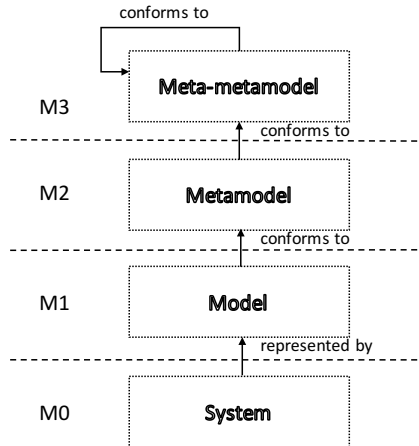


Figure 2.1: Four level architecture proposed by the OMG.

conformance relation between both of them. Besides, since a metamodel is a model itself, a metamodel is written in the language defined by its meta-metamodel. In this way, a recursive process is used to define models conformed by other models and it ends when a model conforms to itself, at a higher level of abstraction.

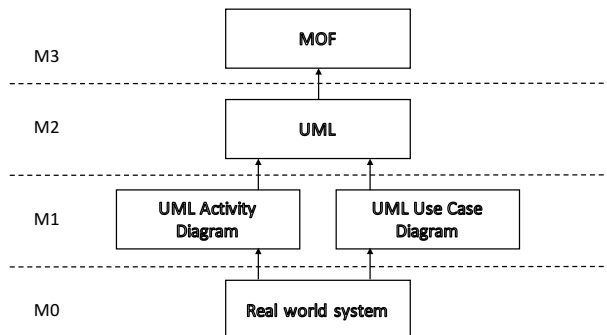


Figure 2.2: Example of UML with the organization in four levels proposed by the OMG.

Regarding this recursive process, the OMG supports Meta-Object Facility (MOF) architecture, organized in four levels, that is pictured in Figure 2.1. The lower level of this architecture (M0) refers to the system in the real world, that is

represented by the model in the next level (M1). This model conforms to its meta-model defined at level M2, whereas the metamodel conforms to the meta-metamodel at level M3, that conforms to itself. A well-known example of modeling language conforming to MOF is UML (Unified Modeling Language). Its instantiation is shown in Figure 2.2. However, MDE is currently adopting a modelling proposal that permits an arbitrary number of meta-levels called Multi-Level Modelling (MLM) or deep modelling [8, 79]. In a multilevel architecture, the dual type/instance nature makes some metamodeling facilities available at each meta-level.

### Domain-Specific Languages

Another important concept in the field of MDE is Domain-Specific Language (DSL). According to Deursen [46], the definition of a DSL is the following: “A **Domain-Specific Language** is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain”.

These languages usually support a higher level of abstraction than general-purpose languages since they are closer to the problem domain than to the implementation domain. In this sense, two main aspects must be taken into consideration when deciding if it is worth to create or reuse an existing DSL to implement the solution of a problem: (i) does the language allow to express the problem more clearly than general purpose languages? and (ii) does the problem appear frequently enough?

The definition of a DSL is divided into three main parts. These parts are related to the syntax (abstract and concrete) and the semantic of the language. In the following we explain them separately:

- **Abstract syntax:** the concepts of the language, the relationships among them and the rules that allow to build the programs, instructions, expressions or models are described in this part. Abstract syntax is usually defined using metamodels in Model-Driven Engineering.
- **Concrete syntax:** it defines the notation used for representing the models that can be described with the language, i.e. it comprises a mapping between

the abstract syntax and its representation. This representation can be visual or textual:

- Visual: this representation describes the models in an intuitive manner representing the objects and relationships by means of symbols. For example, using rectangular boxes to represent objects and lines to represent relationships. An example of visual representation is UML.
  - Textual: this representation allows to describe models in a more expressive way than visual representations. To achieve this, it uses sentences composed by strings. An example of textual representation is OCL.
- Semantics: for a better understanding of the significance of the model, the semantics defines its meaning. Depending on how the semantics is to be used, it can be defined in denotational, operational or axiomatic manners:
    - Denotational: each sentence or model of the language is translated or transformed in a sentence or model in another language with well-defined semantics. In this case, the target language is usually a mathematical formalism known as semantic domain.
    - Operational: the behavior of the system models is described in an explicit way. This is possible using a language based on actions or defining operations whose behavior is specified.
    - Axiomatic: it describes a set of rules, axioms or predicates that models must meet to check if they are well-formed. The models interpret them in a logic where it is possible to reason about them.

Some frequent examples of DSL are HTML (for web pages design), Mathematica and Maxima (for symbolic mathematics) or SQL (for relational database queries).

### Model transformations

Since models are the key part in MDE, it is important to provide mechanisms to modify and create them automatically. In this way, Model Transformations (MTs) allows a model to be manipulated and transformed using mechanisms to specify

how to produce an output model from an input model. Although there is not a common definition about Model Transformation, one of the most popular is given by Kleppe et al. [72]. They expose the following: “A **Model Transformation** generates a target model from a source model automatically and according to a transformation definition”. Both the source and target models conform to their respective metamodels. The transformation definition is set with respect to those metamodels and executed on the concrete models. Besides, a MT can have one or more source and target models and the source and target models can conform to the same or different metamodels. According to these features, there are several types of MT depending on different criteria:

- MT language: they can be *declarative*, *imperative* and *hybrid*, which means a combination of the first two features.
- Directionality: MTs can be *Unidirectional* or *Bidirectional*. In the first case, the transformation rules are executed in one direction, whereas in the second case the rules can be applied in both directions. Note that unidirectional transformations clearly differentiate an input and an output model, and in bidirectional transformations both models behave like input and output model.
- Metamodels involved: according to this criteria, there are *exogenous* and *endogenous* transformations. For exogenous metamodels, source and target models conform to their own metamodels. For endogeneous metamodels, source and target model share the same metamodel.
- Type of target model: in this case, there are *Model to Model* (M2M) and *Model to Text* (M2T) transformations. M2M transformations generate output models from input models and M2T transformations (also called injectors) generate text from models.

## 2.2 Data Streaming Applications

Data streaming applications were conceived to handle data flows generated from different sources, such as social networks, geolocation systems or ecommerce pages. These sources generate information at high rates (often gigabytes of data

per second). For instance, Twitter usage statistics estimate around 6,000 tweets per second on average, which means over 350,000 tweets per minute and 500 million tweets per day [63]. The information generated from this kind of sources is a data stream. Gürcan and Berigel [60] define data stream as follows: “A **data stream** is a continuous, real-time, and unbounded series of data items. The process of streaming divides non-stop flowing input data into distinct units for advanced processing”. Therefore, the processing of these items is known as stream processing. For this term, Gürcan and Berigel [60] gives the following definition: “**Stream processing** is a low-latency processing approach and analyzing of streaming data. Stream processing is about real-time processing of nonstop streams of data in a workflow”.

Regarding these definitions, we highlight three main challenges when processing the data stream: (i) the information is **real-time** (since this information can reach rates of several gigabytes per second, as stated at the beginning of this section, its processing implies managing a huge amount of data), (ii) the stream processing has to be carried out on **low-latency** basis (it is not a simple task since the amount of information to be processed is very large) and (iii) the information has to be **divided into distinct units** for the processing (it is necessary to incorporate additional mechanisms to make the divisions). Therefore, the pairing of real-time information and low-latency processing imposes stringent requirements on resources consumption (execution time and memory). Hence, processing systems usually divide the information into smaller units in order to obtain a faster response with lower memory consumption. However, this involves several questions: how to divide the stream without compromising the accuracy of the results? how much information should each partition contain to obtain a low-latency? which units should be analyzed at every moment?

Data-streaming applications usually include mechanisms to overcome the restrictions imposed for stream processing. In this thesis, we have classified them in two types:

- **Distributed computing:** in this approach, the information is divided into subsets that are processed by different machines (usually called workers) in parallel. Workers are organized into clusters. This type of solutions incorporates a communication mechanism between machines for coordinating



the actions. Some popular technologies that use distributed computing are *Apache Kafka* [68] and *Apache Spark* [108].

- Subsets processing: in this case, just a subset of the information is selected to be queried. This subset contains the most relevant information to obtain valid results and discards the rest in order to increase the performance of the processing. Some approaches that use this technique are *Complex Event Processing* [38, 49, 77, 78] or *Approximate Query Processing* [32, 53, 75, 87].

Both mechanisms are not exclusive. In fact, they are generally used together in order to achieve the best performance. However, this dissertation is mainly focused on the solutions comprised in the second type, since we are interested in software solutions for applications that do not need extremely accurate results instead of hardware solutions that need several machines to process all the data. More precisely, our approach proposes a solution that incorporates the main features of both Complex Event Processing and Approximate Query Processing systems. For this reason, they are explained in more detail in this section.

### 2.2.1 Complex Event Processing

Complex Event Processing (CEP) is a technique developed in early 90's for processing, analyzing and correlating streams of information. Cugola et al. [39] state that the goal of CEP is to define and detect situations of interest, from the analysis of low-level event notifications. According to the Event Processing Technical Society [50], these notifications are called *simple events*. In this way, real-time information is represented as sequences of simple events that are processed as they arrive, as it is represented in Figure 2.3. Conclusions are obtained when summarizing, representing or denoting a set of simple events, and they are represented in the form of *complex events*. Therefore, complex events are inferred from simple events, but they could also be added to the streams for enabling more powerful queries. Furthermore, simple and complex events have a *type*, a set of *attributes* and a *timestamp* with the instant when they occur. Therefore, events are atomic and happen instantaneously.

Regarding the definition of complex events, CEP allows to implement rules that use a *pattern* to combine simple or complex events. Whenever the pattern is

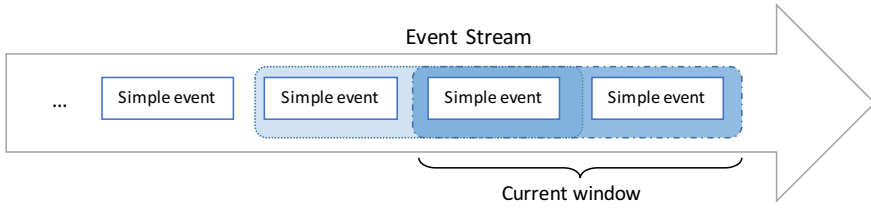


Figure 2.3: CEP stream diagram.

detected in the stream (i.e., it is *satisfied* by the events in the stream), the complex event is created. For example, observe Figure 2.4, where a fire detection system is represented. The system uses the readings of three sensors to detect a fire danger: environmental temperature, humidity percentage and CO<sub>2</sub> concentration. The sensors are continuously sending these parameters in the form of simple events. In order to detect the fire danger as soon as possible, a rule with the following patterns is defined: (i) A simple event from the temperature sensor higher than 50 degrees, (ii) a simple event from the humidity sensor lower than 40% and (iii) a simple event from the CO<sub>2</sub> sensor higher than 5000 ppm. When these events are received in a specific period of time (about 2 seconds) a complex event is created to notify the fire danger.

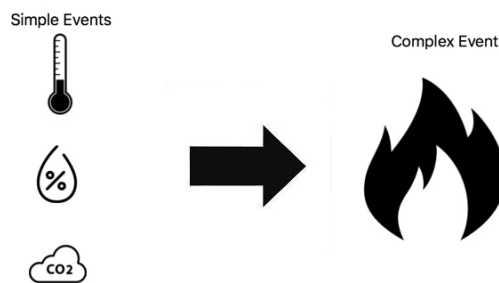


Figure 2.4: Example of CEP system for fire detection.

CEP rules are implemented by Event Processing Languages (EPLs), such as Esper language [48]. Although several EPLs exist, they all share the following representative elements of CEP patterns:

- **Windows:** They are used to restrict the scope to which the patterns apply on. Windows could be classified according to two main aspects:
  - Time or length: a *time window* keeps the events that arrive within a period of time  $T$ , whereas a *length window* keeps a specific number of  $N$  latest events.
  - Batch or sliding: a *batch window* has fixed starting and ending points, whereas a *sliding window* moves the interval (i.e. its ending point coincides with the current position of the pointer traversing the stream).

Observe how the window presented in Figure 2.3 is a sliding length window with  $N = 2$ .

- **Temporal sequencing of events:** A key CEP operator is *followedBy*, which introduces a temporal ordering between two events. Events related by this operator do not need to be consecutive, i.e. a pattern that contains *A followedBy B* only implies that the rule will search for an event *A* that occurs some time before an event *B*.
- **Pattern combination:** Same as other DSLs, EPL patterns can be combined in different ways by using logical operators (such as **or**, **and**, etc.) and temporal connectors (such as **until**, **while**, etc.), among others. Negation is also possible, representing the fact that an event has not happened. In addition, windows can be combined, restricting their scope.

Note that since the stream can be considered infinite, CEP systems restrict the elements by means of the windows. Therefore, CEP is a technology that use subsets processing and they are selected according to the window scope. Furthermore, CEP elements are indeed very close to those of endogenous model transformation rules (cf. Section 2.1.2), with two main additions: (i) the scope defined by the windows and (ii) the timestamp, that may turn the stream into a linear sequence of ordered events. For this reason, our approach is based on CEP architecture but properly modified to work with graph-structured information [13].

### 2.2.2 Approximate Query Processing

As stated in section 2.2, stream processing is about real-time processing, meaning we need to obtain results with a low latency over a huge amount of data. This requirement is rather costly for conventional systems. To alleviate this problem, some solutions propose the decreasing of the accuracy of the processing outcomes in order to improve performance. These solutions are part of the scope of Approximate Query Processing. Liu [76] gives the following definition for this term: “**Approximate Query Processing (AQP)** is an alternative way that returns approximate answers using information which is similar to the one from which the query would be answered”.

Then, AQP obtains approximate answers (but precisely enough to get correct conclusions), querying only a subset of the whole information in order to improve the performance of the processing. AQP is primarily designed for aggregated queries (such as count, sum and avg, etc.). However, some works propose AQP with non-aggregated queries [12, 14, 52]. Since the results are not fully accurate, the error made with the approximation has to be calculated. The relative error for aggregated queries is calculated as following [76]:

$$Error = \left| \frac{x - x'}{x} \right|$$

Where  $x$  is the accurate answer and  $x'$  is the approximate answer.

Existing AQP techniques are generally classified in two categories [32, 53, 75]:

- **Online:**

Online AQP selects samples of the complete information at the same instant of the query run. Then, the query is performed over the samples. Since the samples are obtained online with the processing, there is no need to store them for future processing since they are already processed right after being extracted from the source data. In this way, online techniques avoid overheads by supressing the need to store samples.

The confidence interval of the results is usually delegated to user control, who decides the most convenient balance between accuracy and speed of the queries. Regarding this, *online aggregation* is very popular. Online aggregation provides an initial approximate answer quickly to the user and

it is refined as more information is processed over the time. Then, the user chooses when the confidence level is satisfactory. One example of online aggregation is used in crowdsourcing techniques [122, 123].

- **Offline:**

Offline AQP creates synopses that contain samples of the source information. These synopses are usually stored in memory or cache and they are generated offline with the processing in a pre-computing process. Then, they are used to answer the online query.

It is necessary a previous knowledge about the structure, the meaning of the information and the queries before creating the synopses. This knowledge is used when deciding the criteria for the inclusion of the data in the samples. In this way, two methods are used for offline AQP:

- *Workload-free* synopsis: the information included in the samples is selected with uniform or stratified samples by analyzing the data. The first developed workload-free technique was the AQUA system [104], that selects a sample of all possible combinations of grouping columns.
- *Workload-aware* synopsis: the information included in the samples is selected based on the results of previous queries, assuming that future data will have a similar distribution. Note that this method generates a synopsis for each query. Some examples of this technique are Histogram [98], Wavelet [58], and Sketch [27], in addition to our SDR algorithm developed to work with graph-structured data [15]

In this thesis, both techniques were explored in order to get an acceptable performance working with streams of graph-structured information [14, 15]

### 2.3 Graph-structured information

Commonly, the information processed by Data Streaming Applications is represented as sequences of ordered events without any relation between them. However, in a real scenario, many of the events generated by different sources are organized in complex structures, such as graphs or trees. Some examples of these

sources are social networks (where users are connected to other users and they share pictures, posts, etc.) or e-commerce pages (where customers order products that belong to different departments, and products can be recommended to the customers in turn, etc.). The scope of this thesis focuses on graph-structured information flows. For this reason, this section comprises how to work with this kind of structures.

### 2.3.1 Models as graphs

A graph can have different definitions depending on the field of study (mathematics, computing, etc.). However, the most general definition is related to graph theory. Hinterberger [61] states the following: “A **graph** is a set of *nodes* (also called points or vertices) connected by *links* called lines or edges or arcs. In an undirected graph, a line from point A to point B is considered to be the same thing as a line from point B to point A. In a directed graph, the two directions are counted as being distinct arcs or directed edges”.

Based on this definition, a graph in computer science is a data structure that represents the information by means of nodes and edges. In this way, graph structures can be considered as a type of models since they are commonly used for representing the data of models, and for performing patterns on them. For this representation, in addition to nodes and edges, graphs are also composed of properties. Therefore, nodes represent *objects* in the model, edges represent *relationships* among them and properties represent *attributes* of both.

According to the definition of Hinterberger [61], relationships can be directional or bidirectional in a model too, and they can also be derived. In the approach presented in this thesis, a graph is composed of a set of objects and a set of relationships, and both represent the *elements* of the graph. Finally, graph patterns are structures used to manipulate graphs in the form of queries.

### 2.3.2 Graph processing platforms

Nowadays, graph processing platforms have become very popular to process large-scale graphs that represents datasets of a huge range of domains, such as social science, computational biology, telecommunications, etc. They are very efficient

to express and execute graph algorithms that are useful to obtain information of interest about the dataset (e.g. PageRank [96] or community detection [74]).

Regarding the types of graph processing platforms, different classifications have been presented in the literature according to different criteria [59, 105]. However, we mainly distinguish two categories in this dissertation:

- Distributed platforms: the graph is located and processed using several machines that are coordinated by one of them, called *master*. This solution usually allows a high scalability, since it can use multiple workers. However, it may exhibit a lower performance than single-node solutions due to distribution and overheads. Some popular examples of distributed graph processing platforms are Pregel [80] (a graph processing system developed by Google), PowerGraph abstraction [55] or the library Graphx [56] from Apache Spark.
- Single-node platforms: the graph is processed and located in a single machine with limited scalability. However, these solutions make an efficient use of the resources in order to get a competitive performance when processing the graph. The most common examples of using a single machine to process graphs are graph databases [128]. Graph databases are a type of NoSQL database designed to store, update and perform queries over graph structures. Several graph databases store the data in disk, such as Neo4j, while others implement in-memory graph databases, such as Memgraph [83] or TinkerGraph [118]. They use a graph-query language to perform queries, that provides an intuitive and fast way to access the information stored in the database. Some examples of graph-query languages are Gremlin [6], Cypher [88] and SPARQL [126]. However, even taking into account that graph databases are a common example of single-node platform, some of them allows distribution in several machines (e.g. Dgraph [47]).

In order to choose the most suitable technology to achieve the goals of this thesis, a detailed study of several platforms is presented in Chapter 3.

# Chapter 3

## Comparison and Performance

### Evaluation of Processing Platforms

Current companies and organizations increasingly demand the processing of data streams generated from different sources, such as social networks and streaming platforms. The main challenge of these applications is to provide efficient-enough responses when dealing with large amounts of real-time information. As mentioned in Chapter 1, despite this kind of information is sometimes represented as flows of single events without any relation among them, it is true that there are cases in which the information is composed of data related among them. In these cases, the sources generate data-structured information in the form of graphs, such as in social networks or geolocation systems domains. Furthermore, data of different nature that come from heterogeneous sources have to be considered. In this way, two types of information are normally taken into account in our approach: *persistent* and *transitory*. The former is permanently stored in the system (e.g. users in a social network or coordinates in a geolocation system), whereas the latter is temporarily stored and expires after some period of time (e.g. tweets or routes). This means



that we need to consider heterogeneous nature and structure of the information, which is indeed a challenge.

The approach presented in this thesis is based on the fact that not all the information that is stored in the system at the current time is useful to obtain valid conclusions. Since data-processing applications typically function by performing queries over the information, the proposal in this thesis is to improve the performance of these queries by selecting only relevant data. Then, it is necessary to find a technology and a language that meet three fundamental requirements: (i) they must allow to perform queries and update the information as quickly as possible in order to provide real-time responses, (ii) they must cope with graph-structured information, and (iii) they must provide a clear syntax in order to be able to study the type of query to be run over the data.

In this chapter, we analyse the benefits and limitations of several processing platforms and query languages when working with high volumes of data. The platforms comprise TinkerGraph [118], Neo4j [89], CrateDB [35], Memgraph [83], JanusGraph [64], OrientDB [29] and GraphFrames [106], whereas the languages comprise Gremlin [6], Cypher [88], SQL and the DSL used for GraphFrames [107]. All these technologies are commonly used in Big Data applications [82]. In order to study their performance, we tested them in two case studies and we measured the execution time of the queries in two scenarios. First, queries are run over the data in order to obtain useful information without implying any side effect in the source information. Second, same queries are run but their results modify the source graph by adding, removing or updating the existing information. In addition, the complexity of the languages is measured and compared by counting the number of characters, operators and internal variables used for each query. Finally, a combination of platform and query language is chosen to carry out our proposal, according to our requirements.

The contribution of this chapter is divided into three principal aspects. First, we evaluate the performance and scalability of the platforms by using two different case studies in different scenarios that work with graph-structured information. These scenarios comprise querying the graph in order to obtain information about the received data as well as modifying the source data as a result of the query over the graph. The scalability is evaluated by using models of different sizes that reach more than 10 million elements. Second, syntax complexity and intuitiveness of the

languages are analysed taking into account the most common operators used to work with graphs. Finally, we provide a comparative analysis that involves three kinds of platforms that work with graph-structured information, namely graph and relational databases and distributed platforms.

The structure of this chapter is as follows. First, in Section 3.1 we present a running example in the domain of social networks to illustrate our proposal. Then, Section 3.2 exposes 7 processing platforms that we compared in our study and their main features, whereas Section 3.3 presents the query languages that are used for these platforms. Platforms and languages are evaluated and compared in Section 3.4 in terms of execution time and complexity of the syntax, by using two case studies where one platform and one language are chosen as the most suitable combination for our proposal. Finally, in Section 3.5 we discuss related work and Section 3.6 summarizes the chapter.

### 3.1 A running example

In order to illustrate the DSL syntax used for each Graph Processing Platform studied in our proposal, consider a system that works with information provided by Flickr<sup>1</sup> and Twitter<sup>2</sup>, and analyses them together to identify some situations of interest [13]. The metamodel is depicted in Figure 3.1, where there is one common class between the two domains, namely **Hashtag**. Note that there is no automated manner to implement a completely reliable relation among the remaining elements (e.g. automatically relating users based on their Twitter and Flickr identifiers is impossible).

Given such a metamodel, we are interested in identifying the following situations of interest:

**Q1. HotTopic:** A hashtag has been used by both Twitter and Flickr users at least 100 times in the last hour. We would like to generate a new **HotTopic** object that refers to this hashtag.

**Q2. PopularTwitterPhoto:** The hashtag of a photo is mentioned in a tweet that receives more than 30 likes in the last hour. A **PopularTwitterPhoto** element is created.

---

<sup>1</sup><https://www.flickr.com/>

<sup>2</sup><https://twitter.com/>



**Q4. NiceTwitterPhoto:** A user, with an  $h$ -index<sup>3</sup> higher than 50, posts

**Q5. ActiveUserTweeted:** Considering the 10000 most recent tweets, search

Note that we have included 5 extra objects (shaded in gray) in the metamodel.

## 3.2 Processing platforms

As exposed in section 2.2, the information of Data Streaming Applications is real-time, which requires low-latency in the processing. Thus, we have studied seven powerful platforms for processing huge amounts of data. These platforms have been tested in two case studies with graph-structured information (explained in the following sections), in order to choose the technology with the lowest latency. In addition to this feature, other aspects have been taken into account when choosing the most suitable technology for our proposal, such as the DSL complexity or the possibility of updating the information.

In this section, an overview of the seven platforms is presented. These technologies include five graph databases (**Neo4j**, **JanusGraph**, **OrientDB**, **inkerGraph** and **Memgraph**), a distributed SQL database (**CrateDB**) and a package for working with distributed graphs provided by Apache Spark (**GraphFrames**). Their features are summarized in Table 3.1.

Platform	Distributed	In-memory	Disk	Updatable	Query languages
Neo4j	No	No	Yes	Yes	Cypher
JanusGraph	Yes	Yes	Yes	Yes	Gremlin
OrientDB	Yes	Yes	Yes	Yes	Gremlin, SQL
TinkerGraph	No	Yes	Yes	Yes	Cypher, Gremlin
Memgraph	Yes	Yes	Yes	Yes	Cypher
CrateDB	Yes	No	Yes	Yes	SQL
GraphFrames	Yes	Yes	No	No	GraphFrames DSL

Table 3.1: Processing platforms used in the experiments of the performance study

### 3.2.1 Neo4j

Neo4j [89] is an open-source graph database that combines native graph storage (since the data are stored as a graph and pointers are used to navigate and traverse its elements), ACID transaction compliance (that implies Atomicity, Consistency, Isolation and Durability) and a scalable architecture designed to perform queries quickly. Neo4j stores the data in disk and its features makes it suitable for production scenarios.

In order to provide an intuitive way to manipulate the database, Neo4j uses *Cypher* [88] as query language. This DSL is explained in Section 3.3.2 in detail.

Some domains where Neo4j is commonly used are financial services, manufacturing, technology companies, etc.

### 3.2.2 JanusGraph

JanusGraph [64] is an open-source scalable graph database designed to be distributed in a multi-machine cluster. This feature makes it suitable to store and query large graphs with hundred of billions of elements between edges and nodes. However, it also allows single-node configuration. JanusGraph can store the data in memory or in disk by making use of a backend with Apache Cassandra, Apache HBase, Google Cloud Bigtable, Oracle BerkeleyDB or ScyllaDB.

Among the most important benefits of JanusGraph we find support for global graph analytics through the Hadoop framework, concurrent transactions and operational graph processing as well as native support for Apache Tinkerpop [5] and *Gremlin* language [6]. This DSL is explained in Section 3.3.3 in detail.

In this technical report, we have implemented JanusGraph with the BerkeleyDB backend (disk storage).

### 3.2.3 OrientDB

OrientDB [29] is an open-source multi-model and NoSQL database that is written in Java and it provides both the power of graphs and the flexibility and scalability of documents into one high-performance operational database.

Among the benefits of OrientDB we find that it allows to distribute the data in a multi-machine cluster, it supports ACID transactions, it supports native management of graphs, HTTP, RESTful protocol, and JSON additional libraries. Besides, it is also compliant with Apache TinkerPop [5] and Gremlin [6] as well as SQL language. Both DSLs are thoroughly explained in Section 3.3. OrientDB supports four storage types: (i) plocal, that persists on disk where the access is made in the same JVM process; (ii) remote, that uses the network to access a remote storage; (iii) memory and (iv) local (this one has been deprecated and replaced by plocal).

Among the possible options provided by OrientDB, in this technical report, we have used memory storage and *Gremlin* language.

### 3.2.4 TinkerGraph

TinkerGraph [118] is a light in-memory graph database designed to run in a single machine, but it also includes the option to persist on disk. It provides Online Transactional Process (OLTP) and Online Analytics Process (OLAP) functionality. TinkerGraph is developed by Apache TinkerPop [5], an open-source graph computing framework used to model domains of data as graph structures. Due to its simplicity, it is commonly used as a reference to understand other methods of TinkerPop. However, it is also suitable for production.

TinkerGraph allows querying, modifying and updating the database using *Gremlin* [6] or *Cypher* [88] languages. Both DSLs are thoroughly explained in Section 3.3.

Some examples of use for TinkerGraph are analysis of large in-memory graphs, analysis of subgraphs extracted from very large graphs that can not be stored in memory, and modifying and transforming graphs (adding, removing or updating nodes, edges or properties).

### 3.2.5 Memgraph

Memgraph [83] is an in-memory graph database that allows ACID transaction compliance, multi-version concurrency control and asynchronous IO. These features lead into real-time responses when querying the information. Although it is implemented in C/C++, it is compatible with many existing languages such as Java, JavaScript, Ruby and PHP. Regarding graph manipulation, Memgraph uses *Cypher* [88] as DSL.

Some examples of use for Memgraph are creating business strategies adapted to the changing economy, transactional analysis, etc.

### 3.2.6 CrateDB

CrateDB [35] is a distributed database that integrates SQL language with certain benefits of NoSQL databases. In this way, it provides intuitiveness when

writing queries of SQL languages along with the scalability and flexibility of NoSQL. These features allow to perform queries in real-time over both independent data and complex data structures. Although CrateDB stores data in disk, it is able to perform queries at in-memory speed [36].

Some examples where CrateDB is often used are applications that require real-time data ingestion and backups, identification of special situations (such as trends or anomalies) by means of time series analysis and querying information for geospatial systems.

### 3.2.7 Apache Spark. GraphFrames

Apache Spark [108] is a framework that distributes the processing using a cluster in order to execute different operations in parallel. Spark divides the information into collections of elements partitioned and it spreads them through the cluster. These collections are called *resilient distributed datasets* (RDDs) and can be persisted in memory to be reused.

Regarding the graph-structured information processing, Spark provides a component for graph-parallel computation called *Graphx* [56]. It is a library that uses Spark RDDs to perform graph-related operations by introducing a set of basic operators, such as *subgraph*, *joinVertices* or *groupEdges*, in addition to operators for graph algorithms or analytic tasks, such as PageRank [96]. However, it does not provide any domain-specific language, but the code has to be developed with Scala or Python. This implies an important limitation for many systems, since it leads to the use of very complex patterns to define information queries [13]. For this reason, Apache Spark developed a package that overcame this problem, called *GraphFrames* [106]. It allows to operate with graphs using the benefits of Graphx but using *Spark DataFrames* instead of RDDs. A *DataFrame* is a distributed collection of data organized in columns labeled by names. It integrates the benefits of RDDs, regarding to information distribution, along with Spark SQL writing queries capabilities. This way, GraphFrames enable users to perform the same operations as Graphx, but in a more intuitive way because it uses DataFrames for handling the data.

### 3.3 Query Languages

Four DSLs have been stated in Section 3.2 in order to manipulate the data with the processing platforms. Now, this section presents an introduction to the most important concepts of their syntax. Note that, in our proposal, these languages are applied to graph-structured information. In the following, we present the different query languages considered in this comparison analysis and use basic examples of queries over the metamodel depicted in Figure 3.1 to clarify the concepts.

#### 3.3.1 SQL

SQL is an ANSI/ISO standard declarative query language designed to manipulate database information. It is based on RDBMS (Relational Database Management System), that allows to store the data into database objects called tables. Tables are collections of related data entries that are structured in columns and rows. In this way, rows are the horizontal entities of the table, that represents the individual entries, whereas columns are the vertical entities, that represents the information associated with each entry.

The most important feature of SQL is that it allows a wide variety of operations and high productivity in coding. SQL command functionalities include specifying integrity restrictions, specifying the limits of a transaction, accessing rights and defining relationship schemas and views. As well as other query languages, such as GraphFrame DSL or Gremlin, it is designed to be embedded into common programming languages, such as C++, C, Java, PHP or Fortran.

Since SQL is a standard, it is supported by most database systems, such as MySQL, Oracle or Postgres. In this thesis, we use SQL with CrateDB database [35] in order to store, query and update graphs. However, SQL does not support graph-structured information and, for this reason, the graph will be stored into tables. Therefore, in this section, we expose how graph-structured information is mapped into the tables in order to be queried and manipulated into a CrateDB database with SQL language.



Main components

As stated before, relational databases store the data into **tables**. Therefore, a CrateDB database may contain one or several tables that are identified by a name. Since graphs are mainly composed of nodes and edges, a possible solution for mapping graph structures into this database is to create two types of tables:

- Node tables: this kind of tables represents the nodes of the graph. They can be called with the label of the nodes they represent, i.e. there will be one node table for each node label. Besides, they will contain a column for each property node.

id	name	location
0	George B.	Dublin
1	Mike R.	London
2	Mary C.	Madrid
3	Cindy M.	Rome

(a) `TwitterUser` table

id	text
5	cold
6	winter

(b) `Hashtag` table

id	text	date
4	Winter is coming	12/12/2019

(c) `Tweet` table

src	dst
0	1
1	2
2	3
2	0

(d) `follows` table

src	dst
3	4

(e) `publishes` table

src	dst
4	5
4	6

(f) `contains` table

Table 3.2: TwitterFlickr example with SQL tables

- Edge tables: these kind of tables will have the same name as the label of the edge that they represent. An edge table will contain at least two columns that represent the source and destination nodes by means of foreign keys of the node tables. In addition, they may contain as many columns as edge properties.

As an example of this solution, Table 3.2 shows six tables that correspond with `TwitterUser`, `Hashtag` and `Tweet` nodes and `follows`, `publishes` and `contains` edges of the metamodel depicted in Figure 3.1. Note that edge tables contain two columns (`src` and `dst`) that represent the source and destination. For instance, first row of Table 3.2d represents an edge `follows` between `TwitterUser` called “George B.” (represented in first row of Table 3.2a) and the `TwitterUser` called “Mike C.” (represented in second row of Table 3.2a), which means that the user *George B.* follows the user *Mike C.*

Listing 3.1: SQL queries for graph-structured information

```

1 // creating tables with SQL
2 CREATE TABLE IF NOT EXISTS Hashtag (id LONG, text STRING);
3 // Basic SQL query
4 SELECT location FROM TwitterUser WHERE name="Mary C.";
5 // Quering graphs with SQL
6 SELECT tu.*, tu2.*
7     FROM TwitterUser tu, follows f, follows f2, TwitterUser tu2
8     WHERE tu.id = f.src
9     AND tu2.id = f.dst
10    AND tu2.name="Mary C."
11    AND f2.src = tu2.id
12    AND f2.dst = tu.id;
13 // inserting data with SQL
14 INSERT INTO Hashtag (id, text) VALUES (7,"winter");
15 INSERT INTO Tweet (id, text) VALUES (8,"Winter is coming");
16 INSERT INTO contains(src, dst) VALUES (8,7);
17 // adding a new property with SQL
18 ALTER TABLE TwitterUser ADD COLUMN description STRING;
19 // updating a property with SQL
20 UPDATE Hashtag SET text = "invierno" WHERE text = "winter";
21 // deleting a row with SQL
22 DELETE FROM Hashtag WHERE text = "invierno";
23 // removing a table with SQL
24 DROP TABLE IF EXISTS Hashtag;

```

SQL provides the command `CREATE TABLE` to create a new table in the database. In our approach, this command is used to create edge and node tables. Line 2 of Listing 3.1 shows an expression example for creating the table `Hashtag` with columns `id` and `text`. Note that columns must indicate a type of data depending on the property (`LONG`, `STRING`, etc.).

### Query basis

SQL queries are generally composed of three main commands:

- **SELECT**: this command extracts data from the database.
- **FROM**: it indicates the tables where the information is selected.
- **WHERE**: it filters the data selected with the **SELECT** command according to different conditions.

An example of SQL query is shown in line 4 of Listing 3.1, where the location of the user named “*Mary C.*” is returned. However, queries over graphs usually involve traversing edges and nodes. Since they are represented with tables in a CrateDB database, it is necessary to join those tables. CrateDB allows inner joins using the **WHERE** command. For instance, observe the query of lines 6 to 12 of Listing 3.1. This query joins tables **TwitterUser** and **follows** in order to obtain a user that follows the user named “*Mary C.*” and it is also followed by her. Note that the expression must traverse the edge **follows** twice (since we are interested in a *user that follows the user Mary C. and Mary C. follows this user too*). For this reason, the query contains four joins (**TwitterUser-follows** and **follows-TwitterUser** twice).

In addition, CrateDB allows to create new data into the database, as well as delete and update existing information. These functionalities will be exposed in the following sections.

### Inserting data

SQL provides the command **INSERT INTO** for inserting new data in the database. In this way, this command may be used to insert new records in edge or node tables. Lines 14 to 16 of Listing 3.1 show three examples of the use of this command when new rows are inserted in **Hashtag**, **Tweet** and **contains** tables. These insertions imply the creation of a new tweet with text “*Winter is coming*” that contains the hashtag “*winter*”.

In addition, it is possible to create a new property for nodes or edge with SQL by means of creating a new column in the table. For this purpose, SQL provides the

command `ALTER TABLE`. CrateDB allows to use this command with the keywords `ADD COLUMN` in order to add a new column to a table. However, this is only possible if the table is empty. In this way, considering that the `TwitterUser` table is empty, line 18 of Listing 3.1 adds the property *description* to this table.

### Updating data

In order to update the rows of a table, SQL provides the command `UPDATE`. This command allows to change the value of a property node or edge, by modifying the value of the corresponding column of its table. Line 20 of Listing 3.1 shows an expression to update the hashtag “*winter*” with the new text “*invierno*”. Note that the syntax of this expression is:

```
UPDATE table_name SET column_name = new_value WHERE condition
```

Where *new\_value* is the new value of the property and *condition* is a condition used to filter the properties to update.

### Deleting data

SQL provides two main commands to delete data in the database: `DELETE FROM` and `DROP`. The former is used to delete rows in a table whereas the latter is used to remove tables or columns in the database. Note that when removing columns from a table it is necessary to use the `DROP` command within the command `ALTER TABLE`. However, CrateDB does not support this expression. Therefore, it is only possible to delete tables and rows in a CrateDB database, which means to delete edge or node objects in the graph. As an example, expressions of lines 22 and 24 of Listing 3.1 are used to delete the hashtag “*invierno*” and table `Hashtag`, respectively.

### Filters and subqueries

As stated in the previous sections, we distinguish two types of filters when querying graph information. In this section, we explain how CrateDB treats these filters with SQL language:

Listing 3.2: SQL filters and aggregation operations for graph-structured information

```

1 // Filtering by property
2 SELECT * FROM Hashtag WHERE text="winter";
3 SELECT * FROM Hashtag;
4 // Filtering by subquery
5 SELECT t.*
6     FROM Tweet t,
7     (SELECT c.src
8         FROM contains c, hashtag h
9         WHERE c.dst = h.id AND h.text = "winter") sq
10    WHERE t.id = sq.src;
11 // Filtering by subquery with negation
12 SELECT t.*
13     FROM Tweet t,
14     (SELECT c.src
15         FROM contains c, hashtag h
16         WHERE AND c.dst = h.id AND h.text = "summer") sq
17    WHERE NOT t.id = ANY(sq.src);
18 // Filtering by subquery with conjunction
19 SELECT DISTINCT t.*
20     FROM Tweet t,
21     (SELECT c.src
22         FROM contains c, hashtag h
23         WHERE AND c.dst = h.id AND h.text = "cold") sq1,
24     (SELECT c.src
25         FROM contains c, hashtag h
26         WHERE AND c.dst = h.id AND h.text = "winter") sq2
27    WHERE sq1.src = t.id AND sq2.src = t.id;
28 // Filtering by subquery with disjunction
29 SELECT DISTINCT t.*
30     FROM Tweet t,
31     (SELECT c.src
32         FROM contains c, hashtag h
33         WHERE AND c.dst = h.id AND h.text = "spring") sq1,
34     (SELECT c.src
35         FROM contains c, hashtag h
36         WHERE AND c.dst = h.id AND h.text = "winter") sq2
37    WHERE sq1.src = t.id OR sq2.src = t.id;
38 //Aggregation operations
39 SELECT COUNT(*) FROM Tweet;
40 SELECT t.*
41     FROM Tweet t,
42     (SELECT c.src, count(c.src) countSRC
43         FROM contains c, Hashtag h
44         WHERE h.id = c.dst
45         GROUP BY c.src) sq
46    WHERE t.id = sq.src AND sq.countSRC >=3;

```

- Filtering by property: since edges and nodes are represented by tables, basic SQL queries can be used for this purpose. Some examples are lines 2 and

3 of Listing 3.2, where the hashtag *“winter”* and all hashtags are returned, respectively.

- **Filtering by subquery:** in this case, data is filtered according to different graph subqueries. CrateDB also allows SQL subqueries in order to filter the information. These subqueries are placed in the **FROM** clause and are treated as an independent table. As an example, observe lines 5 to 10 of Listing 3.2. The subquery contained in this query selects the source ids of all edges **contains** that have the hashtag *“winter”* as destination (lines 7 to 9). Then, it selects the tweets that match with these ids (line 10). However, several subqueries can be combined in order to filter the information. New combinations can be created by using the following operators:
  - **NOT:** it implies the negation of a subquery to select the data. An example is shown in lines 12 and 17 of Listing 3.2, where it selects all tweets that do not contain the hashtag *“summer”*. Note that the **ANY** clause indicates that there is no value that checks the condition.
  - **AND:** it implies the conjunction of two or more subqueries to select the data, i.e. the query must satisfy all conditions linked with this operator. An example is shown in lines 19 and 27 of Listing 3.2, where it selects all tweets that contain the hashtag *“winter”* and the hashtag *“cold”*.
  - **OR:** it implies the disjunction of two or more subqueries to select the data, i.e. the query must satisfy at least one of the conditions linked with this operator. An example is shown in lines 29 and 37 of Listing 3.2, where it selects all tweets that contain the hashtag *“winter”* or the hashtag *“spring”*.

## Aggregation Operations

Finally, CrateDB also allows aggregation operations with SQL language. According to our approach, this kind of operation is applied in two manners when querying graphs. In this section, we expose how SQL manages queries over graphs with aggregation operations:

- Aggregation as results: SQL provides several functions to obtain aggregated results (such as `count()`, `sum()`, `max()`, etc.). An example of query that uses `count()` operator is shown in line 39 of Listing 3.2, where the number of tweets of table `Tweet` are returned.
- Aggregation as filters: in this case, an aggregated result is used to filter the information. An example of this use is shown in lines 40 to 46 of Listing 3.2, that returns the tweets that contain at least 3 hashtags. Note that the aggregated result is obtained using the `GROUP BY` command, that groups the information according to one or more properties of the tables. In this case, data is grouped according to the `src` column of the `contains` table. Since this column corresponds with a tweet *id* value, it is possible to obtain the number of hashtags contained for each tweet, using `count()` function.

### 3.3.2 Cypher

Cypher is an open-source and declarative query language that allows to store and query data from graph databases. Its syntax and learning curve resemble those of SQL, but optimized for graph structures. Some of the most important features of this language are: efficiency to represent nodes and edges, constant time traversals (depth and breadth) in big graphs, flexibility to adapt the graph schema according to the needs, and availability of different drivers for Java, JavaScript, Python and other common programming languages.

Cypher is backed by openCypher project [91], that aims to provide an open language specification along with a technical compatibility kit. The language is inspired in SQL language and based on ASCII art. It provides an easier visual and logical syntax understanding and it makes matching patterns in graph easier too. In this section, we expose the most basic concepts of the syntax. However, many other operators and functionalities are available for Cypher. For more information about them, please refer to [88].

#### Main components

The main components of Cypher language are nodes, edges and properties.

**Nodes** are represented in the form of `(n:Node)`, where *Node* is the label of

the node and  $n$  is a variable to refer to that node. Note that the parentheses look similar to a circle, since it is the visual representation generally used for nodes in a graph. This circumstance brings intuitiveness when implementing queries in Cypher. Besides, variables are optional and they are used when the node needs to be referred later, providing more expressiveness to the language. Line 2 of Listing 3.3 shows a node representation example with variable  $tu$  and label *TwitterUser*.

Listing 3.3: Nodes, edges and properties in Cypher

```

1 // node
2 (tu:TwitterUser)
3 // node with property
4 (t:Tweet {text:'Winter is coming'})
5 // edge
6 (t:Tweet) -[r:CONTAINS]->(h:Hashtag)
7 // edge with property
8 (tu:TwitterUser) -[r:LIKES {date:'12/12/2020'}]->(t:Tweet)

```

**Edges** between nodes are represented in the forms of  $-[e:Edge]->$  or  $<-[e:Edge]-$ , depending on the direction of the edge. In this case, *Edge* is the label of the edge and  $e$  is the variable to refer to it (that can be optional too). In this case, Cypher is also intuitive since the arrow looks like the lines that commonly represent edges in a visual graph. Besides, it is possible to represent undirected edges removing the character ' $>$ ' or ' $<$ ', which means that the edge can be traversed in both directions in the query. Line 6 of Listing 3.3 shows an example of an edge with label *CONTAINS* from a node *Tweet* to a node *Hashtag*. This example can be read as follows: selects *tweets that contains hashtags*.

As stated in Section 2.3.1, edges and nodes can contain **properties**. In Cypher, properties are represented as name-value pairs surrounded by curly braces in the form of  $\{p:'property'\}$ , where  $p$  is the property name and *property* is its value. Properties may belong to nodes (represented in the form of  $(n:Node \{ p:'property'\})$ ) or edges (represented as  $-[e:Edge \{ p:'property'\}]->$ ). Lines 4 and 8 of Listing 3.3 show a node and an edge with a property, respectively.

In some cases, edge or node labels are not relevant for the query. Cypher allows to specify anonymous edges and nodes in the forms of  $-$ ,  $->$  or  $<-$  (for anonymous edges), and empty parentheses  $()$  (for anonymous nodes).



### Query basis

**Patterns** in Cypher are built as sequences of nodes and edges, that are called *paths*. A pattern can have a continuous path or several small patterns separated by commas. As an example, observe line 8 of Listing 3.3. This pattern represents a path that searches a *twitter user* that *likes* a *tweet* on *December 12th*. However, this pattern does not indicate the action we want to do with this pattern (inserting it, finding it in the database, etc.). In order to indicate the actions to carry out with a pattern, several **keywords** are used in a query. Cypher provides two main keywords to implement queries:

- **MATCH**: this keyword is similar to the keyword **SELECT** in SQL. It is used to search a pattern in the database. As an example, observe line 2 of Listing 3.4 that indicates that we want to search *tweets* that *contains* the *hashtag* with text “*Winter*”.
- **RETURN**: this keyword indicates the values to be returned by the query. In line 2 of Listing 3.4, nodes labeled with **Tweet** are returned by this query.

In addition to search data in the database, Cypher patterns allow to insert, update and delete data in the graph. Some other keywords are used to achieve this. We explain the most common cases in the following.

### Inserting data

Cypher provides the keyword **CREATE** to insert elements in the graph. It allows to insert nodes, edges or patterns into the database. **CREATE** can be used as main operation or as a result for a query. For instance, observe line 4 of Listing 3.4. In this query, a node labeled with **TwitterUser** that contains the name property “*Mary C.*” is created. **CREATE** is the main operation of the query. In lines 6 to 8, an edge is inserted between this node and the node labeled with **Tweet** with the text property “*Winter is coming*”. Therefore, **CREATE** is the consequence of the query composed by patterns of lines 6 and 7.

Listing 3.4: Cypher basics

```

1 //Basic pattern
2 MATCH (t:Tweet)-[r:CONTAINS]->(h:Hashtag {text:'Winter'}) RETURN t;
3 //Pattern to create a node
4 CREATE (tu:TwitterUser {name: 'Mary C.'}) RETURN tu;
5 //Pattern to insert an edge
6 MATCH (tu:TwitterUser {name: 'Mary C.'})
7     MATCH (t:Tweet {text: 'Winter is coming'})
8     CREATE (tu)-[r:PUBLISHES]->(t);
9 //Pattern to update a property
10 MATCH (h:Hashtag {text: 'Winter'})
11     SET h.text = 'Invierno'
12     RETURN h;
13 //Delete patterns
14 MATCH (h:Hashtag {text: 'Winter'})
15     DELETE h;
16 MATCH (t:Tweet)-[r:CONTAINS]->(h:Hashtag {text:'Winter'})
17     DELETE r;
18 MATCH (tu:TwitterUser {name: 'Mary C.'})
19     REMOVE tu.location;
20 MATCH (tu:TwitterUser {name: 'Mary C.'})
21     DETACH DELETE tu;

```

## Updating data

The keyword used to update an element is **SET**. It allows to modify a property of a node or edge. **SET** is used as a consequence of a **MATCH** clause, since it is necessary to search the element to be updated before modifying its property. As an example of the use of **SET**, observe lines 10 to 12 of Listing 3.4. In this query, the hashtag with the property “*Winter*” is searched in line 10. This node is updated modifying its property *text* in line 11 and finally it is returned in line 12.

## Deleting data

Cypher provides three keywords to delete data in the graph:

- **DELETE**: this keyword is used to delete nodes or edges in the graph. For example, observe in lines 14 and 15 of Listing 3.4 how the node labeled with **Hashtag** with the name property “*Winter*” is deleted. In addition, query of lines 16 and 17 deletes the edges that connect a tweet with the hashtag “*Winter*”.
- **REMOVE**: it is used to delete a property of a node or an edge. For example, query depicted in lines 18 and 19 of Listing 3.4 removes the property *location*

of the node labeled with `TwitterUser` and with name property “*Mary C.*”.

- **DETACH**: it allows to delete all edges connected to a node and the node itself. For instance, observe lines 20 and 21 of Listing 3.4. This query deletes the node labeled with `TwitterUser` and with name property “*Mary C.*” and all edges that are connected to it.

### Filters and subqueries

Queries usually contain filters to select, delete, insert or update the information of the database. In Cypher, the main operator used to filter the data is **WHERE** clause. In our approach, we distinguish two main types of filters with this clause:

- **Filtering by property**: in this case, elements are searched according to a property value. For example, observe lines 2 to 4 in Listing 3.5. This query returns the *tweets* that *contains* the *hashtag* “*Winter*”. Note that this query obtains the same result from the query depicted in line 2 of Listing 3.4. Therefore, this type of filter can be usually replaced by a property in the main **MATCH** clause.
- **Filtering by subquery**: in this case, a function `exists()` appears after the **WHERE** clause that contains the subquery used to filter the information. For instance, observe the query of lines 6 to 8 in Listing 3.5. In this case, the main **MATCH** clause (line 6) selects nodes labelled with `Tweet`, whereas the **WHERE** clause filters the tweets that contain the hashtag “*Winter*”. Note that this query has also the same result as the query depicted in line 2 of Listing 3.4 and query depicted in lines 2 to 4 of Listing 3.5. Therefore, it is possible to simplify the query replacing the **WHERE** clause for a property filter in the main **MATCH** clause. This simplification is possible when the query conditions only involve one subquery. However, some queries filter by two or more subqueries (e.g. obtaining the tweets that contain the hashtag “*Winter*” or “*Spring*”) or they imply the non-existence of a path between two nodes (e.g. obtaining the tweets that do not contain the hashtag “*Summer*”). For these cases, the **WHERE** clause is combined with the following keywords:

- **AND**: it implies the conjunction of two or more subqueries. For example, observe lines 10 to 13 of Listing 3.5. In this query, the **WHERE** clause filters the nodes labelled with **Tweet** (line 10) that satisfy two conditions: (i) they contain the hashtag “*Winter*” (line 11) and (ii) they contain the hashtag “*Cold*” (line 12).
- **OR**: it implies the disjunction of two or more subqueries. For example, observe lines 15 to 18 of Listing 3.5. In this query, the **WHERE** clause filters the nodes labelled with **Tweet** (line 15) that satisfy at least one of two conditions: (i) they contain the hashtag “*Winter*” (line 16) or (ii) they contain the hashtag “*Spring*” (line 17).
- **NOT**: it implies the negation of a subquery. For example, observe the query depicted in lines 20 to 22 of Listing 3.5. The **WHERE** clause filters the nodes labelled with **Tweet** (line 20) that do not contain the hashtag “*Summer*” (line 21).

## Aggregation Operations

Some queries contain aggregation operations, such as counts, sums, maximum or minimum. In our approach, we have classified the use of these operations in two groups:

- Aggregation operations as results: in this case, the goal of the aggregation operation is to obtain an aggregated result of some aspects of the database. As an example of this use, observe line 24 of Listing 3.5. Note how this query returns a count of all nodes labelled with **Tweet** in the graph.
- Aggregation operation as filters: in this case, the aggregation operation is used to filter the information of the database in order to obtain a result that does not need to contain the result of this operation. For instance, observe lines 25 to 28 of Listing 3.5. First, this query selects all nodes labelled with **Tweet** and the number of hashtags they contain (lines 25 and 26). Second, the query filters by the number of hashtags contained for each node **Tweet** (line 26). Finally, it returns the nodes **Tweet** that satisfy this condition (line

Listing 3.5: Filtering, subqueries and aggregated patterns in Cypher

```

1 //Filtering by property
2 MATCH (t:Tweet)-[r:CONTAINS]->(h:Hashtag)
3   WHERE h.text = 'Winter'
4   RETURN t;
5 //Filtering by subquery
6 MATCH (t:Tweet)
7   WHERE exists((t)-[r:CONTAINS]->(h:Hashtag{text:'Winter'}))
8   RETURN t;
9 //Filtering by subquery with conjunction
10 MATCH (t:Tweet)
11   WHERE exists((t)-[r:CONTAINS]->(h:Hashtag{text:'Winter'}))
12   AND exists((t)-[r:CONTAINS]->(h2:Hashtag{text:'Cold'}))
13   RETURN t;
14 //Filtering by subquery with disjunction
15 MATCH (t:Tweet)
16   WHERE exists((t)-[r:CONTAINS]->(h:Hashtag{text:'Winter'}))
17   OR exists((t)-[r:CONTAINS]->(h2:Hashtag{text:'Spring'}))
18   RETURN t;
19 //Filtering by subquery with negation
20 MATCH (t:Tweet)
21   WHERE NOT exists((p)-[r:CONTAINS]->(h:Hashtag{text:'Summer'}))
22   RETURN t;
23 //Aggregation patterns
24 MATCH (t:Tweet) RETURN count(*);
25 MATCH (t:Tweet)-[:CONTAINS]->(h:Hashtag)
26   WITH t, size(collect(h.text)) AS hashtagsList
27   WHERE hashtagsList >= 3
28   RETURN t;

```

28). Note how this query uses the aggregation operation as a filter instead of as a result.

### 3.3.3 Gremlin

Gremlin [6] is a vendor-agnostic, graph traversal language that enables users to express complex traversal queries and manipulate graph databases. Gremlin can be written in imperative, declarative or hybrid manner. This feature provides flexibility to users when expressing queries and it allows to efficiently evaluate traversals.

In contrast to other query languages, such as SQL or Cypher, Gremlin was designed to be embedded in common programming languages. Therefore, it can be represented using their constructs. Thus, several Gremlin language variants are available for the most common programming languages, such as Gremlin-Java, Gremlin-Python or Gremlin-Scala.

Gremlin is distributed by Apache TinkerPop [5], an open-source graph computing framework that allows Online Transactional Process (OLTP) and Online Analytics Process (OLAP). This way, it provides mechanisms to work with both transactional in-memory graph databases or multi-machine distributed graph databases.

Many operations and functionalities are available for Gremlin language [118]. However, we expose the most basic concepts of the Gremlin syntax in the following.

### Main components

Gremlin distinguishes two parts in the graph computing: (i) the structure, that represents the model defined by nodes, edges and properties, and (ii) the process, that corresponds to the means by which the structure is analyzed (typically called traversals). Consequently, there are different components depending on the context:

- Components in the *structure* context. As stated in Section 2.3.1, graphs are composed by nodes (or vertices), edges and properties. For this reason, since Gremlin information is graph-structured, the main components of the structure are the following:
  - **Graph**: it is composed by a set of vertices and edges.
  - **Element**: elements are property collections and a string label that refers to the element type. **Element** has two inheriting components: **Vertex** and **Edge**. Vertices maintain a set of (incoming or outgoing) edges whereas edges maintain a set of (incoming or outgoing) vertices.
  - **Property<V>**: properties are pairs of key-value (being **V** the value of the property) and may belong to vertices or edges.
- Components in the *process* context. As previously mentioned in this section, Gremlin allows OLAP and OLTP. According to this feature, the process can have two main components:
  - **TraversalSource**: it is used to generate **Traversal** instances for a particular graph. A **Traversal** instance represents a directed walk over a graph. In this sense, **TraversalSource** is OLTP oriented.

Depending on the domain, the graph can be represented using different concepts (e.g. social graphs can be represented by people and cities). According to this, Gremlin provides `GraphTraversalSource` and `GraphTraversal`, that are a traversal source and traversal DSL oriented to the semantics of the graph (where concepts are vertices, edges, etc.), respectively.

- **GraphComputer**: it is used to process the graph in parallel and distributed over a multi-machine cluster. In this way, **GraphComputer** is OLAP oriented. **GraphComputer** uses two components: **VertexProgram** and **MapReduce**. First, **GraphComputer** executes **VertexProgram** component over all vertices in the graph in parallel (with intercommunication via message passing). Second, it executes a set of **MapReduce** jobs over the vertices to obtain a single result.

### Query basis

A Gremlin query expression is composed by a traversal (`GraphTraversal` in the domain of graphs) and, as said before, traversals are generated from a traversal source (`GraphTraversalSource` in graph domain). In this way, `GraphTraversalSource` provides two methods to start a `GraphTraversal`:

- `GraphTraversalSource.V(Object... ids)`: it generates the traversal starting from the vertices of the graph indicated by the objects `ids`.
- `GraphTraversalSource.E(Object... ids)`: it generates the traversal starting from the edges of the graph indicated by the objects `ids`.

Notice that the objects `ids` are optional and, therefore, if they are not specified, the traversal is generated starting from all vertices or edges in the graph. In order to clarify the process of generating a traversal, observe the fragment of the Gremlin console shown in Figure 3.2. First, a toy graph example is loaded in the variable `graph`. Second, a `GraphTraversalSource` is obtained from this graph using the method `traversal()` and it is stored in variable `g`. Then, since `g` is a `GraphTraversalSource`, it is able to generate a `GraphTraversal` using `E()` or `V()` methods.

```

      \, , , /
      (o o)
-----o00o-(3)-o00o-----
plugin activated: tinkrpop.server
plugin activated: tinkrpop.utilities
plugin activated: tinkrpop.tinkergraph
[gremlin> graph = TinkerFactory.createModern()
==>tinkergraph[vertices:6 edges:6]
[gremlin> g = graph.traversal()
==>graphtraversalsource[tinkergraph[vertices:6 edges:6], standard]
[gremlin> g.V()
==>v[1]
==>v[2]
==>v[3]
==>v[4]
==>v[5]
==>v[6]
[gremlin> g.E()
==>e[7][1-knows->2]
==>e[8][1-knows->4]
==>e[9][1-created->3]
==>e[10][4-created->5]
==>e[11][4-created->3]
==>e[12][6-created->3]

```

Figure 3.2: GraphTraversalSource and GraphTraversal in Gremlin console

Once the traversal is generated, it is possible to implement a query. **GraphTraversal** supports function composition, which means that it maintains many methods that return a **GraphTraversal** in turn. These methods are called *steps* and each step modulates the results of the previous one. Depending on the effect that each step has over the previous step, it can be classified into three main types:

- *Map steps*: they transform the resulting objects of the previous step to other objects. The most common map steps are `out()` and `in()` methods, that are used to get the vertices that are outgoing and incoming adjacent to the vertices of the previous step.
- *Filter steps*: they remove objects of the result of the previous step. Some examples of filter step are `has()`, `and()`, `or()`, `where()` and `not()` methods, that will be explained in the following sections in detail.
- *SideEffect steps*: they yield some computational side effect in the process. A common example of sideEffect step is `groupCount()` method, that will be thoroughly explained in the following sections.

Finally, the **GraphTraversal** must be executed to obtain a final result. To achieve this, Gremlin provides terminal steps. Some examples of terminal steps



are `first()`, that returns the first object of the query result, or `toList()`, that returns a list with all objects of the query result.

Listing 3.6: Gremlin basis

```

1 //Basic Query
2 graph.traversal().V()
3   .hasLabel("Tweet")
4   .in("contains")
5   .toList();
6 //Traversals for creating vertices
7 v1= g.addV("Hashtag").property("text","winter").next();
8 v2= g.addV("Tweet").property("text","Winter is coming").next();
9 //Traversal for creating an edge
10 g.V(v2).addE("contains").to(v1).iterate();
11 //Traversal for updating a property
12 v1.property("text","invierno").next();
13 //Traversals for deleting
14 g.V().hasLabel("Tweet").drop();
15 g.E().drop();
16 g.V().properties("text").drop();

```

In summary, a query in Gremlin is mainly composed of three parts: (i) a `GraphTraversalSource`, (ii) a `GraphTraversal`, generated from the `GraphTraversalSource` and composed by steps, and (iii) a result. As a basic example, observe lines 2 to 5 of Listing 3.6. This query generates the traversal from a traversal source (expression `graph.traversal()`) starting from the vertices (line 2). Besides, it filters vertices that have the label *Tweet*, using a `hasLabel()` step (filter step). Then, it gets the vertices that are incoming adjacent to them (line 4) via *contains* edge (map step). Finally, the query returns all the results in a list (line 5) with `toList()` method (terminal step). In this way, the source of the traversal is composed by `graph.traversal()` expression, the traversal is composed by the expressions of lines 3 and 4, and the result corresponds to line 5.

In addition to obtaining results, queries in Gremlin allow to insert, update and delete vertices, edges or properties in the graph. We explain these operations in the following as well as the most common methods to implement queries.

### Inserting data

Gremlin allows to add new vertices, edges or properties to the graph by means of methods. These methods are `map/sideEffect` steps, since they transform the

object of the previous graph to another object and imply a side effect in the database. The three main methods to insert data in the database are the following:

- **addV()**: it allows to create a new vertex in the graph, indicating its label between the parenthesis. Two examples of the use are shown in lines 7 and 8 of Listing 3.6<sup>4</sup>. First traversal (line 7) adds a vertex labelled as *Hashtag* to the database, whereas second traversal (line 8) adds a vertex labelled as *Tweet*. Note that properties are not settled with this method. However, Gremlin provides the method **property()** to add them to the vertices.
- **property()**: this method allows to create (or update) a vertex or edge property. The property is written between the parenthesis of this method as a key-value pair separated by a comma. In lines 7 and 8 of Listing 3.6, the property with the key *text* and the values “*winter*” and “*Winter is coming*” are added for the new hashtag vertex and tweet vertex, respectively.
- **addE()**: this method is used to add a new edge between two vertices in the graph. As an example, observe line 10 of Listing 3.6. This pattern creates an edge labelled as *contains* from vertex *v2* (added in line 8) to vertex *v1* (added in line 7).

Note that these methods have to be followed by a terminal step, to execute the traversal.

### Updating elements

Graphs can also be updated by modifying vertices or edges properties. As stated in the previous section, the **property()** method allows to create a new property or update it if its key exists for that vertex or edge already. As an example, observe line 12 of Listing 3.6, where a property *text* is settled with the value “*invierno*” for vertex *v1* (added in line 7). Since this vertex already had that property, the method will change its value from “*winter*” to “*invierno*”. Note how, in this case, it is also necessary to add a terminal step at the end of the traversal, in order to execute the property updating.

<sup>4</sup>We assume that *g* represents the traversal source from now on.

## Deleting elements

In order to delete vertices, edges or properties in the graph, Gremlin provides the method `drop()`. This method is a map/sideEffect step, and thus it does not need a terminal step to execute its side effect. Lines 14 to 16 of Listing 3.6 show an example for deleting vertices, edges and properties, respectively. Then, the traversal of line 14 deletes all vertices labelled as *Tweet* in the database, the traversal of line 15 deletes all edges from the graph and, finally, the traversal of line 16 deletes the property with key *text* for all vertices in the graph. Note that in the example of line 16 the method used to select the properties is `properties()` (map step) instead of `property()`. The `properties()` method is used to extract one or more properties from the object of the previous step, whereas `property()` method is used to add or update a property to the object of the previous step.

## Filters and subqueries

Queries usually involve filters in order to remove non-relevant information from the graph. According to our approach, and similarly to the Cypher filters classification, we differentiate two types of filtering:

- Filtering by property: vertices and edges can be filtered according to their property values. This way, Gremlin allows several methods and operations to achieve this purpose. The most common method used to filter information by a property value is `has()` method (filter step). This method allows to filter the objects of the previous step that contain a specific value for a property. This property is specified as a key-value pair inside the parenthesis and separated by a comma. As example is shown in line 2 of Listing 3.7. A query filters all vertices of the graph that have the value “*winter*” for the property *text*. An important variance of `has()` method is `hasLabel()`, that is used to filter objects by their label. An example of this method is shown in line 3 of Listing 3.7, where vertices with label *Hashtag* are filtered. In addition, there are more variances from `has()` method [114], such as `hasId()` or `hasKey()`.

Listing 3.7: Filtering, subqueries and aggregated operations in Gremlin

```

1 //Filtering by property
2 g.V().has("text","winter").next();
3 g.V().hasLabel("Hashtag").next();
4 //Filtering by subquery
5 g.V().hasLabel("Tweet").where(
6     __.out("contains").has("text","winter")).next();
7 //Filtering by subquery with negation
8 g.V().hasLabel("Tweet").not(
9     __.out("contains").has("text","summer")).next();
10 //Filtering by subquery with conjunction
11 g.V().hasLabel("Tweet").and(
12     __.out("contains").has("text","winter"),
13     __.out("contains").has("text","cold")).next();
14 //Filtering by subquery with disjunction
15 g.V().hasLabel("Tweet").or(
16     __.out("contains").has("text","winter"),
17     __.out("contains").has("text","spring")).next();
18 //Aggregation operations
19 g.V().hasLabel("Tweet").count();
20 g.V().hasLabel("Hashtag").in("contains").hasLabel("Tweet")
21     .groupCount().unfold()
22     .where(__.select(values).is(P.gte(3))).next();

```

- Filtering by subquery: in this case, one or more subqueries are used to filter the objects in the traversal. So, the previous objects of the filter step must satisfy the conditions imposed by this step. There are many Gremlin methods that involve filtering by subqueries [118], however, we highlight the following since they are the most significant for our approach:
  - **where()**: this method checks that the objects of the previous step follow a specific path (described by a subquery inside the parenthesis) to be selected in the filtering. As an example, lines 5 and 6 of Listing 3.7 show a query that filters the tweets that contain the hashtag “winter”. In this way, those vertices must have the path described in line 6.
  - **not()**: in this case, the method implies the negation of a subquery. In other words, it checks that the objects of the previous step do not follow the path contained between the parenthesis to be selected in the filtering. Lines 8 and 9 of Listing 3.7 show an example of the use of this method. This query obtains all tweets in the graph that do not contain the hashtag ‘summer’. In this way, vertices can not have the

path described in line 9 to be included in the filtering.

- **and()**: this method has the same behaviour as **where()** method but it implies the conjunction of two or more subqueries in the filter. In this way, the objects of the previous step have to satisfy all subqueries contained between brackets. For example, the query of lines 11 to 13 of Listing 3.7 filters all tweets that contain the hashtags *winter* and *cold*.
- **or()**: similar to **and()** method, this method includes two or more subqueries in the filtering. However, it implies the disjunction of these subqueries i.e. the objects of the previous step have to satisfy at least one subquery contained between brackets. For example, lines 15 to 17 of Listing 3.7 show a query that filters the tweets that contain the hashtags *winter* or *spring*.

Note that these methods are also used to filter by property [113, 115, 116, 117]. However, our approach uses the **has()** method and its variants.

### Aggregation Operations

Aggregation operations are commonly used to implement queries. They imply a map step that transforms the objects into aggregated results. Some examples of these operations are counting objects or summing property values. Similar to Cypher language, we classify the use of aggregation operations in two manners:

- Aggregation as results: Gremlin provides several methods to obtain aggregated results, such as **count()**, **max()**, **min()**, etc. All of them are map steps. As an example, observe line 18 of Listing 3.7, where the query counts the number of tweets contained in the database.
- Aggregation as filters: the aggregation is used to filter the information. A common method for this purpose is **groupCount()** (map/sideEffect step). This method counts how many times the previous object appears in the traversal and returns a list with the objects and the number of appearances for each object. This list can be filtered in turn, in order to obtain just a part of the objects that it contains. As an example, observe lines 20 to 22 of

Listing 3.7. This query counts the number of hashtags for each tweet (lines 20 to 21) and it returns only those tweets that contain at least 3 hashtags (line 22).

### 3.3.4 GraphFrames

GraphFrames [106] is a package for Apache Spark that allows to work with graphs using DataFrames. It provides powerful tools to implement and run queries and algorithms, that are commonly used for graph analytics (e.g. PageRank [96]).

As stated in Section 3.2.7, DataFrames integrates the benefits of Spark RDDs and Spark SQL. Therefore, GraphFrames syntax is inspired in SQL language, which means that it provides a logical syntax that is easy to understand. Similar to Gremlin language, GraphFrames are designed to be embedded in common programming languages. Consequently, it provides APIs for Scala, Java and Python.

In this section, we expose the most common functionalities of GraphFrames. However, many more operators are provided by Apache Spark for this package. These operators can be found in [107].

#### Main concepts

GraphFrames are generally created from two DataFrames:

- **Vertex DataFrame:** this DataFrame contains at least one column called *id*, which specifies an unique ID for each vertex in the graph.
- **Edge DataFrame:** it contains at least two main columns: (i) *src* column, that refers to the source vertex ID, and (ii) *dst* column, which refers to the destination vertex ID.

In addition, both components can have arbitrary number of other columns, that represent vertex and edge properties. In this way, GraphFrames are created by indicating a vertex DataFrame and an edge DataFrame. Listing 3.8 shows an example that creates a GraphFrame from a vertex DataFrame and an edge DataFrame with Scala code. This example represents a simplified version of the TwitterFlickr case study that contains four *TwitterUser* vertices, one *Tweet* and

two *Hashtag* vertices. From lines 2 to 10 the vertex DataFrame is created from a list that contains the vertices of the graph. Each vertex has five properties represented by columns (*label*, *name*, *location*, *text* and *date*) and the column *id*. Note that some property values are empty strings. This is because these properties do not belong to the object of that vertex instance, as can be observed in the metamodel depicted in Figure 3.1. For instance, observe how the object *TwitterUser* does not have a property *text*. However, since DataFrames are collections of data organized as columns and all vertices of the graph are represented in the same DataFrame, they have the same structure and the same properties. In this way, the properties that are not included in the metamodel for a specific object will be represented as empty strings in its tuples.

Listing 3.8: GraphFrames basic

```

1 // Vertex DataFrame
2 val v = sqlContext.createDataFrame(List(
3   (0, "TwitterUser", "George B.", "Dublin", "", ""),
4   (1, "TwitterUser", "Mike R.", "London", "", ""),
5   (2, "TwitterUser", "Mary C.", "Madrid", "", ""),
6   (3, "TwitterUser", "Cindy M.", "Rome", "", ""),
7   (4, "Tweet", "", "", "Winter is coming", "12/12/2019"),
8   (5, "Hashtag", "", "", "cold", ""),
9   (6, "Hashtag", "", "", "winter", "")
10 )).toDF("id", "label", "name", "location", "text", "date")
11 // Edge DataFrame
12 val e = sqlContext.createDataFrame(List(
13   (0, 1, "follows"),
14   (1, 2, "follows"),
15   (2, 3, "follows"),
16   (2, 0, "follows"),
17   (3, 4, "publishes"),
18   (4, 5, "contains"),
19   (4, 6, "contains")
20 )).toDF("src", "dst", "relationship")
21 // Create a GraphFrame from vertex and edge DataFrames
22 val g = GraphFrame(v, e)

```

On the other hand, in lines 12 to 20, the edge DataFrame is created and it represents the connections among the vertices contained in vertex DataFrame. Edge DataFrame contains *src* and *dst* columns and an additional column to describe the label of the edge (column *relationship*). Finally, in line 22, the GraphFrame is created from *v* and *e* values.

## Query basis

GraphFrames uses a simple DSL for expressing queries. Since this DSL is SQL based, it is very similar to Cypher. More specifically, GraphFrame queries are usually built from two methods:

- **find()**: this method indicates the path to find in the graph in the form of small patterns separated by semicolons. Similar to Cypher syntax, nodes are represented by a label referring to the node inside parenthesis (e.g. `(n)`), whereas edges are represented in the form of `-[e]->`, where `e` is also a label referring to the edge.
- **filter()**: this method filters the information according to the properties of nodes and edges contained in the pattern of **find()** method. For this purpose, it uses the labels settled in **find()** method to refer to nodes and edges.

An example of query over a GraphFrame is depicted in lines 2 to 5 of Listing 3.9. This query searches the users that follow the user with name “*Mary C.*” and that are also followed by her. It returns a DataFrame with four columns called *a*, *e*, *b* and *e1*. Note how the pattern is specified in the **find()** method separated by semicolons, whereas the property filters are specified in **filter()** method separated by **AND** operators.

In addition, GraphFrames queries present some other similarities with Cypher language such as they do not identify distinct elements with different labels, i.e. two elements with different labels may refer to the same graph element. For example, in the patterns `(a)-[e]->(b)` and `(b)-[e1]->(c)`, `a` and `c` may refer to the same node. However, a restriction can be added to the **filter()** method to avoid this situation (such as `a.id != c.id`). Furthermore, GraphFrame queries can also express anonymous edges or nodes by indicating them with empty brackets or empty parenthesis, respectively.

However, since GraphFrame does not allow to modify the graph, it does not provide methods to insert, delete or update the information. Therefore, if an update of the information is needed, it is necessary to create a new graph and override the existing one.



## Filters and subqueries

Many complex queries can be expressed using `filter()` and other similar methods. Within the framework of our proposal, GraphFrame queries represent two types of filters:

- Filtering by property: nodes and edges can be filtered by their property values. To achieve this, GraphFrames provide two methods, in addition to `filter()` method exposed in the previous section: (i) `filterVertices()` and (ii) `filterEdges()`. These methods allow to filter the graph according to the property value of nodes or edges respectively. Two examples of these methods are shown in lines 7 and 8 of Listing 3.9. The query of line 7 selects the vertices that contain the property *text* with the value “winter”, whereas the query of line 8 selects the graph composed by the edges with the value “follows” for the property *relationship*.
- Filtering by subquery: in this case, the information is filtered according to one or several patterns. Unlike other query languages, GraphFrames do not provide methods to filter by subquery. This type of filtering must be performed by describing the pattern using AND and OR operations over properties and edges in `filter()` method. In order to follow the same line of work for all query languages exposed in this chapter, lines 10 to 30 show the GraphFrame version of the queries exposed for `where()`, `not()`, `and()` and `or()` operators in Cypher. In this way, lines 10 to 13 show a query that filters according to a single pattern, whereas lines 15 to 18 show a query that filters according to the non-existence of a pattern. Furthermore, queries of lines 20 to 25 and lines 27 to 30 filter the information according to two different patterns. However, the former implies the conjunction of both patterns to filter the information, whereas the latest implies the disjunction of the patterns.

## Aggregation Operations

GraphFrames also provides methods to express aggregation operations. According to our approach, these operations can be applied in two manners:

Listing 3.9: GraphFrames queries

```

1 // Basic query
2 g.find("(a)-[e]->(b); (b)-[e1]->(a)")
3   .filter("e.relationship = 'follows'
4         AND e1.relationship = 'follows'
5         AND b.name='Mary C.'");
6 //Filter by property
7 g.filterVertices("text = 'winter'");
8 g.filterEdges("relationship = 'follows'");
9 //Filter by subquery
10 g.find("(a)-[e]->(b)")
11   .filter("a.label = 'Tweet'
12         AND e.relationship = 'contains'
13         AND b.text = 'winter'");
14 //Filter by subquery with negation
15 g.find("!(a)-[]->(b)")
16   .filter("a.label = 'Tweet'
17         AND e.relationship = 'contains'
18         AND b.text = 'summer'");
19 //Filter by subquery with conjunction
20 g.find("(a)-[e]->(b); (a)-[e1]->(c)")
21   .filter("a.label = 'Tweet'
22         AND e.relationship = 'contains'
23         AND e1.relationship = 'contains'
24         AND b.text = 'winter'
25         AND c.text = 'cold'");
26 //Filter by subquery with disjunction
27 g.find("(a)-[e]->(b)")
28   .filter("a.label = 'Tweet'
29         AND e.relationship = 'contains'
30         AND (b.text = 'winter' OR b.text = 'cold')");
31 //Aggregation operations
32 g.vertices.filter("label = 'Tweet'").count();
33 g.find("(a)-[e]->(b)")
34   .filter("a.label = 'Tweet'
35         AND e.relationship = 'contains'
36         AND b.label = 'Hashtag'")
37   .groupBy("a").count().select("a").where("count >= 3");

```

- Aggregation as results: GraphFrames syntax provides several methods to obtain aggregated results, such as `count()`, `min()`, `max()`, `inDegrees` (it counts the number of incoming edges) or `outDegrees` (it counts the number of outgoing edges). An example of query that uses a method to obtain an aggregated result is shown in line 32 of Listing 3.9, where the number of tweets in the graph is returned.
- Aggregation as filters: same as for Gremlin and Cypher languages, GraphFrames syntax allows to filter by aggregation operators. For this purpose, `groupBy()` method is used. This method groups the information according

to a node, edge or property indicated between the parenthesis. An example of this use is depicted in lines 33 to 37 of Listing 3.9, where the information is filtered by the tweets that contain at least 3 hashtags. Note that after the use of `groupBy()` method, the method `count()` is used to count the number of occurrences of a node in the pattern, and `where()` method selects those that occur at least three times.

### 3.4 Performance Analysis and Evaluation

In this section, we discuss the performance of technologies and DSLs proposed in Section 3.2 and Section 3.3. In order to study the application of queries in very-large graphs in terms of performance and complexity of the syntax, we have selected two case studies from the literature [13, 111]. In this approach, performance is understood as the execution time of the queries and complexity of the syntax refers to how verbose and difficult to write typical graph queries are, depending on the DSL.

#### 3.4.1 Research Questions

In order to choose the technology that suits our proposal in a better way, we are interested in answering the following research questions:

- **RQ1: How is the latency of each proposed technology when performing queries over very large graphs?** Our hypothesis is that all technologies allow to get execution times for responses that make them suitable for real-time processing. However, we are interested in knowing an approximate average value of the execution times of the queries on very large models in order to compare all technologies. We consider that a very large model contains at least 2 million elements between objects and relationships. In addition, since real-time processing imposes a continuous processing of the data, we consider that the most suitable technology for these types of applications will be the one that takes less time to get the results (with a scale of seconds at most).
- **RQ2: Which overhead implies modifying the graph as result of the query for each proposed technology?** Queries in our approach do

not simply return nodes in the graph, but they typically have a side effect on the graph, such as adding, deleting or updating information in the model. For this reason, we are interested in measuring the execution times of these side effects over the graph. Our hypothesis is that they do not imply a big overhead regarding the whole execution time of the query, meaning these technologies could be used for real-time processing.

- **RQ3: How is the syntax complexity for each proposed DSL?** Our hypothesis is that DSLs for graph databases provide less complexity when querying graphs than the rest of DSLs exposed in this chapter, since they are designed to manipulate graphs by replacing table joins of relational databases by mechanisms to traverse relationships between objects. However, we are interested in studying which language allows to write queries in a simpler manner.
- **RQ4: Which DSL and processing platform provide the best combination regarding complexity of the language and latency of the query?** Since low-latency is one of our most important requirements, we choose the technology for our proposal according to this parameter. Besides, an expressive DSL is needed in order to classify and write queries over graphs. Thus, both features will be compared for all proposed technologies in order to select the most appropriate.

### 3.4.2 Case studies

This section describes two examples that have been used to evaluate and compare the platforms and DSLs exposed in Section 3.2 and Section 3.3. These examples were chosen to capture different situations that can be interesting in real scenarios since they deal with very-large graphs. In addition, queries cover the main concepts of the query languages explained in Section 3.3.

#### TwitterFlickr case study

The first case study corresponds with the metamodel and queries of the running example exposed in Section 3.1. This example was proposed in a previous work [13],

where an extension of a CEP system was presented to work with graph-structured information.

### TrainBenchmark case study

This case study and its queries have been extracted from a work of Szárnyas et al. [111]. The metamodel of this example is depicted in Figure 3.3, where Figure 3.3a represents the main hierarchy and references and Figure 3.3a represents the supertype relationships among the objects. This metamodel represents a railway that is composed of **Routes**, that require **Sensors** in order to provide safety. A **Route** is considered any logical path of the railway. **Sensors** monitor the occupancy of the **Track Elements**, that can be **Segments** or **Switches**. Each route follows certain **Switch positions**. A switch position describes the prescribed position of a switch belonging to the route. Positions can be straight or diverging. Routes can specify different positions for the same switch too. So, a route is active if all its switches are in the position prescribed by the switch positions followed by the route. Finally, there are **Semaphores** in entry and exit points of the routes that emit a signal (**GO** or **STOP**).

This case study pretends finding different anomalies in the railway and repairing them by using the following six queries:

- **Q1. PosLength.** All segments in the graph must have a positive length. Then, this query selects all segments with a length less than or equal to zero and updates them with a positive length.
- **Q2. SwitchMonitored.** Every switch must have at least one sensor connected to it. In this way, this query selects all switches that do not have associated sensors. In order to repair this problem, a new sensor is created and connected to the switch.
- **Q3. RouteSensor.** In the railway, a sensor associated with a switch that is located in a route must also be associated to that route. Therefore, this query obtains all routes that follow a switch position that targets a switch that is monitored by a sensor but they are not connected to the sensor via **requires** relationship (cf. Figure 3.3). In these cases, the query will insert

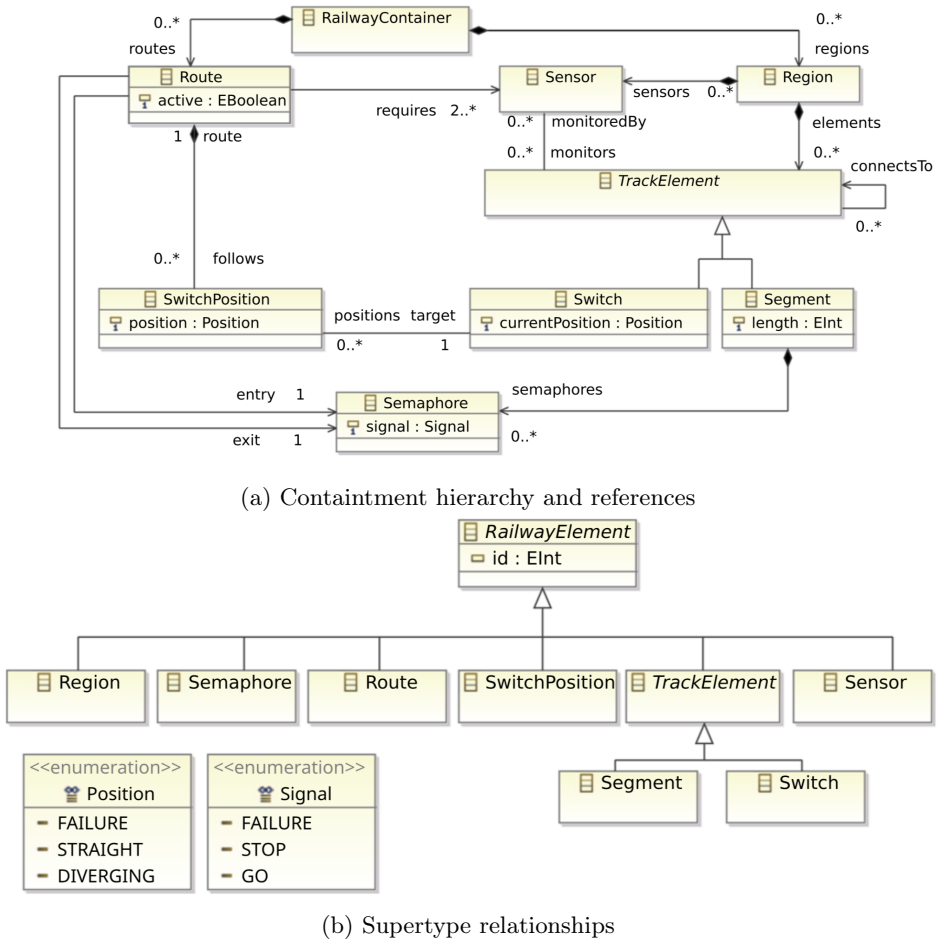


Figure 3.3: TrainBenchmark metamodel [111]

a new **requires** relationship from the route to the sensor, in order to repair the problem.

- **Q4. SwitchSet.** A semaphore shows the signal **GO** when all switches of the corresponding route are in the position prescribed by the route. This query selects all routes that have an entry semaphore with **GO** signal and their switch positions define a different position than the current position of their associated switches. Then, it repairs this problem updating the *currentPosition* property of the switch to the position described by the switch position.

- **Q5. ConnectedSegments.** Each sensor in the network must have five segments associated to it at most. Then, this query selects all sensors that have at least six segments attached to them (called segment1, segment2, segment3, segment4, segment5 and segment6), deletes segment2 and connect segment1 and segment3 between them via *connectsTo* relationship (cf. Figure 3.3).
- **Q6. SemaphoreNeighbor.** Two or more routes that are connected through a pair of sensors and a pair of track elements must belong to the same semaphore. Therefore, this query selects all routes that have an exit semaphore and a sensor that is connected to a track element, which is connected to a second track element. In addition, it checks that the second track element is connected to a second sensor that defines a second route. If this route does not have the semaphore as its entry, the query connects the second route with the semaphore via *entry* relationship (cf. Figure 3.3).

Note that all queries of TwitterFlickr example contain aggregation operations, among other operators, whereas TrainBenchmark example queries only contain filters. Then, all possible operators exposed in the previous sections are represented by these queries.

### 3.4.3 Experimental Setup

In this section, we expose all the parameters used to perform our experiments. These parameters refer to the source models, the methods used to measure the execution times of the queries and the execution environment where they were run.

#### Source Models

Our experiments have been executed on models of different sizes in order to analyze the performance of the queries with the platforms and DSLs exposed in Section 3.2 and Section 3.3. The number of objects and relationships for each model are shown in Table 3.3. Note that models of the different case studies conform to different metamodels and, for this reason, the size of the models is quite different in both examples. Note that the largest models contain between 13 and 14 millions of elements (adding relationships and objects). The name of the models is assigned

Case study	Name	Objects	Relationships
<b>TwitterFlickr</b>	2M	63,561	2,047,431
	2M5	71,225	2,351,293
	3M	83,212	2,780,912
	4M	96,287	3,909,397
	6M5	126,009	6,468,907
	14M	265,102	13,635,513
<b>TrainBenchmark</b>	3K	741	2,135
	8K	2,038	5,898
	22K5	5,777	16,786
	45K	11,499	33,408
	90K	23,233	67,464
	200K	50,623	147,076
	420K	107,352	312,053
	820K	210,789	612,566
	1M5	414,744	1,182,298
	3M	839,057	2,392,113
	6M5	1,672,750	4,861,332
	13M	3,347,214	9,727,504

Table 3.3: Summary of the models used in the experiments.

according to the approximate sum of the number of objects and relationships (e.g. model 2M of TwitterFlickr case study contains around 2 millions of elements and model 2M5 contains 2.5 millions of elements approximately). Note that the models of the TwitterFlickr case study have been created manually according to the metamodel depicted in Figure 3.1, whereas the models of the TrainBenchmark example have been extracted from the sources provided by [111].

### Measurement methods

In order to study the latency of the queries for the case studies exposed in Section 3.4.2, four sets of experiments have been performed. These experiments have been designed according to two parameters:

- **Parallel vs Single executions.** For each case study and platform, two types of executions have been designed for the queries. First, they have been run in a single execution without any other query running at the same time. Then, they have been run in parallel with the rest of the queries.



- **Query with and without effect.** Since queries presented in both case studies modify the initial graph as a result (creating, updating or deleting elements), they can be run in two different manners. First, queries were run returning the filtered elements without any effect over the graph, i.e. removing the part of the query that is in charge of modifying the graph. Second, they were run generating their respective effects over the graph.

This way, the four sets of experiments consist on: (i) single executions of queries without effect, (ii) single executions of queries with effect, (iii) parallel executions of queries without effect and (iv) parallel executions of queries with effect over the graph.

Execution times were measured running the queries and registering their computation times using the `System.currentTimeMillis()` Java method. In order to isolate our results from any transitory load of the machine where tests were run, all experiments were run up to 6 times each. Average performance values were extracted from last 3 runs. In addition, the standard deviation and the coefficient of variation have also been calculated in order to study the variations that occur among the run measurements.

### Execution environment

All experiments have been run on a machine with running operating system Ubuntu 16.04.5 LTS 64 bits, Linux kernel 4.4.0-151-generic, with 64GB of RAM, and an Intel Xeon CPU E5-2680 processor with 16 cores of 2.7 GHz. Besides, our experiments have been run setting 30G for maximum memory allocation pool for the JVM.

Java version 1.8.0\_144 with Oracle JDK vendor was used for all implementations. For TinkerGraph, OrientDB and JanusGraph implementation we used Gremlin-java version 2.6.0 and TinkerGraph-gremlin 3.3.4, orientdb-gremlin 3.1.0 and janusgraph-berkeleyje and janusgraph-lucene 0.3.0, respectively. Crate-jdbc 2.3.1 was used for CrateDB implementation. Scala version 2.11.12, Spark version 2.3.2, GraphFrames package version 0.6.0 and Spark-SQL version 2.0.0 were used for GraphFrames implementation. Finally, neo4j-java driver version 1.4.4 and Neo4j version 3.1.0 were used for both Memgraph and Neo4j implementations.

Finally, recall that Memgraph, TinkerGraph, OrientDB and GraphFrames implementations run in memory whereas CrateDB, Neo4j and JanusGraph implementations store the information in disk.

#### 3.4.4 Results

In this section, we answer the four research questions and discuss the results of the experiments performed to compare the technologies and DSLs described in the previous sections.

##### RQ1: Query latency

To answer this research question, let us focus on the charts depicted in Figure 3.4 and Figure 3.5, which display the execution times for queries without effect of TwitterFlickr example for single and parallel executions respectively. We will also focus on Figure A.1 and Figure A.2 of Appendix A.2, which display the same information for TrainBenchmark example. Besides, Table 3.4 shows a summary of the execution times for each processing platform regarding model size. These values have been obtained calculating the average of the execution times of all queries for each model size in both case studies. In this table, rows 3 to 9 and 19 to 25 show the average execution times for TwitterFlickr and TrainBenchmark examples with single runs, whereas rows 10 to 16 and 26 to 32 show the average execution times for TwitterFlickr and TrainBenchmark examples with parallel runs, respectively. Note that some values are not shown in this table. This is because these experiments caused memory overflow errors because of the source model size. Finally, we noticed that the experiments with parallel runs showed interference among queries that affected the execution times and, therefore, they presented high values for the standard deviation. In order to study the percentage of variation between the measurements, we calculated the coefficient of variation of both case studies for parallel executions as follows:

$$CV = \frac{\sigma}{\mu} \cdot 100$$

Where  $\sigma$  is the standard deviation and  $\mu$  is the execution time average. Results are shown in Table 3.5 and Table A.1 of Appendix A.2.

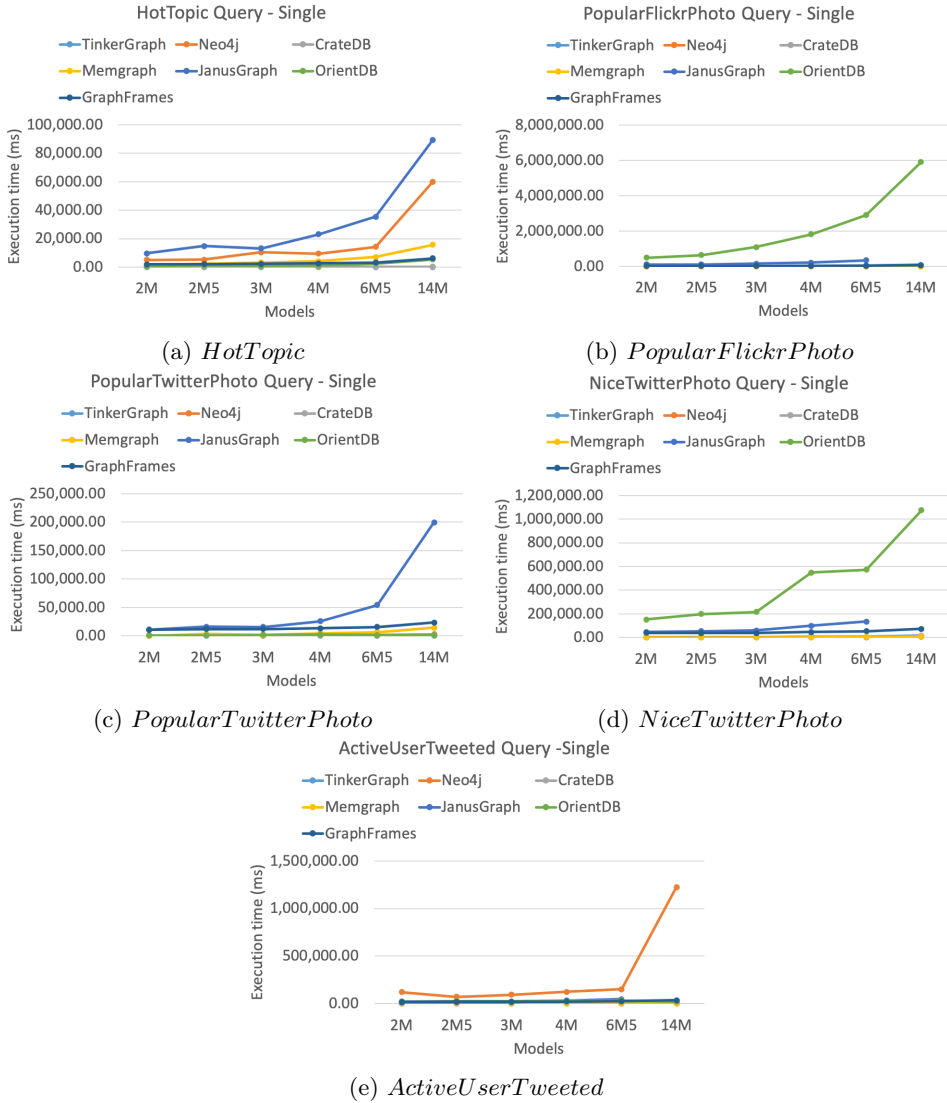


Figure 3.4: Execution time results for queries without effect of TwitterFlickr example with single runs

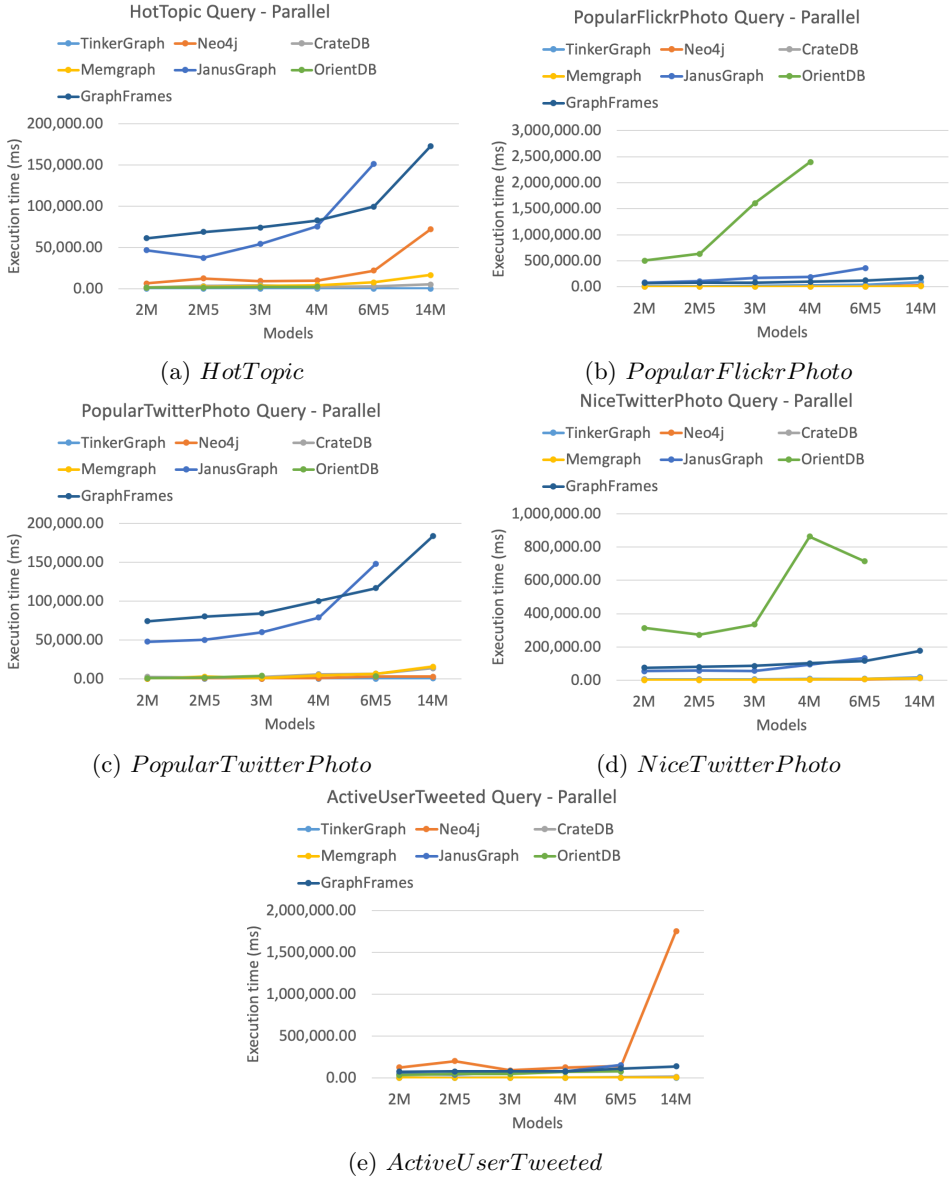


Figure 3.5: Execution time results for queries without effect of TwitterFlickr example with parallel runs

First, we can see in Figure 3.4 that CrateDB presents the lowest execution times for almost all single execution queries of TwitterFlickr example. However, in Figure A.1, it presents the highest execution times for single execution queries of TrainBenchmark example. The reason is that CrateDB is not designed for graph computation and, since TrainBenchmark queries are composed by filters and path traversals, their performance is worse than for TwitterFlickr queries, which are mainly composed of aggregation operators. In this way, since SQL provides optimised functions to filter by aggregation operators, CrateDB obtains a high performance in TwitterFlickr queries. In addition, observe in Table 3.4 how parallel runs make the execution time worse for CrateDB results to a greater extent than the rest of platforms. In order to clarify this circumstance, see how the coefficients of variation of Tables 3.5 and A.1 for CrateDB platform are the highest in all cases. This is due to the existence of a great interference between query executions when they run in parallel, since the execution time measurements obtained are very different from each other.

On the other hand, observe how Memgraph, TinkerGraph and OrientDB present lower execution times than Neo4j and JanusGraph in the majority of charts of Figures 3.4 and 3.5. This is because their implementation is in memory, contrarily to Neo4j and JanusGraph that store the information in disk. However, observe the execution times results of OrientDB in Figures 3.4b and 3.5b and Figures 3.4d and 3.5d that show the results for PopularFlickrPhoto and NiceTwitterPhoto queries, respectively. In these cases, OrientDB presents higher execution times than Neo4j or JanusGraph. Note that these queries contain two aggregation operations (cf. Section 3.1). For this reason, we can conclude that performance in OrientDB gets worse as a higher number of aggregation operations are used in a query. Regarding TrainBenchmark example, execution time results of queries ConnectedSegments, RouteSensor and SemaphoreNeighbor (Figures A.1c, A.2c, A.1e, A.2e, A.1f and A.2f) present similar values for OrientDB, Neo4j and JanusGraph. In addition, observe in Table 3.4 that the execution time averages for TwitterFlickr example with OrientDB are higher than execution time averages of the rest of technologies; whereas average times for TrainBenchmark example with this technology present a steeper growth curve than the rest of technologies, excepting for CrateDB. This means the penalty that comes from the slowest queries implies that even disk implementations outperform OrientDB. Regarding the other

### 3.4 Performance Analysis and Evaluation

Experiment	Tech	Models					
		2M	2M5	3M	4M	6M5	14M
TF Single	TinkerGraph	3,061	3,591	5,044	7,193	10,608	22,468
	Neo4j	25,422	15,922	21,257	27,667	34,787	263,737
	JanusGraph	36,182	40,652	52,754	76,643	122,244	-
	OrientDB	133,314	171,452	267,617	478,438	701,625	1,402,423
	CrateDB	448	575	645	1,039	1,446	3,050
	Memgraph	421	1,232	1,177	2,842	4,623	10,492
	GraphFrames	17,233	17,466	18,567	21,881	26,645	39,429
TF Parallel	TinkerGraph	3,499	3,994	4,747	6,910	10,969	22,550
	Neo4j	26,595	43,230	20,702	28,393	36,867	373,875
	JanusGraph	55,906	58,372	80,972	103,398	189,243	-
	OrientDB	169,732	191,531	398,669	-	-	-
	CrateDB	2,316	2,531	3,260	4,755	5,930	11,556
	Memgraph	428	1,298	1,231	3,052	5,226	11,572
	GraphFrames	71,467	76,936	81,621	92,317	113,373	169,070
Experiment	Tech	Models					
		420K	820K	1M5	3M	6M5	13M
TB Single	TinkerGraph	151	316	584	1,270	2,575	4,821
	Neo4j	4,930	9,737	4,158	7,885	10,742	21,692
	JanusGraph	6,833	12,558	37,598	72,843	142,712	-
	OrientDB	3,090	5,383	11,265	34,434	113,140	-
	CrateDB	9,507	56,867	196,715	803,027	3,193,544	-
	Memgraph	829	1,589	3,212	6,502	13,523	26,886
	GraphFrames	6,811	7,824	10,242	66,310	-	-
TB Parallel	TinkerGraph	185	375	681	1,741	4,624	8,166
	Neo4j	5,943	11,689	5,307	9,663	13,252	25,175
	JanusGraph	9,406	20,510	51,213	115,522	277,571	-
	OrientDB	3,572	9,245	17,569	52,835	82,502	-
	CrateDB	29,840	213,914	1,060,149	3,285,652	14,330,287	-
	Memgraph	960	1,893	3,732	7,592	15,572	33,241
	GraphFrames	30,417	38,632	52,880	376,672	-	-

Table 3.4: Execution time averages (ms) depending on size model

two in-memory graph database implementations, Memgraph outperforms TinkerGraph for TwitterFlickr example, whereas TinkerGraph outperforms Memgraph for TrainBenchmark example (cf. Table 3.4).

Finally, we can observe in Figures 3.4, 3.5, A.2 and A.1 and Table 3.4 that Neo4j queries are faster than JanusGraph queries, which means that Neo4j outperforms JanusGraph. In addition, note how execution times obtained with Neo4j are usually lower than execution times obtained with GraphFrames for single and parallel runs (cf. Figures 3.4, 3.5, A.1 and A.2). Therefore, Neo4j outperforms GraphFrames and JanusGraph in most cases, as it was previously shown in Table 3.4.

With all this information, we can answer RQ1 as follows:

- TinkerGraph and Memgraph present the best performance averages. The reason is they have in-memory implementations.
- CrateDB obtains a high performance with queries that imply aggregation operations. However, it presents a low performance for queries that require traversing edges in a graph.
- CrateDB presents serious interference when running queries in parallel.
- GraphFrames, JanusGraph and Neo4j obtain an intermediate performance with respect to the rest of technologies when querying graphs. However, Neo4j usually outperforms GraphFrames and JanusGraph.
- OrientDB presents a similar performance than Neo4j and JanusGraph, even when it uses in-memory implementation.
- Among disk implementations, Neo4j and JanusGraph outperform CrateDB in most cases. Furthermore, Neo4j queries are faster than JanusGraph queries.

### RQ2: Update latency

Now, let us focus on the charts depicted in Figures 3.6 and 3.7, which display the execution times for queries with effect of TwitterFlickr example for single and parallel executions, respectively; and Figures A.3 and A.4 of Appendix A.2, which display the same information for TrainBenchmark example. Note that some values

Query Name	Tech	Models					
		2M	2M5	3M	4M	6M5	14M
HotTopic	TinkerGraph	5.58	1.63	6.00	5.12	1.16	2.34
	Neo4j	2.55	6.94	2.41	8.17	3.54	3.24
	JanusGraph	26.01	17.75	22.06	16.44	15.02	-
	OrientDB	54.63	42.68	26.14	66.00	-	-
	CrateDB	120.29	120.59	17.49	98.57	93.82	56.14
	Memgraph	4.85	3.15	2.18	0.84	5.72	4.39
	GraphFrames	28.86	27.42	24.93	44.06	39.82	16.65
PopularFlickrPhoto	TinkerGraph	11.49	6.88	4.05	5.94	3.35	0.94
	Neo4j	14.97	5.65	29.58	4.66	6.16	2.87
	JanusGraph	7.14	1.85	25.80	3.77	2.72	-
	OrientDB	1.67	1.47	19.91	-	-	-
	CrateDB	74.29	37.09	76.64	15.25	19.41	25.27
	Memgraph	7.22	9.06	2.00	9.65	2.69	2.97
	GraphFrames	10.44	14.94	12.62	10.24	3.82	8.10
PopularTwitterPhoto	TinkerGraph	2.25	5.02	8.66	8.23	5.44	13.99
	Neo4j	13.11	5.57	4.43	29.58	40.34	20.26
	JanusGraph	25.22	12.23	6.40	5.66	27.93	-
	OrientDB	41.71	70.32	26.48	25.30	81.16	-
	CrateDB	59.03	73.70	98.57	53.54	85.92	30.50
	Memgraph	0.83	6.25	1.88	3.48	6.49	7.51
	GraphFrames	13.77	11.68	7.56	9.24	12.19	2.82
NiceTwitterPhoto	TinkerGraph	1.83	6.28	7.47	6.40	14.38	0.214
	Neo4j	20.95	16.89	6.73	4.03	3.12	3.87
	JanusGraph	6.37	2.45	2.77	3.98	6.25	-
	OrientDB	7.58	0.76	2.00	5.01	14.28	-
	CrateDB	90.82	34.32	77.39	10.28	39.83	69.25
	Memgraph	4.73	0.86	4.67	2.38	4.94	2.26
	GraphFrames	12.56	16.65	14.37	9.53	9.71	9.09
ActiveUserTweeted	TinkerGraph	1.01	4.59	15.86	7.24	4.86	14.65
	Neo4j	3.77	1.73	2.38	1.10	1.02	0.22
	JanusGraph	6.37	2.45	2.77	3.98	6.25	-
	OrientDB	7.58	0.76	2.00	5.01	14.28	-
	CrateDB	62.36	38.88	13.82	54.75	77.89	27.30
	Memgraph	0.51	1.60	2.46	3.77	22.30	9.16
	GraphFrames	76.91	81.95	80.46	76.26	72.23	80.70

Table 3.5: Coefficient of variation (%) of TwitterFlickr queries without effect and parallel runs.



are not depicted for some model sizes and case studies. This is because parallel experiments presented concurrency problems when modifying the graph with several queries running in parallel (e.g. queries for Neo4j and Memgraph experiments). In addition, we have not run any experiment with OrientDB queries in parallel with effect over the graph, since multi-threading is not supported when using Gremlin language as it is expose in its documentation [28]. Regarding queries in single experiments, some of them take too long and the trend line can already be inferred, so there is no real need to compute them (e.g. CrateDB, Neo4j and JanusGraph experiments for TrainBenchmark example). Besides, Table 3.6 shows a summary of the execution times for each processing platform regarding model size. These values have been obtained in the same manner as those in Table 3.4. Therefore, rows 3 to 8 and 17 to 22 show the average execution times for TwitterFlickr and TrainBenchmark examples with single runs, whereas rows 9 to 14 and 23 to 28 show the average execution times for TwitterFlickr and TrainBenchmark examples with parallel runs, respectively. Finally, Table 3.7 displays how much execution time increases when the effect of the query is reflected in the graph—percentages are shown. These values are calculated in the following manner:

$$\Delta T = 100 \cdot \frac{\bar{t}_M - \bar{t}_Q}{\bar{t}_Q} \quad (3.1)$$

where  $\bar{t}_M$  is the average of the execution times for queries with effect over the graph and  $\bar{t}_Q$  is the average of the execution times for queries without effect over the graph.

Note that some values are also not shown in these tables, for the same reasons stated for the graphics. Besides, since GraphFrames do not provide functions to modify the graph, there are not lines referring to this technology in the graphics.

First, observe how Neo4j and JanusGraph present the highest execution times for almost all graphics of Figures 3.6 and 3.7 for TwitterFlickr example. The reason is that they store the graph into disk and accessing disk is more costly than accessing memory. Regarding TrainBenchmark example, CrateDB presents the highest execution times, since it is not designed to work with graphs and TrainBenchmark queries are mainly composed by filters that traverse edges (cf. **RQ1**). However, if we consult the results of Table 3.6, we can observe that Neo4j and OrientDB obtain the highest average execution time in both case studies and

### 3.4 Performance Analysis and Evaluation

Experiment	Tech	Models					
		2M	2M5	3M	4M	6M5	14M
TF Single	TinkerGraph	3,186	4,227	4,768	7,260	12,175	25,162
	Neo4j	25,870	40,453	20,559	29,667	38,319	370,804
	JanusGraph	38,913	49,455	53,177	81,612	117,246	-
	OrientDB	135,967	174,957	230,198	448,458	664,727	1,636,927
	CrateDB	516	720	773	1,268	1,609	3,443
	Memgraph	413	855	1,009	1,904	2,954	7,991
TF Parallel	TinkerGraph	3,378	4,150	4,845	7,201	11,418	25,574
	Neo4j	25,768	42,180	22,433	-	-	-
	JanusGraph	51,585	67,487	77,478	135,398	218,998	-
	OrientDB	-	-	-	-	-	-
	CrateDB	515	710	790	1,232	1,825	3,535
	Memgraph	439	957	-	-	-	-
Experiment	Tech	Models					
		420K	820K	1M5	3M	6M5	13M
TB Single	TinkerGraph	152	290	615	1,187	2,632	5,253
	Neo4j	88,410	397,606	1,637,751	-	-	-
	JanusGraph	7,442	14,074	32,190	61,800	128,325	-
	OrientDB	2,684	5,666	14,177	42,713	146,364	1,186,161
	CrateDB	13,447	61,167	226,460	936,874	-	-
	Memgraph	140,121	588,041	-	-	-	-
TB Parallel	TinkerGraph	150	275	685	-	4,566	8,098
	Neo4j	99,823	405,467	-	-	-	-
	JanusGraph	11,367	17,678	56,544	112,923	272,512	-
	OrientDB	-	-	-	-	-	-
	CrateDB	35,389	106,603	227,880	954,774	-	-
	Memgraph	-	-	-	-	-	-

Table 3.6: Execution time averages (ms) depending on model size (with effect over the graph)

they present concurrency problems when modifying the graph with parallel queries, as explained before. Furthermore, since OrientDB does not support multi-threading with Gremlin and taking into account the results obtained in **RQ1**, we decided to stop the experiments with this technology. Besides, Memgraph also presents concurrency problems for both case studies when modifying and updating the graph in parallel. Finally, TinkerGraph and JanusGraph results show that queries with effect over the graph behave similarly as queries without effect over the graph (cf. **RQ1**). However, execution times of TinkerGraph queries are faster than JanusGraph queries.

Now, observe results of Table 3.7. Note that some values are negative per-

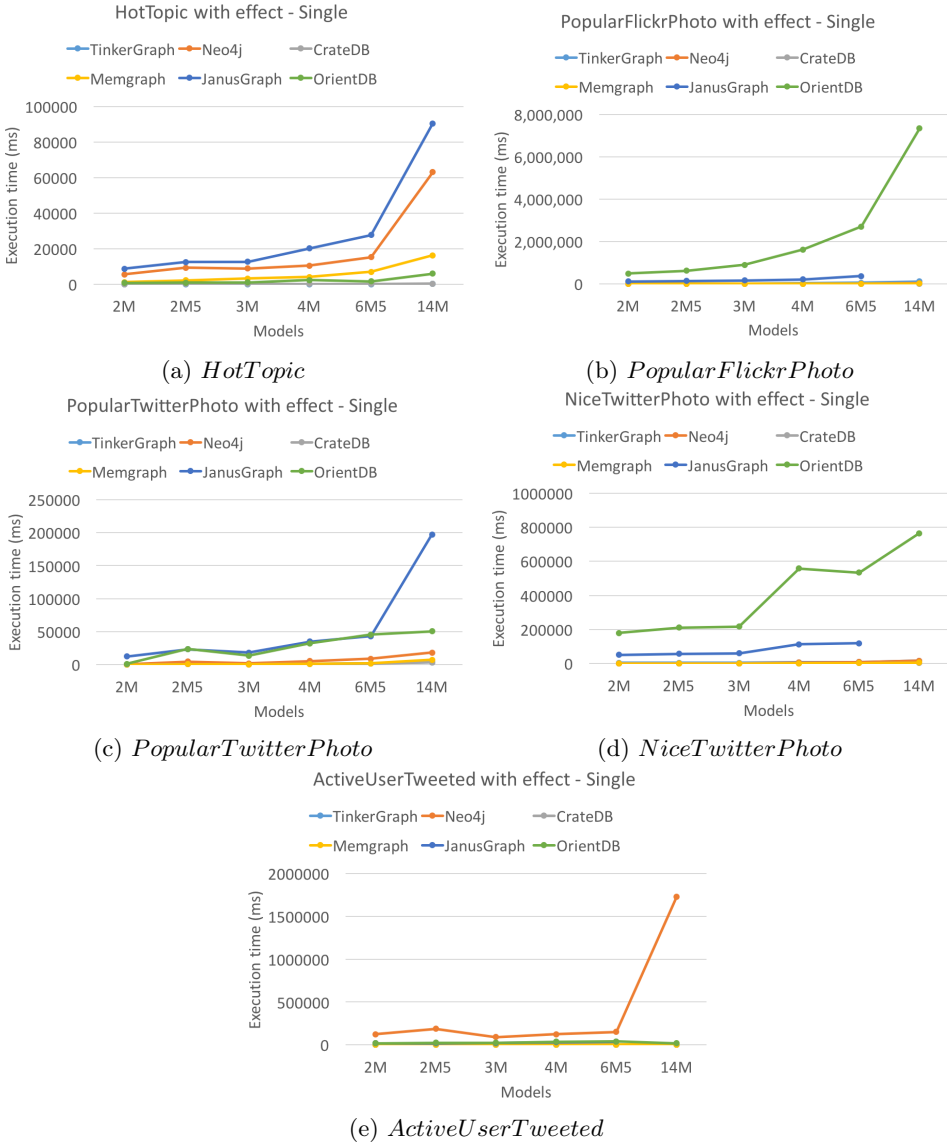


Figure 3.6: Execution time results for queries with effect of TwitterFlickr example with single runs

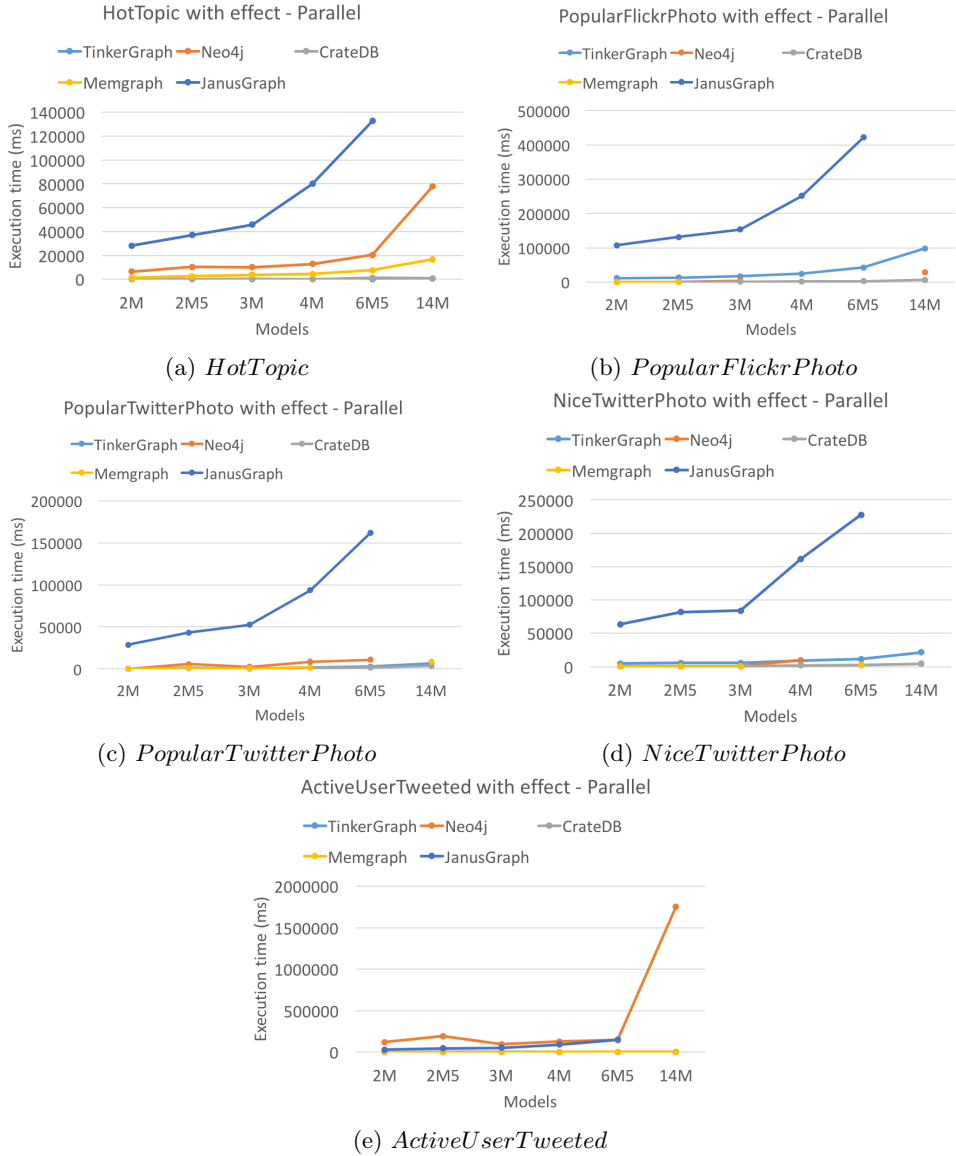


Figure 3.7: Execution time results for queries with effect of TwitterFlickr example with parallel runs

Experiment	Tech	Models					
		2M	2M5	3M	4M	6M5	14M
TF Single	TinkerGraph	4.05	17.71	-5.48	0.91	14.77	11.99
	Neo4j	1.76	154.06	-3.28	7.22	10.15	40.59
	JanusGraph	7.54	21.65	0.80	6.48	-4.08	-
	OrientDB	1.98	2.04	-13.98	-6.26	-5.25	16.72
	CrateDB	14.93	25	19.84	21.99	11.18	12.86
	Memgraph	-1.91	-30.66	-14.29	-33.02	-36.11	-23.84
TF Parallel	TinkerGraph	-3.46	3.88	2.06	4.21	4.08	13.40
	Neo4j	-3.10	-2.42	8.36	-	-	-
	JanusGraph	-7.73	15.61	-4.31	30.94	15.72	-
	OrientDB	-	-	-	-	-	-
	CrateDB	-77.74	-71.95	-75.77	-74.08	-69.22	-69.41
	Memgraph	2.31	-26.24	-	-	-	-
Experiment	Tech	Models					
		420K	820K	1M5	3M	6M5	13M
TB Single	TinkerGraph	0.44	-8.11	5.19	-6.56	2.20	8.96
	Neo4j	1693.02	3983.12	39284.80	-	-	-
	JanusGraph	8.92	12.07	-14.38	-15.15	-10.08	-
	OrientDB	-13.13	5.25	25.84	24.04	29.36	-
	CrateDB	41.43	7.56	15.12	16.66	-	-
	Memgraph	16799.02	36905.74	-	-	-	-
TB Parallel	TinkerGraph	-18.70	-26.59	0.53	-	-1.25	-0.83
	Neo4j	1579.68	3368.51	-	-	-	-
	JanusGraph	20.85	-13.80	10.40	-2.25	-1.82	-
	OrientDB	-	-	-	-	-	-
	CrateDB	18.59	-50.16	-78.50	-70.94	-	-
	Memgraph	-	-	-	-	-	-

Table 3.7: Overhead of execution time average (%) for updating the graph depending on model size

centages, which means that queries with effects obtain a better performance than queries without effect. This is because queries without effect obtain the results and store them in a list, whereas queries with effect create or update the information of the source graph. Then, negative percentages mean that storing results in a list is more costly than updating the graph with the effect specified by the query. In this way, we can see that the execution time averages of TwitterFlickr example are much lower than execution times averages of TrainBenchmark example for Neo4j and Memgraph experiments. The reason is that TwitterFlickr queries only add new objects and relationships to the graph as a result, whereas TrainBenchmark queries usually update the information already stored. For this reason, we can state that Neo4j and Memgraph get better performance when creating new elements than when updating existing information. Besides, CrateDB, TinkerGraph and JanusGraph—and also OrientDB queries in single execution—do not present a high performance decrease when updating the graph with respect to obtaining the results in a list with queries without effect. However, TinkerGraph obtains lower execution times for queries, as can be viewed in Table 3.6.

With all this information, we can answer RQ2 as follows:

- Neo4j and Memgraph obtain a higher decrease in the performance when updating existing information than when creating new elements in the graph.
- CrateDB, TinkerGraph, JanusGraph and OrientDB obtain similar performances when modifying the graph and obtaining the results of queries without effect, i.e., the effects do not greatly increase execution times.
- Memgraph and Neo4j present concurrency problems when modifying the graph in parallel. In addition, OrientDB does not allow modifying the graph with parallel queries when using Gremlin.
- TinkerGraph obtains the best performance for almost all cases when modifying the graph with queries with effect.

### RQ3: Syntax complexity

In order to answer this question, the number of operators, characters and internal variables have been calculated for each query and platform. These features

allow to compare the DSLs exposed in Section 3.3. The parameters have been calculated as follows:

- **Operators:** arithmetic, logical and relational operators as well as methods and functions explained in Section 3.3 have been included in this parameter. Note that the methods or functions that traverse the edges of the graph (e.g. `in()` and `out()` functions in Gremlin code or the operator `-[]->` in Cypher code) are also considered operators and, therefore, they have been taken into account to calculate the number of operators. As an example, observe the **HotTopic** query of TwitterFlickr example depicted in Listing 3.10 in Cypher language. This query contains 12 operators (`MATCH`, `-[tagsE:tags]->`, `<-[containsE:contains]-`, `WITH`, `COUNT`, `+`, `COUNT`, `AS`, `WHERE`, `>`, `CREATE` and `-[:EVENT]->`).
- **Characters:** since some DSLs, such as Gremlin, are designed to be embedded into a general purpose language, and the experiments with the platforms exposed in Section 3.2 are implemented in one of these languages too, we have considered the characters involved in the execution of the queries with the language used for each platform. Therefore, we have counted the number of characters that are contained in the instructions that run a query, without including spaces. In Listing 3.10, the number of characters includes all characters from line 2 to line 7 without including the spaces (222 characters in total). This parameter provides a measurement of the verbosity of the DSL, which is useful to understand the complexity of the language when writing queries for graphs.
- **Variables:** this parameter includes the number of internal variables used by the query syntax in order to obtain the result or update the graph, depending on the effect of the query. The example of Listing 3.10 includes 6 variables (`p`, `tagsE`, `h`, `containsE`, `t`, `sumHT`).

Listing 3.10: HotTopic query for Neo4j

```
1 // HotTopic
2 StatementResult result = session.run(
```

```

3  "MATCH (p)-[tagsE:tags]->(h:Hashtag)<-[containsE:contains]-(t)
4      WITH h,(COUNT(tagsE) + COUNT(containsE))
5      AS sumHT WHERE sumHT>100
6  CREATE (:HotTopic{date:timestamp()})-[:EVENT]->(h)";
7  result.consume();

```

All queries exposed in Section 3.4.2 are depicted in Appendix A for each platform. Furthermore, the results of the number of operators, variables and characters are depicted in Table 3.8 and Table A.2 of Appendix A.2, that summarize all syntax features of TwitterFlickr and TrainBenchmark queries, respectively. These tables are divided into four columns. First column indicates the platform used for running the query, second column shows the name of the query, third column represents the features of the queries with effect over the graph and fourth column represents the features of the queries without effect. Third and fourth columns are also divided into three subcolumns, that show the number of operators, characters and internal variables for each query, respectively. In addition, rows depicted in bold represent the average of these parameters for each platform. Note that JanusGraph, TinkerGraph and OrientDB queries have the same syntax since they use Gremlin language. Besides, although Neo4j and Memgraph use Cypher as DSL, Memgraph does not allow all methods included in Cypher language. For this reason, the tables expose the results of both technologies separately. From now on, when we write *CypherM* and *CypherN* we refer to the queries implemented with Memgraph and Neo4j, respectively.

Firstly, let us observe the results of the column *Update* of Table 3.8. Gremlin queries and CypherM and CypherN obtain the lowest number of characters per query which are similar to each other (being CypherM results slightly lower), whereas SQL average values are higher than previously mentioned languages (over 150 units). Regarding the number of variables, CypherM, CypherN and SQL show an average from 6 to 7 variables, whereas Gremlin shows half the number of variables. However, Gremlin queries have 28 units in average, while CypherM, CypherN and SQL queries have around 8 less operators. Now, we focus on column *Query* of Table 3.8. In this case, Gremlin presents the lowest numbers of characters, operators and variables per query as it decreases these numbers when queries do not modify the graph: around 150 characters, 10 operators and 2.5 variables less. CypherM and CypherN decrease the number of characters in around 65 and SQL decreases them in around 75, whereas the number of operators and variables



remains almost the same. Besides, GraphFrames DSL shows the highest numbers for the three parameters with almost 835 characters, 43 operators and 11 variable as average.

Secondly, see column *Update* of Table A.2. CypherM and CypherN obtain the lowest numbers of characters per query following by Gremlin and SQL. Note the difference between the number of characters for SQL with respect to the rest of DSLs, which is higher than in Gremlin example. This is because SQL language is not designed for querying graphs. As stated in the previous sections, TrainBenchmark example presents queries that mainly involve traversals through the edges of the graph, whereas TwitterFlickr example involves aggregation operations. For this reason, the number of characters needed for TrainBenchmark queries with SQL is higher than the number of characters needed for TwitterFlickr example, since SQL does not provide methods that traverse edges but uses joins to achieve this (cf. Section 3.3.1). In addition, SQL also shows the highest number of operators and variables. However, whereas Gremlin presents almost double the number of operators than CypherM and CypherN, all these languages present a similar number of variables. Now, let us focus on column *Query*. Gremlin significantly decreases the number of operators and characters, whereas CypherM and CypherN present almost the same results when the queries do not involve effects over the graph again. Besides, SQL also presents a significant decrease of operators and characters (almost 11 operators and 400 characters less). However, the number of characters for CrateDB is higher than the number of characters of GraphFrames, presenting both technologies the highest values for the three parameters.

In conclusion, Gremlin significantly reduces the number of operators and characters when querying the graph with respect to updating the graph as a result of a query in both examples. However, these numbers do not reduce for TrainBenchmark example when it is compared with the significant reduced number of variables involved in TwitterFlickr example. Cypher does not obtain important changes when querying the graph with respect to updating it and it also presents similar results for Neo4j and Memgraph platforms. In addition, even taking into account that GraphFrames DSL is designed to work with graphs, it provides a more complex syntax than Gremlin and Cypher for both case studies. Besides, this platform does not allow updating the graph. Since SQL is not designed to work with graphs, it provides less intuitiveness than Cypher, Gremlin and GraphFrames

Tech	Query	Update			No Update		
		Op	Char	Var	Op	Char	Var
TinkerGraph, JanusGraph and OrientDB (Gremlin)	HotTopic	15	202	2	8	89	0
	PopularTwitterPhoto	24	328	3	13	144	0
	PopularFlickrPhoto	21	284	2	14	149	0
	NiceTwitterPhoto	46	593	5	35	412	2
	ActiveUserTweeted	34	446	3	23	258	0
	<b>AVG</b>	<b>28</b>	<b>370.6</b>	<b>3</b>	<b>18.6</b>	<b>210.4</b>	<b>0.4</b>
Neo4j (Cypher)	HotTopic	12	222	6	11	177	6
	PopularTwitterPhoto	15	275	6	14	199	6
	PopularFlickrPhoto	16	281	5	15	222	5
	NiceTwitterPhoto	28	538	11	27	465	11
	ActiveUserTweeted	29	535	8	28	463	8
	<b>AVG</b>	<b>20</b>	<b>370.2</b>	<b>7.2</b>	<b>19</b>	<b>305.2</b>	<b>7.2</b>
CrateDB (SQL)	HotTopic	17	476	8	16	418	8
	PopularTwitterPhoto	12	373	3	11	294	3
	PopularFlickrPhoto	14	360	3	13	292	3
	NiceTwitterPhoto	29	731	8	28	655	8
	ActiveUserTweeted	31	651	6	30	563	6
	<b>AVG</b>	<b>20.6</b>	<b>518.2</b>	<b>5.6</b>	<b>19.6</b>	<b>444.4</b>	<b>5.6</b>
Memgraph (Cypher)	HotTopic	12	224	6	11	178	6
	PopularTwitterPhoto	15	276	6	14	200	6
	PopularFlickrPhoto	16	281	5	15	222	5
	NiceTwitterPhoto	28	525	11	27	452	11
	ActiveUserTweeted	29	535	8	28	463	8
	<b>AVG</b>	<b>20</b>	<b>368.2</b>	<b>7.2</b>	<b>19</b>	<b>303</b>	<b>7.2</b>
GraphFrames	HotTopic	-	-	-	11	151	4
	PopularTwitterPhoto	-	-	-	38	745	10
	PopularFlickrPhoto	-	-	-	30	480	8
	NiceTwitterPhoto	-	-	-	74	1,517	18
	ActiveUserTweeted	-	-	-	62	1,273	15
	<b>AVG</b>	-	-	-	<b>43</b>	<b>833.2</b>	<b>11</b>

Table 3.8: Summary of DSL features for TwitterFlickr case study

DSL when the queries involve traversing edges (such as TrainBenchmark example), but it presents a better result than GraphFrames DSL when the queries contain aggregation operations (such as TwitterFlickr example). Then, Gremlin and Cypher represent queries over graph in a simpler manner than SQL or GraphFrames DSL, since they allow to write queries with a lower number of characters, variables and operators. DSLs for graph databases provide more intuitiveness and less complexity than the rest of DSLs.

With all this information, we can answer RQ3 as follows:

- Gremlin significantly reduces the number of operators and characters when querying the graph compared to updating the graph as a result of a query. The number of reduced variables will depend on the query.
- The Cypher syntax when querying or updating the graph does not present significant modifications.
- SQL provides a simpler syntax for queries that imply aggregation operations than queries that imply the traversing of edges in a graph. However, it is more complex than query languages for graph databases in all cases.
- GraphFrames DSL provides a more complex syntax than Gremlin, Cypher or SQL and it does not allow updating the graph.
- Gremlin and Cypher represent queries over graphs in a simpler manner as SQL or GraphFrames DSL. In this way, DSLs for graph databases provide more intuitiveness and less complexity than the rest of DSLs. Therefore, our main hypothesis is confirmed.
- Regarding simplicity and intuitiveness of the syntax, Cypher and Gremlin present similar results. Therefore, choosing the most suitable DSL for an application will depend on the nature of the experiments.

#### **RQ4: Best combination of DSL and processing platform**

Note that none of the solutions presented along this chapter can be stated as the best for all cases. The choice will ultimately depend on the requirements of the application itself. Therefore, in order to answer this question, we have analysed

the conclusions presented in the previous questions according to the requirements defined on the proposal of this thesis.

First, this thesis pretends to design a classification of queries that can be performed over graph-structured data. For this reason, the query language should have a clear syntax that enables the easy identification of the appropriate query patterns. This made us discard SQL due to the complexity of the query expressions, that presented the highest numbers of characters, operators and internal variables for almost all cases (cf. Tables 3.8 and A.2), and the absence of a clear syntax for writing graph patterns. Furthermore, CrateDB platform showed the worst performance for TwitterFlickr and TrainBenchmark case studies (cf. **RQ1** and **RQ2** of this chapter). For this reason, the combination CrateDB-SQL has to be discarded for our approach.

Second, given that this thesis is focused on systems that are constantly updated as new information arrives, and whose data can be modified as consequence of the queries (e.g., to incorporate newly generated complex events into the data stream), we discard GraphFrame because it does not currently support efficient operations to update the graph. Besides, our results showed that GraphFrame performance when querying the graph is worse than the rest of technologies performance (cf. **RQ1** and **RQ2** of this chapter), and GraphFrames DSL is more complex than query languages for graph databases such as Gremlin or Cypher (cf. **RQ3** of this chapter).

On the other hand, as discussed at the beginning of this chapter, one of our main requirements is the need to process data in real time. The experiments with graph databases have proven that they are useful to store and query very large amounts of data. However, the requirement for real-time processing is too restrictive for solutions that need to access disk, such as Neo4j or JanusGraph. To demonstrate this, notice how the results of **RQ1** and **RQ2** show that Neo4j performs worse than TinkerGraph and Memgraph. For this reason, we conclude that only in-memory solutions are viable, and hence we decide to discard the combinations Neo4j-Cypher and JanusGraph-Gremlin since they store the information in disk. In addition, even when OrientDB was developed with in-memory implementation, we also decided to discard it as execution time averages were the highest. Considering these results and understanding that SQL is not suitable for our proposal due to its complexity, we also concluded that it is not worth to use OrientDB using a

different implementation with SQL language or disk implementation.

Taking into consideration that both Cypher and Gremlin languages show the simplest syntax of the proposed languages and they present similar results in both case studies (cf. **RQ3**), our highest priority for choosing a technology is the speed. For this reason, we select the faster and most efficient alternative between TinkerGraph and Memgraph. Even when both technologies are suitable for implementing real-time applications, the results showed that TinkerGraph is usually faster than Memgraph. In addition, Memgraph started showing some concurrency problems when creating new elements in parallel with models of around 4 million elements ((cf. **RQ1** and **RQ2**). Some studies (e.g., [62]) also show that even if Cypher is usually easier to learn, the implementation of queries that imply graph *vicinity* is easier and more efficient with Gremlin. For these reasons, we choose TinkerGraph-Gremlin combination to implement our approach.

With all this information, we can summarize our conclusions for RQ4 as follows:

- CrateDB-SQL combination is discarded for our proposal since it shows a lower performance and higher syntax complexity compared to the rest of solutions.
- GraphFrames are not suitable for our proposal since they do not provide efficient operations to update the graph.
- OrientDB is discarded because it does not allow multi-threading with Gremlin and it presents a low performance compared to the rest of technologies.
- Neo4j and JanusGraph presents lower performance compared to TinkerGraph and Memgraph graph databases because they store the information in disk. For this reason, only in-memory solutions are adequate for our proposal.
- The most suitable combination between platform and DSL for our proposal is TinkerGraph with Gremlin language.

### 3.4.5 Threats to validity

In this section we discuss the threats that can affect the validity of our proposal and results. We describe four types of threats according to Wohlin et al. [129].

### **Construct validity threats**

These threats are concerned with the relationship between theory and what is observed. Since the main issue with data streaming applications is the execution time detriment, we have put a special focus on this measure when evaluating and comparing the processing platforms presented in Section 3.2. Regarding comparison among DSLs presented in Section 3.3, we have considered the number of characters, operators and internal variables in our analysis. Given that we have analyzed the results from different perspectives, we consider that this threat can be neutralized.

### **Conclusion validity threats**

Conclusion threats are related to the factors that may affect the ability to draw correct conclusions from the results of the experiments. The main issue that can affect the validity of our conclusions is the transitory load of the machine where the experiments were run. To mitigate the effect this can have on the performance results, we run the experiments 6 times and obtained the average of the 3 last runs.

### **Internal validity threats**

These threats are related to those factors that might affect the results of our evaluation. To try to mitigate these threats, we have used models of different sizes as source data for the experiments. Furthermore, since our analysis is focused on data streaming applications that deal with huge amounts of information, most of our models have a very-large size. In particular, our models size range from 2,000 to 14 million elements (objects and relationships together). Besides, since models belong to two different case studies, the topology of the models in the different case studies is very likely to be different, what allows us to analyze the behavior of our approach with data of different nature.

Finally, there can be different applications for data streaming applications. For this reason, our queries and models have been tested with seven different platforms and in two different manners (queries without effect over the graph and queries that modify the graph). Therefore, we mitigate this threat by analyzing how our approach behaves in different scenarios.

### External validity threats

External validity threats have to do with the extent to which it is possible to generalize the findings of the experiments. The first threat is that the results of our experiments have been obtained with two case studies, which externally threatens the generalizability of our results. To mitigate this threat, we have tried to select case studies from different contexts. The first one is on the domain of social networks and has been created by us, namely the TwitterFlickr case study (cf. Section 3.1). In that case study, we have tried to reflect the main parts of the Twitter and Flickr social networks, and have created models of different sizes in which connections among objects are similar to the ones we could have in models containing real data. The other case study is a framework for controlling a railway (cf. Section 3.4.2), extracted from a work that measures the performance of continuous model transformations [111].

### 3.5 Related work

Many works compare different platforms that can be used for querying graphs [59, 60, 105, 111]. In [111] Szárnyas et al. present a benchmark for comparing 10 different tools (Neo4j and TinkerGraph among them) from the domains of EMF, graph databases, RDF and SQL. These tools are compared using three scenarios (batch, inject and reparation). The authors expose the TrainBenchmark case study that we use in this paper in order to run the experiments (cf. Section 3.4.2). It differs from this paper in two main aspects: (i) the experiments are designed for a single case study and (ii) the authors do not cover the comparison between the query languages. In [59] Guo et al. compare six platforms (Neo4j among them) for processing large graphs based on four features (raw performance, resource utilization, scalability, and overhead) and using five graph algorithms (general statistics, breadth-first search, connected component, community detection, and graph evolution). This also differs from our work in two main aspects: (i) they do not compare the performance of the platforms when querying graphs but only when running algorithms over them and (ii) they do not consider the comparison between the query languages. In addition, two overviews [60, 105] identify the challenges and tasks of real-time big data processing and compares more than

10 different platforms commonly used for this processing. However, since these works only expose the state of the art of big data processing, they do not present experiments to compare the platforms. Besides, these works only consider general purpose languages instead of DSLs.

Some works analyse and compare the features of different query languages that can be used for graphs. For instance, Holzschuher and Peinl [62] compare Gremlin and Cypher in order to find out which one has better performance and a simpler syntax. Besides, both syntax are also compared with SQL. However, they consider one case study only with less than half a million elements (between objects and relationships) whereas our approach uses datasets with several millions of elements. In addition, Barceló [11] studies the complexity of several general purpose navigational query languages. In this way, the author considers two semantics (simple and arbitrary paths). Nevertheless, this approach does not consider the comparison of the platforms but only the languages and the way of analyzing the syntax of the language is quite different from our proposal.

Regarding the manipulation of large graphs, NeoEMF [41] provides a multi-database model persistence framework for very-large models, and [42] defines a language to perform OCL queries on graph databases that outperforms (in terms of memory footprints and time) other existing solutions. In addition, [40] defines a mapping from ATL to Gremlin, which enables model transformations on large models stored in graph databases. Regarding the use of NoSQL languages in model transformations, Daniel et al. [40] develops a framework to map from ATL to Gremlin using graph databases. Another case of using Gremlin to transform large models is in [43], where authors generate Gremlin scripts and compute queries with Mogwai tool, but they consider at most 80,000 elements between nodes and edges and our implementation uses more than 1 million. In [1] the authors present an approach to run queries on encrypted graph databases, in order to protect the data privacy, using Neo4j and Cypher as query language. First, they translate the graph into encrypted form to be executed on a server without decrypting the data. Then, the encrypted results are sent back to a client to be decrypted.

Among the papers about performing queries over large graphs, some other works deal with very large or even infinite models [34] as well as streaming models [37]. These works select only a portion of elements to be available at any given moment in time. To achieve this, they use the concept of *sliding window*, that comprises the



information to be queried at the current instant. In addition, *approximate* model transformations [121] or techniques of Approximate Query Processing (AQP) [53, 87] are also used to perform queries over large graphs. These techniques obtain approximate results but accurate enough to draw valid conclusions, i.e. they sacrifice accuracy of results in order to improve performance.

Finally, many other graph query languages exist in the model-driven community, such as GROOVE [54], Henshin [7] and other TGG tools, and VIATRA [44]. These languages are designed to specify graph patterns that define graph queries. However, we have analysed the most frequently used query languages in this paper.

### 3.6 Summary

In this chapter we have presented an overview of the most common platforms used to work with large volumes of data (TinkerGraph, Neo4j, CrateDB, Memgraph, JanusGraph, OrientDB and GraphFrames) along with 4 DSLs used to handle the data (Gremlin, Cypher, SQL and the GraphFrames DSL). Our main goal is to find the best platform and DSL combination that suits the following requirements in a better way: (i) it allows to query and update the data in real time, (ii) it allows to work with graph-structured information, and (iii) it provides a clear syntax that allows to define a classification of query types over graphs.

Platforms were tested in two case studies that contained graph-structured information. Experiments were designed to obtain the performance based on the execution time. Furthermore, queries of each experiment were implemented using the different DSL proposed and we studied the complexity of each language in terms of number of characters, operators and internal variables.

Results show that the most efficient technologies to work with graphs are graph databases. Their DSLs present the simplest syntax too. However, our results conclude that in-memory graph databases Memgraph and TinkerGraph are the most suitable for our approach as they are both faster than in-disk implementations. Finally, comparing both platforms, TinkerGraph and Gremlin language were chosen as the most adequate combination for our needs, since Memgraph presented concurrency problems when modifying a graph with more than 4 million elements in parallel.

# Chapter 4

## Improving Performance with Online Techniques

---

The proposal of this thesis is based on the claim that most of the data that needs to be processed for decision making is not significantly relevant, particularly with large volumes of data. Therefore, the goal is to select the relevant data subset that would still yield valid results on the queries performed, i.e. performing AQP techniques. Applying AQP to our data might result in accuracy loss, since not all elements and connections will be considered in the approximation. In contrast, trying to account for all the relevant information may result in an unacceptable response time, or the need to count on more resources (e.g. memory) than we currently have. Then, the goal is to find the right balance between the performance of running the queries and the accuracy of their results. For this we need to answer two questions: (a) how to select the subset of data that is relevant for a given query; and (b) how to estimate the error we are making when discarding some of the input data. This problem is of application in those systems that deal with large amounts of data and do not need extremely accurate results but

require fast response times (e.g. recommendation systems on Facebook, Netflix or Amazon). Here, discarding some information involves approximating incoming data. Consequently, the accuracy of our results might be compromised. As presented in Chapter 2, two types of AQP techniques are used in order to select a subset of the incoming information generated from different sources, namely online and offline techniques. The former are performed when processing the data whereas the latter perform an initial computation before execution and stores this information to be used in the processing. Recall that our approach considers data in terms of models. For this reason, the graph is composed of elements, that refer to objects and relationships together (cf. Section 2.3.1). Then, in this chapter, we apply online AQP techniques to elements that are related among them in the form of graphs structures.

We explore different approximation possibilities (random, temporal and spatial approximations) depending on (i) how the data is organized and (ii) what information needs to be obtained from the data. Furthermore, we propose a method that allows estimating the errors produced when applying approximations, with the goal of finding the right balance between performance gain and accuracy loss when approximating data. To illustrate our proposal, we use a simplified version of the Amazon ordering service. According to the results presented in Chapter 3, we implemented the solution using TinkerGraph [118] as processing platform and Gremlin language [6].

The contributions of the current chapter are twofold. First, three online AQP techniques are proposed, which select a subset of the source data and allow to reduce processing execution time. Second, we define the error produced by these techniques using well-known terms, namely accuracy, recall and precision. Both contributions allow to find a trade-off between accuracy and performance that must be settled by the user according to her requirements.

The structure of this chapter is as follows. First, in Section 4.1 we present the running example. Then, Section 4.2 describes some main concepts needed to understand the proposal as well as three different types of approximations and how to calculate the error induced by them. Section 4.3 evaluates and discusses the results of the experiments. Finally, Section 4.4 relates our work to other similar proposals and Section 4.5 summarizes the chapter.

## 4.1 A running example

To illustrate our proposal, let us use a simplified version of the Amazon ordering service, but complex enough since it considers many of its relevant features. The metamodel of this example is depicted in Figure 4.1, that defines nine types of objects and different kinds of relationships among them. Then, in this service there are **Customers** that can place **Orders**. **Orders** contain one or several **Items** and each **Item** corresponds to a certain **Product**. In addition, **Customers** can write **Comments** on the **Products**, and the system may suggest **Products** to the **Customers** by using **Offers**. The system can also create marketing campaigns to advertise some **Products**, called **AdCampaigns**. Finally, **Products** belong to **Departments**, and each **Customer** lives in a **GeographicalArea**.

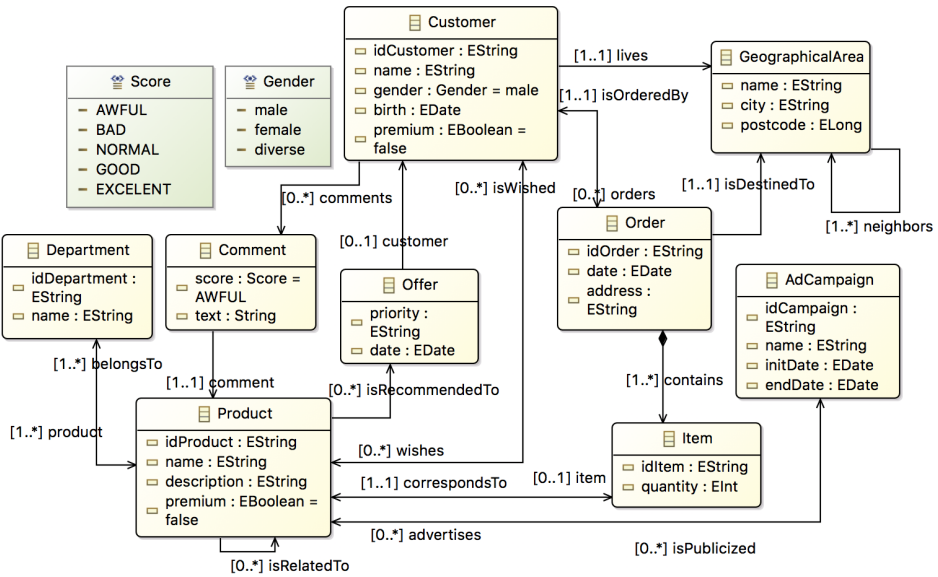


Figure 4.1: The Amazon Example Metamodel.

Considering this system, we are interested in defining some queries that represent situations of interest when processing data in this context. According to these situations, queries modify the source model by adding new elements or provide some other data such as returning a specific set of products. Then, queries are the following:

**Q1. CreateAdCampaign:** if a **Product** has been ordered more than 1,000

times during an advertising campaign period, and a relationship *isPublicized* between the **Product** and the **AdCampaign** does not exist, then the query creates it.

**Q2. UnpopularStock:** this query returns all **Products** that have been ordered by less than 3 **Customers** during last month.

**Q3. RelatedProducts:** if two **Products** have been included in at least 100 common **Orders** during the last month, then the query creates a link *isRelatedTo* between the **Products**.

**Q4. OlympicGamesTrending:** considering we have a Rio de Janeiro Olympic Games **AdCampaign**, the query obtains the **Products** that were ordered at least 100 times in Rio de Janeiro since the beginning of August 2016 until the end of the celebration of the Olympic Games. In this case, the query adds a relationship *isPublicized* between the **Products** and the Olympic Games **AdCampaign**.

**Q5. RecommendsPack:** if a **Customer** has ordered *Product1* at least 5 times in different **Orders** in the last month and this **Product** is related to *Product2* (through a *isRelated* connection), then an **Offer** for *Product2* is created for the **Customer**. Such an **Offer** has a priority of 1—highest priority. If *Product1* is related to *Product3* indirectly through another product, then an **Offer** for *Product3* with priority 2 is created for the **Customer**. In this case, we say that *Product1* is related with *Product3* in two hops. Similarly, if *Product1* is related to *ProductN* in n hops, the query would create an **Offer** with priority n. However, we only consider **Offers** from priority 1 to 3 in this query.

We can observe two main aspects of these queries that are interesting to evaluate our proposal. First, they create objects of type **Offer** and relationships of type *isRelatedTo* and *isPublicized*, that are not critical for the appropriate functioning of the service. Therefore, it is possible to approximate these elements in order to improve the performance. Second, they are not static, i.e., their result depend on when queries are performed. In this way, **Q2**, **Q3** and **Q5** consider data of the *last month*, **Q1** is associated to a specific advertising campaign period and **Q4** depends on when the celebration of the Olympic Games ends.

## 4.2 Approach

As we stated in previous chapters, we distinguish between two types of information depending on their nature: persistent and transient. The former is stored persistently in the system, while the latter is discarded after some time. In our proposal, all the information is stored in the source graph. Examples of persistent data in our running example are **GeographicalAreas**, **Customers** or **Products**, whereas examples of transient data are **Orders**, **Items** and **Offers**. Since transient data quickly change and increase or decrease in time, they can be considered infinite and have to be processed applying temporal windows. Furthermore, taking decisions from such data can be very time consuming. For this reason, approximating these data aims to decrease the amount of transient data to be processed. However, considering a subset of the complete model may lead to a lack of precision in the results that needs to be measured.

In this section, first we introduce several concepts in order to clarify our proposal. Then, we propose three techniques to approximate the source model in order to select just the part that is relevant for the query. These techniques are online with the processing, i.e. they are performed when the query is launched. Finally, we expose how to measure the outcome error produced because of these techniques.

### 4.2.1 Main concepts

We introduce the following terms for classifying the different kinds of models that we consider in our approach:

- **Source Model.** It refers to the complete data model that serves as input for the queries. It can be so large that it is not possible to be considered in full.
- **Pattern Model.** Queries made on this type of systems, such as the queries presented in our running example (cf. Section 4.1), typically focus on a part of the model. Specifically, the Pattern Model is limited according to two features: (i) a time range and (ii) the specific elements that are indicated in the query. For instance, **Q3** described in Section 4.1 focuses on the data

placed during the last month and only considers orders and products. Then, the Pattern Model is the subset of the Source Model that considers the elements filtered by the time range and element types that participate in the query.

- **Approximate Model.** Since the Pattern Model can still be very large when working with data streaming applications, we want to perform approximations in order to execute queries faster. Then, an Approximate Model is a subset of the Pattern Model, that it is selected by using AQP techniques (cf. Section 2.2.2). In our approach, we explored three techniques that we explain below. Note that there can be as many Approximate Models as subsets can be selected from the Pattern Model.
- **Optimal Model.** *Optimal Models* refer to those Approximate Models that meet the *best* trade-off between performance and accuracy. Please note that several Approximate Models could be considered Optimal Models, and this decision is ultimately made by the user. She decides so depending on the extent up to which the performance must be optimized in detriment of the accuracy.

As a summary, there is only one Source Model for each specific scenario and there is only one Pattern Model for each query of the scenario. In addition, there can be many Approximate Models for each Pattern Model, that are obtained applying different AQP techniques. Finally, one or more Approximate Models can be considered as Optimal Models.

### 4.2.2 Online AQP Techniques

In the literature, there are many different AQP techniques that perform online with the processing [32, 53, 75]. However, we consider the following three in our proposal:

#### Spatial Approximations

As we have mentioned along this document, information sources typically provides data structured as a graph. Please remember that we use models terminology,

so we use “objects” and “relationships” instead of “nodes” and “arcs”, respectively (cf. Chapter 2). Therefore, graph-structured data imply that objects are linked among each other through different types of relationships. In this way, we can navigate a model by starting in one object and traversing through the existing relationships. To clarify this, we define the concept of *hop*. A hop is the navigation from one object to another by the relationship that links them. For instance, in our running example and starting from one order (cf. Figure 4.1), we can determine in one hop the geographical area the order is destined to, by navigating the *isDestinedTo* relationship. Also, objects can be connected to other objects of the same type. For instance, from a geographical area we can reach, in one hop, all its neighboring geographical areas, through the *neighbors* relationship. Also, in two hops, we can reach all geographical areas that are neighbors of its neighbors, and so on. In this way, we can obtain *spatial windows* starting from one object and considering other objects reachable in  $n$  hops.

The concept of spatial window, which considers the idea of *spatial vicinity*, was presented as an extension of CEP windows [13]. Indeed, there are different strategies to define the vicinity graph of an element, depending on how we navigate through the graph structure, and the goal we pursue. Representative examples of algorithms for creating relevant vicinity graphs of nearby elements are used for finding related pages in the WWW [45, 71]. These algorithms use different strategies, e.g., going through the parents and children of a page, and then visiting the children and parents of those—using a backward-forward and forward-backward strategy. We could also traverse the graph moving only forward or backward, or using any other traversal strategy: in-breadth, in-depth, topological, hybrid, etc. Traversal could be done through any kind of link, or we could navigate the graph through some selected kinds of relationships.

Listing 4.1: Q4 with Spatial approximation.

```

1 // Select Olympic Games campaign
2 graph.traversal().V().hasLabel("AdCampaign")
3   .has("name", P.eq("Olympic Games")).as("campaign")
4 // Take property "endDate"
5   .values("endDate").as("end")
6 // Select Geographical Area with postal code 24495L
7   .V().hasLabel("GeographicalArea").has("postcode", P.eq(24495L))
8 // Traverse the graph through relationship "neighbors" with vicinity

```



```

9      .repeat(__.out("neighbors"))
10      .times(hops).emit().as("area").dedup("area").select("area")
11 //Select orders destined to the area
12 // and ordered before "endDate" property
13      .in("isDestinedTo").filter(__.values("date").where(P.lte("end")))
14 // Select products contained by the orders
15      .out("contains").as("product")
16 //Check there is not a previous relationship "isPublicized"
17 //between products and campaign
18      .not(__.select("product")
19          .outE("isPublicized").inV()
20          .where(P.eq("campaign"))))
21 //Count the number of matches between products and campaign
22 // and filter when they are at least 100
23      .select("campaign","product").groupCount().unfold()
24      .where(__.select(values).is(P.gte(100)))
25 // Add new elements to the graph
26      .select(keys).addE("isPublicized")
27      .from("product").to("campaign").dedup().iterate();

```

In order to illustrate how spatial approximations can be defined in our queries, Listing 4.1 shows the implementation of **Q4** in Gremlin language. In this query, a spatial window for geographical areas is shown from lines 7 to 10. The window starts from the area with postal code 24495, which is located in the centre of Rio de Janeiro in our models, and the `repeat-times` block indicates the number of hops to consider from this area in order to cover the complete area of Rio de Janeiro. In this case, the spatial approximation is indicated by the parameter *hops* in the `times` clause.

## Temporal Approximations

Since incoming data are typically tagged with the timestamp when they occur and data flows can be considered infinite (think for instance of all the information stored in Facebook during its lifetime), we can build *temporal windows* filtering by the data timestamp. A temporal window will be typically determined by the query, since queries are normally focused on a specific time range. The idea is to narrow down the Source Model by selecting the subset of the model indicated by the temporal window. Then, the Pattern Model is obtained. However, we can be interested in applying a further temporal approximation to the Pattern Model in order to obtain the Approximate Models. In this way, having a temporal window  $(t_i, t_e)$  of size  $N$  where  $t_i$  is the initial time and  $t_e$  is the end time of the

Listing 4.2: Q3 for Random and Temporal approximation.

```

1 // Select Product elements
2 graph.traversal().V().hasLabel("Product").as("product1")
3 // Select Orders that contains the Products inside a temporal window
4   .in("contains")
5   .where(__.values("date").is(P.inside(initTime, endTime)))
6 // Filter Orders by probability with coin step (Random approximation)
7   .coin(prob).as("order1")
8 // Select products in the same order
9   .out("contains").as("product2").where(P.neq("product1"))
10 // Check there is not a previous relationship "isRelatedTo"
11 // between products
12   .not(__.select("product1").outE("isRelatedTo").inV()
13     .where(P.eq("product2")))
14 // Count the number of matches between products
15 // and filter when they are at least 100
16   .select("product1", "product2").groupCount().unfold()
17   .where(__.select(values).is(P.gte(100)))
18 // Add new elements to the graph
19   .select(keys).addE("isRelatedTo")
20   .from("product1").to("product2").iterate();

```

window, it is possible to process just a sub-period of time  $(t_{a_i}, t_{a_e})$  with size  $n$  where  $(t_{a_i}, t_{a_e}) \subseteq (t_i, t_e)$  and  $n < N$ .

To illustrate how temporal approximations are integrated into our queries, observe Listing 4.2 that shows the code corresponding to **Q3** in Gremlin language. In line 5 a temporal window is implemented inside the **where** clause. In this case, the temporal approximation is applied narrowing down the parameters *initTime* and *endTime*, which correspond to the initial and ending time of the temporal window, respectively.

### Random Approximations

Approximate Models can also be obtained by applying random sampling techniques. This means that the decision on which elements of the Pattern Model will conform the Approximate Model is randomly made. For instance, we can assign a probability to each element of the Pattern Model to be included in the Approximate Model. Also, we can do approximations by element type. For example, we could determine that only 30% of the orders should be included in the Approximate Model. Many other random approximation techniques can be applied, as it is proposed in [121]. This is also a good approach when only transient data

needs to be approximated. Of course, random approximations can be combined with the other two.

To illustrate how random approximations are integrated in our queries, observe the query shown in Listing 4.2 again. A random approximation has been implemented applying the `coin` clause offered by Gremlin [6]. This function allows to run the query on the Approximate Model in which orders are considered depending on a probability. We can see in line 7 how the `coin` step is applied so that orders are processed depending on the probability value set with the parameter *prob*.

### 4.2.3 Measures for accuracy

In the following, we describe the metrics that we use in our proposal for computing the errors of the results, expressed as the differences between the expected and the resulting query outputs.

Since we are trading accuracy for performance, a very important aspect in our approach is to be able to measure both. We consider performance in terms of execution time of the queries. Regarding accuracy, we use the measures of *precision*, *recall* and *accuracy* [81]. These three measures are defined by formulas that include the concepts of true positives, false positives, false negatives and true negatives. In our context, we define and calculate them as follows:

- True Positives (TPs): number of elements created or returned as the result of a query on both the Approximate Model and the Pattern Model.
- False Positives (FPs): number of elements created or returned as the result of a query on the Approximate Model but not created or returned when running it on the Pattern Model.
- False Negatives (FNs): number of elements created or returned as the result of running a query on the Pattern Model but not created or returned when running it on the Approximate Model.
- True Negatives (TNs): number of elements that are neither created nor returned as the result of running a query on both the Approximate Model and the Pattern Model. While the calculation of the other three values is straightforward, the calculation of TNs is more complex. First, we need

to consider the maximum number of elements the query could create or return. For instance, in **Q2**, which returns all products ordered by less than 3 customers, if we have a total of 500 products, the total amount of products that could be returned in principle is 500. Let us name this amount as  $P_{re}$  (possibly returned elements). From this number, we need to subtract the amount of elements that are created or returned when the query is run on the Approximate Model, which is reflected in  $(TP + FP)$ . In summary, TNs are calculated as:  $TN = P_{re} - (TP + FP)$ .

Then, the three accuracy measures can be calculated as follow [81]:

- Accuracy: it is the most usual performance measure. In our context, it describes the effect of FPs and FNs when running queries on the Approximate Model. It is calculated as follows:  $Accuracy = (TP + TN)/(TP + TN + FN + FP)$ .
- Precision: this measure is useful to determine how accurate the model is when FPs are costly. For example, in email spam detection, a FP may cause loss of important information when a non-spam email is identified as spam. It is calculated as follows:  $Precision = TP/(TP + FP)$ .
- Recall: this measure computes how accurate the model is when FNs are costly. As an example, a FN on illness detection may cause catastrophic consequences on the life of the patient. It is calculated as follows:  $Recall = TP/(TP + FN)$ .

### 4.3 Performance Analysis and Evaluation

In this section, we discuss the performance of our approach. In order to evaluate our proposal, we executed the queries of Section 4.1 several times with the techniques exposed in Section 4.2.2 and different Source Model sizes. Then, the accuracy of the results were measured in the terms exposed in Section 4.2.3. Some charts are depicted in this chapter in order to explain and analyze the performance of our approach. However, since we have analyzed many different scenarios and obtained a large number of charts, they are all available in Appendix B.

### 4.3.1 Research Questions

In order to evaluate how the online AQP techniques that we have proposed behave when working with big graphs, we are interested in answering the following research questions (RQs):

- **RQ1 - How is performance improved when considering Approximate Models?** Running queries on Approximate Models is faster than running them on the Pattern Model. However, we want to check how much performance is gained depending on the sizes of the Pattern and Approximate Models, the type of approximation applied, and the distribution of the source data.
- **RQ2 - Are the 3 accuracy measures enough for identifying the Optimal Model?** Since we want to improve the performance of queries without compromising their accuracy, we want to discover whether the three measures presented in Section 4.2.3 are appropriate for measuring such accuracy.
- **RQ3 - Which approximation method provides the best trade-off between accuracy and performance?** Since there are different ways of approximating the Pattern Model, we want to find out which one is better, in terms of trade-off between performance and accuracy, depending on the source data.

### 4.3.2 Experimental Setup

In this section, we expose the source models and all parameters used to perform our experiments.

#### Source Models

Data stored in the models that we handle can be distributed in many forms. Considering the concept of time described in temporal approximations of Section 4.2.2, the data contained in our models can be concentrated in certain periods of time. For example, people are more likely to order products in their spare time, so evenings will normally concentrate more data than mornings. In this case, we

Distribution Batch	Name	Nodes	Edges
A	31K	286804	2399746
	62K	424,368	4,113,948
	125K	699,517	7,547,815
	250K	1,251,025	14,431,225
B	31K	287,731	2,477,232
	62K	425,836	4,201,686
	125K	699,945	7,635,425
	250K	1,252,316	14,543,380

Table 4.1: Summary of the models used in the experiments.

say that data is *focused* on evenings. As for the concept of vicinity introduced in spatial approximations of Section 4.2.2, more orders are likely to be made in Europe than in Africa, so data will be more concentrated along geographical areas in Europe. In this case, we say that data is *focused* in Europe. Now, if we consider only the data in the evenings or the data in Europe, we may have a more uniform distribution.

For experimentation purposes, we have manually created (according to the metamodel depicted in Figure 4.1) models of different sizes that contain information of the orders of Amazon Brazil in August 2016, and we suppose we are executing the queries in September 2016—i.e., *last month* in the queries (cf. Section 4.1) refers to August 2016. Furthermore, we have grouped source models in two batches, as we can see in Table 4.1. In batch A, data is uniformly distributed along the month, while in batch B data is mainly *focused* on the first week. In the table, the name of the model is assigned according to the number of **Customers** (cf. Figure 4.1). In this way, the smallest models contain 31K customers and around 255K objects linked among them by around 2.4M relationships. The largest models contain 250K customers and around 1M objects linked among them by around 14.5M relationships.

Being these models of different and considerable sizes and counting on different data distributions, we want to evaluate how approximating such models improves performance, and how much this compromises confidence in terms of precision, recall and accuracy values when executing the different queries.

### Queries and approximations

In the following we will consider the five queries presented in Section 4.1, which we think take into account aspects of queries that we could have in the real world. Regarding the different approximations to be made on the source data, we consider the three types of approximations defined in Section 4.2: time, spatial and random.

### Measurement method

Execution times of queries were measured running them and registering their computation times using the `System.currentTimeMillis()` Java method. In order to isolate our results from any transitory load of the machine where tests were run, all experiments were run up to 6 times each. Average performance values were extracted from last 3 runs.

### Experiments and data collected

Figures 4.2 to 4.6 show the execution times and accuracy obtained for each query, using different model sizes. The information displayed in each chart is the following:

- Data approximation. The type of approximation that is used in each experiment is displayed in the X axis. For instance, in the charts of Figure 4.2 a random approximation is applied. Thus, the X axis shows how much of the Pattern Model is being considered (i.e., it indicates the probability of each element of the Pattern Model to be included in the Approximate Model). The chart in Figure 4.3c shows a temporal approximation, so the X axis indicates the elements of the Pattern Model that are considered for the Approximate Model according to elements timestamps (cf. temporal approximations of Section 4.2.2). Finally, the chart in Figure 4.5c displays a spatial approximation, where the X axis indicates the number of hops (cf. spatial approximations of Section 4.2.2) taken from an initial object.
- Execution time. Whenever the execution time of the query is displayed in a chart, its values are shown on the left-hand-side Y axis, such as in the chart of Figure 4.2a. The performance evolution depending on the Approximate Model used is displayed with a continuous blue line.

- Number of elements returned by the query. Depending on the query, they can create relationships (such as **Q1**, **Q3** and **Q4**), objects and relationships (**Q5**) or return objects (**Q2**). The quantity of elements (objects and relationships) that are either returned or created by the queries is shown on the right-hand-side Y axis of the charts. The evolution of this quantity of elements depending on the Approximate Model used is displayed with a dashed orange line, such as in the chart of Figure 4.2a.
- Precision and Recall. When they appear in a chart, their values are shown in the left-hand-side Y axis (chart in figures 4.2b and 4.2d for instance). Precision and recall evolutions are shown with a continuous blue line.
- Accuracy. When it appears in a chart, its values are shown on the right-hand-side Y axis (such as in the chart in Figure 4.2b). Its evolution is displayed with a dashed gray line in the chart.

The rest of the results of the experiments are placed in Appendix B. In addition, the implementation of the queries and all the experiments can be accessed on our Git repository [16].

### Pattern Model vs Approximate Model vs Optimal Model

In the charts, as we move along the X axis, we see the results for different Approximate Models. For instance, let us focus in the chart of Figure 4.2a. When we see 0.45 in the X axis, it means that we are running **Q1** with an Approximate Model that includes 45% of the elements in the Pattern Model. Therefore, the right-most value in the X axis (1.0 in this chart) shows the result of the query when considering the whole Pattern Model. Regarding the Optimal Models, they are the Approximate Models such that, when running the query on them, obtain the *best* result. This means the best balance between performance and accuracy. Please note that the focus of our approach is not to automatically identify the Optimal Models, but to provide enough data (i.e., elements generated or retrieved by the query, and performance and accuracy values) for deciding what the Optimal Models should be depending on the user's needs and for knowing when they have been obtained.



## Execution environment

All experiments have been run on a MacBook running operating system macOS Sierra version 10.13.2 64-bit, with 16GB of RAM memory, and an Intel Core i7-6700HQ processor with 8 cores of 2.6 GHz. We used TinkerGraph-Gremlin version 3.3.4 [118] in our implementation, Java version 1.8.0\_144 and Gremlin-java version 2.6.0.

Experiments' replicability for this chapter is presented in Appendix D.1

### 4.3.3 Results

#### RQ1. Performance improvement

First of all, let us focus on the performance figures, shown in the charts of Figures 4.2a, 4.2c, 4.3a, 4.3c, 4.4a, 4.4c, 4.5a and 4.5c. In all of them, the smaller the Approximate Model considered, the faster the execution time. This means that the time taken by the Gremlin engine to filter the data that compose our conceptual Approximate Models pays off, since the engine runs the queries faster with smaller models.

Now, let us take the type of approximation into consideration. In random approximations, displayed in Figures 4.2a, 4.2c, 4.3a, 4.4a and 4.5a, the execution time increases linearly as the size of the Approximate Model grows. This happens in all cases, no matter how data is distributed in the source models. For instance, Figures 4.3a and 4.4a show the performance evolution for **Q3** applying a random approximation on the source model *62K*. Figure 4.3a shows the result with model *62K – BatchA* and Figure 4.4a with model *62K – BatchB*. We can see that there is not much difference in the execution times.

Regarding temporal approximations, shown in Figures 4.3c and 4.4c, we can see that execution times do not present much variation depending on how source data is distributed either, and that execution time also grows linearly. In a spatial approximation, such as the one shown in Figure 4.6a for **Q5**, we can see that execution time grows faster than linearly. This is reasonable, since a linear increase in the number of hops does not mean a linear increase of the data considered for the Approximate Model, but an exponential one. For instance, in **Q5**, hops are taken according to the *isRelatedTo* relationship among products. Since one product

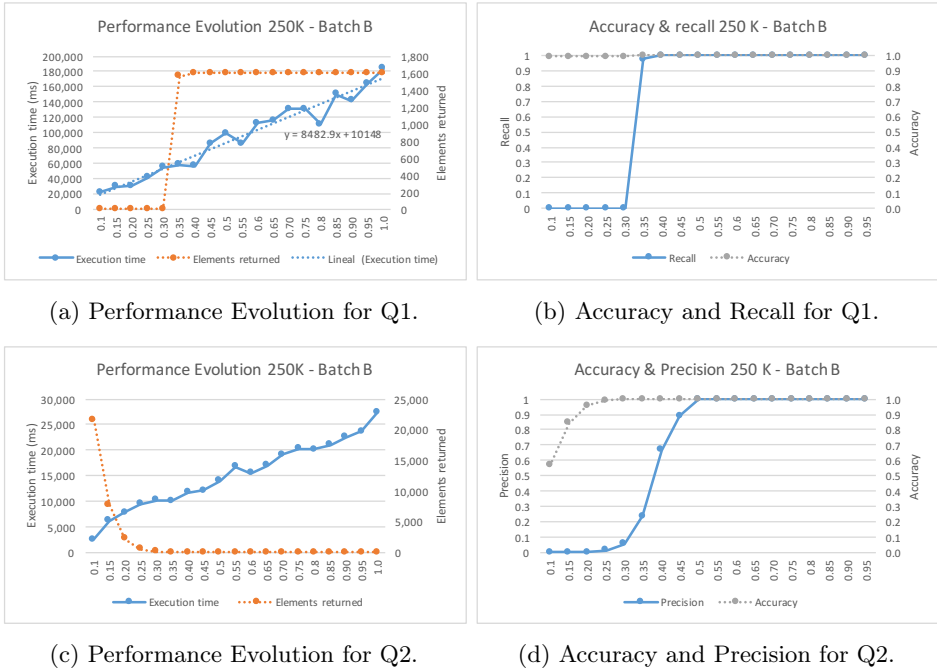
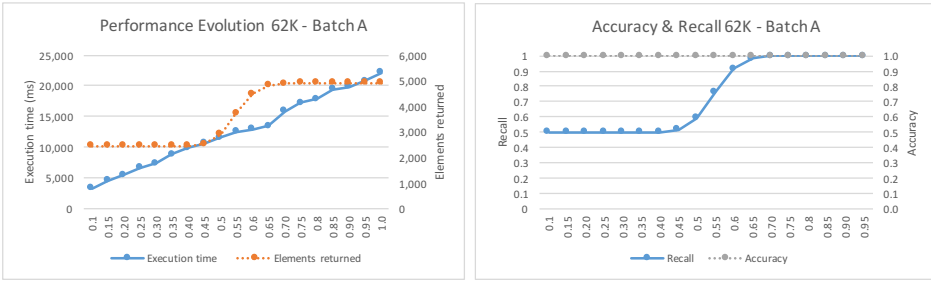


Figure 4.2: Accuracy, Precision and Recall with Random Approximations.

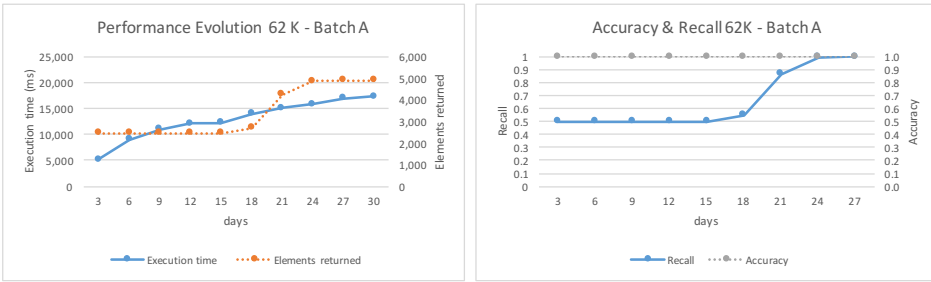
can be linked through this relationship with many others, doing one hop may imply considering many more elements; and even more if we take 2 hops, since the growth is exponential.

With all this data, we can answer RQ1 as follows:

- Performance improvement is noticeable when the amount of data considered for the Approximate Model is smaller than the amount of data in the Pattern Model.
- The execution time is directly proportional to the approximate model size considered.
- The time taken by the engine to filter data for obtaining the Approximate Model is not significant and it pays off.



(a) Performance Evolution for Q3 with Random Approximation. (b) Accuracy and Recall for Q3 with Random Approximation.



(c) Performance Evolution for Q3 with Temporal Approximation. (d) Accuracy and Recall for Q3 with Temporal Approximation.

Figure 4.3: Comparison between Temporal and Random Approximations with uniformly distributed data.

## RQ2. Validity of Accuracy Measures

Let us now consider **Q1** and **Q2**. **Q1** creates an *isPublicized* relationship when a product appears at least 1,000 times during an advertising campaign period. For this query, as the volume of data in Approximate Models decreases, some elements that should be returned are not, which means we get some FNs and no FP. Therefore, the precision value is 1 and the deviation from the query result with the Pattern Model can only be represented by the recall value. In Figure 4.2a we present the performance evolution for this query when run on model 250K – BatchB applying a random approximation. In this figure, probabilities go from 0.1 to 1 with increments of 0.1. As previously mentioned, execution time has a lineal increase. Regarding the elements created by the query, their number grows as the probability value of the horizontal axis increases, until the line eventually stabilizes and, at this point, we could consider we have reached the amount of

data considered for an Optimal Model. In order to double-check this, results for accuracy and recall values are shown in Figure 4.2b. The value of accuracy is not significant, since it is always 1. As for recall, it reaches 1 precisely when the number of elements returned gets stable.

Consider now query **Q2**, in which a product is returned if it has been ordered less than three times in a month. In this case, as the volume of data in the Approximate Model increases, the number of elements returned decreases. This means that we will have no FNs and therefore a recall value of 1, and therefore the precision value is the most significant for calculating accuracy. Performance evolution when running the query on model  $250K - BatchB$  is depicted in Figure 4.2c, while Figure 4.2d shows the accuracy and precision values for this query. For this experiment we have used a random approximation. If we expect a precision of 1 for considering an amount of data that conforms the Optimal Model, then we can see that we get it when we approximate half of the data of the Pattern Model.

Note that in most cases accuracy values are very close to 1. This is due to the influence of TNs in the accuracy equation. Consequently, the accuracy value is not descriptive enough to represent the deviation of running the query on Approximate Model versus running it on the Pattern Model. Therefore, we can come to a response for RQ2 as follows:

- Accuracy is not well suited as a measure for determining the amount of data to be considered in the Optimal Model.
- Precision and recall are valid measures when we get FPs and FNs, respectively.

### **RQ3. Trade-off between Accuracy and Performance**

Since we have different approximation possibilities for obtaining the data of Approximate Models, we want to find out which one is more convenient depending on the situation. First of all, temporal and spatial approximations only make sense when the query filters according to time, or to some spatial concept, respectively. Let us focus first on **Q3**. We have used random and temporal approximations for running it and have used the two types of source models presented in Section 4.3.2,

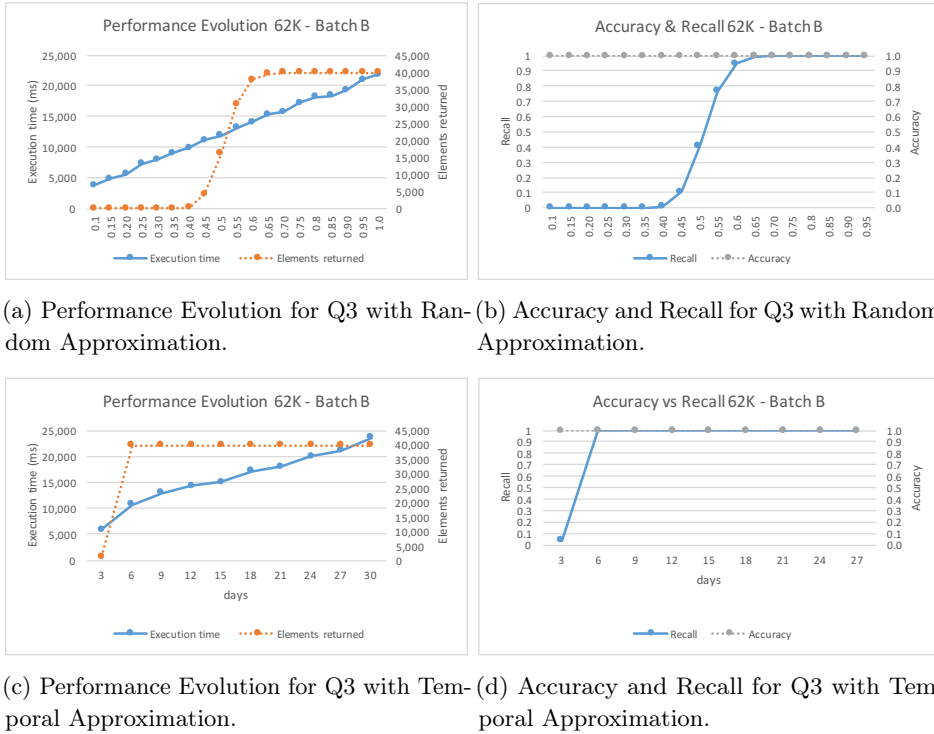
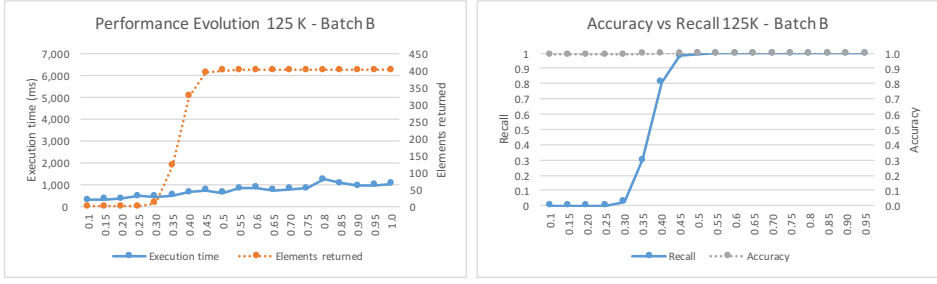


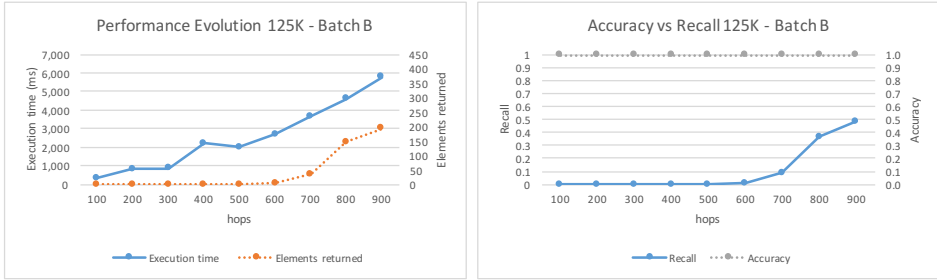
Figure 4.4: Comparison between Temporal and Random Approximation with temporal focus on the data.

this is, those where data is uniformly distributed along the month (Batch A) and those where there is a temporal focus on the first week (Batch B). In the random approximation, Approximate Models start considering 10% of the model, with increments of 5%, until we consider the Pattern Model (same as in the charts discussed before). In the temporal approximation, the increments contain the information of 3 days, i.e. the first Approximate Model considers the first 3 days in the month, the second one considers 6 days, the third one 9 days, and so on.

Figures 4.3 and 4.4 display execution results for models 62K – BatchA and 62K – BatchB, respectively. Let us focus first on Figure 4.3. Figures 4.3b and 4.3d display the recall (and accuracy, but, as we said before, we do not take this one into account) with random and temporal approximations, respectively. We can see that recall reaches the value 1 first in the random approximation, specifically, when 65% of the Pattern Model is considered in the approximation. In the temporal approximation, recall does not reach the value 1 until the approximation contains



(a) Performance Evolution for Q4 with Ran- (b) Accuracy and Recall for Q4 with Random Approximation.



(c) Performance Evolution for Q4 with Spa- (d) Accuracy and Recall for Q4 with Spatial Approximation.

Figure 4.5: Comparison between Spatial and Random Approximations.

80% of elements of the Pattern Model. Furthermore, in Figures 4.3a and 4.3c, we see that the execution time is lower in the random approximation when the number of elements returned stabilizes with respect to when the number of elements stabilizes in the temporal approximation.

Let us now consider Figure 4.4, which displays the results with model 62K – *BatchB*. Recall that data in this model has a temporal focus in the first week. Figures 4.4c and 4.4d reflect this. Indeed, we can see that recall reaches 1 when the approximation considers the first 7 days of the month, and we also see that the number of elements returned by the query stabilizes in this time. In the random approximation (Figures 4.4a and 4.4b), however, it occurs like in the example before: query result and recall stabilize when the approximation considers 65% of the model. In fact, we see that random approximations always behave the same, no matter how data is distributed in the source models. In temporal approximations, the accuracy of the query result depends on the temporal distribution of the data in the model.

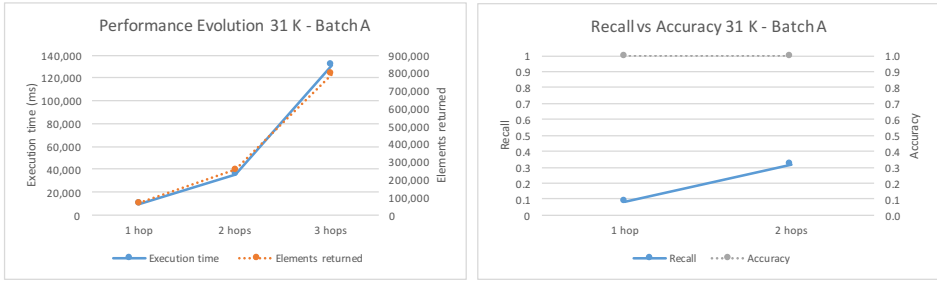
For comparing random and spatial approximations, we use **Q4** (Figure 4.5). Spatial approximation is made by considering geographical areas. In particular, we start from postal code 24495 (cf. Listing 4.1), which is located in the centre of Rio de Janeiro. In every increment in the approximation, we consider those areas within the next 100 hops (cf. Spatial Approximations in Section 4.2.2). Figure 4.5 shows the results with both approximations for model 125K – *BatchB*. Note that when running experiments applying spatial approximation, results do not stabilize. In fact, we discovered that Gremlin can only process a limited number of hops. In our case, we could not perform more than 900 hops, so the Approximate Model does not cover the complete area of Rio de Janeiro.

Apart from this limitation, we can see in Figures 4.5a and 4.5c that the execution time when considering spatial approximation is higher. This is because it is more expensive to traverse the model by applying hops (the model is traversed through objects and relationships among them) instead of filtering by attributes. We see that, in this case, random approximation is more efficient and accurate. However, there might be cases where it is not possible to filter by property, such as in some social networks analysis, where some conclusions can only be taken by traversing the model, so the use of a spatial window is essential.

Let us now consider **Q5** for further studying the performance evolution of spatial approximations. In this case, the starting point of the spatial window corresponds to each product which has been ordered at least 5 times by a customer, i.e. there are more than one object as initial point of the spatial window. Results for batch A and size 31K are shown in Figure 4.6. Please note that, since execution time results for this query have a very high value, we have run the experiments only for models 31K and 62K. As we see in Figure 4.6a, execution time seems to increase exponentially as the number of hops grows. Indeed, as we see in Figure 4.6b, recall grows slowly in comparison with the execution time. This implies that spatial approximations are also too costly when the initial point involve more than one object.

With all this information, we can answer RQ3 as follows:

- There is no approximation method that always provides the best trade-off between performance and accuracy.
- Random approximations typically behave the same no matter how data is



(a) Performance Evolution for Q5 with Spa- (b) Accuracy and Recall for Q5 with Spatial Approximation

Figure 4.6: Spatial Approximations with several sources

distributed in source models.

- Temporal approximations behave differently depending on how source data is temporally distributed.
- Approximating using hops (i.e. spatial approximations) is expensive in terms of runtime, but there might be systems or situations where it is the only possible approximation.

#### 4.3.4 Discussion

This section discusses some of the issues that we have found during the specification of the queries and the evaluation of their behavior. First of all, the concepts defined in Section 4.2 serve for explaining our approach from a conceptual point of view. However, we do not try to obtain first a Pattern Model and then Approximate Models. In our implementation, these tasks are implicitly done by the query, as we have shown in Listings 4.1 and 4.2.

As we have mentioned, the automatic computation of the Optimal Models is out of the scope of our approach. These Approximate Models offer the *best* balance between performance and accuracy. Since this balance can be subjective, this decision must be ultimately taken by the user.

Although performance is usually defined in terms of execution time and memory consumption, this chapter is focused on the first feature, since we consider it is the main concern in data processing applications. However, in order to have a first



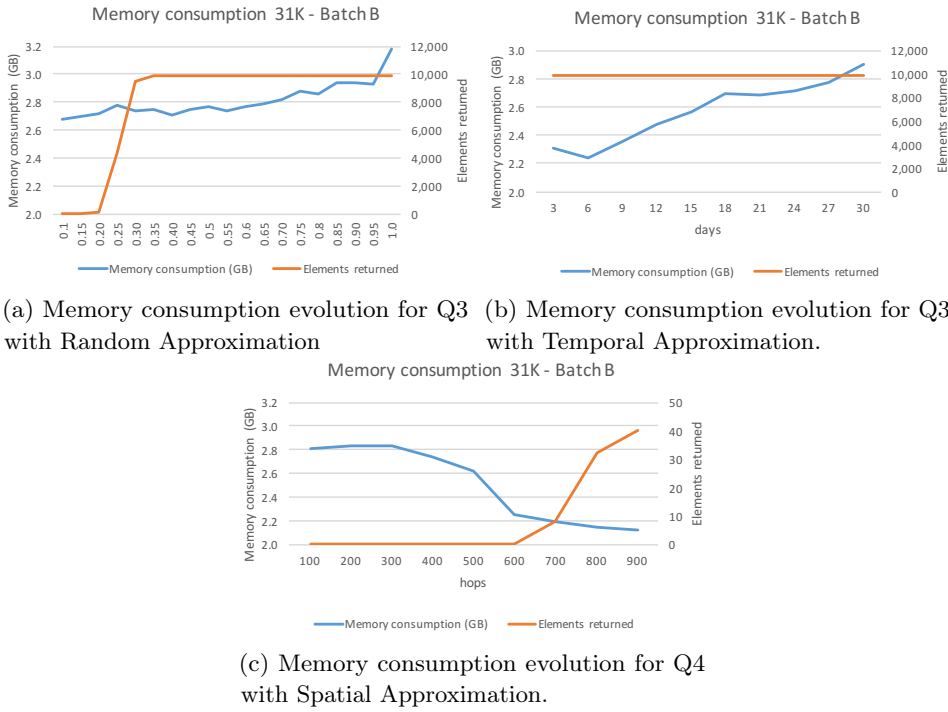


Figure 4.7: Memory consumption for Q3 and Q4.

estimation, we have measured memory consumption in model 31K of batch B for **Q3** with Temporal and Random approximations, and **Q4** with Spatial approximation. Results of the experiments are shown in Figure 4.7. As we can observe in Figures 4.7a and 4.7b, memory consumption grows as the Approximate Model increases its size for Temporal and Random approximations. However, in Figure 4.7c, memory consumption decreases as more hops are considered in the Spatial approximation. This may be due to the fact that some parts of the model are discarded as we do more hops.

The conclusions presented in the previous section can be summarized as follows:

- Random approximations are the best option when a query does not contain temporal or spatial filtering.
- Results of applying random approximations are similar no matter how the source data is distributed.
- If a query contains a temporal filter and the data is distributed with a

temporal *focus*, then it is convenient to use a temporal approximation centered on the focus.

- If a query contains a temporal filter but the source data is uniformly distributed, then random approximations seem to perform best.
- Spatial approximations by means of hops are very expensive in terms of runtime. They are only recommended when there is no other option.

#### 4.3.5 Threats to validity

In the following we describe the four types of threats that can affect the validity of our study, according to Wohlin et al. [129].

##### Construct validity threats

They are concerned with the relationship between theory and what is observed. Performance is typically measured in terms of memory consumption and execution time. In this chapter, we have only used execution time for performance comparison between different types of approximation techniques, with different queries and data distribution. Therefore, a possible construct validity threat is the use of only execution time for performance measurement. However, since one of the critical points of many data processing applications is to respond to the situations described by the queries as quickly as possible, we believe that this measure is descriptive enough for this work.

##### Conclusion validity threats

Threats to the conclusion validity are concerned with the issues that affect the ability to draw correct conclusions from the data obtained from the experiments. In our experiments, the transitory load of the machine where the experiments were executed can influence the execution time results. Besides, the elements returned for Random approximation experiments may have small variations depending on the input data. To mitigate both threats, we have run all experiments 6 times on the same machine and taken the average execution time and elements returned by the 3 last runs.

### Internal validity threats

These threats are related to those factors that might affect the results of our evaluation. In our experiments, we have considered five queries. Although they present a certain degree of variability, having considered more queries could have yielded different results. Furthermore, for classifying such queries, we have considered the type of window according to the definition of the query, and conclusions have been drawn according to such type of window. However, other features could have been considered in the classification, such as number and type of filters, action resulting from the query (deletion, insertion, update...) or traversal path. These concepts might also have an influence on the type of approximation recommended for each query in order to obtain the best trade-off between accuracy and performance. Finally, the temporal or spatial *focuses* present in the models influence the number of elements returned by Temporal and Spatial approximations. We have considered models with different types of focus. However, if we had consider a higher variability of these focuses, we could have obtained different results.

We will aim for the mitigation of these threats by considering the different dimensions mentioned for classifying the queries, as well as by considering more models with higher variability.

### External validity threats

These threats have to do with the extent to which it is possible to generalize the findings of the experiments. The first threat is that the results of our experiments have been obtained with one case study, which externally threatens the generalizability of our results. To mitigate this threat, we have tried to select five queries that consider different situations with different approximations. Furthermore, the case study has been selected from a real situation. Second, we have used Gremlin and TinkerGraph as the technologies for implementing our approach, since we concluded that they were the most suitable for our proposal (cf. Chapter 3). In any case, we do believe the same conclusions would have been drawn with similar technologies, such as Neo4j and Cypher. Finally, the results of the experiments depend on the size and distribution of the data. To mitigate this threat, all experiments have been run with different data sizes and distributions.

## 4.4 Related Work

The approach of this chapter was published in [14] and it derived from two previous works. First, [121] introduces the concept of Approximate Model Transformation (AMT), and the error from applying AMTs is calculated with statistical formulas. However, data are not related to each other, but they are just a stream of simple events. Then, [13] provides a prototypical solution based on CEP to process graph-structured information. The system was implemented with Spark and the *GraphX* tool for graph-parallel computation. The concept of *spatial window* was introduced using vicinity graphs. This chapter complements these works by exploring different strategies for selecting the subset of data to perform the approximations with graph-structured information (including spatial windows for Spatial approximations), and by providing mechanisms to estimate the accuracy of the different options.

Some studies have applied approximations similar to the ones we propose in this chapter. For instance, some works have used spatial windows when dealing with streaming models [37] and very large or even infinite models [34], where a *sliding window* limits the data to be processed at any given moment in time. Similarly, the work [67] uses a grid to represent a forest-fire spreading example, where the spreading is represented with vicinity links between the cells. Although our approach is in the context of these works, we specially focus on providing the necessary mechanisms for obtaining the accuracy of the results when applying different types of approximations. Regarding random approximations, other authors use sampling-based approaches to improve the execution time in large models [22]. However, their proposal is limited to models with up to 20,853 elements, clearly insufficient for the kinds of models our proposal targets, with more than 2 millions of elements.

There are some works that apply crowd-sourcing techniques [122, 123] and develop statistical tools to find a trade-off between cost/time and completeness of results. However, while the query result is constructed incrementally (the query is performed on an initial small dataset and the result is refined as more source data arrives to the system), in our proposal an initial large dataset is approximated at once. There are more works that deal with *incremental* queries and transformations [20, 65, 66, 99, 110, 124], where the input model changes with

time. However, they do not compute the error produced when not considering the complete model for the transformation, what is of key importance in our approach. There are some other works peripheral to ours. For instance, in [73] the authors select only a subset of the information in order to manage information overload, so they mainly aim for system survival, but sacrificing system reliability.

Some studies have proposed to apply precomputation in the context of AQP in database systems [32, 75]. This consists of storing a summary of the source data with interesting information for the query (offline precomputation step) and then using this summary to approximate the source information and performing the query. However, there are not many solutions that use these techniques with complex data structures such as graph-based structures [32, 75].

Finally, technologies such as the distributed streaming platform of Apache Kafka [68] or the streaming extension of Apache Spark [108] are aimed for processing information flows. They can generate streams of data as well as handle them. Our work does not pretend to replace these technologies, but it can rather complement them. In fact, our main goal is to obtain the best trade-off between execution time and information loss when processing large amounts of graph-structured information. An approach for reducing information to improve performance in complex stream processing scenarios has already been developed for Streaming Transformations for XML (STX) [33]. It consists of performing transformations to XML documents, which are provided as sequences of XML events, accessing just a part of the entire document in order to run the query. The part to be processed is selected with precomputing tasks. This work differs from our approach in two aspects: it works with XML-tree-like structures instead of graph structures and makes use of preprocessing tasks to perform the approximation of the source data.

### 4.5 Summary

In this chapter we have explored three different online AQP techniques for queries performed on models containing large amounts of data: (i) temporal, (ii) spatial and (iii) random approximations. Then, we have proposed a method for measuring the accuracy of the approximations, so that the user can find the right balance between accuracy loss and performance gain. The accuracy loss is calculated in terms of accuracy, recall and precision. We have also analyzed how

the distribution of the input data affects these approximations.

Approximation techniques were tested in a simplified Amazon ordering service case study, with different model sizes and two different data distributions.

Our experiments conclude that performance can indeed be improved. In fact, an optimal accuracy value can be acquired when considering only part of the source model.



# Chapter 5

## Improving performance with Offline techniques

---

As exposed in Section 2.2.2, AQP techniques are used to speed up the processing of the information of applications that work with huge amounts of data, e.g. Data Streaming Applications. These techniques select a subset of the dataset using different criteria, e.g., sampling techniques. The goal is to obtain only approximate results from the query. Although not necessarily correct, results should be precise enough to draw valid conclusions from them. One common feature of these approaches is that they deal with streams of data that represent sequences of loosely related events. However, this is not the case in many other applications, in which the information to be processed is structured as a graph of highly interconnected elements. It is well known that considering the relationships among the system elements may have a significant impact on the performance of the queries [111]. Our work focuses on these kinds of graph-structured information flows. In the literature, these kinds of graphs are commonly referred to as *dynamic graphs* [24, 25]. Although there is also the term *streaming graphs*, it typically



refers to settings in which there is no initial data and streaming is unbounded, i.e., one does not see the entire graph at any given time. Although the difference is sometimes blurred, we consider that the context of our work is that of *dynamic graphs*. Nevertheless, we will still use the term ‘data streaming applications’ to refer to those applications where new data constantly arrive.

As mentioned along this document, AQP is divided in two techniques, namely online and offline (cf. Section 2.2.2). In Chapter 4 we proposed three different online AQP techniques and estimated the accuracy of the results. The current chapter proposes an offline technique for querying graph-structured information flows that is able to deal with persistent and transient data at the same time. This technique aims at improving the performance of the queries by reducing the dataset to be processed. However, in contrast to conventional AQP techniques, it reduces the dataset without compromising the accuracy of the results, i.e., being able to produce fully correct answers. For this, we have developed a new algorithm, called Source Dataset Reduction (SDR) algorithm, that obtains a subgraph of the complete dataset (i.e., the model) with the relevant elements for a given query. In this way we are able to achieve speedups of more than 100x for some types of queries, even in already-optimized systems. The algorithm is executed before the query is performed for the first time. After that, an incremental version of the algorithm has also been developed, so that the relevant query subgraph is automatically updated, at a very low cost, when new information arrives, or the system data changes.

Our proposal has been evaluated using three case studies: (i) a simplified version of the Amazon ordering service; (ii) the New Yorker cartoon caption contest application, and (iii) a machine learning application for finding objects in Youtube videos. The first one is already used in Section 4.1 and illustrates our proposal. Remember that this case study uses synthetic datasets in order to have control over the possible configurations and sizes of the source data. The datasets used in the other two case studies are the real ones [100, 131]. Our solution has been implemented using the TinkerGraph in-memory graph database [118], since its execution time is lower than other similar solutions, and Gremlin has been used as a query language because it presents some relevant benefits over other graph query languages (cf. Chapter 3).

The contributions of this chapter are twofold. First, we develop a precomputing

algorithm (SDR algorithm) to reduce the source data in very large graphs. Second, we develop an incremental mechanism to handle graph data streams (Incremental SDR). Both contributions allow to improve the performance of the processing when working with Data Streaming Application with graph-structured information.

The structure of this chapter is as follows. Section 5.1 presents a case study that is used to illustrate our proposal. Section 5.2 describes a classification of query patterns that we have defined for the development of the algorithm that builds the subset of the relevant data depending on the query. Then, Section 5.3 presents the algorithm, which is evaluated in Section 5.4. Finally, Section 5.6 relates our work to other similar proposals, and Section 5.7 summarizes the chapter.

## 5.1 A running example

To explain our approach, we use the Amazon ordering service already presented in Section 4.1. However, in this case, we have designed six queries that represent different features of interest to our proposal. In this way, queries contain several operators that allow to study the behavior of our algorithm with six types of patterns that we found in the queries. These patterns are exposed in Section 5.2.

**Q1. ProductPopularity:** considering a specific product (e.g., the product with `idProduct = '10'`), this query returns the customers who have ordered that product. With this query, we can obtain the popularity of a product within the Amazon ordering network. It represents a query with a single selection filter or, alternatively, one with a conditional expression.

**Q2. AlternativeCustomer:** given a featured event, for example the Olympic Games, and a list of products that are known to be more frequently ordered than others during the event, this query obtains the customers who do not have any order that contains these products at that time. This query can be useful to improve advertisement campaigns in order to increase their success, recommending their products to those customers who have not ordered them yet. It represents a query that contains a negative application condition (NAC), i.e., a negation pattern.

**Q3. PackagePopularity:** considering two different products, i.e., with distinct `idProduct`, this query computes the customers who have ordered both. With this query, we obtain information about the frequency with which a customer orders two specific products, something that can be useful to create recommendations to

customers who have ordered only one of them. This query implements a conjunction of filters.

**Q4. SimProductsPopularity:** given two specific products that are known to be similar (for example two types of sports socks), this query gets the customers that who ordered at least one of them. This query is useful in order to discover the popularity of products with common attributes. It represents a pattern that implements a disjunction of filters.

**Q5. PrefCustomer:** This query returns the customers who have ordered a specific popular product more than 3 times. With this query we can create offers to customers according to the products that they often buy. This query implements an aggregation of filters.

**Q6. PrefCustomerSimProducts:** given two specific popular products that are similar, this query obtains the customers that have ordered one of them at least three times. This query is similar to **Q5**, since it is also helpful for suggesting offers to customers, but it uses an aggregation of selection filters.

Note that these queries only obtain information about the content of the Amazon ordering service, but they do not alter the system. Once this information is obtained, we would be able to create, remove or update elements in the system accordingly. Our approach is focused on optimizing the way information is queried, i.e., we are interested in optimizing the search for the elements that are retrieved by a query.

## 5.2 Classification of queries

In order to reduce the source data set according to the content of a query, we need to follow a strategy, which will depend on the type of the query. This is why we have defined a classification of queries that will allow us to decide how the algorithm should behave. It is not trivial to provide a classification for all the possible queries that can be defined by a user. Besides, it is important to decide whether queries are classified syntactically, semantically or both. Providing a semantic classification of the queries would depend on the case study and the context in which queries are applied. For this reason, we discard a semantic classification and focus on a syntactic classification. In this sense, we need to take into account the operators and clauses that constitute the query. Of course,

different patterns may appear in the same query. For instance, if we find a *where* clause, we talk about *conditional pattern*. Within the *where* clause, we can find other queries that follow other patterns. In the following, we describe the patterns that are relevant to our algorithm. Each pattern is atomically treated, i.e., they are described omitting any other pattern that could also be present in the query.

### 5.2.1 Simple filter pattern

Queries that follow the *simple filter pattern* sieve the information using only incoming and outgoing relationships and property filters. By incoming and outgoing relationship we mean a simple navigation step through an association. A property filter is used to obtain one or more elements of the graph according to the value of a property, or the type of object or relationship. Examples of property filters in the Amazon case study are a filter that obtains all customers older than 25, or one that obtains all objects of type `Product`.

Listing 5.1: Implementation of `Q1.ProductPopularity`.

```

1 g.V().as("customers")    // element type filter
2 .out("orders")           // relationship step
3 .out("contains")         // relationship step
4 .has("idProduct", "10") // property filter step
5 // returns the customers of the first step:
6 .select("customers")

```

Listing 5.1 shows a possible implementation in Gremlin of query **Q1** that follows this pattern. First, it selects all objects (line 1) and then it navigates through `orders` and `contains` outgoing relationships (lines 2 and 3). After that, it selects only those products whose `idProduct` is ‘10’ (property filter, line 4). Note how the `as` and `select` operators (lines 1 and 6) make the query return only those objects labeled as a variable `customers` that have ordered the products filtered in lines 2-4.

### 5.2.2 Condition pattern

Queries that follow the *condition pattern* select the information using a **where** clause, which specifies a sub-query with the condition that defines the filter.

Listing 5.2: Q1.*ProductPopularity* with **where** operator.

```

1 g.V() // element type filter - all objects
2 .where( // "where" step starts
3   __.out("orders") // relationship step
4   .out("contains") // relationship step
5   .has("idProduct","10") // property filter step
6 ) // "where" step ends

```

Listing 5.2 shows an implementation of query **Q1** that follows this pattern. It filters the objects that have a path indicated within the **where** clause (line 2). This path is composed by two relationships (lines 3 and 4) and a property filter (line 5), like in the previous example.

### 5.2.3 Negation pattern

Queries that follow the *negation pattern* sieve the information using a negative condition, selecting those elements that do not fulfil the condition expressed in a **not** clause. Listing 5.3 shows an implementation of query **Q2** that follows this pattern. In this case, the query selects the customers whose orders do **not** (line 3) contain any product of a list called **idProducts** (lines 4 and 5).

Listing 5.3: Implementation of Q2.*AlternativeCustomer*.

```

1 g.V().as("customers") // element type filter
2 .out("orders") // relationship step
3 .not( // "not" step starts:
4   __.out("contains") //relationship step
5   .has("idProduct", P.within(idProducts)) // property filter step
6 ) // "not" step ends.
7 // returns the customers of the first step:
8 .select("customers")

```

### 5.2.4 Conjunctive pattern

Queries that follow the *conjunctive pattern* select the information with an **and** clause that contains two or more predicates. The query selects those elements that satisfy all predicates.

Listing 5.4: Implementation of Q3.*PackagePopularity*.

```

1 g.V() // element type filter
2 .and( // "and" step starts:
3 // PREDICATE 1
4   __.out("orders")           //relationship step
5   .out("contains")           //relationship step
6   .has("idProduct", "10"), //property filter step
7 // PREDICATE 2
8   __.out("orders")           //relationship step
9   .out("contains")           //relationship step
10  .has("idProduct", "20") //property filter step
11 ) // "and" step ends

```

To illustrate an example of a query that follows this pattern, Listing 5.4 shows an implementation of query **Q3**. This query is composed of two predicates (lines 4-6 and 8-10). The first one filters customers who have ordered the product with `idProduct = '10'` (line 6) and the second one filters those who have ordered the product with `idProduct = '20'` (line 10).

### 5.2.5 Disjunctive pattern

Queries that follow the *disjunctive pattern* select the information with an **or** clause that contains two or more predicates. The resulting elements must meet at least one of these predicates.

Changing the **and** clause of Listing 5.4 (line 2) by an **or** clause, we obtain an implementation for query **Q4**, which is an example of a query that follows this pattern. The query selects the customers who have ordered a product with `idProduct='10'` or `idProduct='20'`.

### 5.2.6 Aggregation pattern

Queries that follow the *aggregation pattern* first group the information with aggregation operators, and then filter it with relational operators.

Listing 5.5: Implementation of *Q5.PrefCustomer*.

```

1 g.V() // element type filter
2 .has("idProduct", "10") // property filter step
3 .in("contains")         // relationship step
4 .in("orders")           // relationship step
5 .groupCount().unfold() // aggregation operation
6 .where(                 // aggregation filter

```

```
7  -- .select(values).is(P.gte(3))  
8  )
```

An example of a query that follows this pattern is presented in Listing 5.5, which shows an implementation of **Q5** in Gremlin. Note the aggregation operator used in line 5. It groups the customers by the number of times that they ordered the product with `idProduct = '10'` (lines 1-5). Then, it selects the customers who ordered this product at least 3 times (line 7).

### 5.3 The SDR algorithm

We have developed an algorithm for optimizing the performance of queries over graph-structured information models, by means of reducing the source dataset to be used by the query to the subset of the information that is relevant to it. Hence the name *Source DataSet Reduction* (SDR) algorithm.

This section describes the two versions of the SDR algorithm that we have developed. The first one is devised to be executed before the query is run for the first time, and computes the appropriate subgraph for the query (Section 5.3.1). The second version incrementally updates that subgraph when new elements arrive to the system, or the persistent information changes (Section 5.3.2).

An overall view of our proposal and all its components are depicted in Fig. 5.1. The implementation of the SDR algorithms and all artefacts used in their evaluation are available from [17].

#### 5.3.1 The main SDR Algorithm

The SDR algorithm for computing the subgraph of the complete information model that is relevant to a given query is inspired by Google's PageRank algorithm [96]. Given a set of web pages, the PageRank algorithm obtains a ranking of the most relevant web pages according to the number of pages that point to them, and their respective weights. To obtain such a ranking, the algorithm assigns a weight (a real number between 0 and 1) to each web page. A page's weight represents the probability that a person randomly clicking on Web links arrives at this particular page. For computing the weights, the algorithm performs several iterations. In the first iteration, it assigns the same probability to all pages: 1

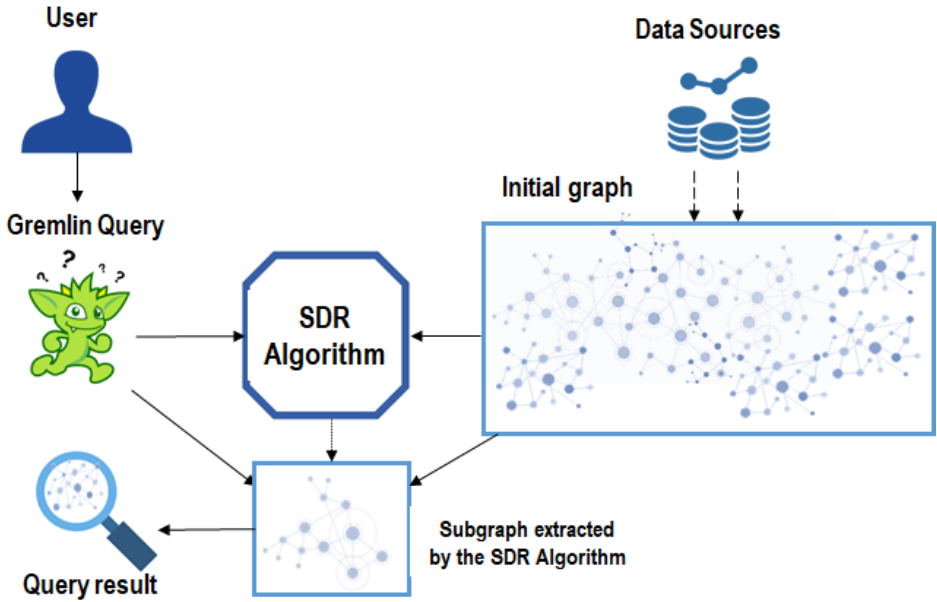


Figure 5.1: Overall view of queries using the SDR algorithm.

divided by the total number of pages. This probability is modified in the next iterations according to the number of links that the web page receives, and the weights of its neighbor pages in the previous iteration.

In a similar way, the SDR algorithm analyzes a query in order to assign a weight to all objects in the graph according to their relevance for the query. The algorithm returns a subgraph with the objects with a weight greater than 0 and the relationships among them. This subgraph contains all the elements that are relevant to the query. Therefore, considering that this thesis relies on MDE, this algorithm uses the information provided by the metamodel, by analyzing the query, in order to build the subgraph with the objects and relationships contained in the model. Note that, since the subgraph is obtained from the objects with a weight greater than 0, the numerical weight could be replaced by a Boolean value. However, even taking into account that this value is not relevant to the approach presented here, its calculation has been designed for future extensions that could integrate approximate algorithms, so that a further reduction in execution time and memory consumption could be achieved. This feature makes the current implementation more flexible.



A query is composed of different clauses, operations and filters, which in the context of this work we will call *steps*. That is, we consider that a step is any kind of clause, filter or operation that is applied to the elements of a model as specified by a query. According to the query patterns presented in Section 5.2, we consider eight types of steps: element type filter, property filter, relationship, *and* operation, *or* operation, *not* operation, *where* operation, and aggregation. A step can be, in turn, divided in sub-steps.

Let us illustrate the steps of a query with the *ProductPopularity* query of the Amazon case study, shown in Listing 5.2. It retrieves the users who have ordered the product with `idProduct='10'`. This query has two main steps, namely an element-type-filter step and a *where* step (lines 1 and 2, respectively). In our proposal, the aim of an element-type-filter step is to make the query focus on either the objects or the relationships of the graph. In this case, it focuses on the objects (line 1). Then, the *where* step selects the relevant objects, using three sub-steps: the **orders** and **contains** relationship steps (lines 3 and 4), which traverse the graph through the **orders** and the **contains** relationships, respectively; and a property filter step (line 5), which filters products by property `idProduct`. Our algorithm traverses all the query steps, starting from the most specific one, in order to assign a weight to the objects that match each step. We consider that the most specific step is always the last one (the *where* step in this example; the contained subquery, in turn, will be traversed starting from the filter-property step). The algorithm starts from the last step and traverses backwards the rest of the query steps.

The algorithm is executed in parallel on every single object. This parallel computation is possible by making use of Tinkerpop's VertexProgram [120], which implements the vertex-centric programming model [69, 80]. This programming model consists in an iterative process over a user-defined function that stops when a satisfying threshold is reached, or after executing a certain number of times. This process is executed in BSP (Bulk Synchronous Parallel) mode, which means the message passing among the objects is synchronized in order to avoid inconsistencies. In this way, VertexProgram is an interface for distributed graph computation, where each object is a “worker” that executes a program in parallel. Then, in each step of the query, the object sends a message through the relationships relevant to the step, and counts the number of messages that its neighbors sent to it in the

previous iteration. The weight is computed using the number of received and sent messages. The complete flow of the SDR algorithm is shown in Algorithm 1, which is described next. The inputs of the algorithm are the query  $Q$  and the graph  $G$ ; the result is the subgraph with the objects that are relevant to  $Q$ .

As stated before, the SDR algorithm traverses the steps of the query in several iterations. To achieve this, the function `SDRVertexCentric(Q,v)` is run in each object in parallel. For each step of the query, it checks whether object  $v$  satisfies the conditions to be assigned a weight. Variable `guardCondition` stores the results of these checks.

---

**Algorithm 1:** The main SDR algorithm

---

**Data:** A query  $Q$  and a Graph  $G(V,E)$

**Result:** A subgraph  $SG(V_{SG}, E_{SG})$

- 1:  $v.weight = \text{SDRVertexCentric}(Q,v) \forall v \in V$
  - 2: *ListSGIds* add  $\{v_w.id, v_w.weight\} \forall v_w \in V$  where  $v_w.weight \neq 0$
  - 3: **return**  $SG = G - \{v_d \in V \text{ where } v_d.id \notin \text{ListSGIds}\}$
- 

Function `SDRVertexCentric(Q, v)`

- 1: Obtain the set  $S$  of steps of  $Q$
  - 2:  $iteration = 0, weight = 0$
  - 3: **while**  $iteration \leq S.size$  **do**
  - 4:    $guardCondition = true$
  - 5:   **if**  $iteration == 0$  **then**
  - 6:      $s = S.get(S.size - 1)$
  - 7:      $weight = \text{WeightInitialisation}(s, v)$
  - 8:   **else**
  - 9:     Select  $s = S.get(S.size - iteration)$
  - 10:    **if**  $iteration == 1$  **then**
  - 11:      $weight = \text{InWeightPropagation}(s, v, weight)$
  - 12:    **else**
  - 13:      $weight = \text{FurWeightPropagation}(s, v, weight)$
  - 14:    **end if**
  - 15:   **end if**
  - 16:    $iteration++$
  - 17: **end while**
  - 18: **return**  $weight$
-

---

---

Function *WeightInitialisation*(*s*, *v*)

```

1: if s is property filter then
2:   if v matches the filter then
3:     pRel = previous relationship step of s
4:     cNeighbors = No. neighbors of v through pRel
5:     guardCondition = cNeighbors > 0?
6:   else
7:     guardCondition = false
8:   end if
9: else if s is a relationship then
10:  cNeighbors = No. neighbors of v through s
11:  guardCondition = cNeighbors > 0?
12: else if s is a TraversalParent filter then
13:  Obtain subqueries SQ from s
14:  for q : SQ do
15:    weightsSQ = SDRVertexCentric(q, v), q ∈ SQ
16:    weight = TraversalParentType(weightsSQ)
17:  end for
18: end if
19: if guardCondition then
20:  weight = weight + cNeighbors
21: end if
22: return weight

```

---

Function *InWeightPropagation*(*s*, *v*, *weight*)

```

1: if s is relationship and weight > 0 then
2:  Send messages through s
3: else if s is property filter or TraversalParent then
4:  pRel = previous relationship of s
5:  iteration++
6:  if weight > 0 then
7:    Send messages through pRel
8:  end if
9: end if
10: return weight

```

---

---



---

**Function** FurWeightPropagation( $s, v, weight$ )

---

```

1:  $cMessages = \text{sum}(\text{received messages})$ 
2: if  $cMessages > 0$  then
3:   if  $s$  is relationship then
4:      $cNeighbors = \text{No. neighbors of } v \text{ through } s$ 
5:      $guardCondition = cNeighbors > 0?$ 
6:     Send messages through  $s$ 
7:   else if  $s$  is a property filter then
8:      $pRel = \text{previous relationship of } s$ 
9:      $iteration ++$ 
10:    if  $v$  match the filters then
11:       $cNeighbors = \text{No. neighbors of } v \text{ thru } pRel$ 
12:       $guardCondition = cNeighbors > 0?$ 
13:      Send messages through  $pRel$ 
14:    else
15:       $guardCondition = \text{false}$ 
16:    end if
17:  end if
18: end if
19: if  $guardCondition$  then
20:    $weight = weight + cNeighbors + cMessages$ 
21: end if
22: return  $weight$ 

```

---

Similar to Google's PageRank algorithm, the first two iterations of the SDR algorithm are slightly different than the rest. PageRank uses an initial iteration, called iteration 0, to count the number of pages. Then, in iteration 1, this number  $N$  is used to calculate the initial weights of the pages (which is the same for all:  $1/N$ ). After this, the pages inform their neighboring pages about their current weight, so that weights can be updated in the following iterations according to the links to the page and the weights of the linked pages. In a similar manner, the SDR algorithm uses the initial iteration (function **WeightInitialisation**( $s, v$ )) to compute an initial weight of those objects that are relevant to the first step of the query. To compute this initial weight, the algorithm counts the number of relationships to the objects that are relevant to the step. Then, in the second iteration (function **InWeightPropagation**( $s, v, weight$ )), the objects inform, through

those relationships, their neighboring objects about their current weight. The remaining iterations (function `FurWeightPropagation(s, v, weight)`) will compute the objects' weights according to their relevance for the query and the relationships with the other relevant objects. In the following, an overview of the algorithm is explained by organizing it in three different subsections. A detailed exemplification of the algorithm with a concrete query is described in Appendix C.1.

### Iteration 0 - Weight Initialisation

When the algorithm starts, it calls the function `SDRVertexCentric(Q, v)` that will run over all objects in parallel (line 1). Then, this function retrieves (line 6 of `SDRVertexCentric`) the last step of the query,  $s$ , and calls the function `WeightInitialisation(s, v)`, (line 7 of `SDRVertexCentric`) that checks the type of  $s$ . Depending on the type of  $s$ , `WeightInitialisation` may proceed in different ways:

- If  $s$  is a **property-filter** step (line 1), it checks whether  $v$  matches the filter (line 2) or not. If not, *guardCondition* is set to false (line 7). Otherwise, the algorithm traverses the query backwards until it finds a relationship step and counts the number of neighbors of  $v$  that can be reached through that relationship. If this number is 0, *guardCondition* is set to false (line 5).
- If  $s$  is a **relationship** step (line 9), the function counts the number of neighbors that the object  $v$  reaches through this relationship and checks whether this number is greater than 0. Otherwise, *guardCondition* is set to false (line 11).
- If  $s$  is a **traversal**<sup>1</sup> step (line 12), the function makes a recursive call to the `SDRVertexCentric` function for each subquery of  $s$  and uses the appropriate strategy to compute the weight depending on the type of traversal (lines 13-17). All the different strategies are explained and exemplified in Appendix C.2.

---

<sup>1</sup>A `TraversalParent` in Gremlin includes steps that imply one or more subqueries, namely *where*, *and*, *or* and *not*.

- Steps of types **element-type-filter** and **aggregation** are not considered because they do not affect the weight of  $v$ . The first ones are only used to select the objects or relationships that will serve as starting point of the query, while the second ones are used for grouping the information obtained in the previous steps. For this reason, the aggregation steps are removed from the query before making any call to the SDR algorithm, so that the algorithm skips this step when analysing the query.

After that, the weight of  $v$  is computed if *guardCondition* is true (line 19 of *WeightInitialisation*). The weight is calculated as the addition of two parameters (line 20 of *WeightInitialisation*): the accumulated weight and the number of neighbors reachable through the relevant relationship to  $s$  (*cNeighbors* value). After updating the weight, the *WeightInitialisation* function concludes and the *SDRVertexCentric* function starts the next iteration (line 16 of *SDRVertexCentric*).

### Iteration 1 - Initial Weight Propagation

After *WeightInitialisation*, all objects have a weight but they are not aware of their neighbors' weights. This is the goal of the *InWeightPropagation*( $s, v, weight$ ) function, which proceeds with the same step  $s$  (lines 9 to 12 of *SDRVertexCentric*). The behavior of *InWeightPropagation* depends on the kind of step, as above:

- If  $s$  is a **relationship** step, and weight is greater than 0 (line 1), it means that  $v$  met the *guardCondition* in the initial iteration, so it sends a message through that relationship to its neighbors (line 2).
- If  $s$  is a **property filter** or a **traversal** step (line 3), there is no relationship through which object  $v$  can send the messages, and therefore the algorithm searches backwards in the query for the next relationship step, *pRel* (line 4), and increments the iteration counter accordingly (line 5). This increment is needed because two steps are analyzed in this case:  $s$  and *pRel*. Note that in cases with multiple calls to property filter steps, they are grouped and considered as a single step. In the same way, a chain of several traversal steps

is processed as an *and* step, which is also considered a single *TraversalParent*.

Then,  $v$  sends the messages through  $pRel$  if its weight is above 0 (line 7).

After that, the `InWeightPropagation` function concludes and the `SDRVertexCentric` function starts the new iteration (line 16 of `SDRVertexCentric`).

### Remaining iterations - Further Weight Propagation

For the rest of the iterations, `SDRVertexCentric` calls the `FurWeightPropagation( $s, v, weight$ )` function (line 13), which checks whether the object  $v$  is relevant to the step  $s$  (i.e., it received messages in the last iteration) or not, and proceeds depending on the type of  $s$ :

- If  $s$  is a **relationship** step (line 3), and  $v$  has neighbors through  $s$ , it sends messages to its neighbors; otherwise *guardCondition* is set to false (line 5).
- If  $s$  is a **property filter** step (line 7), it finds the preceding relation through  $s$ ,  $pRel$ , and proceeds as in the `InWeightPropagation` function.

Then, `FurWeightPropagation` updates the value of *weight* by adding parameters *cNeighbors* and *cMessages* (the number of messages received in the last iteration), as shown in lines 19 to 21. Once `SDRVertexCentric` is executed on the objects of the initial model, the subgraph composed of the non-zero weight objects and the relationships among them will contain the subgraph that is relevant to the query. This subgraph is calculated in two steps. First, we create a list that contains all the ids of the non-zero weight objects and store it in memory (line 2 of Algorithm 1). We call this list *ListSGIds*. Second, we obtain the induced subgraph from the objects that appear in the list (line 3 of Algorithm 1).

Note that computing such a subgraph can be done in parallel, since different threads can calculate the weights of distinct subsets of objects.

### 5.3.2 Incremental SDR Algorithm

Our approach is designed for dynamic systems that are constantly updated with new information. Executing the main SDR algorithm (Section 5.3.1) on all objects every time the graph changes would be too costly in terms of time and

memory. For this reason, we have developed a so-called *Incremental SDR algorithm* that updates the weights of the graph nodes when new elements are added or existing elements are updated or discarded. This represents the arrival of new information to the system, changes in the graph persistent data or the removal of old information. This way, the main SDR algorithm needs to be executed only once, and then updated every time the graph information changes.

---

**Algorithm 2:** The Incremental SDR algorithm

---

**Data:** A set of objects  $V_n$ , a query  $Q$  and a Graph  $G(V, E)$   
**Result:** A subgraph  $SG(V_{SG}, E_{SG})$

- 1: Obtain steps  $S$  from  $Q$
- 2: Initialise an empty subgraph  $SG_i(V_i, E_i)$
- 3: **for**  $s : S$  **do**
- 4:   **if**  $s$  represents a relationship **then**
- 5:      $SG_i = SG_i \cup \text{createSubGraph}(s, V_n)$
- 6:   **else if**  $s$  represents a TraversalParent **then**
- 7:     Obtain subqueries  $SQ$  from  $s$
- 8:     **for**  $q : SQ$  **do**
- 9:       Obtain steps  $S_{SQ}$  from  $q$
- 10:       **for**  $s_{SQ} : S_{SQ}$  **do**
- 11:          **if**  $s_{SQ}$  represents a relationship **then**
- 12:            $SG_i = SG_i \cup \text{createSubGraph}(s_{SQ}, V_n)$
- 13:          **end if**
- 14:       **end for**
- 15:     **end for**
- 16:   **end if**
- 17: **end for**
- 18:  $v_i.\text{weight} = \text{SDRVertexCentric}(Q, v_i) \forall v_i \in V_i$
- 19:  $ListWeights = \text{get weight and id from } SG_i$
- 20: Update  $ListSGIds$  with  $ListWeights$
- 21: **return**  $SG = G - \{v_d \in V \text{ where } v_d.id \notin ListSGIds\}$

---

When new elements are added, updated or removed from the graph, we analyze the query and obtain a list of the neighbors of these objects that can be reached through the relationships of the query, with the aim of updating their weights. This is performed by the Incremental SDR algorithm shown in Algorithm 2. Our approach only updates the weight of the objects that arrive or are modified in the system, since removed objects do not need to have their weights updated. It also



updates the weight of the objects that can be affected because of a change in the graph structure, i.e. the objects that can be reached from the added, updated or removed objects through the relationships of the query. Note that these changes in the graph are always at model level. Therefore, the incremental SDR algorithm analyzes the query again to re-compute the subgraph with the new data of the model, i.e. it works at metamodel/model level to compute the new subgraph, as we already explained in Section 5.3.1

Typically, more than one object will be added, discarded or modified in the graph at the same time, because events usually arrive in batches. Thus, the inputs of Algorithm 2 are a set of objects  $V_n$ , the query  $Q$  and the graph  $G(V, E)$ . The set  $V_n$  contains those objects that are added or updated in the graph, plus the set of neighbors of the recently removed objects. The Incremental SDR algorithm traverses the relationship steps of the query (lines 3-17) to find all objects that are connected through these relationships with the objects of set  $V_n$ . With this information, it creates a subgraph  $SG_i$  and calls the vertex-centric function of the SDR algorithm with the objects contained in  $SG_i$  and  $Q$  as inputs (line 18). Once the SDR algorithm finishes, it returns the subgraph  $SG_i$  with its corresponding weights. Then, the algorithm extracts from  $SG_i$  the objects' ids and their corresponding weights (line 19). After the execution of the Incremental SDR algorithm, the resulting *ListWeights* is analyzed to obtain the updated weights of the objects of  $SG_i$ , and the list *ListSGIds* (cf. Section 5.3.1) is updated with the new weights of these objects (line 20). Finally, the algorithm returns the updated subgraph  $SG$  to be queried (line 21).

The Incremental SDR algorithm uses the function `createSubGraph`, whose pseudo-code is shown in Algorithm 3. It receives as inputs the current relationship step  $s$  of the query and the set of objects  $V_n$ , and returns all neighbors of the objects  $V_n$  that can be reached through  $s$  in the query. First, the function selects the step  $s$  and its forward and backward relationships (lines 2 and 3) in order to get the neighbors of  $V_n$  that can be reached through them (lines 4–6), and returns the subgraph composed by the set  $V_n$  and their neighbors. Note that forward and backward relationships do not necessarily imply outgoing and incoming edges, respectively. They refer to the relationship steps found when we traverse the query forwards and backwards. For instance, if we start to analyze the query `g.V().hasLabel("Order").in("orders")` from the `hasLabel` step, the next step

obtained when we traverse the query in the forward direction is the **in** step, which implies an incoming relationship.

---

**Algorithm 3:** Function createSubGraph
 

---

**Data:** A step  $s$  and an a set of objects  $V_n$   
**Result:** A subgraph  $SG(V_{SG}, E_{SG})$

- 1: Initialise an empty subgraph  $SG(V_{SG}, E_{SG})$
- 2:  $next_r = s \cup \text{forward relationships of } s$
- 3:  $previous_r = \text{backward relationships of } s$
- 4: **for**  $n : next_r \cup p : previous_r$  **do**
- 5:    $SG = SG \cup \text{neighbors of } V_n \text{ through } n \text{ and } p$
- 6: **end for**
- 7: **return**  $SG$

---

The fact that the Incremental SDR algorithm has to update only the weights of the neighbors of the newly added, updated or discarded objects from the graph does not represent a significant performance overhead, since the complexity of the algorithm is  $\mathcal{O}(v \cdot r \cdot n)$ , where  $v$  is the size of  $V_n$  (i.e., the number of new, updated or neighbors of discarded elements),  $r$  is the number of relationships of  $s$ , and  $n$  is the number of neighbors of  $V_n$  through  $s$ . Given that these numbers are normally small, the execution time of this algorithm is not significant when compared to the execution of the query. Besides, this incremental algorithm is executed in parallel with the queries, so it does not affect their performance. The only introduced penalty is due to the final update of the query subgraph after the execution of the incremental algorithm, since the update procedure uses a lock to avoid inconsistencies in the subgraph. In this way, if a new modification (addition, deletion or update) takes place in the source graph while the SDR algorithm is running, the algorithm finishes its execution on the data that was available when the execution was launched. Then, the new modification occurs and the SDR algorithm is launched again in order to calculate the new subgraph.

## 5.4 Performance Analysis and Evaluation

In this section, we discuss the performance of the SDR algorithm exposed in Section 5.3. In order to evaluate the algorithm, we tested it with three case studies. One of them was generated from synthetic data (cf. Section 5.1) and the rest were

extracted from real datasets [100, 131]. Then, we measured the performance in terms of execution time and memory consumption.

#### 5.4.1 Research Questions

To evaluate our proposal, we are interested in answering the following research questions:

- **RQ1: How much is the graph reduced by the SDR algorithm?**  
Given a query and a graph, applying the SDR algorithm returns a subgraph with the information needed for running the query. Our hypothesis is that the ratio of size reduction is related to the type of patterns used in the query. Therefore, we want to know the relation between the query patterns and the ratio of size reduction.
- **RQ2: What is the performance gain when running the query on the subgraph, instead of running it on the original graph?**  
Our hypothesis is that running queries on the reduced subgraph is much faster and consumes less memory than running them on the original graph. However, depending on the pattern followed by the query the performance improvement might differ, and be more or less significant. We want to analyze this.
- **RQ3: Considering data streaming applications, what is the break-even point of our approach?** The SDR algorithm implies additional time and memory costs when initially computing the subgraph. Our hypothesis is that these initial costs are compensated as soon as the query is executed several times. We want to analyze the break-even point, i.e., how many queries are needed to amortize such initial costs, making our approach worthwhile.

#### 5.4.2 Case Studies

In order to evaluate our proposal and to try to generalize the results, we have performed our experiments in three case studies, which are explained in the following.

The first case study corresponds with the metamodel depicted in Figure 4.1 and queries of the running example exposed in Section 5.1.

This case study is extracted from the New Yorker caption contest dataset [131]. This dataset provides approximately 89 million ratings over 750,000 captions in 155 contests. The contests are part of the “cartoon caption contest”, where users have to rate cartoons and captions according to how funny they are through two types of questions:

- [illegible]

The metamodel of this case study is depicted in Figure 5.2. It is conformed by **Participants** who give two types of **Answers**: **Choices** in a **Dueling** question or

**Ratings** in a **Cardinal** question. In addition, each **Question** belongs to a **Contest** and is generated from an **Algorithm**. In this case, the following situations of interest are taking into consideration:

- **Q1. RecentPart:** considering all **Contests** in the system, getting the number of **Participants** who have answered at least one **Question** in a contest in the last month.
- **Q2. ContestPart:** considering a specific **Contest**, obtaining all the **Participants** who have only answered one **Question**.
- **Q3. UnchosenCap:** considering a specific **Caption**, counting how many times a caption appeared in a dueling contest question and it was not eventually chosen.
- **Q4. FunniestCaption:** getting the highest scored **Caption** in a cardinal contest. The highest scored caption is considered as the most voted caption tagged as ‘funny’.
- **Q5. Abandon:** obtaining all **Participants** that answered one **Question** only. This query might be useful when deleting participants’ answers considered as irrelevant.
- **Q6. FunniestCaptionU:** same as **FunniestCaption**, obtaining the highest scored **Caption** taking into account all **Questions** generated by a random **Algorithm** only. In this way, the result of this query is unbiased.

### Youtube Videos

This case study uses the YouTube-BoundingBoxes dataset [100], which consists of approximately 380,000 video segments of 15 to 20 seconds extracted from 240,000 Youtube videos. In these segments, the presence or absence of 23 different objects were annotated by humans. The dataset is aimed at training machine learning algorithms.

The metamodel of this example is shown in Figure 5.3. It is conformed by **Videos**, which are composed of **Segments** where the **Object** is searched. Each

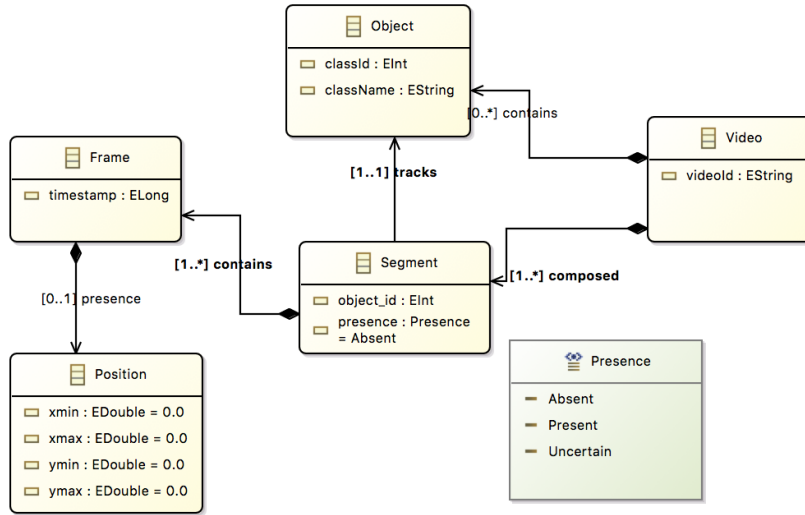


Figure 5.3: Youtube Videos metamodel

**Segment** contains one or several **Frames** where the **Object** may appear in a specific **Position**. In this case, we are interested in the following situations:

- **Q1. GetAnimalVideos:** obtaining all **Videos** that contain an animal. Animal tags in this dataset are the following: “cat”, “dog”, “bird”, “zebra”, “cow”, “bear”, “horse”, “giraffe” and “elephant”.
- **Q2. NotPresent:** getting the **Segments** where the **Object** is not present in any of its frames.
- **Q3. AnimalPerson:** returning all **Videos** that contain at least an animal and a person.
- **Q4. PresentSoon:** obtaining all **Videos** where the **Object** is present during the first 3 seconds.
- **Q5. Pets:** getting all **Frames** that contain a cat or a dog.
- **Q6. InCast:** returning all **Videos** where the **Object** is present in at least 10 **Segments**.

### 5.4.3 Experimental Setup

In this section, we expose the source models and all parameters used to perform our experiments.

#### Source Models

Our experiments have been run on models of different sizes in order to analyze the performance of our approach. We have created models for the three case studies exposed in Section 5.4.2. First, we have used the models of batch B that were created for the study of Chapter 4 (cf. Section 4.3.2) for the experiments with Amazon case study. Second, we have parsed to TinkerGraph format and imported the models from the sources provided in [131] for the Contest case study. Finally, we have also parsed to TinkerGraph format and imported the models from the sources provided in [100] for the Youtube case study. The number of objects and relationships for each model are shown in Table 5.1. Since the models of the different case studies conform to different metamodels, the size of the models have been chosen to have a similar growth curve. Note that the smaller models have between 1.5 and 2.5 million elements (adding objects and relationships), while the larger models contain between 12 and 16 million elements.

Models are named according to the approximate sum of the number of their objects and relationships.

#### Queries

As described in Section 5.2, queries can follow different patterns. To determine the performance of our proposal we have defined several queries, each one following a different pattern. The number of steps of the queries ranges between 3 and 11. The analysis of our approach with queries that combine more than one pattern is left as part of future work.

Table 5.2 summarizes all the queries we have used. They are fully described and implemented on our Git repository [17].

Note that the objects of queries that involve a specific object (e.g., a particular product in **ProductPopularity** or a particular contest in **ContestPart**) consider the worst-case scenario, i.e., they select the object with a higher number of rela-

Case study	Name	Objects	Relationships
<b>Amazon</b>	2M	286,804	2,399,746
	4M	424,368	4,113,948
	8M	699,517	7,547,815
	15M	1,251,025	14,431,225
<b>Contest</b>	1M	279,170	929,010
	4M	1,162,164	3,591,820
	9M	2,240,240	6,789,472
	12M	3,096,948	9,333,592
	16M	4,010,120	12,048,874
<b>YouTube</b>	2M	944,945	971,781
	4M	1,888,351	1,942,056
	6M	2,830,563	2,911,132
	8M	3,775,098	3,882,562
	10M	4,717,843	4,852,181
	12M	5,661,552	5,822,785

Table 5.1: Summary of the models used in the experiments.

tionships with the rest of the network. This makes our algorithm build the largest possible subgraph.

### Execution environment

All experiments have been executed on a machine running the Ubuntu operating system 16.04.5 LTS 64 bits, Linux kernel 4.4.0-151-generic, with 64GB of RAM, and an Intel Xeon CPU E5-2680 processor with 16 cores of 2.7 GHz. Our implementation used TinkerGraph-Gremlin version 3.3.4 [118], Java version 1.8.0\_144 with Oracle JDK vendor and Gremlin-Java version 2.6.0. Besides, we set to 30G the memory allocation pool of the JVM to obtain the maximum size.

Experiments' replicability for this chapter is presented in Appendix D.2

#### 5.4.4 Experiments and data collected

We have performed two sets of experiments. The first one focuses on querying static information on large models. The second one queries new information as it is added to the model, i.e., it deals with streams of information.

Note that we consider additions in these experiments, since they imply an



Case study	Query name	Query pattern
<b>Amazon</b>	ProductPopularity	Simple
	ProductPopularityC	Conditional
	AlternativeCustomer	Negation
	PackagePopularity	Conjunction
	SimProductsPopularity	Disjunction
	PrefCustomer	Aggregation
	PrefCustomerSimProducts	Aggregation
<b>Contest</b>	RecentPart	Simple
	ContestPart	Conditional
	UnchosenCap	Conjunction
	FunniestCaption	Aggregation
	Abandon	Aggregation
	FunniestCaptionU	Aggregation and Conjunction
<b>YouTube</b>	GetAnimalVideos	Conditional
	NotPresent	Negation
	AnimalPerson	Conjunction
	PresentSoon	Conjunction
	Pets	Disjunction
	InCast	Aggregation

Table 5.2: Summary of the queries used in the experiments.

increment in the volume of the graph, i.e., they are the most costly operation when working with streams. Then, results reflect the behavior of our approach in the worst-case scenario. This way, we aim to evaluate both our SDR Algorithm (Section 5.3.1) and its incremental version (Section 5.3.2). Both sets of experiments are described next.

### Experiments with static information

The idea of these experiments is to perform queries on both the original graph and the subgraph obtained by the SDR algorithm. We want to compare three aspects, namely (i) execution time, (ii) memory consumption and (iii) number of elements in the graphs.

For this, we applied the SDR algorithm to all models and queries listed in Tables 5.1 and 5.2, respectively. Table 5.3 shows the ratio of elements that are removed from the original graph as a result of running the SDR algorithm in each specific case study for each particular query. Columns 1, 2 and 3 indicate

the case study, the name of the query, and its type, respectively. Columns 4 to 9 show the ratio  $R$  of elements that are removed for each model. That ratio is calculated as  $R = 1 - \#T_{sg}/\#T_g$ , where  $\#T_g$  and  $\#T_{sg}$  represent the number of elements (objects and relationships) in the graph and subgraph, respectively. Hence,  $R = 0.94$  means that the subgraph contains only 6% of the elements of the original graph.

In addition, Figures 5.4, C.2 and C.3 show the results of memory consumption and execution time for all queries of the three case studies. The information displayed in each chart is the following:

- Each chart is labeled with the pattern followed by the query used for the experiment.
- The model size is displayed on the X axis using the names indicated in Table 5.1.
- The values of the *execution times* are displayed on the left-hand-side of the Y axis in milliseconds. As indicated in the charts captions, the blue solid line represents the execution time of the query over the subgraph, whereas the orange dotted line represents the execution time of the query when executed over the original graph.
- The values for *memory consumption* are displayed on the right-hand-side of the Y axis in Gigabytes. The yellow dashed line represents the memory consumption of the query over the graph, whereas the gray dashed line represents the memory consumption of the query over the subgraph.

To avoid measurement disruptions due to the warm up phase and transitory loads, all experiments were executed six times on the same machine, and the resulting values have been calculated as the average of the last three runs.

Table 5.4 summarizes in tabular format the information displayed in Figures 5.4, C.2 and C.3 with the times (in ms) of the queries when executed on the complete graph ( $T_g$ ), on the reduced subgraph as calculated by the SDR algorithm ( $T_{sg}$ ), and the corresponding speedups (S). Note that we are able to obtain results below 1 second in most cases, when the queries on the complete graph took much longer. Recall that the results shown in Figures 5.4, C.2 and C.3 and Table 5.4

Case study	Query Name	Pattern	Models					
			2M	4M	8M	15M		
Amazon	ProductPopularity	Simple	0.9912	0.9949	0.9973	0.9926		
	ProductPopularityC	Cond.	0.9912	0.9949	0.9973	0.9926		
	AlternativeCustomer	Neg.	0.4739	0.5140	0.4423	0.5206		
	PackagePopularity	Conj.	0.9861	0.9921	0.9959	0.9880		
	SimProductsPopularity	Disj.	0.9817	0.9895	0.9945	0.9859		
	PrefCustomer	Aggr.	0.9039	0.8902	0.8815	0.8757		
	PrefCustomerSimProducts	Aggr.	0.8970	0.8858	0.8790	0.8734		
			1M	4M	9M	12M	16M	
Contest	RecentPart	Simple	0.9663	0.9806	0.9898	0.9926	0.9942	
	ContestPart	Cond.	0.9226	0.9803	0.9896	0.9924	0.9941	
	UnchosenCap	Conj.	0.9086	0.9668	0.9825	0.9872	0.9901	
	FunniestCaption	Aggr.	0.8427	0.7657	0.7444	0.7429	0.7435	
	Abandon	Aggr.	0.7721	0.7564	0.7525	0.7513	0.7506	
	FunniestCaptionU	Aggr.&Conj.	0.8658	0.9548	0.9584	0.9603	0.8634	
			2M	4M	6M	8M	10M	12M
YouTube	GetAnimalVideos	Cond.	0.9951	0.9951	0.9951	0.9951	0.9951	0.9951
	NotPresent	Neg.	0.9688	0.9683	0.9683	0.9685	0.9685	0.9685
	AnimalPerson	Conj.	0.9946	0.9945	0.9945	0.9945	0.9945	0.9945
	PresentSoon	Conj.	0.9815	0.9817	0.9817	0.9817	0.9817	0.9816
	Pets	Disj.	0.9588	0.9582	0.9574	0.9573	0.9574	0.9578
	InCast	Aggr.	0.5456	0.5460	0.5464	0.5462	0.5464	0.5464

Table 5.3: Elements savings ratio when running the SDR algorithm.

are obtained with static experiments, i.e., we consider that the first execution of the SDR algorithm has already been performed. For this reason, we only compare the execution times of the query on the subgraph with the execution times of the query on the entire graph. Dynamic experiments are explained in the following section, which consider the first run of the SDR algorithm in their results.

## Experiments with streams of information

The second set of experiments is devoted to analyze our approach when dealing with dynamic graphs. For this, we need to mimic the arrival of new information. In particular, we consider the arrival of new *records*, where a record is composed of a set of elements that may be related to already existing information. In each case study, a record implies a different number of elements, approximately 2, 8 and 5 in the Amazon, Contest and YouTube applications, respectively.

To evaluate the speedup achieved by the incremental SDR algorithm, we have performed queries after a certain number of records arrive at the system. For these experiments, we have followed two approaches:

- ***CG* execution:** the query is always performed in the complete graph without running the SDR algorithm, i.e., in the graph that contains the initial information plus the new records.
- ***SubG* execution:** the SDR algorithm is run once at the beginning on the initial graph, and the query is performed on the resulting subgraph. As new records arrive to the graph, the incremental version of the SDR algorithm is run in order to keep the subgraph updated. The time taken by the initial SDR algorithm is included in the analysis, which means that the results measured for this type of execution are calculated as the sum of the execution time of the initial SDR algorithm plus the execution time of the queries in the subgraph. However, the time taken by the incremental algorithm is not included in the analysis because it is executed in parallel with the queries. Note that the incremental SDR algorithm is always listening for a change in the graph (addition, modification or deletion of elements). Whenever there is a change, it is executed.

The results of the experiments for the three case studies are shown in Table 5.5 and Tables C.9 and C.10 of Appendix C.4. Queries are executed every time  $\alpha$  new records arrive, i.e., are added to the graph. In order to limit the number of records that arrive at the system, and to evaluate our approach with the arrival of different numbers of records, executions are stopped after  $\beta$  new records have arrived. For instance, if  $\alpha = 5$  and  $\beta = 100$ , it means that the query is executed every time 5 new records arrive, and the experiment finishes after 100 new records are finally added (and the query finishes).

Each table displays the results of the experiments with a different case study. The numbers represent the ratio of execution time gain. When it is negative, it means that our approach (*SubG*) is slower than *CG*. The execution times in absolute terms for all experiments are shown on our project's website [18] and Appendix C.3. As we can see in the tables, results are organized by values of  $\alpha$  and  $\beta$ , queries, and size of models. For each value of  $\alpha$  and each model, the results are to be read vertically. Values in bold represent the first value of  $\beta$  where the execution time of *SubG* is faster than *CG*. For instance, consider the values for the query *ProductPopularity* in Table 5.5 with  $\alpha = 5$  and model *2M*. When the experiment is executed with  $\beta = 50$  (50 new records), the value is  $-0.0274$ ,

meaning *SubG* is 2.74% slower than *CG*. However, as we let more records arrive to the system, the time gained by running the query on the subgraph starts to compensate. For instance, if we consider 100 new records ( $\beta = 100$ ), *SubG* saves 11.98% of the time taken by *CG*. The formula used to represent the time gain is  $T_{gain} = 1 - T_{SubG}/T_{CG}$ , where  $T_{SubG}$  and  $T_{CG}$  represent, respectively, the times taken by the query using our subgraph and the complete graph. Recall that  $T_{SubG}$  is calculated as the sum of the time taken by the SDR algorithm to calculate the initial subgraph plus the times taken by the query in this subgraph, that is updated with the incremental algorithm.

Note that some numbers are not shown in Table C.9. This is because the queries performed on *CG* take too long and the break-even point has already been reached.

Table 5.6 summarizes the number of times the query needs to be executed for our algorithm to pay off, i.e., when our approach is worthwhile. In some cases such a number is 1, meaning that we achieve a better performance from the very first query. Of course, the larger the model the better our algorithm performs. This will be discussed in the next section.

As previously, all queries were executed six times on the same machine, and the results have been calculated as the average of the last three runs.

### 5.4.5 Functional Correctness

In order to test the proposed algorithms and to check that their behavior is correct, we have conducted extensive functional tests that try to ensure that all queries always return the same results for both the original graph and the subgraph.

Since the query is the same for the graph and the subgraph, the only way to get a wrong answer would be if the subgraph did not contain some elements or relations of the original graph that were relevant to that query. However, what our algorithm tries to obtain is *precisely* the set of elements and relationships that are relevant to the query, discarding those that are not. In this sense, our algorithm tries to generate a subgraph that is correct by construction.

Having said that, and despite the functional tests that checked that the algorithm was correct using different query suites and models, a formal proof of correctness could be interesting as part of our future work.

## 5.5 Results

This section answers the three research questions and discusses the results of the experiments described in the previous section. Threats to the validity of our study and an overview about the SDR algorithm's relationship to indexing are also discussed at the end.

### 5.5.1 RQ1: Graph size reduction

To answer the first research question, Table 5.3 displays how much the size of the subgraph obtained by the SDR algorithm is reduced compared to the size of the original graph. Each row shows the saving of elements for a specific query and for different graph sizes. The values are practically constant in each row. Also note that since the SDR algorithm obtains the subgraph according to the query structure, the saving of elements is independent of the model size.

The influence of the type of query pattern on the graph size reduction is also of interest. In this regard, queries that follow simple, conditional, conjunctive and disjunctive patterns achieve a reduction of more than 90% in all cases and nearly 100% in many of them. This suggests that, in these cases, the SDR algorithm obtains a subgraph that is close to the minimal subgraph required for matching the query. In contrast, the results are not that good for queries that follow the aggregation pattern. This is due to the fact that the algorithm does not consider the aggregation step when obtaining the subgraph (cf. Section 5.3.1). In addition, the size reduction also depends on the number of elements that pass the query filters before the aggregation operator, so the more restrictive the filters, the better.

Finally, the reduction achieved in the case of queries that follow a negative pattern directly depends on the number of elements that match the predicate of the *not* clause, because the subgraph will contain the complement of the set of such elements. This explains the different reduction results obtained for the two queries that follow a negative pattern (Amazon-AlternativeCustomer and YouTube-NotPresent).

### 5.5.2 RQ2: Performance improvement

Figures 5.4, C.2 and C.3 display the results for memory consumption and execution time for the three case studies when executing queries that follow different patterns. In all cases, both the execution times and the memory consumption are smaller because of the reduction achieved for the graph, as expected.

Charts for queries that follow the simple (Figures 5.4a and C.2a), conditional (Figures 5.4b, C.2b and C.3a), conjunctive (Figures 5.4d, C.2c, C.3c and C.3d) and disjunctive (Figures 5.4e and C.3e) patterns show that the execution time and memory consumption when executing the queries on the original graph increase as the model size grows. However, for the subgraph, these values are almost constant. This is because of the high reduction performed by the SDR algorithm on the original graph, as shown in Table 5.3, which in these cases reduces almost 99% of the elements. Note that, for these patterns, the query on the subgraph takes only a few tenths of second, which yields a speedup higher than 15 in most cases.

The performance of queries that follow aggregation patterns is highly dependent on the time and memory taken for resolving the query aggregation operators and filters (see lines 5–7 of Listing 5.5 for an example). Figures 5.4f, 5.4g, C.2e and C.3f show situations where the performance using the graph and subgraph is practically the same, because these steps are very costly (in Table 5.4, their speedups are nearly 1). In contrast, the performance of other queries, such as those that return only one element, is much better (Figures C.2d and C.2f), mainly because the aggregation filter is solved faster. In these queries, the speedup is above 40 in all cases (Table 5.4).

Figures 5.4c and C.3b show the results for queries that follow a negative pattern. Again, the more elements matching the pattern, the better the performance improvement.

### 5.5.3 RQ3: Execution time gains with data streams

Tables 5.5, C.9 and C.10 show the results of running the algorithms when dealing with data streaming applications (cf. Section 5.4.4), where queries are executed while new data is constantly arriving and being added to the model. These results are summarized in Table 5.6. Recall that  $\beta$  is the total number of records added per experiment, while  $\alpha$  represents the size of the new records batch

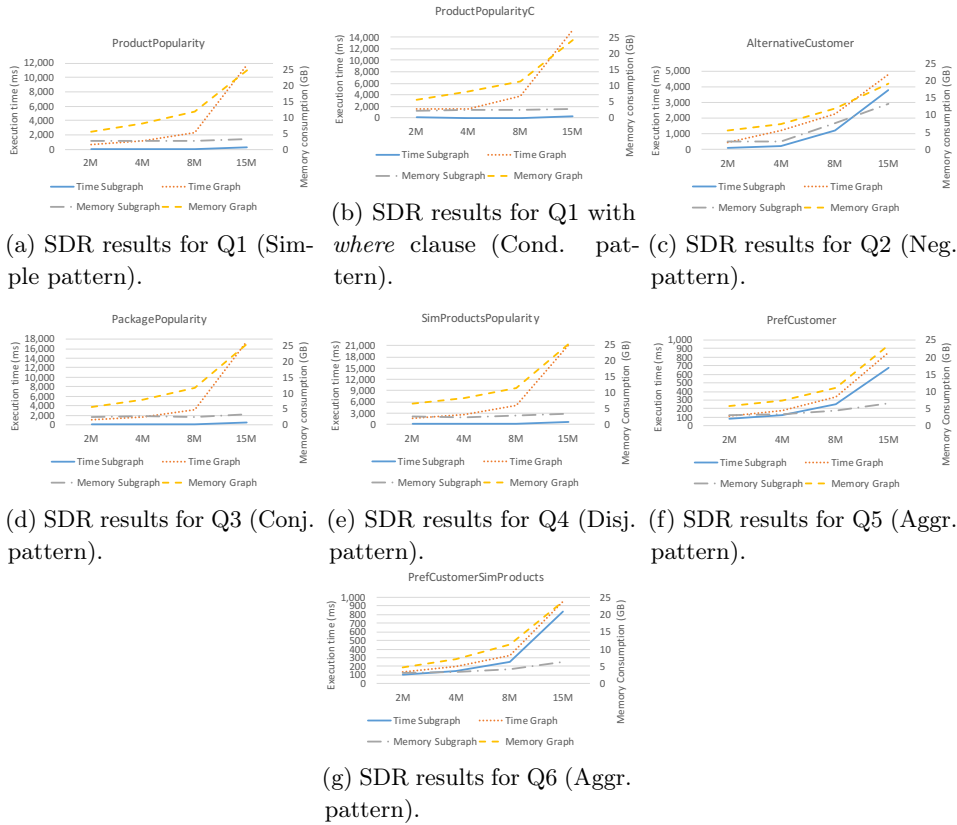


Figure 5.4: Performance results of the SDR algorithm for the Amazon queries.



Query Name	Pattern	Models											
		2M			4M			8M			15M		
Amazon Case		$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S
ProductPopularity	Simple	684	43	15.91	1,121	36	31.14	2,333	37	63.05	11,793	291	40.53
ProductPopularityC	Cond.	1,495	40	37.38	1,607	35	45.91	3,855	31	124.35	15,178	327	46.46
AlternativeCustomer	Neg.	439	82	5.35	1,182	239	4.95	2,258	1,194	1.89	4,785	3,788	1.26
PackagePopularity	Conj.	1,027	75	13.69	1,641	53	30.96	3,152	67	47.04	17,561	505	34.77
SimProductsPopularity	Disj.	1,739	105	16.56	2,556	83	30.80	4,985	104	47.93	21,283	625	34.05
PrefCustomer	Aggr.	117	81	1.44	175	120	1.46	333	252	1.32	852	674	1.26
PrefCustomerSimProducts	Aggr.	131	98	1.34	197	146	1.35	324	247	1.31	958	834	1.15
Contest Case		1M			4M			9M			12M		
		$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S
RecentPart	Simple	298	96	3.10	1,470	78	18.85	4,047	78	51.88	4,250	74	57.43
ContestPart	Cond.	610	88	6.93	2,954	81	36.47	5,642	64	88.16	6,479	86	75.34
UnchosenCap	Conj.	497	62	8.02	2,562	123	20.83	4,832	94	51.40	5,613	92	61.01
FunnistCaption	Aggr.	56,377	499	112.98	197,117	4,045	48.73	375,390	8,641	43.44	556,320	12,167	45.72
Abandon	Aggr.	245	189	1.30	1,695	893	1.90	2,844	1,939	1.47	4,365	2,808	1.55
FunnistCaptionU	Aggr. & Conj.	15,535	312	49.79	21,617	314	68.84	39,343	669	58.81	40,790	757	53.88
YouTube Case		2M			4M			6M			8M		
		$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S	$T_g$	$T_{sg}$	S
GetAnimalVideos	Cond.	1,485	15	99.00	2,851	21	135.76	3,931	22	178.68	4,283	35	122.37
NotPresent	Neg.	280	17	16.47	795	30	26.50	929	49	18.96	1,818	72	25.25
AnimalPerson	Conj.	1,400	29	48.29	3,470	39	88.97	3,432	49	70.04	4,513	54	83.57
PresentSoon	Conj.	1,200	46	26.09	2,743	87	31.53	3,122	95	32.86	3,849	109	35.31
Pets	Disj.	2,166	138	15.70	4,075	256	15.92	8,199	379	21.63	10,696	552	19.38
InCast	Aggr.	688	289	2.38	1,817	864	2.10	2,745	1,081	2.54	2,883	1,444	2.00

Table 5.4: Execution times (ms) of queries with the complete graph ( $T_g$ ), the subgraph ( $T_{sg}$ ), and the corresponding speedups (S).

that have arrived at the system each time the query is run. This means that, for a constant value of  $\alpha$ , the higher the value of  $\beta$ , the higher number of times the query of each experiment is executed.

To analyze the results for each type of query pattern, recall that, in the tables, the point at which the time gain becomes positive, which depends on the value of  $\beta$ , is highlighted in bold. This is what we call the break-even point. In Tables 5.5, C.9 and C.10, the break-even point is shown as gain ratio, while Table 5.6 displays the break-even point in number of query executions, i.e., how many executions of the query are necessary for the gain to be positive.

Our hypothesis is that time gain—in other words, how fast the break-even point is reached—is directly proportional to the value of  $\beta$  and inversely proportional to the value of  $\alpha$ . Another hypothesis is that time gain also increases with the model size. Tables 5.5, C.9 and C.10 confirm both hypothesis since, in general, time gains increase with the increase of (i) model size, (ii) data arrival and (iii) number of queries execution. This is also the tendency according to Table 5.6. We can see that for some queries and some model sizes, the break-even point is reached after only one execution of the query, which is a very good result.

Having a look at the different query patterns, we can observe that, generally,

Query Name		Models							
		$\alpha = 5$				$\alpha = 10$			
	$\beta$	2M	4M	8M	15M	2M	4M	8M	15M
ProductPopularity (Simple)	50	-0.0274	-0.0104	<b>0.0486</b>	<b>0.0155</b>	-0.2043	-0.1492	-0.0649	-0.0986
	100	<b>0.1198</b>	<b>0.1834</b>	0.2032	0.1875	-0.0176	-0.0220	<b>0.0760</b>	<b>0.0602</b>
	150	0.1874	0.2274	0.2647	0.2718	-0.0101	<b>0.0827</b>	0.1272	0.1585
	200	0.2021	0.2771	0.3018	0.3363	<b>0.0656</b>	0.1215	0.1821	0.1920
ProductPopularityC (Conditional)	250	0.2053	0.3302	0.3414	0.3772	0.0721	0.1658	0.2175	0.2469
	50	-0.0334	-0.0244	<b>0.0119</b>	<b>0.0666</b>	-0.2080	-0.1541	-0.1271	-0.0677
	100	<b>0.0704</b>	<b>0.1319</b>	0.1468	0.2048	-0.0708	-0.0163	<b>0.0456</b>	<b>0.0572</b>
	150	0.1546	0.2095	0.2110	0.2915	<b>0.0347</b>	<b>0.0522</b>	0.1095	0.1537
AlternativeCustomer (Negative)	200	0.1930	0.2479	0.2586	0.3392	0.0232	0.0828	0.1402	0.1910
	250	0.2145	0.2913	0.3001	0.3508	0.0588	0.1312	0.1728	0.2315
	50	-0.2713	-0.2519	-0.1435	-0.1760	-0.2514	-0.2908	-0.2229	-0.1628
	100	-0.0989	-0.1047	-0.0367	-0.0555	-0.1472	-0.1421	-0.1212	-0.0990
PackagePopularity (Conjunctive)	150	-0.0805	-0.0218	<b>0.0006</b>	<b>0.0034</b>	-0.0985	-0.0880	-0.0805	-0.0337
	200	-0.0461	<b>0.0186</b>	0.0283	0.0522	-0.0588	-0.0535	-0.0486	-0.0290
	250	-0.0280	0.0614	0.0728	0.1121	-0.0581	-0.0192	-0.0311	-0.0116
	50	-0.0778	-0.0850	-0.0113	-0.0107	-0.3188	-0.2222	-0.1674	-0.0938
SimProductsPopularity (Disjunctive)	100	<b>0.0704</b>	<b>0.1187</b>	<b>0.1596</b>	<b>0.1632</b>	-0.0485	-0.0714	-0.0594	<b>0.0112</b>
	150	0.1396	0.2072	0.2123	0.2764	-0.0445	<b>0.0305</b>	<b>0.0692</b>	0.0792
	200	0.1730	0.2486	0.2499	0.3607	<b>0.0123</b>	0.0585	0.0910	0.2189
	250	0.1849	0.2927	0.2936	0.3875	0.0455	0.1282	0.1289	0.2496
PrefCustomer (Aggregation)	50	-0.0207	-0.0092	<b>0.1372</b>	<b>0.1514</b>	-0.2326	-0.1121	-0.1092	-0.0574
	100	<b>0.1525</b>	<b>0.1721</b>	0.2617	0.2718	-0.0111	<b>0.0096</b>	<b>0.0463</b>	<b>0.1414</b>
	150	0.2337	0.3121	0.3335	0.3965	<b>0.0477</b>	0.1059	0.1632	0.2507
	200	0.2722	0.3508	0.3838	0.4497	0.0969	0.1861	0.1834	0.2659
PrefCustomerSimProducts (Aggregation)	250	0.3029	0.3918	0.4038	0.4753	0.0828	0.2052	0.2215	0.3278
	50	-0.3316	-0.3102	-0.3002	-0.3041	-0.3724	-0.3595	-0.3509	-0.3238
	100	-0.2860	-0.2479	-0.2088	-0.1554	-0.2951	-0.2926	-0.2500	-0.2115
	150	-0.2145	-0.1989	-0.1652	-0.0751	-0.2395	-0.2121	-0.1850	-0.1140
PrefCustomerSimProducts (Aggregation)	200	-0.2006	-0.1295	-0.1283	-0.0547	-0.2191	-0.1526	-0.1230	-0.0772
	250	-0.1826	-0.0999	-0.0932	-0.0185	-0.2061	-0.1125	-0.0984	-0.0440
	50	-0.2663	-0.2892	-0.2294	-0.3203	-0.3024	-0.3723	-0.2916	-0.3652
	100	-0.2282	-0.2215	-0.1509	-0.1806	-0.2734	-0.2464	-0.2089	-0.2137
PrefCustomerSimProducts (Aggregation)	150	-0.1746	-0.1753	-0.1190	-0.1055	-0.2083	-0.1865	-0.1715	-0.1146
	200	-0.1550	-0.0871	-0.1061	-0.0601	-0.1808	-0.1361	-0.1128	-0.0745
	250	-0.1377	-0.0807	-0.0696	-0.0194	-0.1775	-0.1097	-0.0987	-0.0379

Table 5.5: Gain ratio when using the incremental algorithm in the Amazon case study.

disjunctive queries achieve the highest gain (see *SimProductsPopularity* in Table 5.5 and *Pets* in Table C.10), followed by simple and conditional queries, which have a similar gain (see *ProductPopularity* and *ProductPopularityC* in Table 5.5, *RecentPart* and *ContestPart* in Table C.9 and *GetAnimalVideos* in Table C.10), where conditional queries have a slightly higher gain than simple queries. Then, conjunctive queries (see *PackagePopularity* in Table 5.5, *UnchosenCap* in Table C.9, and *AnimalPerson* and *PresentSoon* in Table C.10) have a higher gain than negative queries (*AlternativeCustomer* in Table 5.5 and *NotPresent* in Table C.10). Regarding aggregation queries, they present very different gain values in the three case studies. For example, observe how the *FunniestCaption* query in Table C.9 has a gain higher than 70% for all  $\alpha$  and  $\beta$  values, whereas *PrefCustomer* in Table 5.5 does not present any positive gain for any  $\alpha$  and  $\beta$  values.

In summary, we conclude that the query patterns in which the break-even points are reached faster are, in this order, disjunctive, conditional, simple, conjunctive and negative. Regarding results for aggregation patterns, they are quite different from each other. Typically, the break-even point of queries following this pattern depends on the overload imposed by the aggregation operators and their corresponding filters: the lighter they are, the sooner the break-even point is reached, and vice-versa.

#### 5.5.4 SDR algorithm and Indexing techniques

Indices are a very popular and efficient technique to improve query performance. In fact, some of the technologies that we studied to develop our proposal (cf. Chapter 3) have some support to implement them. Some examples are the indexing of objects and relationships from TinkerGraph [119], the indexing of labels and properties from Neo4j [90], or Memgraph label and label-property indices [84].

According to the classification presented in Section 5.2, a valid indexing schema for our queries needs to provide two fundamental features: (i) efficient lookups to identify the initial objects of the query, i.e. the objects that match with the last step of the query, and (ii) it may guide the traversals during query evaluation. However, although some works provide mechanisms to create graph indexing techniques [85], this is still an open issue to be addressed [125]. For this reason, in the present chapter, we have addressed the improvement of query performance on graphs from a different perspective that does not use indices. Nevertheless, our work does

Case study	Query Name	Pattern	Models			
			2M	4M	8M	15M
Amazon	ProductPopularity	Simple	15	11	6	8
	ProductPopularityC	Conditional	16	13	9	6
	AlternativeCustomer	Negative	37	32	30	29
	PackagePopularity	Conjunctive	19	17	12	10
	SimProductsPopularity	Disjunctive	13	11	7	5
	PrefCustomer	Aggregation	68	51	49	37
	PrefCustomerSimProducts	Aggregation	67	46	50	37
			1M	4M	9M	12M
Contest	RecentPart	Simple	41	12	1	1
	ContestPart	Conditional	31	6	2	1
	UnchosenCap	Conjunctive	38	10	4	1
	FunniestCaption	Aggregation	1	1	1	1
	Abandon	Aggregation	38	15	5	3
	FunniestCaptionU	Aggregation & Conjunctive	1	2	2	2
			2M	4M	6M	8M
YouTube	GetAnimalVideos	Conditional	49	3	1	1
	NotPresent	Negative	67	24	4	1
	AnimalPerson	Conjunctive	47	11	6	1
	PresentSoon	Conjunctive	43	11	5	3
	Pets	Disjunctive	5	4	1	1
	InCast	Aggregation	45	22	15	1

Table 5.6: Number of query executions needed to obtain a positive gain for each query.

not pretend to replace indexing techniques, but to complement them in order to achieve further improvements. In this way, a possible approach may use the efficient indexing searches in order to identify the parameterized objects of our queries (e.g those that refer to the most specific step of the query, which is the last step in our approach), together with the dataset reduction obtained from the SDR algorithm. In addition, since the objects weight calculated with the SDR algorithm is a numerical value, our approach is designed to be applied in the context of approximate queries. This application is not contemplated by indexing techniques, so it may complement them too in order to speed up the queries. However, all these applications are out of the scope of this thesis, so we consider them as future work.

### 5.5.5 Threats to Validity

In this section we discuss the threats that can affect the validity of our proposal and results. We describe four types of threats according to Wohlin et al. [129].

#### Construct validity threats

These threats are concerned with the relationship between theory and what is observed. A common construct validity threat, known as the mono-method bias, is related to the use of one single metric in the evaluation. In our experiments, we have considered different metrics, namely execution time, memory consumption and source data set reduction. Given that results obtained by the different metrics are consistent when drawing the conclusions, we consider the mono-method bias threat neutralized.

#### Conclusion validity threats

The main issue that can affect the validity of our conclusions is the transient effects of noise by other components of the system under study. To mitigate this, we ran the experiment 6 times and took the average of the last 3 runs. Furthermore, the raw data and scripts for replicating our experiments are available on our project's website [17, 18].

#### Internal validity threats

These threats are related to those factors that might affect the results of our evaluation. To mitigate them, we have used models of different size. Since our approach is targeted at optimizing queries when the volume of information is high, all models were large (with between 1.5 to 16 million objects and relationships). Besides, we analyze the behavior of our approach with data of different nature, since they belong to three case studies whose graphs have different topology.

The way we have tried to mimic the arrival of new information to the initial data set might have also affected the validity of our results. In order to mitigate this threat, we have analyzed how our approach behaves in different dynamic scenarios, and combined (i) the amount of information that arrives at every time step, (ii)

how often such information arrives and (iii) the use of models of different sizes (cf. Tables 5.5, C.9 and C.10).

### External validity threats

External validity threats have to do with the extent to which it is possible to generalize the findings of the experiments. The first threat is that the results of our experiments have been obtained with three case studies, which externally threatens the generalizability of our results. To mitigate this, we have tried to select case studies from different and real contexts, where only one has been created by us. In that case study, we tried to reflect the main parts of the Amazon ordering service, and created models of different sizes in which connections among objects are similar to the ones we could have in models containing real data. The other two case studies have been taken from real data sets, so that this threat is minimized.

Although we checked that all queries returned in all cases the same results for both the graph and the subgraph, and conducted exhaustive functional tests on the algorithm, formally proving the correctness of the algorithm could be of interest, too.

A third threat to the external validity of our solution is related to the language and technologies used to implement our approach. As described in Chapter 3, we studied different technologies and selected the ones that we considered most appropriate, namely TinkerGraph and Gremlin. While we believe our approach can be implemented with other technologies, doing so might lead to slightly different performance results.

The final threat to external validity identified is related to the classification of queries provided. In fact, our SDR algorithm works depending on the type of query, which in turn depends on the constructs offered by the query language. Should we have provided a different classification for the queries, the implementation of our algorithm would have been different and the results might have varied.

## 5.6 Related Work

This work was published in [15]. It mainly derives from three previous studies. First, we introduced the concept of *Approximate Model Transformations* and the

statistical error calculation for streams of data [121]. However, this work does not consider graph-structured data but simple events that are not related to each other. The second study is a previous work [13], in which we propose a prototypical solution based on CEP and adapted to graph-structured information. It uses the *Graphx* tool of Spark for graph-parallel computation. In that work, the queries are developed using Scala language, what increases the complexity of the query implementation. Furthermore, the work does not deal with approximations. The last study, also developed by us [14], exposes the online method described in Chapter 4. However, as opposed to the approach presented in this chapter, the online method does not consider the content of the queries in the approximation. The present work extends all these three works by proposing a solution that applies a precomputation (i.e. offline AQP technique) that takes into account the syntax of the query. Result of this precomputation is a subset of the incoming graph-structured information that is relevant for the query. By executing the query on this subset, execution time and memory consumption are decreased. Besides, the approach proposed in Chapter 4 is complementary to this work, since we can seek greater performance improvement by applying approximations to the subset obtained.

Two surveys about approximate query processing mention a precomputation step in order to select important information for the query before it is executed [32, 75]. This information is stored as a summary of the source data and it is used to perform the query faster. Other works propose precomputation with sampling techniques in order to select only part of the information with the aim of speeding up queries [2, 3, 10, 30, 31]. However, these precomputation proposals typically differ from ours in three aspects, namely (i) they only consider queries that return an aggregated result, (ii) they are not applied to graph-structured information, and (iii) the accuracy of the aggregated answer is not optimal, since this aggregation does not contain all relevant information to the query. Up to our knowledge, the closest work to ours regarding precomputation has been proposed by Fan et al. [52], who study how to query a graph with bounded resources. They propose an algorithm to calculate an approximation that depends on the query and on a parameter that indicates the limit of resources. The algorithm assigns a weight to each object according to their importance for the query. The approximation contains as many objects as the parameter of the limit indicates, taking the most

relevant ones and discarding the rest. In this way, they get the minimum possible error considering the bounded resources. However, differently from the approach presented in this chapter, they only consider static graphs and not data streaming applications with information continuously coming in.

Other related proposals do consider the arrival of new information. In a previous work by Fan et al. [51], they define algorithms for incremental graph pattern matching when the graph is updated. However, their evaluation considers graph sizes of 65,000 elements at most (counting objects and relationships). Moreover, our approach uses the type of pattern followed by the query to improve graph reduction and query performance, whereas they do not make use of this information.

There are other works that deal with incremental queries using crowdsourcing techniques [122, 123]. However, these works have a different perspective. Crowdsourcing techniques construct the results incrementally starting from an initial small dataset. At this point, since the source information does not contain enough relevant data, the results have a low accuracy. As new information arrives to the system, the results are refined. Our approach, instead, considers all relevant information to the query from the beginning (typically a large dataset) so that an accurate result can be obtained at first. We use the incremental SDR algorithm to maintain the subgraph updated.

Some other works that deal with the arrival of new information propose incremental transformations, where the input model changes over time [20, 65, 66, 99, 110, 124]. They present partial and incremental model transformations using EMF-IncQuery and EMF-IncQuery-D frameworks [20, 110, 124], an incremental algorithm for ATL [66], a framework for the instant and incremental transformation of changes among models [65], and a partial evaluator prototype called QvtMix [99]. Therefore, these papers are not focused on graph databases. In addition, they only consider two types of queries: simple (with two elements at most) and complex (with more than two elements that are linked through one or more joins). In this way, our proposal uses a more exhaustive query classification schema with six different types of query patterns.

Bergmann et al. [21] present a solution that supports incremental queries over models in the VIATRA2 framework. The implementation is based on the RETE algorithm, which improves speed at the expense of consuming more memory. Their solution stores the pattern matches, and updates them as new changes occur in



the model. Evaluation results report an average scale of up to 9% with respect to normal executions, which implies a speedup of about 11. This approach works differently from ours because it propagates the changes of the model to the resultset (so a resultset must always be available), while our approach propagates them to the dataset that will be queried, which corresponds to the subgraph. Besides, the SDR algorithm works with graph databases while VIATRA2 works with models. For these reasons, and since both approaches pursue a similar goal, we believe they are complementary.

Other projects propose incremental queries with graph databases, such as the *ingraph* query engine [112] and OrientDB's *LiveQuery* [95]. The first difference is that the SDR algorithm is implemented for the Gremlin language, whereas *ingraph* works with Cypher and *LiveQuery* uses a SQL dialect. Also, *ingraph* propagates the changes of the graph to the resultset, while *LiveGraph* returns the latest changes (but not the complete resultset). This is similar to the VIATRA2 approach, as mentioned earlier.

Two other works classify graph queries according to their structure and calculate their complexity. First, Barceló et al. [11] propose a classification of queries according to the paths they contain. However, they do not consider the property filters or the operator types. Angles et al. [4] propose a more complete classification that considers the operations that can be found in a query. The authors mainly distinguish between basic and complex graph patterns. The former ones cover the property filters that can be queried with variables or constants; the latter ones extend the basic graph patterns with different operations like union, projection or difference. They describe each type of pattern and illustrate them using three of the most popular graph query languages, namely Gremlin, SPARQL and Cypher. This approach is very similar to ours, since it also considers filters. However, our classification further divides complex graph patterns into six individual subcategories, namely condition, negation, conjunctive, disjunctive, and aggregation. This refinement is relevant for analysing the behavior and performance of the proposed algorithm.

Finally, our algorithm was developed considering the rationale behind Google's PageRank algorithm [96]. This algorithm calculates a probability for each web page according to its importance but without considering the context of any search. In our approach, instead, the relevance of graph objects is influenced by the query

contents. In a similar way, Richardson and Domingos propose a probabilistic model for a more intelligent PageRank algorithm [101] that calculates the probability that a web page contains the terms of a specific search query. However, they do not consider the structure and operators of the query itself, which we have seen have a significant impact on the results.

## 5.7 Summary

In this chapter, we have designed and developed an algorithm that implements an offline technique to optimize the performance of queries on graph-structured data streams. This algorithm selects a subgraph of the original model that contains the relevant data for the query, and on which the query can be more efficiently executed. Furthermore, as new information arrives and is added to the system, the subgraph is updated using another algorithm, that we have called Incremental SDR. In order to study the performance depending on the content of the query, we have identified and classified six different patterns that can be found in queries over graph-structured data according to their structure.

Our experiments show that querying the subgraph obtained with the SDR algorithm instead of the entire source graph achieves a performance improvement for all query patterns. We have also demonstrated that these improvements increase with the original graph size, as well as with the number of times the query is run. However, queries that follow aggregation patterns behave slightly different than the rest, since they depend on the aggregation filters and operators that they contain. Therefore, aggregation patterns need to be studied in detail as future work.



# Chapter 6

## Conclusions and Future Work

---

This chapter summarizes the proposal that has been explained throughout this dissertation. First, we highlight the main conclusions of our work in Section 6.1. Second, we expose the publications derived from our contributions in Section 6.2. Finally, we describe our future work in Section 6.3.

### 6.1 Summary and Conclusions

As mentioned throughout the lines of this dissertation, one of the main challenges when working with Data Streaming Applications is to get a low-latency processing, which implies fast responses. This feature motivated the first contribution of this thesis, presented in Chapter 3. We presented a comparative study of 7 processing platforms that are commonly used to work with large volumes of data, namely TinkerGraph, Neo4j, CrateDB, Memgraph, JanusGraph, OrientDB and GraphFrames. In addition, 4 DSLs to write the queries that process the information were also compared, namely Gremlin, Cypher, SQL and the GraphFrames DSL. We

aimed to get the best combination of DSL and processing platform that suits the following requirements: (i) they allow to perform queries and update the information as quickly as possible in order to provide real-time responses, (ii) they cope with graph-structured information, and (iii) the DSL provides a clear syntax in order to be able to study the type of query to be run over the data. All technologies were evaluated using two case studies with graph-structured information. Performance, in terms of execution time, and complexity of the language, in terms of number of characters, operators and internal variables, were compared in the experiments. Results showed that graph databases are the most efficient technologies to work with graphs. Besides, the DSLs used with this kind of databases present the simplest syntax. We concluded the most suitable combination for our requirements was TinkerGraph and Gremlin.

The second main contribution was presented in Chapter 4. This contribution addresses online AQP when working with graph-structured information flows. Three techniques were proposed for improving the performance when querying large models, called temporal, spatial and random approximations. Temporal and spatial approximations select a subset of the source information by means of reducing the temporal and spatial ranges, respectively, whereas random approximations add a probability to each element of the graph to be included in the subset. In order to find the right balance between accuracy loss derived from these approximations and performance gain, we proposed a method for measuring the accuracy. This method is based on the terms of accuracy, recall and precision.

Two different data distributions were used in the experiments of Chapter 4, in order to analyze how this feature affects the approximations. Results concluded that performance can be improved with approximations and an optimal accuracy value can be acquired when considering only part of the source model. Nevertheless, temporal approximations are the most convenient option when the data present a temporal focus. In addition, random approximations showed a similar performance regardless data distribution and their experiments showed that they are the best option when a query does not contain temporal or spatial filtering. Finally, spatial approximation results showed that they are very expensive in terms of runtime and they only pay off when there is not other option.

The last contribution was presented in Chapter 5. It addresses offline AQP in order to improve the performance of Data Streaming Applications. To achieve

this, we designed and developed the SDR algorithm. This algorithm optimizes the performance when querying graph-structured data streams by selecting a subgraph of the source model. This subgraph contains the data that is relevant for the query and, therefore, the query can be more efficiently executed. Since the elements contained in the subgraph depend on the query structure, we identified six patterns that can be found in graph queries, namely simple, conditional, conjunction, disjunction, negative and aggregation patterns. Since our approach is designed to work with information flows, we developed an incremental version of the algorithm, called Incremental SDR. This algorithm updates the subgraph as new information arrives to the system.

Three case studies were used to validate our proposal. Results showed that querying the subgraph obtained with the SDR algorithm instead of the entire source graph achieves a performance improvement for all query patterns. In fact, some query patterns only need to query a subgraph that contains only 1% of the elements of the original graph. Specifically, the query patterns in which the time gain is higher are, in this order, disjunctive, conditional, simple, conjunctive and negative. However, queries that follow aggregation patterns behave slightly different than the rest, since they depend on the aggregation filters and operators that they contain. Furthermore, we also showed that the performance improvements increase with the original graph size, as well as with the number of times the query is run.

Therefore, our approach has demonstrated that it is possible to obtain the best trade-off between the correctness of the results and the performance of the processing. For this reason, we consider that this thesis provides an answer to the research question proposed in Section 1.1 (*Can we obtain a good (or the optimal) trade-off between performance and accuracy loss when processing very-large amounts of information?*) and it also achieves the expected goals.

## 6.2 Publications

In this section we expose the publications that support this thesis as well as other research contributions where the author of this dissertation has participated over the course of her Ph.D.

### 6.2.1 Publications Supporting this Dissertation

This section shows the publications that are closely related with the approaches presented in this thesis.

#### International Conferences

- Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo, “Extending complex event processing to graph-structured information” in Proc. of Model Driven Engineering Languages and Systems - 21th International Conference. ACM, pp. 166–175, 2018 [13]. A preliminary version of a system that manages streams of graph-structured data is published in this paper. This system is based on CEP and it involves the basis of the approach presented in Chapters 4 and 5.

#### International Journals

- Gala Barquero, Javier Troya, and Antonio Vallecillo, “Trading accuracy for performance in data processing applications”, Journal of Object Technology, vol. 18, no. 2, pp. 9:1–24, 2019 [14]. This paper supports the contents presented in Chapter 4. This publication obtained the **Best Paper Award** of the European Conference on Modelling Foundations and Applications (ECMFA2019).
- Gala Barquero, Javier Troya and Antonio Vallecillo, “Improving Query Performance on Dynamic Graphs”, Software and System Modeling pp. 1619–1374, 2020 [15]. The algorithm and the approach presented in Chapter 5 is published in this paper. In addition, this paper comprises a summary of the research presented in Chapter 3.

### 6.2.2 Further Publications

This section shows two further contributions that were published in international conferences, although they are not directly related to the lines of research presented in this dissertation.

- Manuel F. Bertoa, Nathalie Moreno, Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo, “Expressing measurement uncertainty in OCL/UML datatypes”, in Proc. of Modelling Foundations and Applications - 14th European Conference, Held as Part of STAF 2018. Springer, pp. 46–62, 2018 [23].
- Nathalie Moreno, Manuel F. Bertoa, Gala Barquero, Loli Burgueño, Javier Troya, Adrián García-López, and Antonio Vallecillo, “Managing uncertain complex events in web ofthings applications”, in Proc. of Web Engineering - 18th International Conference. Springer, pp. 349–357, 2018 [86].

## 6.3 Future Work

We consider that the work presented in this thesis is mature enough to give an answer to a specific problem. However, the approaches presented in this dissertation may be extended in several directions in order to improve the research.

### 6.3.1 Online techniques

According to the results presented in Chapter 4, performance can indeed be improved with the proposed online AQP techniques (temporal, spatial and random approximations). In fact, an optimal accuracy value can be acquired when considering only part of the source model. Nevertheless, we plan to conduct further experiments. To begin with, we are interested in investigating how the presence of more than one data focus in different time intervals can affect the approximations. Similarly, we want to investigate spatial data focuses located in different points of the model and their effects in the approximations. We pretend to bear out the hypothesis that temporal and spatial focuses do not affect random approximations but spatial and temporal approximations.

Regarding the algorithm used to traverse the graph for spatial approximations, Gremlin applies a depth-first approach by default. It is interesting to study how applying other algorithms affects accuracy of the approximations as well as execution time, so it is something we plan to explore in future work. In this line, we also plan to consider what benefits model indexing technologies can bring to processing performance.



Depending on the query of the running example exposed in Chapter 4, temporal, spatial and random approximations imply FPs or FNs. However, other examples may imply FPs and FNs in the same query. We plan to study the evolution of performance in approximations with this kind of examples, and how to obtain the amount of data for the Optimal Models. In this case, both precision and recall will be evaluated for the query and the Optimal Model would be chosen regarding the stabilization of both parameters. In this way, although our current results are enough to draw conclusions about the effect of using approximations in data flows, more case studies may be tested to better assess our proposal.

We also plan to address a new problem: how to automatically determine the Optimal Models. We envision the application of search-based algorithms for determining the Pareto front of Optimal Models, according to a set of optimization criteria, such as decreasing execution time and memory consumption, and improving precision and recall. Although execution time is the most restrictive aspect when working with Data Streaming Applications (cf. Section 2.2), we plan to do a more exhaustive evaluation of memory consumption with many more experiments and study how the different approximation types can affect this feature.

### 6.3.2 Offline techniques

On the other hand, experiments presented in Chapter 5 show that performance can be also improved with the SDR algorithm. Recall that although this algorithm is an offline AQP technique, we achieved a performance improvement without compromising the accuracy of our results. Nevertheless, queries that follow aggregation patterns behave slightly different than the rest, since they depend on the aggregation filters and operators that they contain. For this reason, we plan to study these kinds of queries more deeply. In this way, we pretend to obtain a subgraph tighter to the query and, therefore, to get a more noticeable performance improvement for aggregation patterns. We also plan to study how to improve the performance of negative patterns since improvement for this query pattern is a bit lower compared to the rest of patterns. For example, we plan to implement a second scan of the query that would remove unnecessary elements in the subgraph using data cleansing techniques. In addition, in this approach we have considered queries that follow mostly one pattern, in order to characterize their behavior. The

analysis of more complex queries with more patterns could also be of interest.

### 6.3.3 Mixed techniques

Taking into account results of the experiments with online and offline AQP techniques, other interesting line of future work is to include approximation techniques of Chapter 4 in the proposal of Chapter 5. In this way, spatial and temporal windows as well as random techniques would be applied to the subgraph obtained with the SDR algorithm in order to get a higher performance in detriment of the accuracy of the results. Since the results would be approximate, we would need to measure the trade-off between performance gain and accuracy loss, and study it depending on the query pattern.

Finally, since temporal, spatial and random approximations and SDR algorithm are technology-independent, we plan to implement them with the technologies presented in Chapter 3 and compare the results.



# Bibliography

---

- [1] Aburawi, N., Lisitsa, A., Coenen, F.: Querying encrypted graph databases. In: Proc. of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018 January 22-24, pp. 447–451 (2018). DOI 10.5220/0006660004470451
- [2] Acharya, S., Gibbons, P.B., Poosala, V.: Congressional samples for approximate answering of group-by queries. In: Proc. of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, pp. 487–498 (2000). DOI 10.1145/342009.335450
- [3] Agarwal, S., Panda, A., Mozafari, B., Iyer, A.P., Madden, S., Stoica, I.: Blink and it’s done: Interactive queries on very large data. PVLDB **5**(12), 1902–1905 (2012). DOI 10.14778/2367502.2367533
- [4] Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1–68:40 (2017). DOI 10.1145/3104031
- [5] Apache: Apache TinkerPop (accessed March 2019). <http://tinkerpop.apache.org/>
- [6] Apache TinkerPop: The Gremlin Graph Traversal Machine and Language (accessed November 2019). <https://tinkerpop.apache.org/gremlin.html>
- [7] Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. of Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010, October 3-8, Part I, *LNCS*, vol. 6394, pp. 121–135. Springer (2010). DOI 10.1007/978-3-642-16145-2\\_9

- [8] Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: «UML» 2001 - Proc. of The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, October 1-5, pp. 19–33 (2001). DOI 10.1007/3-540-45441-1\\_3
- [9] Avila-García, O., Cabot, J., Muñoz, J., Romero, J.R., Vallecillo, A.: Desarrollo de Software Dirigido por Modelos (DSDM, 7ª ed.). RA-MA Editorial (2010)
- [10] Babcock, B., Chaudhuri, S., Das, G.: Dynamic sample selection for approximate query processing. In: Proc. of the 2003 ACM SIGMOD International Conference on Management of Data, June 9-12, pp. 539–550 (2003). DOI 10.1145/872757.872822
- [11] Barceló, P.: Querying graph databases. In: Proc. of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, June 22 - 27, pp. 175–188 (2013). DOI 10.1145/2463664.2465216
- [12] Barceló, P., Libkin, L., Romero, M.: Efficient approximations of conjunctive queries. SIAM J. Comput. **43**(3), 1085–1130 (2014). DOI 10.1137/130911731
- [13] Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Extending complex event processing to graph-structured information. In: Proc. of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, October 14-19, pp. 166–175. ACM (2018). DOI 10.1145/3239372.3239402
- [14] Barquero, G., Troya, J., Vallecillo, A.: Trading accuracy for performance in data processing applications. Journal of Object Technology **18**(2), 9:1–24 (2019). DOI 10.5381/jot.2019.18.2.a9
- [15] Barquero, G., Troya, J., Vallecillo, A.: Improving query performance on dynamic graphs. Software and System Modeling pp. 1619–1374 (2020). DOI 10.1007/s10270-020-00832-3
- [16] Barquero, G., Troya, J., Vallecillo, A.: Approximate Transformation git repository (accessed March 2019). <https://github.com/atenearesearchgroup/approximateTransformation.git>.

- [17] Barquero, G., Troya, J., Vallecillo, A.: SDR algorithm git repository (accessed November 2019). <https://github.com/atenearesearchgroup/SDRalgorithm>.
- [18] Barquero, G., Troya, J., Vallecillo, A.: SDR algorithm website (accessed November 2019). <http://atenea.lcc.uma.es/projects/SDRAlg.html>.
- [19] BBVA: The impact of the Mobile World Congress in a dynamic visualization by BBVA and CartoDB (2013). <https://www.bbva.com/en/impact-mobile-world-congress-dynamic-visualization-bbva-cartodb/>. (accessed March 2019)
- [20] Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Proc. of Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, October 3-8, Part I, pp. 76–90 (2010). DOI 10.1007/978-3-642-16145-2\\_6
- [21] Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA model transformation system. In: Proc. of the Third International Workshop on Graph and Model Transformations, GRAMOT'08, pp. 25–32. ACM (2008). DOI 10.1145/1402947.1402953
- [22] Bernard, J., Héam, P., Kouchnarenko, O.: An approximation-based approach for the random exploration of large models. In: Proc. of Tests and Proofs - 12th International Conference, TAP@STAF 2018, June 27-29, pp. 27–43 (2018). DOI 10.1007/978-3-319-92994-1\\_2
- [23] Bertoa, M.F., Moreno, N., Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Expressing measurement uncertainty in OCL/UML datatypes. In: Proc. of Modelling Foundations and Applications - 14th European Conference, ECMFA@STAF 2018, June 26-28, pp. 46–62 (2018). DOI 10.1007/978-3-319-92997-2\\_4
- [24] Besta, M., Fischer, M., Kalavri, V., Kapralov, M., Hoefer, T.: Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. CoRR **abs/1912.12740** (2019)

- [25] Besta, M., Peter, E., Gerstenberger, R., Fischer, M., Podstawski, M., Barthels, C., Alonso, G., Hoefer, T.: Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. CoRR **abs/1910.09017** (2019)
- [26] Bran Selic: Abstraction Patterns in Model-Based Engineering (accessed February 2020). <https://openmodelica.org/images/docs/modprod2011-talks-day2/modprod2011-day2-talk1-keynote-Bran-Selic-Abstraction.pdf>
- [27] Braverman, V., Ostrovsky, R.: Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams. In: Proc. of Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, August 21-23, pp. 58–70 (2013). DOI 10.1007/978-3-642-40328-6\\_5
- [28] Callidus Software Inc.: OrientDB Documentation. Concurrency. (accessed July 2020). <https://orientdb.org/docs/3.0.x/general/Concurrency.html>
- [29] Callidus Software Inc.: OrientDB. The database designed for the modern world. (accessed June 2020). <https://orientdb.com/>
- [30] Chaudhuri, S., Das, G., Datar, M., Motwani, R., Narasayya, V.R.: Overcoming limitations of sampling for aggregation queries. In: Proc. of the 17th International Conference on Data Engineering, April 2-6, pp. 534–542 (2001). DOI 10.1109/ICDE.2001.914867
- [31] Chaudhuri, S., Das, G., Narasayya, V.R.: A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries. In: Proc. of the 2001 ACM SIGMOD international conference on Management of data, May 21-24, pp. 295–306 (2001). DOI 10.1145/375663.375694
- [32] Chaudhuri, S., Ding, B., Kandula, S.: Approximate query processing: No silver bullet. In: Proc. of the 2017 ACM International Conference on Man-

- agement of Data, SIGMOD Conference 2017, May 14-19, pp. 511–519 (2017). DOI 10.1145/3035918.3056097
- [33] Cimprich, P., Becker, O., Nentwich, C., Jiroušek, H., Batsis, M., Brown, P., Kay, M.: Streaming Transformations for XML (STX). Working Draft (accessed May 2019). <http://stx.sourceforge.net/documents/>
- [34] Combemale, B., Thirioux, X., Baudry, B.: Formally Defining and Iterating Infinite Models. In: Proc. of Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, September 30–October 5, *LNCS*, vol. 7590, pp. 119–133. Springer (2012). DOI 10.1007/978-3-642-33666-9\\_9
- [35] CrateDB: CrateDB: The distributed SQL database for machine data (accessed February 2020). <https://crate.io/>
- [36] CrateDB: What is CrateDB? (accessed March 2020). <https://crate.io/products/cratedb/>
- [37] Cuadrado, J.S., de Lara, J.: Streaming Model Transformations: Scenarios, Challenges and Initial Solutions. In: Proc. of Theory and Practice of Model Transformations - 6th International Conference, ICMT@STAF 2013, June 18-19, *LNCS*, vol. 7909, pp. 1–16. Springer (2013). DOI 10.1007/978-3-642-38883-5\\_1
- [38] Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* **44**(3), 15:1–15:62 (2012). DOI 10.1145/2187671.2187677
- [39] Cugola, G., Margara, A., Pezzè, M., Pradella, M.: Efficient analysis of event processing applications. In: Proc. of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, June 29 - July 3, pp. 10–21. ACM (2015). DOI 10.1145/2675743.2771834
- [40] Daniel, G., Jouault, F., Sunyé, G., Cabot, J.: Gremlin-ATL: a scalable model transformation framework. In: Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, October 30 -



- November 03, pp. 462–472. IEEE Computer Society (2017). DOI 10.1109/ASE.2017.8115658
- [41] Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., Cabot, J.: Neoemf: A multi-database model persistence framework for very large models. *Sci. Comput. Program.* **149**, 9–14 (2017). DOI 10.1016/j.scico.2017.08.002
- [42] Daniel, G., Sunyé, G., Cabot, J.: Mogwai: A framework to handle complex queries on large models. In: Tenth IEEE International Conference on Research Challenges in Information Science, RCIS 2016, June 1-3, pp. 1–12 (2016). DOI 10.1109/RCIS.2016.7549343
- [43] Daniel, G., Sunyé, G., Cabot, J.: Scalable Queries and Model Transformations with the Mogwai Tool. In: Proc. of Theory and Practice of Model Transformation - 11th International Conference, ICMT@STAF 2018, June 25-26, pp. 175–183 (2018). DOI 10.1007/978-3-319-93317-7\\_9
- [44] Dávid, I., Ráth, I., Varró, D.: Streaming Model Transformations By Complex Event Processing. In: Proc. of Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, September 28 - October 3, *LNCS*, vol. 8767, pp. 68–83. Springer (2014). DOI 10.1007/978-3-319-11653-2\\_5
- [45] Dean, J., Henzinger, M.R.: Finding related pages in the world wide web. *Comput. Netw.* **31**(11-16), 1467–1479 (1999). DOI 10.1016/S1389-1286(99)00022-5
- [46] van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices* **35**(6), 26–36 (2000). DOI 10.1145/352029.352035
- [47] DGraph: DGraph (accessed January 2020). <https://dgraph.io/>
- [48] Esper: Esper Tech (accessed March 2019). <http://www.espertech.com/esper/>

- [49] Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Company (2010)
- [50] Event Processing Technical Society: Event Processing Glossary, Version 2.0 (2011). [http://www.complexevents.com/wp-content/uploads/2011/08/EPTS\\_Event\\_Processing\\_Glossary\\_v2.pdf](http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf)
- [51] Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., Wu, Y.: Graph pattern matching: From intractable to polynomial time. *PVLDB* **3**(1), 264–275 (2010). DOI 10.14778/1920841.1920878
- [52] Fan, W., Wang, X., Wu, Y.: Querying big graphs within bounded resources. In: International Conference on Management of Data, SIGMOD 2014, June 22–27, pp. 301–312 (2014). DOI 10.1145/2588555.2610513
- [53] Garofalakis, M.N., Gibbons, P.B.: Approximate query processing: Taming the terabytes. In: Proc. of 27th International Conference on Very Large Data Bases, VLDB 2001, September 11–14 (2001)
- [54] Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. *STTT* **14**(1), 15–40 (2012). DOI 10.1007/s10009-011-0186-x
- [55] Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In: 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, October 8–10, pp. 17–30 (2012)
- [56] Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, October 6–8, pp. 599–613 (2014)
- [57] Greenfield, J., Short, K.: Software factories: assembling applications with patterns, models, frameworks and tools. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26–30, pp. 16–27 (2003). DOI 10.1145/949344.949348

- [58] Guha, S., Harb, B.: Wavelet synopsis for data streams: minimizing non-euclidean error. In: Proc. of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 21-24, pp. 88–97 (2005). DOI 10.1145/1081870.1081884
- [59] Guo, Y., Biczak, M., Varbanescu, A.L., Iosup, A., Martella, C., Willke, T.L.: How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, May 19-23, pp. 395–404 (2014). DOI 10.1109/IPDPS.2014.49
- [60] Gurcan, F., Berigel, M.: Real-Time Processing of Big Data Streams: Lifecycle, Tools, Tasks, and Challenges. In: 2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), pp. 1–6 (2018). DOI 10.1109/ISMSIT.2018.8567061
- [61] Hinterberger, H.: Graph, pp. 1260–1261. Springer US, Boston, MA (2009). DOI 10.1007/978-0-387-39940-9\_1374
- [62] Holzschuher, F., Peinl, R.: Performance of graph query languages: comparison of Cypher, Gremlin and native access in Neo4j. In: Proc. of Workshop Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, March 22, pp. 195–204 (2013). DOI 10.1145/2457317.2457351
- [63] Internet Live Stats: Twitter Usage Statistics (accessed January 2020). <https://www.internetlivestats.com/twitter-statistics/>
- [64] JanusGraph Authors: JanusGraph. Distributed, open source, massively scalable graph database. (accessed June 2020). <https://janusgraph.org/>
- [65] Johann, S., Egyed, A.: Instant and incremental transformation of models. In: 19th IEEE International Conference on Automated Software Engineering (ASE 2004), September 20-25, pp. 362–365 (2004). DOI 10.1109/ASE.2004.10047
- [66] Jouault, F., Tisi, M.: Towards incremental execution of ATL transformations. In: Proc. of Theory and Practice of Model Transformations, Third

- International Conference, ICMT 2010, June 28-July 2, pp. 123–137 (2010). DOI 10.1007/978-3-642-13688-7\\_9
- [67] Jukss, M., Verbrugge, C., Elaasar, M., Vangheluwe, H.: Scope in model transformations. *Software and System Modeling* **17**(4), 1227–1252 (2018)
- [68] Kafka, A.: Apache Kafka. A distributed streaming platform (accessed May 2019). <https://kafka.apache.org/intro>
- [69] Kalavri, V., Vlassov, V., Haridi, S.: High-level programming abstractions for distributed graph processing. *IEEE Trans. Knowl. Data Eng.* **30**(2), 305–324 (2018). DOI 10.1109/TKDE.2017.2762294
- [70] Karau, H., Konwinski, A., Wendell, P., Zaharia, M.: *Learning Spark*. O’Reilly (2015)
- [71] Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. *J. ACM* **46**(5), 604–632 (1999). DOI 10.1145/324133.324140
- [72] Kleppe, A., Warmer, J., Bast, W.: *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley (2003). URL <http://www.informit.com/store/mda-explained-the-model-driven-architecture-practice-9780321194428>
- [73] Knight, J.C., Strunk, E.A.: Achieving critical system survivability through software architectures. In: *Architecting Dependable Systems II - [the book is a result of the ICSE 2003 Workshop on Software Architectures for Dependable Systems]*, pp. 51–78 (2003). DOI 10.1007/978-3-540-25939-8\\_3
- [74] Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Math.* **6**(1), 29–123 (2009). DOI 10.1080/15427951.2009.10129177
- [75] Li, K., Li, G.: Approximate query processing: What is new and where to go? - A survey on approximate query processing. *Data Science and Engineering* **3**(4), 379–397 (2018). DOI 10.1007/s41019-018-0074-4

- [76] Liu, Q.: Approximate Query Processing, pp. 113–119. Springer US, Boston, MA (2009). DOI 10.1007/978-0-387-39940-9\_534
- [77] Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2002)
- [78] Luckham, D.C.: Event Processing for Business: Organizing the Real-Time Enterprise. Wiley (2012)
- [79] Macías, F., Guerra, E., de Lara, J.: Towards Rearchitecting Meta-Models into Multi-level Models. In: Proc. of Conceptual Modeling - 36th International Conference, ER 2017, November 6-9, pp. 59–68 (2017). DOI 10.1007/978-3-319-69904-2\_5
- [80] Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proc. of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, June 6-10, pp. 135–146 (2010). DOI 10.1145/1807167.1807184
- [81] Manning, C.D., Raghavan, P., Schütze, H.: Introduction to information retrieval. Cambridge University Press (2008). DOI 10.1017/CBO9780511809071
- [82] Marz, N., Warren, J.: Big Data: Principles and best practices of scalable realtime data systems. Manning Publications Company (2015)
- [83] Memgraph Ltd: Memgraph graph database (accessed November 2019). <https://memgraph.com/>
- [84] Memgraph Ltd: Memgraph Indexing (accessed September 2020). <https://docs.memgraph.com/memgraph/concepts-overview/indexing>
- [85] Mhedhbi, A., Gupta, P., Khaliq, S., Salihoglu, S.: A+ indexes: Lightweight and highly flexible adjacency lists for graph database management systems. CoRR **abs/2004.00130** (2020)
- [86] Moreno, N., Bertoa, M.F., Barquero, G., Burgueño, L., Troya, J., García-López, A., Vallecillo, A.: Managing uncertain complex events in web of things applications. In: Proc. of Web Engineering - 18th International Conference,

- ICWE 2018, June 5-8, pp. 349–357 (2018). DOI 10.1007/978-3-319-91662-0\\_28
- [87] Mozafari, B., Niu, N.: A handbook for building an approximate query engine. *IEEE Data Eng. Bull.* **38**(3), 3–29 (2015)
- [88] Neo4j: Cypher Query Language (accessed February 2020). <https://neo4j.com/developer/cypher-query-language/>
- [89] Neo4j: Neo4j Graph Platform (accessed November 2019). <https://neo4j.com/>
- [90] Neo4j: Neo4j - Indexes for search performance (accessed September 2020). <https://neo4j.com/docs/cypher-manual/current/administration/indexes-for-search-performance/index.html>
- [91] Neo4j Inc: The openCypher project (accessed February 2020). <https://www.opencypher.org/>
- [92] OMG: Model Driven Architecture - A Technical Perspective (2001). URL <http://www.omg.org/docs/ormsc/01-07-01.pdf>
- [93] OMG: MDA Guide V.1.0.1 (2003). URL <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>
- [94] OMG: UML 2.3.1 Superstructure specification (2010). URL <http://www.omg.org/spec/UML/2.3/Superstructure/PDF>
- [95] OrientDB: LiveQuery (accessed July 2020). <https://orientdb.com/nosql/livequery/>
- [96] Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project (1998)
- [97] Perliger, A., Pedahzur, A.: Social network analysis in the study of terrorism and political violence. *PS: Political Science and Politics* **44**(1), 45–50 (2011). DOI 10.1017/S1049096510001848

- [98] Piatetsky-Shapiro, G., Connell, C.: Accurate estimation of the number of tuples satisfying a condition. In: Proc. of Annual Meeting, SIGMOD'84, June 18-21, pp. 256–276 (1984). DOI 10.1145/602259.602294
- [99] Razavi, A., Kontogiannis, K.: Partial evaluation of model transformations. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, pp. 562–572 (2012). DOI 10.1109/ICSE.2012.6227160
- [100] Real, E., Shlens, J., , Pan, S.M.X., Vanhoucke, V.: YouTube-BoundingBoxes Dataset (accessed October 2019). <https://research.google.com/youtube-bb/>
- [101] Richardson, M., Domingos, P.M.: The Intelligent surfer: Probabilistic Combination of Link and Content Information in PageRank. In: Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8], pp. 1441–1448 (2001)
- [102] Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006). DOI 10.1109/MC.2006.58
- [103] Seidewitz, E.: What Models Mean. IEEE Software **20**(5), 26–32 (2003). DOI 10.1109/MS.2003.1231147
- [104] Sellis, T.K.: Review - the aqua approximate query answering system. ACM SIGMOD Digital Review **2** (2000)
- [105] Shawi, R.E., Batarfi, O., Fayoumi, A.G., Barnawi, A., Sakr, S.: Big Graph Processing Systems: State-of-the-Art and Open Challenges. In: First IEEE International Conference on Big Data Computing Service and Applications, BigDataService 2015, March 30 - April 2, pp. 24–33 (2015). DOI 10.1109/BigDataService.2015.11
- [106] Spark, A.: GraphFrames (accessed February 2020). [https://graphframes.github.io/graphframes/docs/\\_site/index.html](https://graphframes.github.io/graphframes/docs/_site/index.html)
- [107] Spark, A.: GraphFrames User Guide (accessed February 2020). [https://graphframes.github.io/graphframes/docs/\\_site/user-guide.html](https://graphframes.github.io/graphframes/docs/_site/user-guide.html)

- [108] Spark, A.: Spark Streaming Programming (accessed May 2019). <https://spark.apache.org/docs/latest/streaming-programming-guide.html>
- [109] Stachowiak, H.: Allgemeine Modelltheorie (1973)
- [110] Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In: Proc. of Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, September 28 - October 3, pp. 653–669 (2014). DOI 10.1007/978-3-319-11653-2\\_40
- [111] Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling* **17**(4), 1365–1393 (2018). DOI 10.1007/s10270-016-0571-8
- [112] Szárnyas, G., Marton, J., Maginecz, J., Varró, D.: Reducing property graph queries to relational algebra for incremental view maintenance. *CoRR abs/1806.07344* (2018)
- [113] TinkerPop: And Step. Apache TinkerGraph (accessed February 2020). <http://tinkerpop.apache.org/docs/current/reference/#and-step>
- [114] TinkerPop: Has Step. Apache TinkerGraph (accessed February 2020). <http://tinkerpop.apache.org/docs/current/reference/#has-step>
- [115] TinkerPop: Not Step. Apache TinkerGraph (accessed February 2020). <http://tinkerpop.apache.org/docs/current/reference/#not-step>
- [116] TinkerPop: Or Step. Apache TinkerGraph (accessed February 2020). <http://tinkerpop.apache.org/docs/current/reference/#or-step>
- [117] TinkerPop: Where Step. Apache TinkerGraph (accessed February 2020). <http://tinkerpop.apache.org/docs/current/reference/#where-step>
- [118] TinkerPop: Apache TinkerGraph (accessed October 2019). <http://tinkerpop.apache.org/docs/current/reference/#tinkergraph-gremlin>



- [119] TinkerPop: TinkerGraph Indices (accessed September 2020). <https://tinkerpop.apache.org/javadocs/3.2.2/full/org/apache/tinkerpop/gremlin/tinkergraph/structure/TinkerGraph.html#vertexIndex>
- [120] Tinkerpop, A.: Interface VertexProgram (accessed Oct 2019). <http://tinkerpop.apache.org/javadocs/3.1.4/core/org/apache/tinkerpop/gremlin/process/computer/VertexProgram.html>.
- [121] Troya, J., Wimmer, M., Burgueño, L., Vallecillo, A.: Towards approximate model transformations. In: Proc. of the Workshop on Analysis of Model Transformations (AMT@MoDELS'14), pp. 44–53. CEUR-WS (2014)
- [122] Trushkowsky, B., Kraska, T., Franklin, M.J., Sarkar, P.: Crowdsourced enumeration queries. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, April 8–12, pp. 673–684 (2013). DOI 10.1109/ICDE.2013.6544865
- [123] Trushkowsky, B., Kraska, T., Franklin, M.J., Sarkar, P.: Answering enumeration queries with the crowd. *Commun. ACM* **59**(1), 118–127 (2016). DOI 10.1145/2845644
- [124] Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* **98**, 80–99 (2015). DOI 10.1016/j.scico.2014.01.004
- [125] Uta, A., Ghit, B., Dave, A., Boncz, P.A.: [Demo] Low-latency Spark Queries on Updatable Data. In: Proc. of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, June 30 - July 5, pp. 2009–2012 (2019). DOI 10.1145/3299869.3320227
- [126] W3C RDF Data Access Working Group: SPARQL Query Language (accessed November 2019). <https://www.w3.org/TR/rdf-sparql-query/>
- [127] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your models Ready for MDA, 2nd edn. Addison-Wesley (2003)

- [128] Webber, J., Robinson, I., Eifrem, E.: Graph Databases. O'Reilly Media (2013)
- [129] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in Software Engineering. Springer (2012). DOI 10.1007/978-3-642-29044-2
- [130] Yang, C.C., Ng, T.D.: Terrorism and Crime Related Weblog Social Network: Link, Content Analysis and Information Visualization. In: Proc. of IEEE International Conference on Intelligence and Security Informatics, ISI 2007, May 23-24, pp. 55–58 (2007). DOI 10.1109/ISI.2007.379533
- [131] Yorker, T.N.: Data from the New Yorker Caption Contest (accessed October 2019). <https://github.com/nextml/caption-contest-data>



# Appendix A

## Results and queries for the comparison of Processing Platforms

---

In this appendix we show some results of the experiments proposed in Chapter 3, as well as the queries for the two case studies (TrainBenchmark and TwitterFlickr). In this way, in Section A.1 we present the implementation of all queries used in the experiments, whereas in Section A.2 we show some figures and tables with the results of the experiments performed for TrainBenchmark example.

### A.1 Queries for processing platforms

The queries presented in this section have two types of implementations: (i) queries without any side effect and (ii) queries that involve an effect over the source data (adding, removing or updating existing elements). Then, we show both implementations in for the two case studies the following.

### A.1.1 TwitterFlickr queries with effect

Listing A.1: Gremlin queries with effect for TwitterFlickr case study

```

1 // HotTopic
2 g.V().hasLabel("Hashtag")
3   .where(
4     __.in("contains","tags")
5     .count().is(P.gte(100))
6   ).as("h")
7   .addV("HotTopic")
8     .property("date", System.currentTimeMillis())
9     .as("ht")
10  .addE("event")
11    .from("ht").to("h").iterate();
12 // PopularTwitterPhoto
13 g.V().hasLabel("Hashtag")
14   .where(
15     __.in("contains")
16     .where(
17       __.hasLabel("Tweet")
18       .in("likes").count().is(P.gte(30))
19     )
20   ).as("h")
21   .in().hasLabel("Photo").as("p")
22   .addV("PopularTwitterPhoto")
23     .property("hashtag",__.select("h").id())
24     .property("date", System.currentTimeMillis())
25     .as("labelPTP")
26   .addE("event")
27     .from("labelPTP").to("p").iterate();
28 // PopularFlickrPhoto
29 g.V().hasLabel("Photo")
30   .where(
31     __.in("favorites")
32     .where(
33       __.hasLabel("FlickrUser").in("follows")
34       .count().is(P.gte(50))
35     ).count().is(P.gte(30))
36     .as("labelPhoto")
37   .addV("PopularFlickrPhoto")
38     .property("date", System.currentTimeMillis())
39     .as("labelPFP")
40   .addE("event")
41     .from("labelPFP").to("labelPhoto").iterate();
42 // NiceTwitterPhoto
43 List tags = g.V().hasLabel("TwitterUser")
44   .where(

```

```

45     __.in("follows")
46     .where(
47         __.in("follows").count().is(P.gte(50))
48         ).count().is(P.gte(50))
49     ).as("tu")
50     .out("publishes").hasLabel("Tweet")
51     .out("contains").hasLabel("Hashtag")
52     .as("h")
53     .select("tu", "h").groupCount()
54     .unfold().where(__.select(values).is(P.gte(3))).select(keys)
55     .select("h").toList();
56 g.V().hasLabel("Hashtag")
57     .is(P.within(tags)).as("h")
58     .in("tags").hasLabel("Photo").as("p")
59     .addV("NiceTwitterPhoto")
60         .property("hashtag", __.select("h").id())
61         .property("date", System.currentTimeMillis())
62         .as("labelNTP")
63     .addE("event")
64         .from("labelNTP").to("p").iterate();
65 // ActiveUserTweeted
66 g.V().hasLabel("Tweet")
67     .order().by("currentTimestamp", asc).tail(10000)
68     .as("t")
69     .in("publishes").hasLabel("TwitterUser")
70     .where(
71         __.out("follows").count().is(P.gte(5000))
72     ).and(
73         __.in("follows")
74         .where(
75             __.in("follows").count().is(P.gte(30))
76         ).count().is(P.gte(30))
77     ).as("tu")
78     .addV("ActiveUserTweeted")
79         .property("tweet", __.select("t").id())
80         .property("date", System.currentTimeMillis())
81         .as("labelActive")
82     .addE("event")
83         .from("labelActive").to("tu").iterate();

```

Listing A.2: Neo4j queries with effect for TwitterFlickr case study

```

1 // HotTopic
2 StatementResult result = session.run(
3     "MATCH (p)-[tagsE:tags]->(h:Hashtag)<-[containsE:contains]-(t)
4     WITH h,(COUNT(tagsE) + COUNT(containsE))
5     AS sumHT WHERE sumHT>100
6     CREATE (:HotTopic{date:timestamp()})-[:EVENT]->(h)");

```

```

7 result.consume();
8 // PopularTwitterPhoto
9 StatementResult result = session.run(
10     "MATCH (u)-[:likes]->(t:Tweet)
11         WITH t, count(1) AS likes WHERE likes >= 30
12     MATCH (t)-[:contains]->(h:Hashtag)
13         WITH DISTINCT h as h
14     MATCH (h)<-[:tags]-(p:Photo)
15     CREATE (:PopularTwitterPhoto{
16         timestamp:timestamp(), idHashtag: h.id})
17         -[:EVENT]->(p)");
18 result.consume();
19 // PopularFlickrPhoto
20 StatementResult result = session.run(
21     "MATCH (fu2:FlickrUser)-[:follows]->(fu:FlickrUser)
22         WITH fu, COUNT(fu2) AS followers WHERE followers >= 50
23     MATCH (fu)-[:favorites]->(p:Photo)
24         WITH p, COUNT(fu) AS favs WHERE favs >= 30
25     CREATE (:PopularFlickrPhoto{timestamp:timestamp()})
26         -[:EVENT]->(p)");
27 result.consume();
28 //NiceTwitterPhoto
29 StatementResult result = session.run(
30     "MATCH (ffollower:TwitterUser)-[:follows]->(follower:TwitterUser)
31         WITH follower, COUNT(ffollower)
32         AS ffollowers WHERE ffollowers >= 50
33     MATCH (follower)-[:follows]->(tu:TwitterUser)
34         WITH DISTINCT tu, COUNT(follower)
35         AS followers
36         WHERE followers >= 50 WITH DISTINCT tu
37     MATCH (tu)-[:publishes]->(t:Tweet)
38         -[:containsE:contains]->(h:Hashtag)
39         WITH DISTINCT h, tu, COUNT(distinct containsE)
40         AS uses WHERE uses >= 3 WITH DISTINCT h
41     MATCH (h)<-[:tags]-(p:Photo)
42     CREATE (:NiceTwitterPhoto{
43         timestamp:timestamp(), idHashtag: h.id})
44         -[:EVENT]->(p)");
45 result.consume();
46 //ActiveUserTweeted
47 StatementResult result = session.run(
48     "MATCH (ffollower:TwitterUser)-[:follows]->(follower:TwitterUser)
49         WITH follower, COUNT(ffollower)
50         AS ffollowers WHERE ffollowers >= 30
51     MATCH (follower)-[:follows]->(tu:TwitterUser)
52         WITH tu, COUNT(follower) AS followers WHERE followers >= 30
53     MATCH (tu)-[:follows]->(tu2:TwitterUser)
54         WITH tu, followers, COUNT(tu2)
55         AS following WHERE following >= 5000

```

```

56 MATCH (t:Tweet) WITH t,tu
57 ORDER BY t.currentTimestamp DESC LIMIT 10000
58 MATCH (tu)-[:publishes]->(t)
59 CREATE (:ActiveUserTweeted{timestamp:timestamp(), tweetid: t.id})
60 -[:EVENT]->(tu)");
61 result.consume();

```

Listing A.3: CrateDB queries with effect for TwitterFlickr case study

```

1 //HotTopic
2 PreparedStatement statement = conn.prepareStatement(
3     "INSERT INTO hottopic (hashtagid,date)
4     (SELECT a.idnode, CURRENT_TIMESTAMP FROM
5     (SELECT DISTINCT
6         h.idNode, h.id, COUNT(c.hashtagid) countContains
7         FROM hashtag h, contains containsE
8         WHERE h.idNode=containsE.hashtagid
9         GROUP BY h.idNode, h.id) a,
10    (SELECT DISTINCT
11        h.idNode, h.id, COUNT(tagsE.hashtagid) countTags
12        FROM hashtag h, tags tagsE
13        WHERE h.idNode=c.hashtagid
14        GROUP BY h.idNode, h.id) b
15    WHERE a.idnode = b.idnode
16    AND a.countContains + b.countTags > 100)");
17 statement.execute();
18 //PopularTwitterPhoto
19 PreparedStatement statement = conn.prepareStatement(
20     "INSERT INTO populartwitterphoto (idphoto, idhashtag, date)
21     (SELECT tagsE.photoid, tagsE.hashtagid, CURRENT_TIMESTAMP
22     FROM tags tagsE
23     WHERE tagsE.hashtagid
24     IN (SELECT DISTINCT containsE.hashtagid
25     FROM contains containsE
26     WHERE containsE.tweetid
27     IN (SELECT DISTINCT l.tweetid
28         FROM likes l
29         GROUP BY l.tweetid
30         HAVING COUNT(l.tweetid) >= 30)))");
31 statement.execute();
32 //PopularFlickrPhoto
33 PreparedStatement statement = conn.prepareStatement(
34     "INSERT INTO popularflickrphoto (idphoto,date)
35     (SELECT photoid, CURRENT_TIMESTAMP FROM
36     favorites,
37     (SELECT DISTINCT fu.idnode
38     FROM follows f, flickruser fu
39     WHERE fu.idnode = f.userid2

```



```

40         GROUP BY fu.idnode, fu.username
41         HAVING COUNT(fu.idnode) >= 50) followers
42     WHERE flickruserid=idnode
43     GROUP BY photoid
44     HAVING COUNT(photoid) >= 30)");
45 statement.execute();
46 //NiceTwitterPhoto
47 PreparedStatement statement = conn.prepareStatement(
48     "INSERT INTO NiceTwitterPhoto (photoid, hashtagid,date)
49     (SELECT tagsE.photoid, tagsE.hashtagid, CURRENT_TIMESTAMP
50     ↪FROM
51     tags tagsE,
52     (SELECT DISTINCT containsE.hashtagid FROM
53     contains containsE,
54     (SELECT DISTINCT tweetid, twitteruserid
55     FROM publishes
56     WHERE twitteruserid IN (SELECT userid2 FROM
57     (SELECT userid2, followers FROM
58     follows,
59     (SELECT
60     userid2 userid,
61     COUNT(userid1) followers
62     FROM follows, twitteruser
63     WHERE userid2=idnode
64     GROUP BY userid2) numFollows
65     WHERE userid1=userid
66     AND followers >= 50)
67     ↪numFollPerFollower
68     GROUP BY userid2
69     HAVING COUNT(userid2) >= 50)) tweets
70     WHERE tweets.tweetid = containsE.tweetid
71     GROUP BY containsE.hashtagid, tweets.twitteruserid
72     HAVING COUNT( *) >= 3) hashtags
73     WHERE tagsE.hashtagid = hashtags.hashtagid)");
74 statement.execute();
75 //ActiveUserTweeted
76 PreparedStatement statement = conn.prepareStatement(
77     "INSERT INTO
78     activeusertweeted (tweetid, twitteruserid, currenttimestamp)
79     (SELECT d.tweetid, a.userid1, CURRENT_TIMESTAMP
80     FROM
81     (SELECT userid1
82     FROM follows
83     GROUP BY userid1
84     HAVING COUNT(userid1) >= 5000) a,
85     (SELECT userid2 FROM
86     (SELECT userid2, followers FROM
87     follows,
88     (SELECT userid2 userid, COUNT(userid1) followers

```

```

87         FROM follows, twitteruser
88         WHERE userid2=idnode
89         GROUP BY userid2) numFollows
90     WHERE userid1=userid
91         AND followers >= 30) numFollPerFollower
92     GROUP BY userid2
93     HAVING COUNT(userid2) >= 30) b,
94     publishes d
95     WHERE a.userid1 = b.userid2
96         AND b.userid2 = d.twitteruserid
97         AND d.tweetid IN
98         (SELECT idnode
99         FROM tweet
100         ORDER BY currenttimestamp
101         DESC LIMIT 10000));
102 statement.execute();

```

Listing A.4: Memgraph queries with effect for TwitterFlickr case study

```

1 // HotTopic
2 StatementResult result = session.run(
3     "MATCH (p)-[tagsE:tags]->(h:Hashtag)<-[containsE:containsA]-(t)
4     WITH h,(COUNT(tagsE) + COUNT(containsE))
5     AS sumHT WHERE sumHT>100
6     CREATE (:HotTopic{date:timestamp()})-[:EVENT]->(h)");
7 result.consume();
8 // PopularTwitterPhoto
9 StatementResult result = session.run(
10     "MATCH (u)-[l:likes]->(t:Tweet)
11     WITH t, count(l) AS likes WHERE likes >= 30
12     MATCH (t)-[:containsA]->(h:Hashtag)
13     WITH DISTINCT h AS h
14     MATCH (h)<-[:tags]-(p:Photo)
15     CREATE (:PopularTwitterPhoto{
16         timestamp:timestamp(), idHashtag: h.id})
17     -[:EVENT]->(p)");
18 result.consume();
19 // NiceTwitterPhoto
20 StatementResult result = session.run(
21     "MATCH (ffollower:TwitterUser)-[:follows]->(follower:TwitterUser)
22     WITH follower, COUNT(ffollower)
23     AS ffollowers WHERE ffollowers >= 50
24     MATCH (follower)-[:follows]->(tu:TwitterUser)
25     WITH DISTINCT tu, COUNT(follower)
26     AS followers WHERE followers >= 50 WITH DISTINCT tu
27     MATCH (tu)-[:publishes]->(t:Tweet)-[c:containsA]->(h:Hashtag)
28     WITH DISTINCT h, tu, c WITH h, tu, COUNT(c)
29     AS uses WHERE uses >= 3 WITH DISTINCT h

```

```

30 MATCH (h) <-[:tags]-(p:Photo)
31 CREATE (:NiceTwitterPhoto{
32     timestamp:timestamp(), idHashtag: h.id})
33     -[:EVENT]->(p)");
34 result.consume();

```

### A.1.2 TwitterFlickr queries without effect

Listing A.5: Gremlin queries without effect for TwitterFlickr case study

```

1 // HotTopic
2 g.V().hasLabel("Hashtag")
3   .where(
4     __.in("contains","tags")
5     .count().is(P.gte(100))
6   ).toList();
7 // PopularTwitterPhoto
8 g.V().hasLabel("Hashtag")
9   .where(
10    __.in("contains")
11    .where(
12      __.hasLabel("Tweet")
13      .in("likes").count().is(P.gte(30))
14    )
15  ).in().hasLabel("Photo").toList();
16 // PopularFlickrPhoto
17 g.V().hasLabel("Photo")
18   .where(
19     __.in("favorites")
20     .where(
21       __.hasLabel("FlickrUser").in("follows")
22       .count().is(P.gte(50))
23     ).count().is(P.gte(30))
24   ).toList();
25 // NiceTwitterPhoto
26 List tags = g.V().hasLabel("TwitterUser")
27   .where(
28     __.in("follows")
29     .where(
30       __.in("follows").count().is(P.gte(50))
31     ).count().is(P.gte(50))
32   ).as("tu")
33   .out("publishes").hasLabel("Tweet")
34   .out("contains").hasLabel("Hashtag")
35   .as("h")

```

```

36     .select("tu", "h").groupBy().groupCount()
37     .unfold().where(__.select(values).is(P.gte(3))).select(keys)
38     .select("h").toList();
39 g.V().hasLabel("Hashtag")
40     .is(P.within(tags))
41     .in("tags").hasLabel("Photo").toList();
42 // ActiveUserTweeted
43 g.V().hasLabel("Tweet")
44     .order().by("currentTimestamp", asc).tail(10000)
45     .in("publishes").hasLabel("TwitterUser")
46     .where(
47         __.out("follows").count().is(P.gte(5000))
48     ).and(
49         __.in("follows")
50         .where(
51             __.in("follows").count().is(P.gte(30))
52         ).count().is(P.gte(30))
53     ).toList();

```

Listing A.6: Neo4j queries without effect for TwitterFlickr case study

```

1 // HotTopic
2 StatementResult result = session.run(
3     "MATCH (p)-[tagsE:tags]->(h:Hashtag)<-[containsE:contains]-(t)
4     WITH h,(COUNT(tagsE) + COUNT(containsE))
5     AS sumHT WHERE sumHT>100
6     RETURN h");
7 result.list();
8 // PopularTwitterPhoto
9 StatementResult result = session.run(
10     "MATCH (u)-[l:likes]->(t:Tweet)
11     WITH t, count(l) AS likes WHERE likes >= 30
12     MATCH (t)-[:contains]->(h:Hashtag)
13     WITH DISTINCT h as h
14     MATCH (h)<-[:tags]-(p:Photo)
15     RETURN p");
16 result.list();
17 // PopularFlickrPhoto
18 StatementResult result = session.run(
19     "MATCH (fu2:FlickrUser)-[:follows]->(fu:FlickrUser)
20     WITH fu, COUNT(fu2) AS followers WHERE followers >= 50
21     MATCH (fu)-[:favorites]->(p:Photo)
22     WITH p, COUNT(fu) AS favs WHERE favs >= 30
23     RETURN p");
24 result.list();
25 //NiceTwitterPhoto
26 StatementResult result = session.run(
27     "MATCH (ffollower:TwitterUser)-[:follows]->(follower:TwitterUser)

```

```

28     WITH follower, COUNT(ffollower)
29     AS ffollowers WHERE ffollowers >= 50
30     MATCH (follower) -[:follows]->(tu:TwitterUser)
31     WITH DISTINCT tu, COUNT(follower)
32     AS followers
33     WHERE followers >= 50 WITH DISTINCT tu
34     MATCH (tu) -[:publishes]->(t:Tweet)
35         -[containsE:contains]->(h:Hashtag)
36     WITH DISTINCT h, tu, COUNT(distinct containsE)
37     AS uses WHERE uses >= 3 WITH DISTINCT h
38     MATCH (h) <-[:tags]-(p:Photo)
39     RETURN p");
40 result.list();
41 //ActiveUserTweeted
42 StatementResult result = session.run(
43     "MATCH (ffollower:TwitterUser) -[:follows]->(follower:TwitterUser)
44     WITH follower, COUNT(ffollower)
45     AS ffollowers WHERE ffollowers >= 30
46     MATCH (follower) -[:follows]->(tu:TwitterUser)
47     WITH tu, COUNT(follower) AS followers WHERE followers >= 30
48     MATCH (tu) -[:follows]->(tu2:TwitterUser)
49     WITH tu, followers, COUNT(tu2)
50     AS following WHERE following >= 5000
51     MATCH (t:Tweet) WITH t,tu
52     ORDER BY t.currentTimestamp DESC LIMIT 10000
53     MATCH (tu) -[:publishes]->(t)
54     RETURN tu");
55 result.list();

```

Listing A.7: CrateDB queries without effect for TwitterFlickr case study

```

1 //HotTopic
2 PreparedStatement statement = conn.executeQuery(
3     "SELECT a.idnode, CURRENT_TIMESTAMP FROM
4     (SELECT DISTINCT
5         h.idNode, h.id, COUNT(c.hashtagid) countContains
6         FROM hashtag h, contains containsE
7         WHERE h.idNode=containsE.hashtagid
8         GROUP BY h.idNode, h.id) a,
9     (SELECT DISTINCT
10        h.idNode, h.id, COUNT(tagsE.hashtagid) countTags
11        FROM hashtag h, tags tagsE
12        WHERE h.idNode=c.hashtagid
13        GROUP BY h.idNode, h.id) b
14    WHERE a.idnode = b.idnode
15        AND a.countContains + b.countTags > 100");
16 //PopularTwitterPhoto
17 PreparedStatement statement = conn.executeQuery(

```

```

18 "SELECT tagsE.photoid, tagsE.hashtagid, CURRENT_TIMESTAMP
19 FROM tags tagsE
20 WHERE tagsE.hashtagid
21 IN (SELECT DISTINCT containsE.hashtagid
22 FROM contains containsE
23 WHERE containsE.tweetid
24 IN (SELECT DISTINCT l.tweetid
25 FROM likes l
26 GROUP BY l.tweetid
27 HAVING COUNT(l.tweetid) >= 30));
28 //PopularFlickrPhoto
29 PreparedStatement statement = conn.executeQuery(
30 "SELECT photoid, CURRENT_TIMESTAMP FROM
31 favorites,
32 (SELECT DISTINCT fu.idnode
33 FROM follows f, flickruser fu
34 WHERE fu.idnode = f.userid2
35 GROUP BY fu.idnode, fu.usrname
36 HAVING COUNT(fu.idnode) >= 50) followers
37 WHERE flickruserid=idnode
38 GROUP BY photoid
39 HAVING COUNT(photoid) >= 30");
40 //NiceTwitterPhoto
41 PreparedStatement statement = conn.executeQuery(
42 "SELECT tagsE.photoid, tagsE.hashtagid, CURRENT_TIMESTAMP FROM
43 tags tagsE,
44 (SELECT DISTINCT containsE.hashtagid FROM
45 contains containsE,
46 (SELECT DISTINCT tweetid, twitteruserid
47 FROM publishes
48 WHERE twitteruserid IN (SELECT userid2 FROM
49 (SELECT userid2, followers FROM
50 follows,
51 (SELECT
52 userid2 userid,
53 COUNT(userid1) followers
54 FROM follows, twitteruser
55 WHERE userid2=idnode
56 GROUP BY userid2) numFollows
57 WHERE userid1=userid
58 AND followers >= 50)
59 numFollPerFollower
60 GROUP BY userid2
61 HAVING COUNT(userid2) >= 50)) tweets
62 WHERE tweets.tweetid = containsE.tweetid
63 GROUP BY containsE.hashtagid, tweets.twitteruserid
64 HAVING COUNT( *) >= 3) hashtags
65 WHERE tagsE.hashtagid = hashtags.hashtagid");
66 //ActiveUserTweeted

```

```

66 PreparedStatement statement = conn.executeQuery(
67     "SELECT d.tweetid, a.userid1, CURRENT_TIMESTAMP
68     FROM
69     (SELECT userid1
70     FROM follows
71     GROUP BY userid1
72     HAVING COUNT(userid1) >= 5000) a,
73     (SELECT userid2 FROM
74     (SELECT userid2, followers FROM
75     follows,
76     (SELECT userid2 userid, COUNT(userid1) followers
77     FROM follows, twitteruser
78     WHERE userid2=idnode
79     GROUP BY userid2) numFollows
80     WHERE userid1=userid
81     AND followers >= 30) numFollPerFollower
82     GROUP BY userid2
83     HAVING COUNT(userid2) >= 30) b,
84     publishes d
85 WHERE a.userid1 = b.userid2
86     AND b.userid2 = d.twitteruserid
87     AND d.tweetid IN
88     (SELECT idnode
89     FROM tweet
90     ORDER BY currenttimestamp
91     DESC LIMIT 10000)");

```

Listing A.8: Memgraph queries without effect for TwitterFlickr case study

```

1 // HotTopic
2 StatementResult result = session.run(
3     "MATCH (p)-[tagsE:tags]->(h:Hashtag)<-[containsE:containsA]-(t)
4     WITH h,(COUNT(tagsE) + COUNT(containsE))
5     AS sumHT WHERE sumHT>100
6     RETURN h");
7 result.list();
8 // PopularTwitterPhoto
9 StatementResult result = session.run(
10    "MATCH (u)-[l:likes]->(t:Tweet)
11    WITH t, count(l) AS likes WHERE likes >= 30
12    MATCH (t)-[:containsA]->(h:Hashtag)
13    WITH DISTINCT h AS h
14    MATCH (h)<-[:tags]-(p:Photo)
15    RETURN p");
16 result.list();
17 // NiceTwitterPhoto
18 StatementResult result = session.run(
19    "MATCH (ffollower:TwitterUser)-[:follows]->(follower:TwitterUser)

```

```

20     WITH follower, COUNT(ffollower)
21     AS ffollowers WHERE ffollowers >= 50
22     MATCH (follower)-[:follows]->(tu:TwitterUser)
23     WITH DISTINCT tu, COUNT(follower)
24     AS followers WHERE followers >= 50 WITH DISTINCT tu
25     MATCH (tu)-[:publishes]->(t:Tweet)-[c:containsA]->(h:Hashtag)
26     WITH DISTINCT h, tu, c WITH h, tu, COUNT(c)
27     AS uses WHERE uses >= 3 WITH DISTINCT h
28     MATCH (h)-[:tags]->(p:Photo)
29     RETURN p";
30 result.list();

```

Listing A.9: GraphFrames queries without effect for TwitterFlickr case study

```

1 //HotTopic
2 graph.find("(a)-[tagsE]->(h)")
3     .filter("tagsE.relationship = 'tags'"
4         OR tagsE.relationship = 'contains'")
5     .groupBy("h").count()
6     .select("h").where("count >= 100");
7 //PopularTwitterPhoto
8 var popularTwitterPhoto =
9     graph.find("(u)-[l]->(t)")
10        .filter("u.node = 'TwitterUser'"
11            AND l.relationship = 'likes'"
12            AND t.node = 'Tweet'")
13        .groupBy("t").count()
14        .where("count >= 30").select("t");
15 var popularTwitterPhoto2 =
16     graph.find("(t)-[containsE]->(h)")
17        .filter("t.node = 'Tweet'"
18            AND containsE.relationship = 'contains'"
19            AND h.node = 'Hashtag'");
20 val popularTwitterPhotoJoin1 =
21     popularTwitterPhoto.join(
22         popularTwitterPhoto2,
23         popularTwitterPhoto.col("t")
24         ==
25         popularTwitterPhoto2.col("t"))
26     .select("h").distinct;
27 popularTwitterPhoto2 =
28     graph.find("(p)-[tagsE]->(h)")
29        .filter("p.node = 'Photo'"
30            AND tagsE.relationship = 'tags'"
31            AND h.node = 'Hashtag'");
32 popularTwitterPhoto =
33     popularTwitterPhotoJoin1.join(
34         popularTwitterPhoto2,

```



```

35     popularTwitterPhotoJoin1.col("h")
36     ==
37     popularTwitterPhoto2("h"))
38     .select("h", "p");
39
40 //PopularFlickrPhoto
41 var popularFlickrPhoto =
42     graph.find("(fu2)-[follows]->(fu)")
43     .filter("fu2.node = 'FlickrUser'
44             AND fu.node = 'FlickrUser'
45             AND follows.relationship = 'follows'")
46     .groupBy("fu").count().select("fu").where("count >= 50");
47 var fav =
48     graph.find("(fUser)-[favorites]->(p)")
49     .filter("fUser.node = 'FlickrUser'
50             AND favorites.relationship = 'favorites'
51             AND p.node = 'Photo'");
52 popularFlickrPhoto =
53     popularFlickrPhoto.join(
54         fav,
55         fav.col("fUser")
56         ==
57         popularFlickrPhoto.col("fu"))
58     .groupBy("p").count().where("count >= 30").select("p");
59 //NiceTwitterPhoto
60 var niceTwitterPhoto =
61     graph.find("(ffollower)-[follows]->(follower)")
62     .filter("ffollower.node = 'TwitterUser'
63             AND follower.node = 'TwitterUser'
64             AND follows.relationship = 'follows'")
65     .groupBy("follower").count
66     .where("count >= 50").select("follower");
67 var niceTwitterPhoto2 =
68     graph.find("(follower1)-[follows]->(tu)")
69     .filter("follower1.node = 'TwitterUser'
70             AND tu.node = 'TwitterUser'
71             AND follows.relationship = 'follows'");
72 var niceTwitterPhotoJoin1 =
73     niceTwitterPhoto2.join(
74         niceTwitterPhoto,
75         niceTwitterPhoto.col("follower")
76         ==
77         niceTwitterPhoto2.col("follower1"))
78     .groupBy("tu").count()
79     .where("count >= 50").select("tu");
80 niceTwitterPhoto2 =
81     graph.find("(tu1)-[publishes]->(t)")
82     .filter("tu1.node = 'TwitterUser'
83             AND publishes.relationship = 'publishes'")

```

```

84         AND t.node = 'Tweet')
85     .select("tu1", "t").distinct;
86 niceTwitterPhotoJoin1 =
87     niceTwitterPhoto2.join(
88         niceTwitterPhotoJoin1,
89         niceTwitterPhotoJoin1.col("tu")
90     )
91     niceTwitterPhoto2.col("tu1"));
92 niceTwitterPhoto2 =
93     graph.find("(t1)-[contains]->(h)")
94     .filter("t1.node = 'Tweet'
95         AND contains.relationship = 'contains'
96         AND h.node = 'Hashtag'");
97 niceTwitterPhotoJoin1 =
98     niceTwitterPhoto2.join(
99         niceTwitterPhotoJoin1,
100        niceTwitterPhotoJoin1.col("t")
101    )
102    niceTwitterPhoto2.col("t1"))
103    .select("tu1", "h", "contains").groupBy("h", "tu1").count
104    .where("count >= 3").select("h").distinct;
105 niceTwitterPhoto =
106     graph.find("(p)-[tags]->(hashtag)")
107     .filter("p.node = 'Photo'
108         AND tags.relationship = 'tags'
109         AND hashtag.node = 'Hashtag'");
110 niceTwitterPhoto =
111     niceTwitterPhoto.join(
112         niceTwitterPhotoJoin1,
113         niceTwitterPhoto.col("hashtag")
114     )
115     niceTwitterPhotoJoin1.col("h"))
116     .select("hashtag", "p");
117 //ActiveUserTweeted
118 var activeUserTweeted =
119     graph.find("(ffollower)-[follows]->(follower)")
120     .filter("ffollower.node = 'TwitterUser'
121         AND follower.node = 'TwitterUser'
122         AND follows.relationship = 'follows'")
123     .groupBy("follower").count
124     .where("count >= 30").select("follower");
125 val activeUserTweeted2 =
126     graph.find("(follower1)-[follows]->(tu)")
127     .filter("follower1.node = 'TwitterUser'
128         AND tu.node = 'TwitterUser'
129         AND follows.relationship = 'follows'");
130 var activeUserTweetedJoin1 =
131     activeUserTweeted2.join(
132         activeUserTweeted,

```

```

133         activeUserTweeted.col("follower")
134         ==
135         activeUserTweeted2.col("follower1"))
136     .groupBy("tu").count()
137     .where("count" >= 30).select("tu");
138 activeUserTweeted =
139     graph.find("(tu1)-[follows]->(tu2)")
140     .filter("follows.relationship = 'follows'
141           AND tu1.node = 'TwitterUser'
142           AND tu2.node = 'TwitterUser'")
143     .groupBy("tu1").count
144     .where("count" >= 5000).select("tu1");
145 activeUserTweetedJoin1 =
146     activeUserTweeted.join(
147         activeUserTweetedJoin1,
148         activeUserTweetedJoin1.col("tu")
149         ==
150         activeUserTweeted.col("tu1"));
151 activeUserTweeted =
152     graph.find("(tUser)-[publishes]->(t)")
153     .filter("tUser.node = 'TwitterUser'
154           AND publishes.relationship = 'publishes'
155           AND t.node = 'Tweet'")
156     .orderBy(desc("t.currentTimestamp")).limit(10000);
157 activeUserTweeted =
158     activeUserTweeted.join(
159         activeUserTweetedJoin1,
160         activeUserTweetedJoin1.col("tu1")
161         ==
162         activeUserTweeted.col("tUser"))
163     .select("t", "tUser");

```

### A.1.3 TrainBenchmark queries with effect

Listing A.10: Gremlin queries with effect for TrainBenchmark case study

```

1 // PosLength
2 List<Vertex> matches =
3     g.V().hasLabel("Segment")
4         .has("length", P.lte(0)).as("segment").toList();
5 for (Vertex n : matches) {
6     n.property("length", -((Integer) n.value("length")) + 1);
7 }
8 // SwitchMonitored
9 List<Vertex> matches =
10    g.V().hasLabel("Switch")

```

```

11     .not(__.outE("monitoredBy")).as("sw").toList();
12 for (Vertex n : matches) {
13     Vertex sensor = graph.addVertex("Sensor");
14     sensor.property("id", 0);
15     n.addEdge("monitoredBy", sensor);
16 }
17 // RouteSensor
18 List<Map<String, Object>> matches =
19     g.V().hasLabel("Route").as("route")
20         .out("follows").hasLabel("SwitchPosition").as("swP")
21         .out("target").hasLabel("Switch").as("sw")
22         .out("monitoredBy").hasLabel("Sensor").as("sensor")
23         .not(__.inE("requires").outV().as("route2"))
24         .where("route2", P.eq("route"))
25         .select("route", "sensor").toList();
26 for (Map<String, Object> n : matches) {
27     ((Vertex) n.get("route"))
28         .addEdge("requires", (Vertex) n.get("sensor"));
29 }
30 // SwitchSet
31 List<Map<String, Object>> matches =
32     g.V().hasLabel("Route")
33         .has("active", true).as("route")
34         .out("entry").hasLabel("Semaphore").as("semaphore")
35         .select("route")
36         .out("follows").hasLabel("SwitchPosition").as("swP")
37         .values("position").as("position").select("swP")
38         .out("target").hasLabel("Switch").as("sw")
39         .values("currentPosition").as("currentPosition")
40         .where("position", P.neq("currentPosition"))
41         .select("semaphore", "route",
42             "swP", "sw", "currentPosition", "position").toList();
43 for (Map<String, Object> n : matches) {
44     String position = (String) n.get("position");
45     ((Vertex) n.get("sw")).property("currentPosition", position);
46 }
47 // ConnectedSegments
48 List<Map<String, Object>> matches =
49     g.V().hasLabel("Sensor").as("sensor")
50         .in("monitoredBy").hasLabel("Segment").as("segment1")
51         .out("connectsTo").hasLabel("Segment").as("segment2")
52         .out("connectsTo").hasLabel("Segment").as("segment3")
53         .out("connectsTo").hasLabel("Segment").as("segment4")
54         .out("connectsTo").hasLabel("Segment").as("segment5")
55         .out("connectsTo").hasLabel("Segment").as("segment6")
56         .out("monitoredBy").as("sensor2")
57         .where("sensor", P.eq("sensor2"))
58         .select("sensor", "segment1", "segment2",
59             "segment3", "segment4", "segment5", "segment6").toList();

```

```

60 for (Map<String, Object> n : matches) {
61     Vertex segment2 = (Vertex) n.get("segment2");
62     segment2.remove();
63     ((Vertex) n.get("segment1"))
64         .addEdge("connectsTo", (Vertex) n.get("segment3"));
65 }
66 // SemaphoreNeighbor
67 List<Map<String, Object>> matches =
68     g.V().hasLabel("Route").as("route1")
69         .out("exit").hasLabel("Semaphore").as("semaphore")
70         .select("route1")
71         .out("requires").hasLabel("Sensor").as("sensor1")
72         .in("monitoredBy").as("te1")
73         .out("connectsTo").as("te2")
74         .out("monitoredBy").hasLabel("Sensor").as("sensor2")
75         .in("requires").hasLabel("Route").as("route2")
76         .where("route2", P.neq("route1")).select("route2")
77         .not(__.out("entry").as("semaphore2"))
78         .where("semaphore2", P.eq("semaphore")))
79         .select("semaphore", "route1", "route2",
80             "sensor1", "sensor2", "te1", "te2").toList();
81 for (Map<String, Object> n : matches) {
82     Vertex semaphore = (Vertex) n.get("semaphore");
83     Vertex route2 = (Vertex) n.get("route2");
84     if (!route2.edges(Direction.OUT, "entry").hasNext()) {
85         route2.addEdge("entry", semaphore);
86     }
87 }

```

Listing A.11: Neo4j queries with effect for TrainBenchmark case study

```

1 // PosLength
2 StatementResult result = session.run(
3     "MATCH (segment:Segment)
4         WHERE segment.length <= 0
5         SET segment.length = -segment.length + 1
6         RETURN segment, segment.length AS length");
7 result.consume();
8 // SwitchMonitored
9 StatementResult result = session.run(
10    "MATCH (sw:Switch), (n) WITH sw, MAX(n.id) AS maxid
11        WHERE NOT (sw)-[:monitoredBy]->()
12        CREATE (sw)-[:monitoredBy]->(:Sensor{id: maxid + 1})");
13 result.consume();
14 // RouteSensor
15 StatementResult result = session.run(
16    "MATCH (route:Route)-[:follows]->(swP:SwitchPosition)
17        -[:target]->(sw:Switch)-[:monitoredBy]->(sensor:Sensor)

```

```

18     WHERE NOT (route) -[:requires]->(sensor)
19     CREATE (route) -[:requires]->(sensor)");
20 result.consume();
21 // SwitchSet
22 StatementResult result = session.run(
23     "MATCH (semaphore:Semaphore) <-[:entry]-(route:Route)
24         -[:follows]->(swP:SwitchPosition) -[:target]->(sw:Switch)
25     WHERE semaphore.signal = "G0"
26     AND route.active = true
27     AND sw.currentPosition <> swP.position
28     SET sw.currentPosition = swP.position
29     RETURN semaphore, route, swP, sw,
30         sw.currentPosition AS currentPosition,
31         swP.position AS position");
32 result.consume();
33 // ConnectedSegments
34 StatementResult result = session.run(
35     "MATCH
36         (sensor:Sensor) <-[:monitoredBy]-(segment1:Segment),
37         (segment1:Segment) -[:connectsTo]->(segment2:Segment)
38         -[:connectsTo]->(segment3:Segment) -[:connectsTo]
39         ->(segment4:Segment) -[:connectsTo]->
40         (segment5:Segment) -[:connectsTo]->
41         (segment6:Segment),
42         (segment2:Segment) -[:monitoredBy]->(sensor:Sensor),
43         (segment3:Segment) -[:monitoredBy]->(sensor:Sensor),
44         (segment4:Segment) -[:monitoredBy]->(sensor:Sensor),
45         (segment5:Segment) -[:monitoredBy]->(sensor:Sensor),
46         (segment6:Segment) -[:monitoredBy]->(sensor:Sensor)
47     DETACH DELETE segment2");
48 result.consume();
49 // SemaphoreNeighbor
50 StatementResult result = session.run(
51     "MATCH (semaphore:Semaphore) <-[:exit]-(route1:Route)
52         -[:requires]->(sensor1:Sensor),
53         (sensor1) <-[:monitoredBy]-(te1) -[:connectsTo]
54         ->(te2) -[:monitoredBy]->(sensor2:Sensor)
55         <-[:requires]-(route2:Route)
56     WHERE NOT (semaphore) <-[:entry]-(route2)
57     AND route1 <> route2
58     CREATE (semaphore) <-[:entry]-(route2)");
59 result.consume();

```

Listing A.12: CrateDB queries with effect for TrainBenchmark case study

```

1 // PosLength
2 ResultSet resultset =
3     statement.executeQuery(

```

```

4      "SELECT id AS segment, length AS length
5      FROM Segment
6      WHERE length <= 0;");
7  long count = 0;
8  while (resultset.next()) {
9      count++;
10     Long segment = resultset.getLong("segment");
11     PreparedStatement statementP = conn
12         .prepareStatement(
13         "UPDATE Segment SET length = -length + 1
14         WHERE id = " + segment);
15     statementP.execute();
16 }
17 // SwitchMonitored
18 ResultSet resultset = statement.executeQuery(
19     "SELECT switch.id AS sw
20     FROM switch\n"
21     LEFT JOIN monitoredBy
22         ON monitoredBy.\"TrackElement_id\" = switch.id
23     WHERE monitoredBy.\"TrackElement_id\" IS NULL;");
24 long count = 0;
25 while (resultset.next()) {
26     Statement statement1 = conn.createStatement();
27     statement1.executeUpdate(
28         "INSERT INTO Sensor(id) (SELECT MAX(id) +1
29         FROM
30         (SELECT id AS id
31         FROM region
32         UNION ALL SELECT id AS id FROM route
33         UNION ALL SELECT id AS id FROM sensor
34         UNION ALL SELECT id AS id FROM switch
35         UNION ALL SELECT id AS id FROM segment
36         UNION ALL SELECT id AS id FROM semaphore
37         UNION ALL SELECT id AS id FROM sensor
38         UNION ALL SELECT id AS id FROM switchposition)
39         maxid)");
40     Statement statement2 = conn.createStatement();
41     ResultSet result = statement2.executeQuery(
42         "SELECT MAX(id) AS id FROM sensor");
43     result.next();
44     Statement statement3 = conn.createStatement();
45     statement3.executeUpdate(
46         "INSERT INTO monitoredBy(\"TrackElement_id\", \"Sensor_id\")
47         VALUES (\"
48         + resultset.getLong("sw") + "\",
49         + result.getLong("id") + ")");
50     count++;
51 }
52 // RouteSensor

```

```

53 int result = statement1.executeUpdate(
54     "INSERT INTO requires(\"Route_id\", \"Sensor_id\")
55     (SELECT Route.id, Sensor.id
56         FROM Sensor
57         INNER JOIN monitoredBy
58             ON monitoredBy.\"Sensor_id\" = Sensor.id
59         INNER JOIN Switch
60             ON Switch.id = monitoredBy.\"TrackElement_id\"
61         INNER JOIN SwitchPosition
62             ON SwitchPosition.target = Switch.id
63         INNER JOIN Route
64             ON Route.id = SwitchPosition.route
65         LEFT OUTER JOIN requires
66             ON requires.\"Route_id\" = Route.id
67             AND requires.\"Sensor_id\" = Sensor.id
68         WHERE requires.\"Sensor_id\" IS NULL);");
59 // SwitchSet
70 ResultSet resultset = statement.executeQuery(
71     "SELECT Semaphore.id AS semaphore,
72         Route.id AS route,
73         SwitchPosition.id AS swP,
74         Switch.id AS sw,
75         SwitchPosition.position AS position,
76         Switch.currentPosition AS currentPosition
77     FROM route
78         INNER JOIN SwitchPosition
79             ON Route.id = SwitchPosition.route
80         INNER JOIN Switch
81             ON SwitchPosition.target = Switch.id
82         INNER JOIN Semaphore
83             ON Route.entry = Semaphore.id
84     WHERE Route.active = 0
85         AND Switch.currentPosition != SwitchPosition.position
86         AND Semaphore.signal = 'GO';");
87 long count = 0;
88 while (resultset.next()) {
89     count++;
90     PreparedStatement statementP = conn.prepareStatement(
91         "UPDATE Switch
92         SET currentPosition = \"\" + resultset.getString("position")
93         + \"\" WHERE id = \" + resultset.getLong("sw"));
94     statementP.execute();
95 }
96 // ConnectedSegments
97 ResultSet resultset = statement.executeQuery(
98     "SELECT mb1.\"Sensor_id\" AS sensor,
99         ct1.\"TrackElement1_id\" AS segment1,
100         ct2.\"TrackElement1_id\" AS segment2,
101         ct3.\"TrackElement1_id\" AS segment3,

```



```

102         ct4.\"TrackElement1_id\" AS segment4,
103         ct5.\"TrackElement1_id\" AS segment5,
104         ct5.\"TrackElement2_id\" AS segment6
105 FROM connectsTo AS ct1
106     INNER JOIN connectsTo AS ct2
107         ON ct1.\"TrackElement2_id\"
108            = ct2.\"TrackElement1_id\"
109     INNER JOIN connectsTo AS ct3
110         ON ct2.\"TrackElement2_id\"
111            = ct3.\"TrackElement1_id\"
112     INNER JOIN connectsTo AS ct4
113         ON ct3.\"TrackElement2_id\"
114            = ct4.\"TrackElement1_id\"
115     INNER JOIN connectsTo AS ct5
116         ON ct4.\"TrackElement2_id\"
117            = ct5.\"TrackElement1_id\"
118     INNER JOIN monitoredBy AS mb1
119         ON mb1.\"TrackElement_id\"
120            = ct1.\"TrackElement1_id\"
121     INNER JOIN monitoredBy AS mb2
122         ON mb2.\"TrackElement_id\"
123            = ct2.\"TrackElement1_id\"
124     AND mb1.\"Sensor_id\"
125        = mb2.\"Sensor_id\"
126     INNER JOIN monitoredBy AS mb3
127         ON mb3.\"TrackElement_id\"
128            = ct3.\"TrackElement1_id\"
129     AND mb1.\"Sensor_id\"
130        = mb3.\"Sensor_id\"
131     INNER JOIN monitoredBy AS mb4
132         ON mb4.\"TrackElement_id\"
133            = ct4.\"TrackElement1_id\"
134     AND mb1.\"Sensor_id\"
135        = mb4.\"Sensor_id\"
136     INNER JOIN monitoredBy AS mb5
137         ON mb5.\"TrackElement_id\"
138            = ct5.\"TrackElement1_id\"
139     AND mb1.\"Sensor_id\"
140        = mb5.\"Sensor_id\"
141     INNER JOIN monitoredBy AS mb6
142         ON mb6.\"TrackElement_id\"
143            = ct5.\"TrackElement2_id\"
144     AND mb1.\"Sensor_id\" = mb6.\"Sensor_id\"
145 WHERE ct1.\"TrackElement1_id\" IN (SELECT id FROM Segment)
146     AND ct2.\"TrackElement1_id\" IN (SELECT id FROM Segment)
147     AND ct3.\"TrackElement1_id\" IN (SELECT id FROM Segment)
148     AND ct4.\"TrackElement1_id\" IN (SELECT id FROM Segment)
149     AND ct5.\"TrackElement1_id\" IN (SELECT id FROM Segment)
150     AND ct5.\"TrackElement2_id\" IN (SELECT id FROM Segment);");

```

```

151 while (resultset.next()) {
152     count++;
153     Statement statement1 = conn.createStatement();
154     statement1.executeUpdate("DELETE FROM segment WHERE id = " +
155         resultset.getLong("segment2"));
156     Statement statement2 = conn.createStatement();
157     statement2.executeUpdate(
158         "DELETE FROM connectsTo
159         WHERE \"TrackElement2_id\" = " +
160             resultset.getLong("segment2")
161         + " AND \"TrackElement1_id\" = " +
162             resultset.getLong("segment1"));
163     Statement statement3 = conn.createStatement();
164     statement3.executeUpdate(
165         "DELETE FROM connectsTo
166         WHERE \"TrackElement2_id\" = " +
167             resultset.getLong("segment3")
168         + " AND \"TrackElement1_id\" = " +
169             resultset.getLong("segment2"));
170     Statement statement4 = conn.createStatement();
171     statement4.executeUpdate("DELETE FROM monitoredBy
172         WHERE \"TrackElement_id\" = " + resultset.getLong("segment2") +
173         " AND \"Sensor_id\" = " + resultset.getLong("sensor"));
174     Statement statement5 = conn.createStatement();
175     statement5.executeUpdate(
176         "INSERT INTO connectsTo(
177             \"TrackElement1_id\",
178             \"TrackElement2_id\")
179         VALUES(
180             + resultset.getLong("segment1") + ",
181             + resultset.getLong("segment3") + ")");
182 }
183 // SemaphoreNeighbor
184 ResultSet resultset = statement.executeQuery(
185     "SELECT Route1.exit AS semaphore,
186         Route1.id AS route1,
187         Route2.id AS route2,
188         requires1.\"Sensor_id\" AS sensor1,
189         requires2.\"Sensor_id\" AS sensor2,
190         ct.\"TrackElement1_id\" AS te1,
191         ct.\"TrackElement2_id\" AS te2
192     FROM Route AS Route1
193         INNER JOIN requires AS requires1
194             ON Route1.id = requires1.\"Route_id\"
195         INNER JOIN monitoredBy AS mb1
196             ON requires1.\"Sensor_id\" = mb1.\"Sensor_id\"
197         INNER JOIN connectsTo AS ct
198             ON mb1.\"TrackElement_id\" = ct.\"TrackElement1_id\"
199         INNER JOIN monitoredBy AS mb2

```

```

200         ON ct.\"TrackElement2_id\" = mb2.\"TrackElement_id\"
201     INNER JOIN requires AS requires2
202         ON mb2.\"Sensor_id\" = requires2.\"Sensor_id\"
203     INNER JOIN Route AS Route2
204         ON requires2.\"Route_id\" = Route2.id
205         AND Route1.id != Route2.id
206         AND Route1.exit IS NOT NULL
207         AND (Route2.entry IS NULL
208             OR Route2.entry != Route1.exit);
209 Statement statement1 = conn.createStatement();
210 long count = 0;
211 while (resultset.next()) {
212     statement1.executeUpdate(
213         \"UPDATE route SET entry = \"
214         + resultset.getLong(\"semaphore\")
215         + \" WHERE id = \" + resultset.getLong(\"route2\"));
216     count++;
217 }

```

Listing A.13: Memgraph queries with effect for TrainBenchmark case study

```

1 //SwitchMonitored
2 StatementResult result = session.run(
3     \"MATCH (sw:Switch), (n)
4         OPTIONAL MATCH (sw)-[m:monitoredBy]->()
5         WITH sw, max(n.id) AS maxid, m
6         WHERE m IS NULL
7         CREATE (sw)-[:monitoredBy]->(:Sensor{id: maxid +1})\" );
8 result.consume();
9 //RouteSensor
10 StatementResult result = session.run(
11     \"MATCH (route:Route)-[:follows]->(swP:SwitchPosition)
12         -[:target]->(sw:Switch)-[:monitoredBy]->(sensor:Sensor)
13         OPTIONAL MATCH (route)-[r:requires]->(sensor)
14         WITH sensor, r, route, sw, swP
15         WHERE r IS NULL
16         CREATE (route)-[:requires]->(sensor);\" );
17 result.consume();
18 //SemaphoreNeighbor
19 StatementResult result = session.run(
20     \"MATCH (semaphore)<-[:exit]-(route1)-[:requires]->(sensor1)
21     MATCH (sensor1)<-[:monitoredBy]-(te1)-[:connectsTo]
22         ->(te2)-[:monitoredBy]->(sensor2)
23         <-[:requires]-(route2)
24     WHERE route1 <> route2
25     OPTIONAL MATCH (semaphore)<-[:entry]-(route2)
26     WITH semaphore, route1, route2,
27         sensor1, sensor2, te1, te2,e

```

```

28 WHERE e IS NULL
29 CREATE (semaphore) <-[:entry]-(route2)");
30 result.consume();

```

#### A.1.4 TrainBenchmark queries without effect

Listing A.14: Gremlin queries without effect for TrainBenchmark case study

```

1 // PosLength
2 List<Vertex> matches =
3     g.V().hasLabel("Segment")
4         .has("length", P.lte(0)).as("segment").toList();
5 // SwitchMonitored
6 List<Vertex> matches =
7     g.V().hasLabel("Switch")
8         .not(__.outE("monitoredBy")).as("sw").toList();
9 // RouteSensor
10 List<Map<String, Object>> matches =
11     g.V().hasLabel("Route").as("route")
12         .out("follows").hasLabel("SwitchPosition").as("swP")
13         .out("target").hasLabel("Switch").as("sw")
14         .out("monitoredBy").hasLabel("Sensor").as("sensor")
15         .not(__.inE("requires").outV().as("route2"))
16         .where("route2", P.eq("route"))
17         .select("route", "sensor").toList();
18 // SwitchSet
19 List<Map<String, Object>> matches =
20     g.V().hasLabel("Route")
21         .has("active", true).as("route")
22         .out("entry").hasLabel("Semaphore").as("semaphore")
23         .select("route")
24         .out("follows").hasLabel("SwitchPosition").as("swP")
25         .values("position").as("position").select("swP")
26         .out("target").hasLabel("Switch").as("sw")
27         .values("currentPosition").as("currentPosition")
28         .where("position", P.neq("currentPosition"))
29         .select("semaphore", "route",
30             "swP", "sw", "currentPosition", "position").toList();
31 // ConnectedSegments
32 List<Map<String, Object>> matches =
33     g.V().hasLabel("Sensor").as("sensor")
34         .in("monitoredBy").hasLabel("Segment").as("segment1")
35         .out("connectsTo").hasLabel("Segment").as("segment2")
36         .out("connectsTo").hasLabel("Segment").as("segment3")
37         .out("connectsTo").hasLabel("Segment").as("segment4")
38         .out("connectsTo").hasLabel("Segment").as("segment5")

```

```

39 .out("connectsTo").hasLabel("Segment").as("segment6")
40 .out("monitoredBy").as("sensor2")
41 .where("sensor", P.eq("sensor2"))
42 .select("sensor", "segment1", "segment2",
43         "segment3", "segment4", "segment5", "segment6").toList();
44 // SemaphoreNeighbor
45 List<Map<String, Object>> matches =
46     g.V().hasLabel("Route").as("route1")
47         .out("exit").hasLabel("Semaphore").as("semaphore")
48         .select("route1")
49         .out("requires").hasLabel("Sensor").as("sensor1")
50         .in("monitoredBy").as("te1")
51         .out("connectsTo").as("te2")
52         .out("monitoredBy").hasLabel("Sensor").as("sensor2")
53         .in("requires").hasLabel("Route").as("route2")
54         .where("route2", P.neq("route1")).select("route2")
55         .not(__.out("entry").as("semaphore2"))
56         .where("semaphore2", P.eq("semaphore")))
57         .select("semaphore", "route1", "route2",
58             "sensor1", "sensor2", "te1", "te2").toList();

```

Listing A.15: Neo4j queries without effect for TrainBenchmark case study

```

1 // PosLength
2 StatementResult result = session.run(
3     "MATCH (segment:Segment)
4       WHERE segment.length <= 0
5       RETURN segment, segment.length AS length");
6 result.list();
7 // SwitchMonitored
8 StatementResult result = session.run(
9     "MATCH (sw:Switch), (n) WITH sw, MAX(n.id) AS maxid
10    WHERE NOT (sw)-[:monitoredBy]->()
11    RETURN sw");
12 result.list();
13 // RouteSensor
14 StatementResult result = session.run(
15     "MATCH (route:Route)-[:follows]->(swP:SwitchPosition)
16    -[:target]->(sw:Switch)-[:monitoredBy]->(sensor:Sensor)
17    WHERE NOT (route)-[:requires]->(sensor)
18    RETURN route, sensor, sw, swP");
19 result.list();
20 // SwitchSet
21 StatementResult result = session.run(
22     "MATCH (semaphore:Semaphore)<-[:entry]-(route:Route)
23    -[:follows]->(swP:SwitchPosition)-[:target]->(sw:Switch)
24    WHERE semaphore.signal = "GO"
25    AND route.active = true

```

```

26         AND sw.currentPosition <> swP.position
27     RETURN semaphore, route, swP, sw,
28         sw.currentPosition AS currentPosition,
29         swP.position AS position");
30 result.list();
31 // ConnectedSegments
32 StatementResult result = session.run(
33     "MATCH
34         (sensor:Sensor) <-[:monitoredBy]-(segment1:Segment),
35         (segment1:Segment) -[:connectsTo]->(segment2:Segment)
36         -[:connectsTo]->(segment3:Segment) -[:connectsTo]
37         ->(segment4:Segment) -[:connectsTo]->
38         (segment5:Segment) -[:connectsTo]->
39         (segment6:Segment),
40         (segment2:Segment) -[:monitoredBy]->(sensor:Sensor),
41         (segment3:Segment) -[:monitoredBy]->(sensor:Sensor),
42         (segment4:Segment) -[:monitoredBy]->(sensor:Sensor),
43         (segment5:Segment) -[:monitoredBy]->(sensor:Sensor),
44         (segment6:Segment) -[:monitoredBy]->(sensor:Sensor)
45     RETURN segment1, segment2, segment3,
46         segment4, segment5, segment6");
47 result.list();
48 // SemaphoreNeighbor
49 StatementResult result = session.run(
50     "MATCH (semaphore:Semaphore) <-[:exit]-(route1:Route)
51         -[:requires]->(sensor1:Sensor),
52         (sensor1) <-[:monitoredBy]-(te1) -[:connectsTo]
53         ->(te2) -[:monitoredBy]->(sensor2:Sensor)
54         <-[:requires]-(route2:Route)
55     WHERE NOT (semaphore) <-[:entry]-(route2)
56     AND route1 <> route2
57     RETURN semaphore, route1, route2,
58         te1, te2, sensor1, sensor2");
59 result.list();

```

Listing A.16: CrateDB queries without effect for TrainBenchmark case study

```

1 // PosLength
2 ResultSet resultset =
3     statement.executeQuery(
4         "SELECT id AS segment, length AS length
5         FROM Segment
6         WHERE length <= 0;");
7 // SwitchMonitored
8 ResultSet resultset = statement.executeQuery(
9     "SELECT switch.id AS sw
10    FROM switch\n"
11    LEFT JOIN monitoredBy

```

```

12         ON monitoredBy.\"TrackElement_id\" = switch.id
13     WHERE monitoredBy.\"TrackElement_id\" IS NULL;");
14 // RouteSensor
15 int result = statement1.executeQuery(
16     "SELECT Route.id, Sensor.id
17     FROM Sensor
18         INNER JOIN monitoredBy
19             ON monitoredBy.\"Sensor_id\" = Sensor.id
20         INNER JOIN Switch
21             ON Switch.id = monitoredBy.\"TrackElement_id\"
22         INNER JOIN SwitchPosition
23             ON SwitchPosition.target = Switch.id
24         INNER JOIN Route
25             ON Route.id = SwitchPosition.route
26     LEFT OUTER JOIN requires
27         ON requires.\"Route_id\" = Route.id
28         AND requires.\"Sensor_id\" = Sensor.id
29     WHERE requires.\"Sensor_id\" IS NULL;");
30 // SwitchSet
31 ResultSet resultset = statement.executeQuery(
32     "SELECT Semaphore.id AS semaphore,
33         Route.id AS route,
34         SwitchPosition.id AS swP,
35         Switch.id AS sw,
36         SwitchPosition.position AS position,
37         Switch.currentPosition AS currentPosition
38     FROM route
39         INNER JOIN SwitchPosition
40             ON Route.id = SwitchPosition.route
41         INNER JOIN Switch
42             ON SwitchPosition.target = Switch.id
43         INNER JOIN Semaphore
44             ON Route.entry = Semaphore.id
45     WHERE Route.active = 0
46         AND Switch.currentPosition != SwitchPosition.position
47         AND Semaphore.signal = 'GO'; ");
48 // ConnectedSegments
49 ResultSet resultset = statement.executeQuery(
50     "SELECT mb1.\"Sensor_id\" AS sensor,
51         ct1.\"TrackElement1_id\" AS segment1,
52         ct2.\"TrackElement1_id\" AS segment2,
53         ct3.\"TrackElement1_id\" AS segment3,
54         ct4.\"TrackElement1_id\" AS segment4,
55         ct5.\"TrackElement1_id\" AS segment5,
56         ct5.\"TrackElement2_id\" AS segment6
57     FROM connectsTo AS ct1
58         INNER JOIN connectsTo AS ct2
59             ON ct1.\"TrackElement2_id\"
60             = ct2.\"TrackElement1_id\"

```

```

61     INNER JOIN connectsTo AS ct3
62         ON ct2.\"TrackElement2_id\"
63            = ct3.\"TrackElement1_id\"
64     INNER JOIN connectsTo AS ct4
65         ON ct3.\"TrackElement2_id\"
66            = ct4.\"TrackElement1_id\"
67     INNER JOIN connectsTo AS ct5
68         ON ct4.\"TrackElement2_id\"
69            = ct5.\"TrackElement1_id\"
70     INNER JOIN monitoredBy AS mb1
71         ON mb1.\"TrackElement_id\"
72            = ct1.\"TrackElement1_id\"
73     INNER JOIN monitoredBy AS mb2
74         ON mb2.\"TrackElement_id\"
75            = ct2.\"TrackElement1_id\"
76     AND mb1.\"Sensor_id\"
77        = mb2.\"Sensor_id\"
78     INNER JOIN monitoredBy AS mb3
79         ON mb3.\"TrackElement_id\"
80            = ct3.\"TrackElement1_id\"
81     AND mb1.\"Sensor_id\"
82        = mb3.\"Sensor_id\"
83     INNER JOIN monitoredBy AS mb4
84         ON mb4.\"TrackElement_id\"
85            = ct4.\"TrackElement1_id\"
86     AND mb1.\"Sensor_id\"
87        = mb4.\"Sensor_id\"
88     INNER JOIN monitoredBy AS mb5
89         ON mb5.\"TrackElement_id\"
90            = ct5.\"TrackElement1_id\"
91     AND mb1.\"Sensor_id\"
92        = mb5.\"Sensor_id\"
93     INNER JOIN monitoredBy AS mb6
94         ON mb6.\"TrackElement_id\"
95            = ct5.\"TrackElement2_id\"
96     AND mb1.\"Sensor_id\" = mb6.\"Sensor_id\"
97     WHERE ct1.\"TrackElement1_id\" IN (SELECT id FROM Segment)
98     AND ct2.\"TrackElement1_id\" IN (SELECT id FROM Segment)
99     AND ct3.\"TrackElement1_id\" IN (SELECT id FROM Segment)
100    AND ct4.\"TrackElement1_id\" IN (SELECT id FROM Segment)
101    AND ct5.\"TrackElement1_id\" IN (SELECT id FROM Segment)
102    AND ct5.\"TrackElement2_id\" IN (SELECT id FROM Segment);");
103 // SemaphoreNeighbor
104 ResultSet resultset = statement.executeQuery(
105     "SELECT Route1.exit AS semaphore,
106         Route1.id AS route1,
107         Route2.id AS route2,
108         requires1.\"Sensor_id\" AS sensor1,
109         requires2.\"Sensor_id\" AS sensor2,

```



```

110     ct.\"TrackElement1_id\" AS te1,
111     ct.\"TrackElement2_id\" AS te2
112 FROM Route AS Route1
113     INNER JOIN requires AS requires1
114         ON Route1.id = requires1.\"Route_id\"
115     INNER JOIN monitoredBy AS mb1
116         ON requires1.\"Sensor_id\" = mb1.\"Sensor_id\"
117     INNER JOIN connectsTo AS ct
118         ON mb1.\"TrackElement_id\" = ct.\"TrackElement1_id\"
119     INNER JOIN monitoredBy AS mb2
120         ON ct.\"TrackElement2_id\" = mb2.\"TrackElement_id\"
121     INNER JOIN requires AS requires2
122         ON mb2.\"Sensor_id\" = requires2.\"Sensor_id\"
123     INNER JOIN Route AS Route2
124         ON requires2.\"Route_id\" = Route2.id
125     AND Route1.id != Route2.id
126     AND Route1.exit IS NOT NULL
127     AND (Route2.entry IS NULL
128         OR Route2.entry != Route1.exit);

```

Listing A.17: Memgraph queries without effect for TrainBenchmark case study

```

1  //SwitchMonitored
2  StatementResult result = session.run(
3      "MATCH (sw:Switch), (n)
4          OPTIONAL MATCH (sw)-[m:monitoredBy]->()
5          WITH sw, max(n.id) AS maxid, m
6          WHERE m IS NULL
7          RETURN sw");
8  result.list();
9  //RouteSensor
10 StatementResult result = session.run(
11     "MATCH (route:Route)-[:follows]->(swP:SwitchPosition)
12         -[:target]->(sw:Switch)-[:monitoredBy]->(sensor:Sensor)
13     OPTIONAL MATCH (route)-[r:requires]->(sensor)
14     WITH sensor, r, route, sw, swP
15     WHERE r IS NULL
16     RETURN route, sensor, swP, sw");
17 result.list();
18 //SemaphoreNeighbor
19 StatementResult result = session.run(
20     "MATCH (semaphore)<-[:exit]-(route1)-[:requires]->(sensor1)
21     MATCH (sensor1)<-[:monitoredBy]-(te1)-[:connectsTo]
22         ->(te2)-[:monitoredBy]->(sensor2)
23         <-[:requires]-(route2)
24     WHERE route1 <> route2
25     OPTIONAL MATCH (semaphore)<-[:e:entry]-(route2)
26     WITH semaphore, route1, route2,

```

```

27     sensor1, sensor2, te1, te2,e
28     WHERE e IS NULL
29     RETURN semaphore, route1, route2, sensor1, sensor2, te1, te2");
30 result.list();

```

Listing A.18: GraphFrames queries without effect for TrainBenchmark case study

```

1  // PosLength
2  graph.find("(segment)")
3      .filter("segment.node = 'Segment' AND segment.length <= 0");
4  // SwitchMonitored
5  var monitored = graph.find("(switch1)-[monitoredBy]->()")
6      .filter("switch1.node = 'Switch'
7          AND monitoredBy.relationship = 'monitoredBy'")
8      .groupBy("fu").count.select("fu").where("count >= 50");
9  switchMonitored = monitored.join(
10      switchMonitored,
11      monitored.col("switch1.id")
12          == switchMonitored.col("switch.id"), "outer")
13      .where("switch1 is null")
14      .groupBy("p").count.where("count >= 30").select("p");
15  // RouteSensor
16  graph.find("(route)-[follows]->(swP);
17      (swP)-[target]->(sw); (sw)-[monitoredBy]->(sensor);
18      !(route)-[]->(sensor)")
19      .filter("route.node = 'Route'
20          AND swP.node = 'SwitchPosition'
21          AND follows.relationship = 'follows'
22          AND target.relationship = 'target'
23          AND sw.node = 'Switch'
24          AND monitoredBy.relationship = 'monitoredBy'
25          AND sensor.node = 'Sensor'");
26  // SwitchSet
27  graph.find("(route)-[entry]->(semaphore); (
28      route)-[follows]->(swP);
29      (swP)-[target]->(sw)")
30      .filter("semaphore.node = 'Semaphore'
31          AND semaphore.signal = 'GO'
32          AND entry.relationship = 'entry'
33          AND route.node = 'Route'
34          AND route.active = true
35          AND follows.relationship = 'follows'
36          AND swP.node = 'SwitchPosition'
37          AND sw.currentPosition != swP.position
38          AND target.relationship = 'target'
39          AND sw.node = 'Switch'").distinct
40  // ConnectedSegments
41  graph.find("(segment1)-[monitoredBy]->(sensor);

```

```

42     (segment1)-[connectsTo1]->(segment2);
43     (segment2)-[connectsTo2]->(segment3);
44     (segment3)-[connectsTo3]->(segment4);
45     (segment4)-[connectsTo4]->(segment5);
46     (segment5)-[connectsTo5]->(segment6);
47     (segment2)-[monitoredBy2]->(sensor);
48     (segment3)-[monitoredBy3]->(sensor);
49     (segment4)-[monitoredBy4]->(sensor);
50     (segment5)-[monitoredBy5]->(sensor);
51     (segment6)-[monitoredBy6]->(sensor) ")
52 .filter("sensor.node = 'Sensor'
53 AND monitoredBy.relationship = 'monitoredBy'
54 AND segment1.node = 'Segment'
55 AND connectsTo1.relationship = 'connectsTo'
56 AND segment2.node = 'Segment'
57 AND connectsTo2.relationship = 'connectsTo'
58 AND segment3.node = 'Segment'
59 AND connectsTo3.relationship = 'connectsTo'
60 AND segment4.node = 'Segment'
61 AND connectsTo4.relationship = 'connectsTo'
62 AND segment5.node = 'Segment'
63 AND connectsTo5.relationship = 'connectsTo'
64 AND segment6.node = 'Segment'
65 AND monitoredBy2.relationship = 'monitoredBy'
66 AND monitoredBy3.relationship = 'monitoredBy'
67 AND monitoredBy4.relationship = 'monitoredBy'
68 AND monitoredBy5.relationship = 'monitoredBy'
69 AND monitoredBy6.relationship = 'monitoredBy'");
70 // SemaphoreNeighbor
71 var SemaphoreNeighbor =
72     graph.find("(route1)-[exit]->(semaphore);
73     (route1)-[requires]->(sensor1);
74     (te1)-[monitoredBy1]->(sensor1);
75     (te1)-[connectsTo]->(te2);
76     (te2)-[monitoredBy2]->(sensor2);
77     (route2)-[requires2]->(sensor2) ")
78 .filter("semaphore.node = 'Semaphore'
79 AND exit.relationship = 'exit'
80 AND route1.node = 'Route'
81 AND requires.relationship = 'requires'
82 AND sensor1.node = 'Sensor'
83 AND monitoredBy1.relationship = 'monitoredBy'
84 AND connectsTo.relationship = 'connectsTo'
85 AND monitoredBy2.relationship = 'monitoredBy'
86 AND sensor2.node = 'Sensor'
87 AND requires2.relationship = 'requires'
88 AND route2.node = 'Route'
89 AND route1 != route2 ")
90 var entry =

```

```
91 graph.find("(route22)-[entry]->(semaphore1)")
92   .filter("route22.node = 'Route'
93         AND entry.relationship = 'entry'
94         AND semaphore1.node = 'Semaphore'");
95 SemaphoreNeighbor = SemaphoreNeighbor.join(
96     entry,
97     SemaphoreNeighbor.col("semaphore.id")
98     == entry.col("semaphore1.id")
99     && SemaphoreNeighbor.col("route2.id")
100    == entry.col("route22.id"), "outer")
101 .where("entry is null")
102 .groupBy("p").count.where("count >= 30").select("p")
```

## A.2 Additional charts and tables displaying TrainBenchmark results

To improve the readability of Chapter 3, this appendix contains some of the tables and figures that show the results of the evaluations. Specifically, Figures A.1 and A.2 show the execution time of the experiments without effect over the graph with single and parallel executions of TrainBenchmark example, respectively. Then, Figures A.3 and A.4 show the execution time of the experiments with effect over the graph with single and parallel executions of TrainBenchmark example, respectively. Finally, Table A.1 shows the coefficient of variation for parallel execution and Table A.2 summarizes all syntax features of queries of TrainBenchmark example.

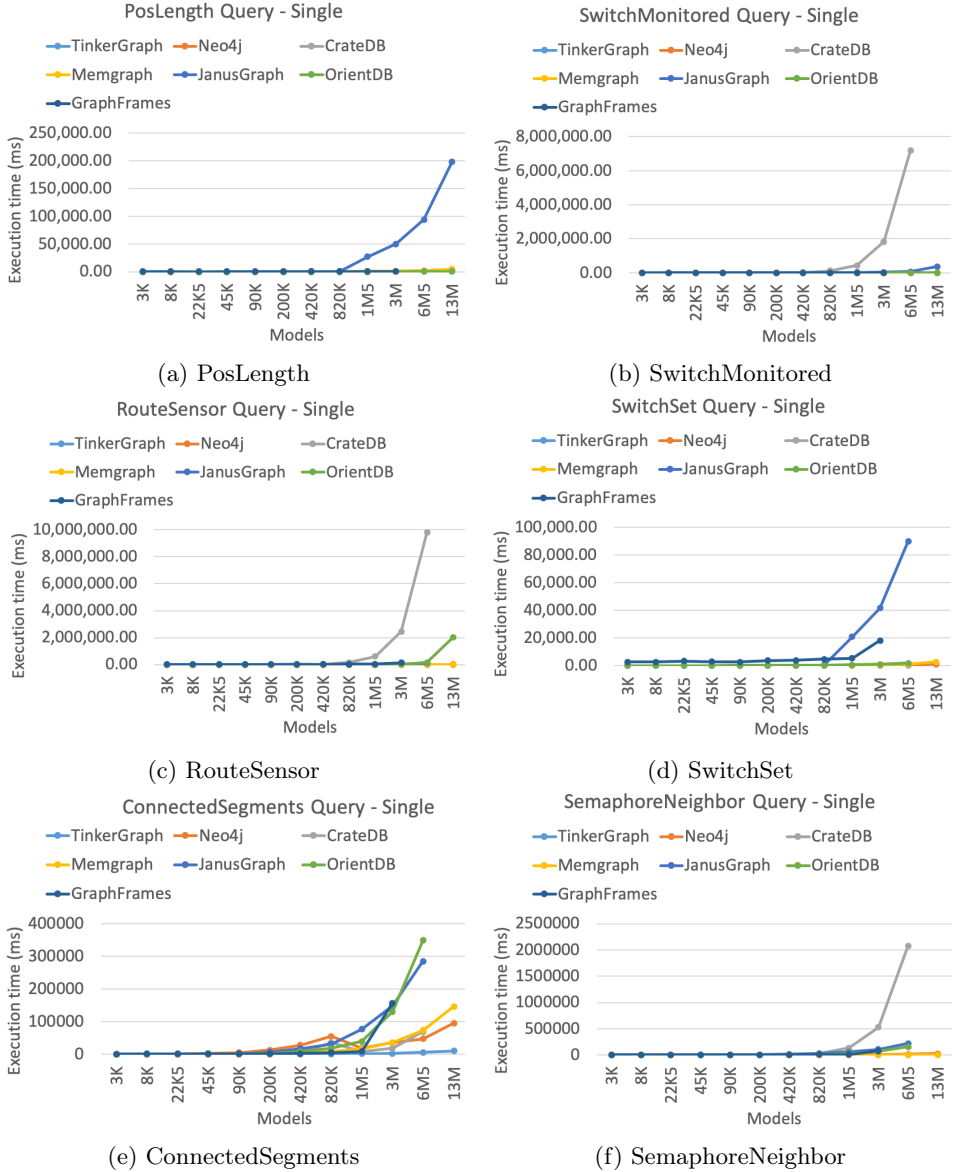


Figure A.1: Execution time results for queries without effect of TrainBenchmark example with single runs

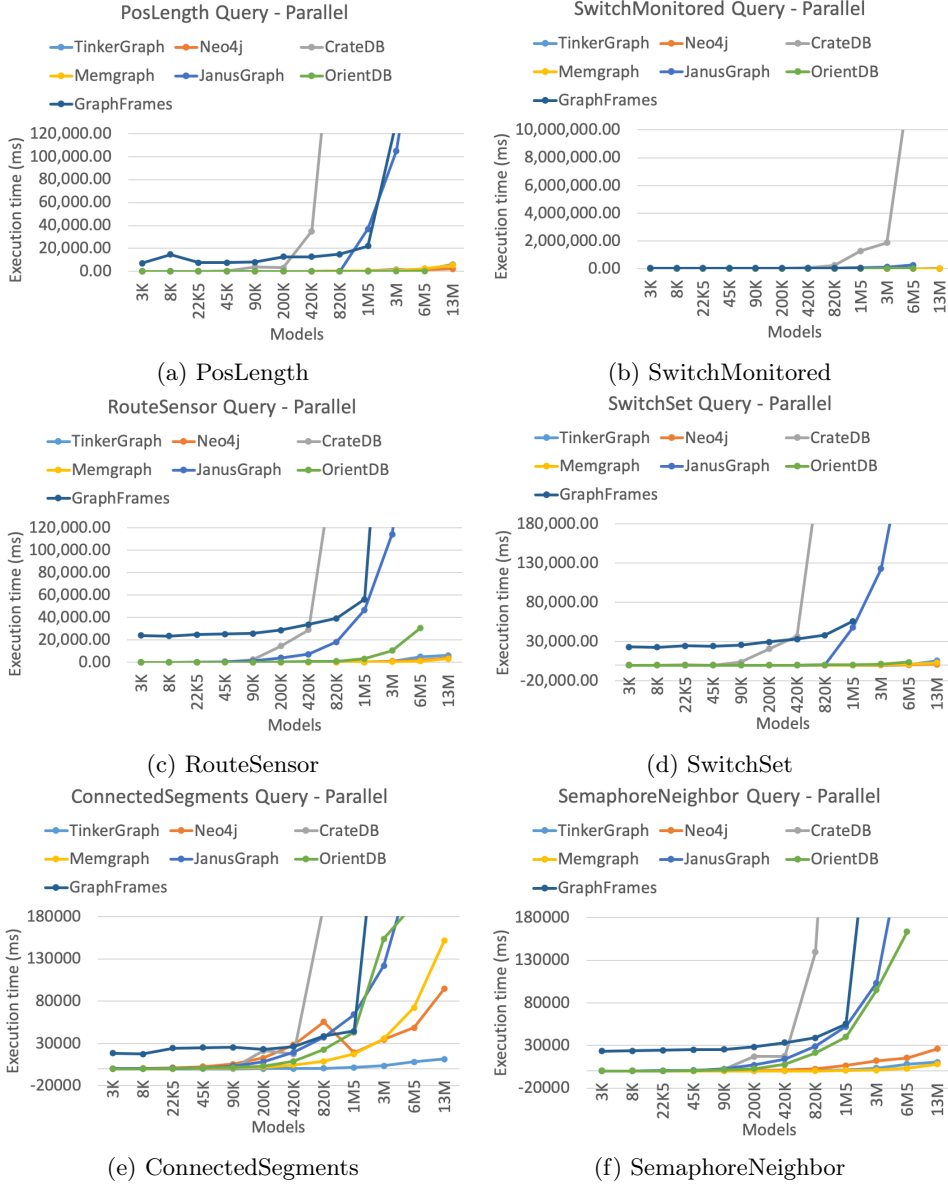


Figure A.2: Execution time results for queries without effect of TrainBenchmark example with parallel runs

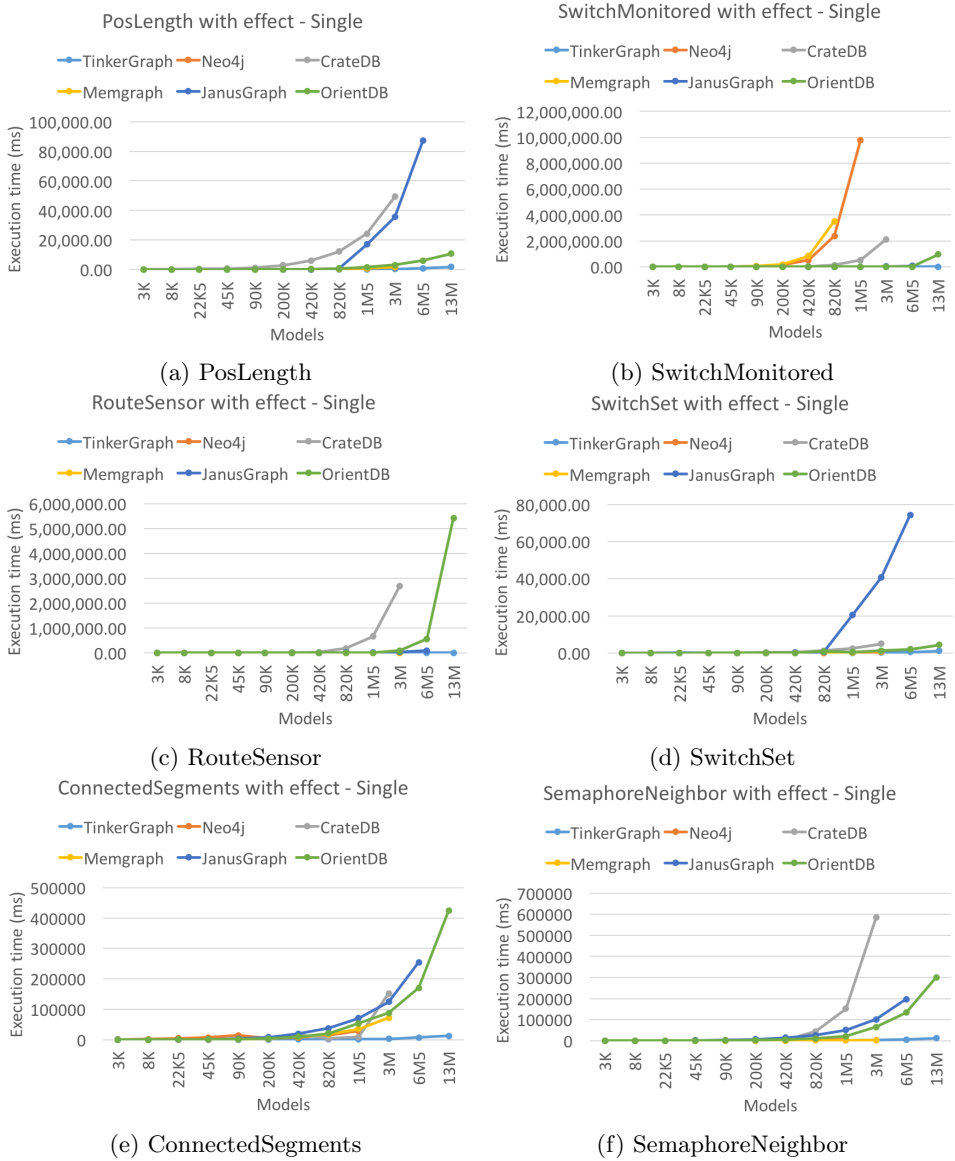


Figure A.3: Execution time results for queries with effect of TrainBenchmark example with single runs



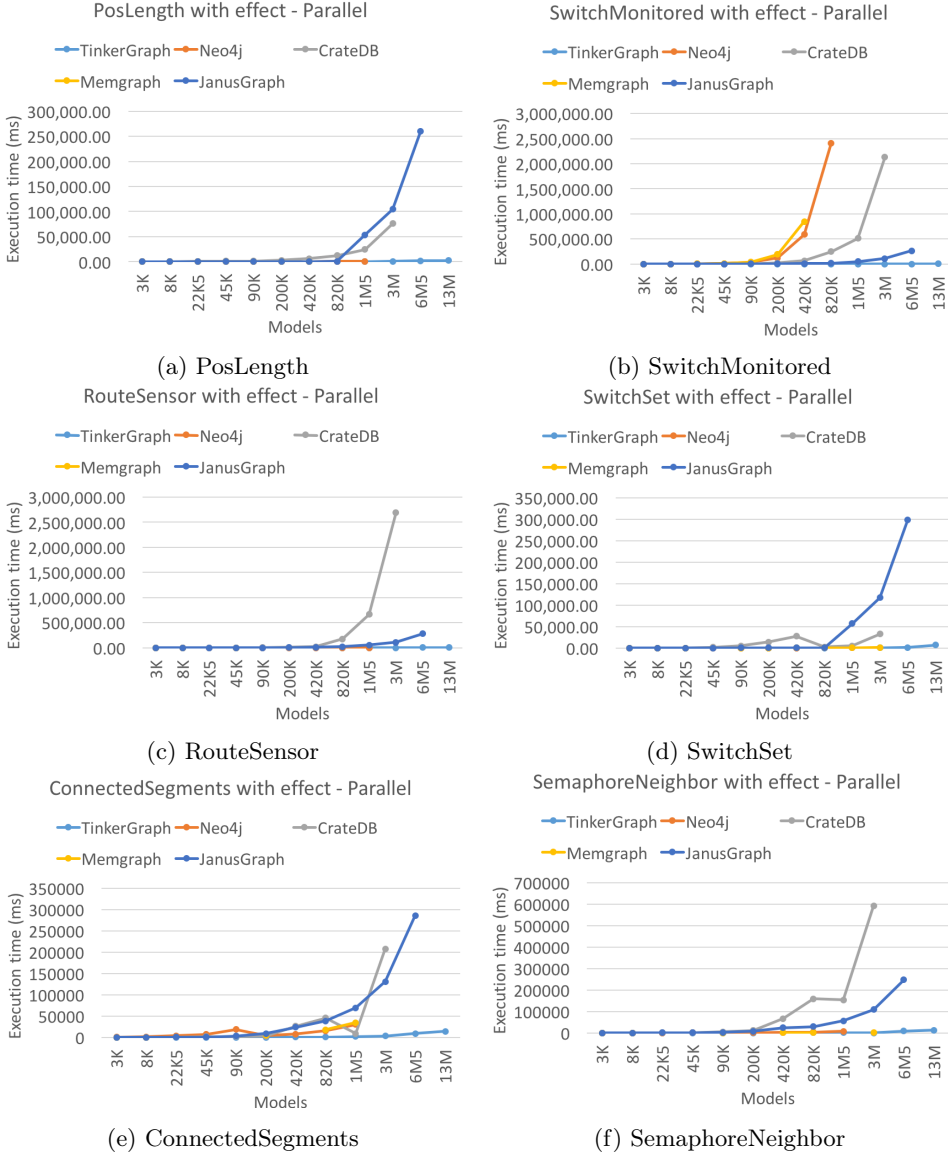


Figure A.4: Execution time results for queries with effect of TrainBenchmark example with parallel runs

## A.2 Additional charts and tables displaying TrainBenchmark results

Query Name	Tech	Models					
		420K	820K	1M5	3M	6M5	13M
PosLength	TinkerGraph	9.90	57.71	36.01	135.44	37.02	116.05
	Neo4j	29.19	17.40	5.12	17.49	13.11	14.83
	JanusGraph	25.10	9.95	13.65	43.29	44.09	-
	OrientDB	14.37	3.98	54.20	11.93	3.54	-
	CrateDB	87.24	94.74	90.96	103.37	96.27	-
	Memgraph	5.17	2.86	1.10	0.85	5.43	14.26
	GraphFrames	82.19	65.51	92.06	79.33	-	-
SwitchMonitored	TinkerGraph	4.33	77.91	89.64	70.61	70.71	87.38
	Neo4j	13.09	5	39.12	8.80	38.97	26.00
	JanusGraph	7.31	11.26	3.80	12.66	4.69	-
	OrientDB	6.96	54.80	1.83	53.88	40.92	-
	CrateDB	13.90	46.52	68.49	2.98	35.82	-
	Memgraph	6.25	9.35	1.24	1.52	4.66	2.41
	GraphFrames	37.88	3.59	6.57	-	-	-
RouteSensor	TinkerGraph	28.96	16.91	4.46	34.72	70.68	17.72
	Neo4j	11.18	11.70	13.71	18.02	10.24	24.07
	JanusGraph	11.63	15.53	18.78	28.22	23.63	-
	OrientDB	52.85	7.90	73.85	38.85	24.49	-
	CrateDB	59.58	30.95	37.02	2.11	44.94	-
	Memgraph	12.22	16.89	3.39	4.61	6.79	11.31
	GraphFrames	7.25	0.42	4.72	0.65	-	-
SwitchSet	TinkerGraph	8.54	98.46	20.82	1.77	61.85	127.90
	Neo4j	35.89	33.04	26.27	1.41	18.81	10.07
	JanusGraph	14.94	8.52	8.07	8.56	6.57	-
	OrientDB	17.93	58.47	4.49	59.14	57.81	-
	CrateDB	73.07	82.57	70.89	29.96	47.48	-
	Memgraph	1.35	16.02	8.83	4.73	4.03	16.21
	GraphFrames	1.25	1.96	10.83	-	-	-
ConnectedSegments	TinkerGraph	12.82	10.02	22.89	19.60	15.67	11.00
	Neo4j	1.05	0.33	1.82	1.89	0.83	2.30
	JanusGraph	2.65	11.05	10.59	10.98	5.36	-
	OrientDB	11.02	7.30	4.65	20.34	9.02	-
	CrateDB	98.65	17.33	45.44	99.57	16.27	-
	Memgraph	1.40	0.20	1.39	0.77	0.50	1.50
	GraphFrames	32.96	2.50	22.35	0.14	-	-
SemaphoreNeighbor	TinkerGraph	2.64	1.16	31.49	22.85	13.19	14.45
	Neo4j	8.89	4.41	7.07	4.09	10.65	4.64
	JanusGraph	0.93	8.89	2.75	7.16	9.76	-
	OrientDB	6.46	18.18		4.30	26.61	-
	CrateDB	125.00	129.94	77.88	1.89	76.65	-
	Memgraph	2.52	2.28	0.70	0.64	20.58	8.96
	GraphFrames	6.88	1.54	5.71	1.05	-	-

Table A.1: Coefficient of variation (%) of TrainBenchmark queries without effect and parallel runs.

Tech	Query	Update			No Update		
		Op	Char	Var	Op	Char	Var
TinkerGraph, JanusGraph and OrientDB (Gremlin)	PosLength	10	167	1	6	92	1
	SwitchMonitored	9	208	1	6	91	1
	RouteSensor	23	424	5	20	321	5
	SwitchSet	26	598	6	22	464	6
	ConnectedSegment	32	714	8	27	545	8
	SemaphoreNeighbor	35	743	8	30	543	8
	<b>AVG</b>	<b>22.5</b>	<b>475.6</b>	<b>4.8</b>	<b>18.5</b>	<b>342.6</b>	<b>4.8</b>
Neo4j (Cypher)	PosLength	9	170	2	5	132	2
	SwitchMonitored	10	177	3	8	134	3
	RouteSensor	9	233	4	8	220	4
	SwitchSet	15	363	6	13	328	6
	ConnectedSegment	13	574	7	13	610	7
	SemaphoreNeighbor	14	340	7	13	354	7
	<b>AVG</b>	<b>11.6</b>	<b>309.5</b>	<b>4.8</b>	<b>10</b>	<b>296.3</b>	<b>4.8</b>
CrateDB (SQL)	PosLength	13	313	2	5	104	2
	SwitchMonitored	31	907	11	7	183	1
	RouteSensor	21	459	0	20	411	0
	SwitchSet	27	673	6	22	451	6
	ConnectedSegment	97	2,525	18	77	1,549	18
	SemaphoreNeighbor	46	957	14	41	750	14
	<b>AVG</b>	<b>39.1</b>	<b>972.3</b>	<b>8.5</b>	<b>28.6</b>	<b>574.6</b>	<b>8.5</b>
Memgraph (Cypher)	PosLength	9	170	2	5	132	2
	SwitchMonitored	11	197	4	9	154	3
	RouteSensor	11	277	5	10	262	5
	SwitchSet	15	363	6	13	328	6
	ConnectedSegment	13	574	7	13	610	7
	SemaphoreNeighbor	17	381	8	16	395	8
	<b>AVG</b>	<b>12.6</b>	<b>327</b>	<b>5.3</b>	<b>11</b>	<b>313.5</b>	<b>5.3</b>
GraphFrames	PosLength	-	-	-	5	77	1
	SwitchMonitored	-	-	-	22	372	4
	RouteSensor	-	-	-	20	315	7
	SwitchSet	-	-	-	24	374	7
	ConnectedSegment	-	-	-	48	1,052	18
	SemaphoreNeighbor	-	-	-	54	983	17
	<b>AVG</b>	-	-	-	<b>28.8</b>	<b>528.8</b>	<b>9</b>

Table A.2: Summary of DSL features for TrainBenchmark case study

# Appendix B

## Results for Online AQP techniques

---

In this appendix we show all results for the experiments exposed in Chapter 4. Recall that these experiments consist on running the queries of the case study presented in Section 4.1 over different model sizes and distributions (Batch A and B). The execution times and accuracy results of three online AQP techniques (Temporal, Spatial and Random approximations) are depicted in the charts for each query run. Besides, all queries were implemented with TinkerGraph technology and Gremlin language. All charts are presented in the following.

### B.1 Results for Batch A

This section presents all results of the experiments performed with Batch A of models. Note that orders contained in these models are uniformly distributed along the models.

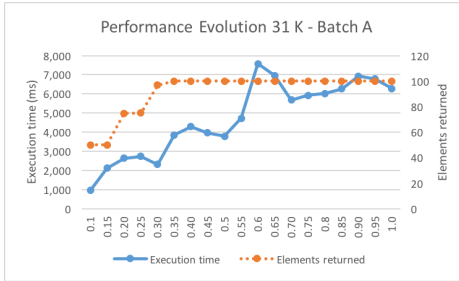
First, let us focus on charts presented in Sections B.1.1 and B.1.2. The former shows Recall value in order to measure the accuracy of the random approximations whereas the latter shows the Precision value for this purpose. This is because the

results of Q1 return more elements as higher is the probability for each element of the graph to be selected in the approximation, while Q2 returns less elements as higher is this probability. However, Precision and Recall values increase with this probability in both cases.

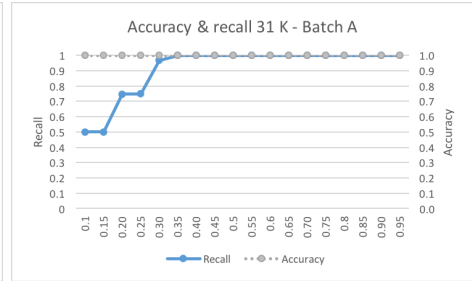
Second, let us observe charts presented in Sections B.1.3 and B.1.4, where random and temporal approximations are shown for query Q3, respectively. For all model sizes, the execution times are lower in the random approximations when the number of elements returned stabilizes with respect to when the number of elements stabilizes in the temporal approximations. This is because orders contained in the models of Batch A are uniformly distributed along the month and temporal approximations need to consider almost the entire month to stabilize the number of elements returned (cf. Section 4.3.3).

Regarding spatial approximations, charts presented in Sections B.1.5 and B.1.6 show how random approximations always present lower execution times than spatial approximations for Q4. However, observe charts for spatial approximation with Q5 depicted in Section B.1.7. In this case, we see how a linear increase in the number of hops imply an exponential increase in the execution times, as previously exposed in Section 4.3.3.

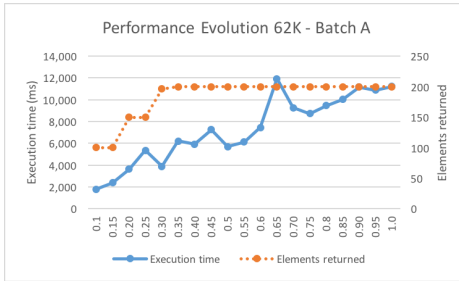
## B.1.1 Q1 - Random approximation



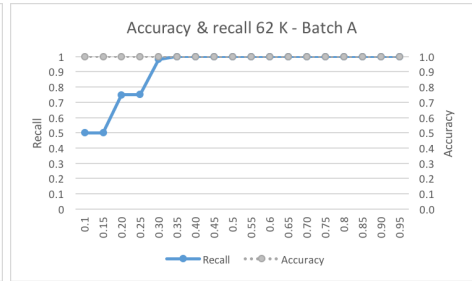
(a) Performance Evolution for 31K.



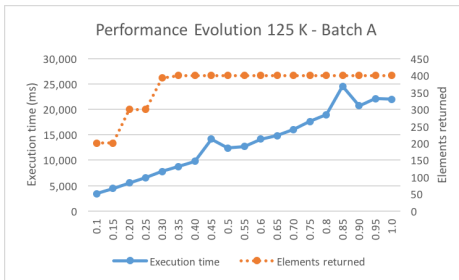
(b) Accuracy and Precision for 31K.



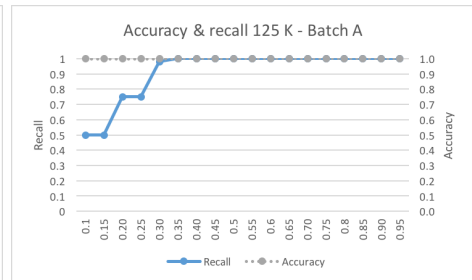
(c) Performance Evolution for 62K.



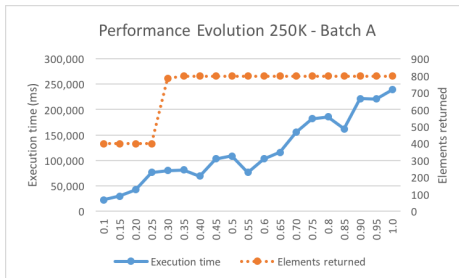
(d) Accuracy and Precision for 62K.



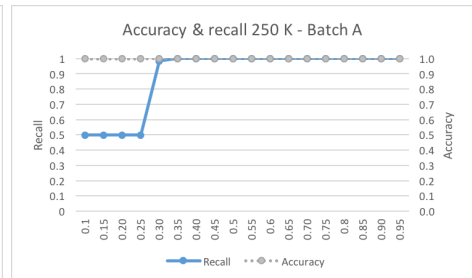
(e) Performance Evolution for 125K.



(f) Accuracy and Precision for 125K.



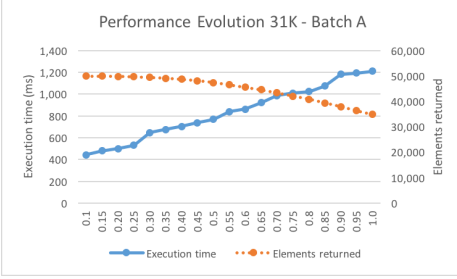
(g) Performance Evolution for 250K.



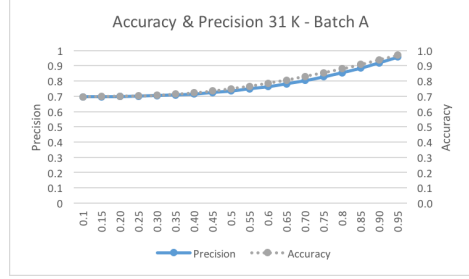
(h) Accuracy and Precision for 250K.

Figure B.1: Q1 Batch A. Accuracy and Precision with Random Approximations.

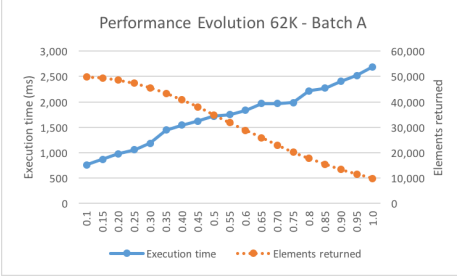
## B.1.2 Q2 - Random approximation



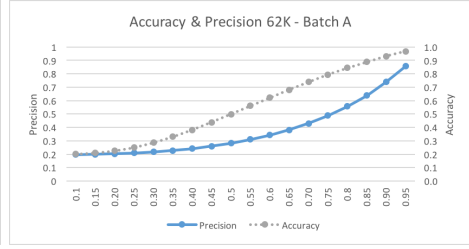
(a) Performance Evolution for 31K.



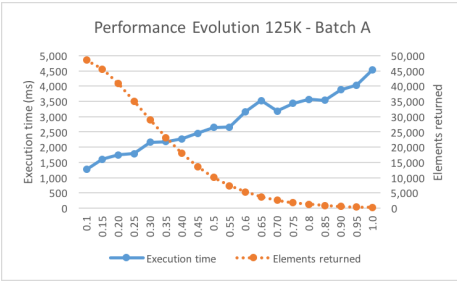
(b) Accuracy and Precision for 31K.



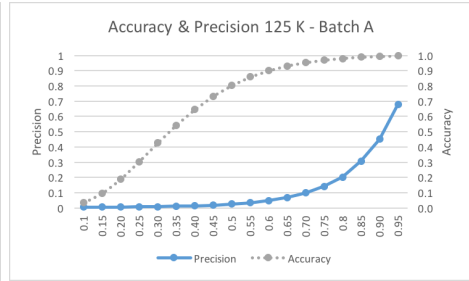
(c) Performance Evolution for 62K.



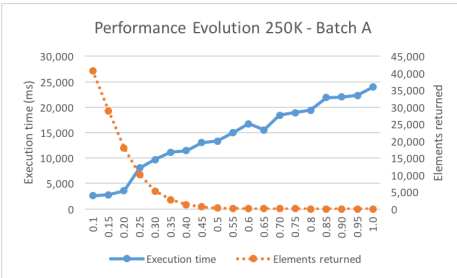
(d) Accuracy and Precision for 62K.



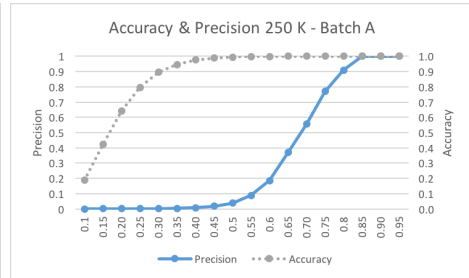
(e) Performance Evolution for 125K.



(f) Accuracy and Precision for 125K.



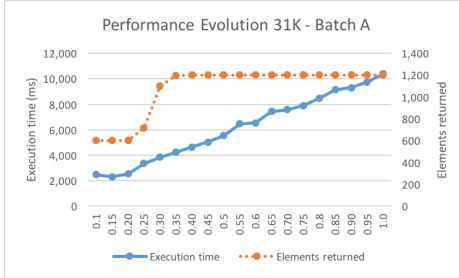
(g) Performance Evolution for 250K.



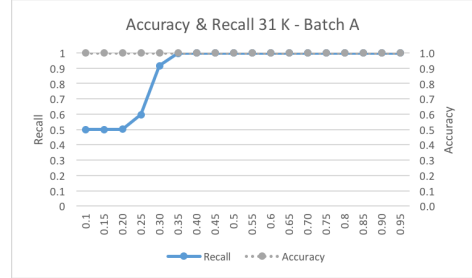
(h) Accuracy and Precision for 250K.

Figure B.2: Q2 Batch A. Accuracy and Precision with Random Approximations.

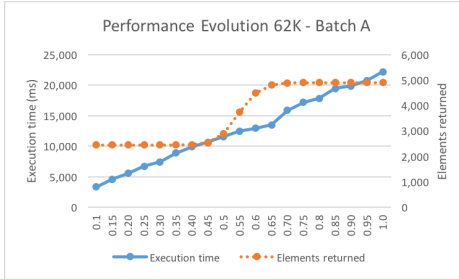
## B.1.3 Q3 - Random approximation



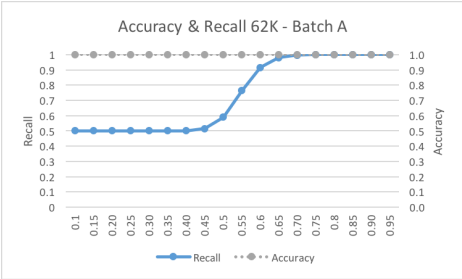
(a) Performance Evolution for 31K.



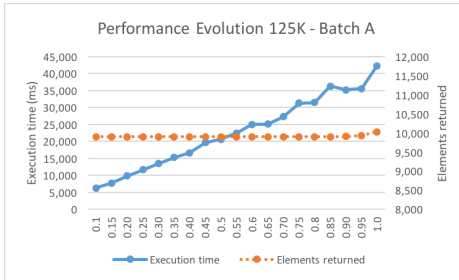
(b) Accuracy and Recall for 31K.



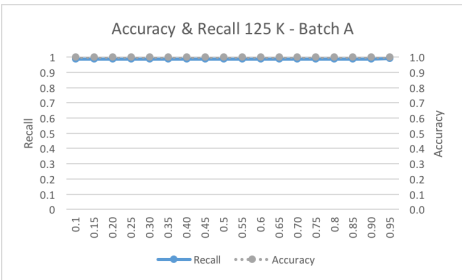
(c) Performance Evolution for 62K.



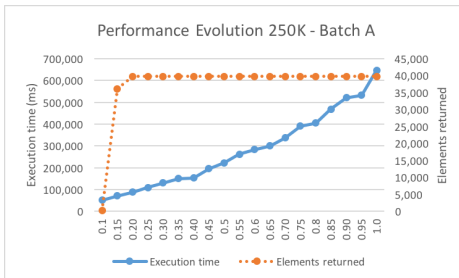
(d) Accuracy and Recall for 62K.



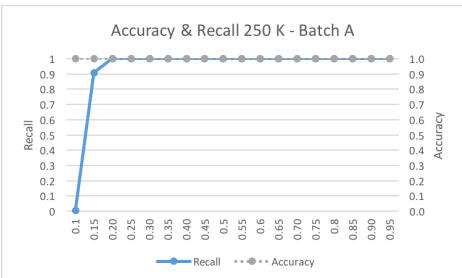
(e) Performance Evolution for 125K.



(f) Accuracy and Recall for 125K.



(g) Performance Evolution for 250K.

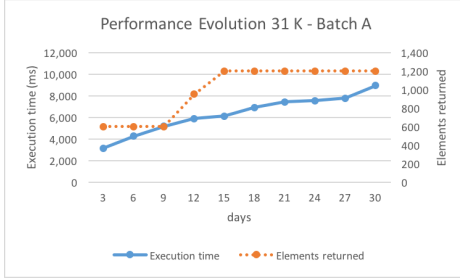


(h) Accuracy and Recall for 250K.

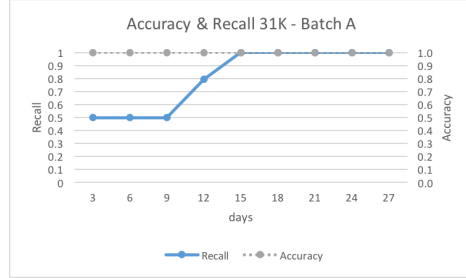
Figure B.3: Q3 Batch A. Accuracy and Recall with Random Approximations.



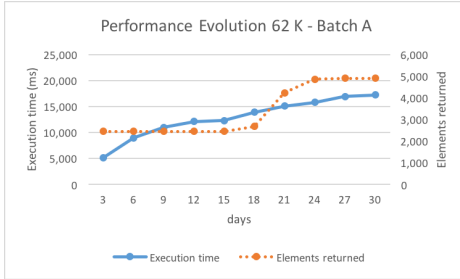
### B.1.4 Q3 - Temporal approximation



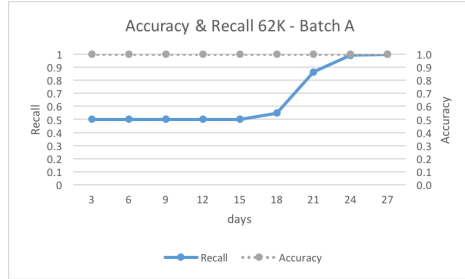
(a) Performance Evolution for 31K.



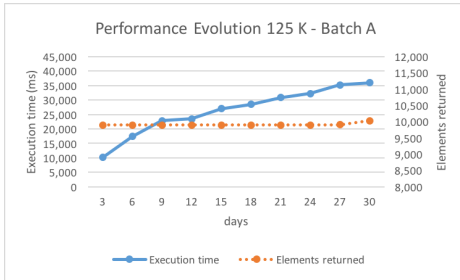
(b) Accuracy and Recall for 31K.



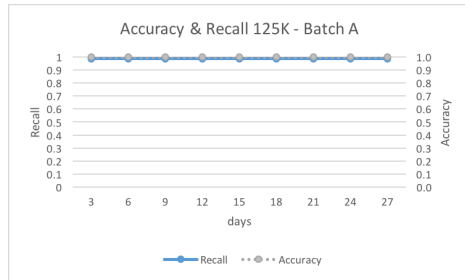
(c) Performance Evolution for 62K.



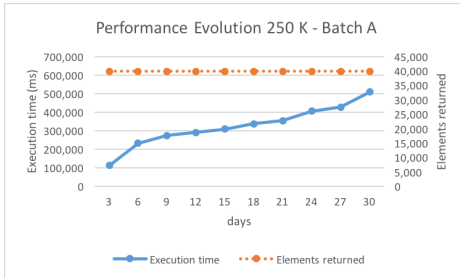
(d) Accuracy and Recall for 62K.



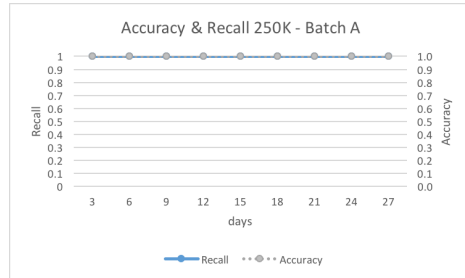
(e) Performance Evolution for 125K.



(f) Accuracy and Recall for 125K.



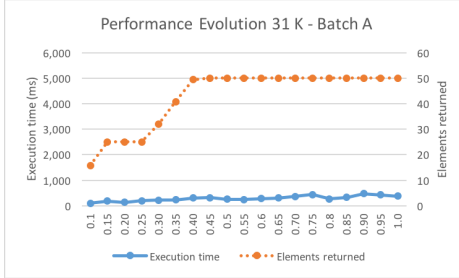
(g) Performance Evolution for 250K.



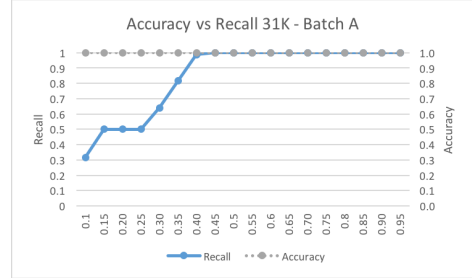
(h) Accuracy and Recall for 250K.

Figure B.4: Q3 Batch A. Accuracy and Recall with Temporal Approximations.

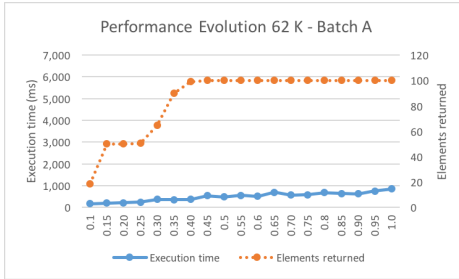
## B.1.5 Q4 - Random approximation



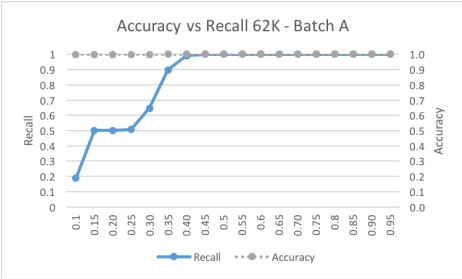
(a) Performance Evolution for 31K.



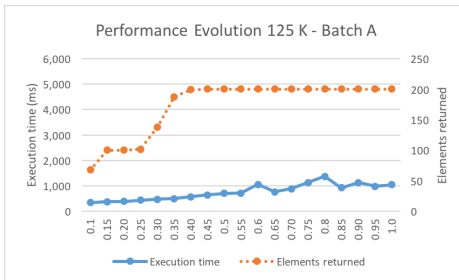
(b) Accuracy and Recall for 31K.



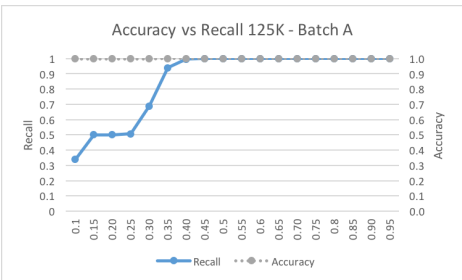
(c) Performance Evolution for 62K.



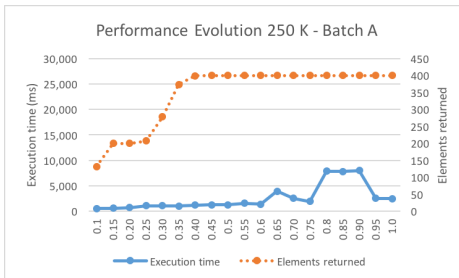
(d) Accuracy and Recall for 62K.



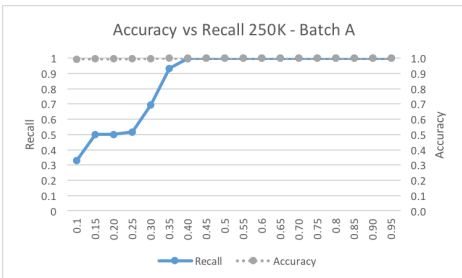
(e) Performance Evolution for 125K.



(f) Accuracy and Recall for 125K.



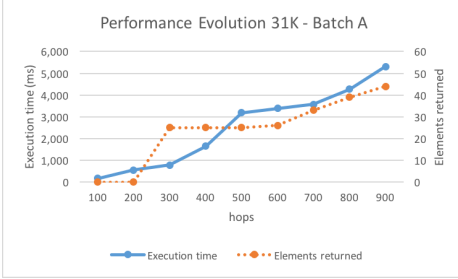
(g) Performance Evolution for 250K.



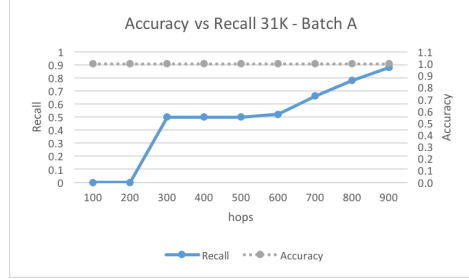
(h) Accuracy and Recall for 250K.

Figure B.5: Q4 Batch A. Accuracy and Recall with Random Approximations.

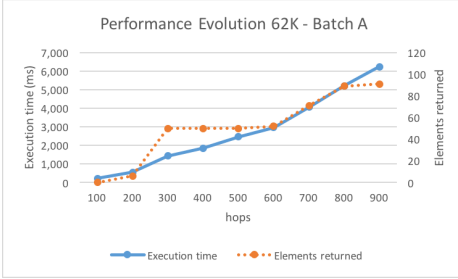
## B.1.6 Q4 - Spatial approximation



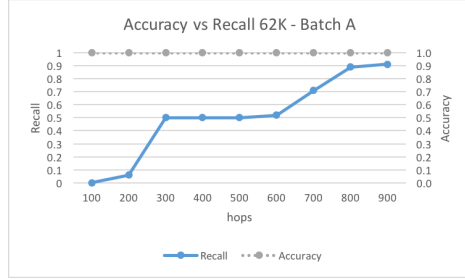
(a) Performance Evolution for 31K.



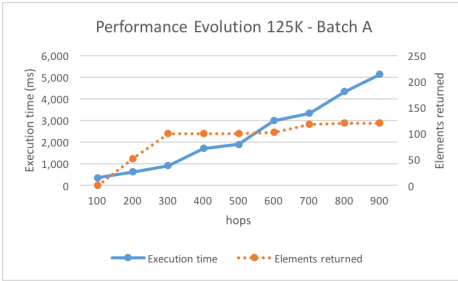
(b) Accuracy and Recall for 31K.



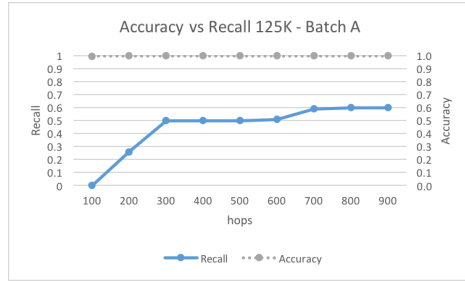
(c) Performance Evolution for 62K.



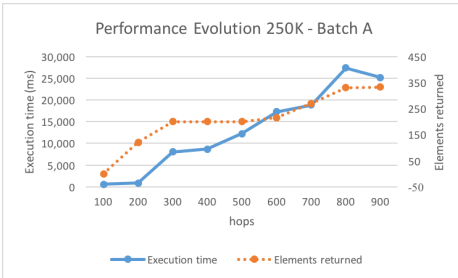
(d) Accuracy and Recall for 62K.



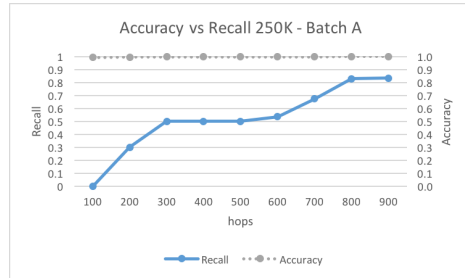
(e) Performance Evolution for 125K.



(f) Accuracy and Recall for 125K.



(g) Performance Evolution for 250K.



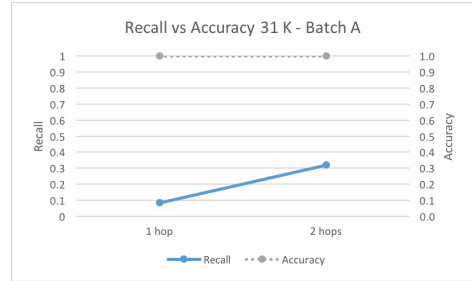
(h) Accuracy and Recall for 250K.

Figure B.6: Q4 Batch A. Accuracy and Recall with Spatial Approximations.

## B.1.7 Q5 - Spatial approximation



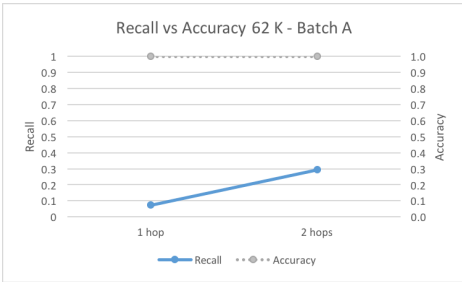
(a) Performance Evolution for 31K.



(b) Accuracy and Precision for 31K.



(c) Performance Evolution for 62K.



(d) Accuracy and Precision for 62K.

Figure B.7: Q5 Batch A. Accuracy and Precision with Spatial Approximations.

## B.2 Results for Batch B

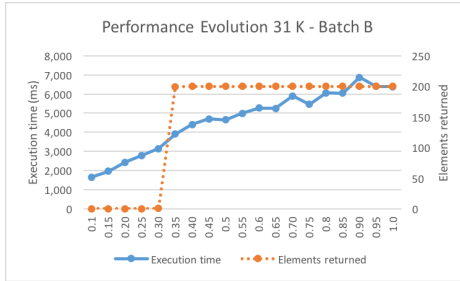
This section presents all results of the experiments performed with Batch B of models. Recall that orders contained in these models are mainly focused on the first week.

First, let us focus on charts presented in Sections B.2.1 and B.2.2. Note how the execution times and accuracy results are very similar to the results for Batch A exposed in Sections B.1.1 and B.1.2. This is because random approximations behave in the same way regardless of data distribution.

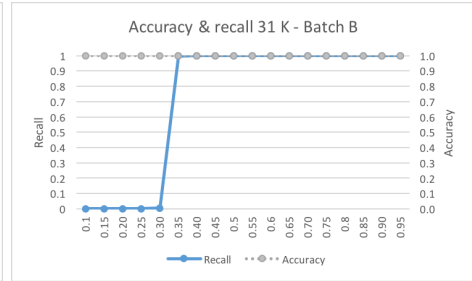
Second, observe charts presented in Sections B.2.3 and B.2.4 for random and temporal approximations with query Q3, respectively. Unlike the results presented in the graphs of Batch A (Section B.1), the execution times are lower in temporal approximations when the number of elements returned stabilizes with respect to when the number of elements stabilizes in random approximations. This is because orders contained in the models of Batch B are focused in the first week of the month. For this reason, the elements returned stabilize when considering only the first days of the month with temporal approximations, that get a value of 100% for the Recall.

Finally, charts presented in Sections B.2.5 and B.2.6 show the results for random and spatial approximations with Q4. In this case, random approximations also present lower execution times than spatial approximations in all cases. In addition, charts for spatial approximation with Q5 depicted in Section B.2.7 show a linear increase in the number of hops that imply an exponential increase in the execution times. All these results are thoroughly exposed in Section 4.3.3.

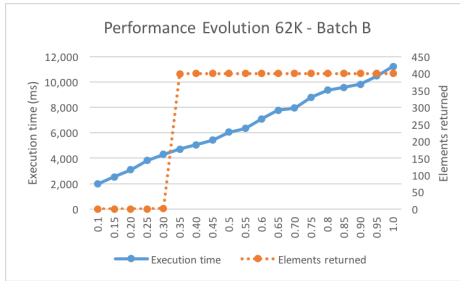
## B.2.1 Q1 - Random approximation



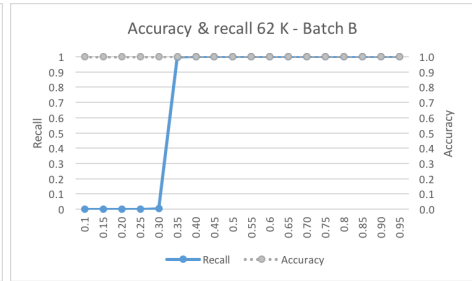
(a) Performance Evolution for 31K.



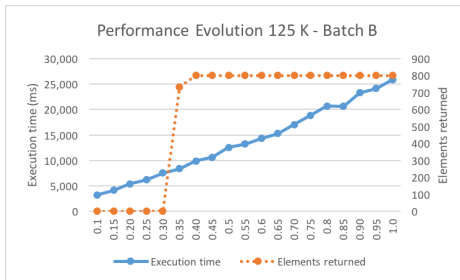
(b) Accuracy and Recall for 31K.



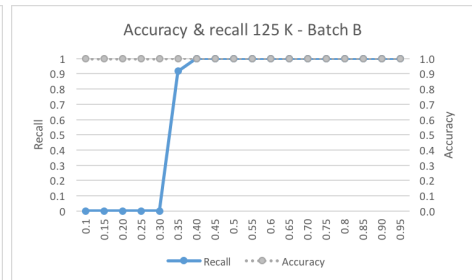
(c) Performance Evolution for 62K.



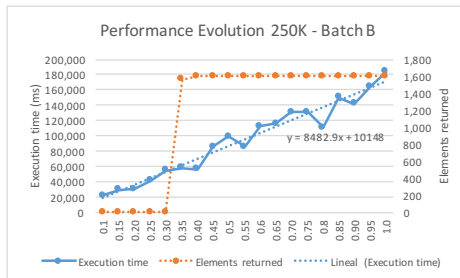
(d) Accuracy and Recall for 62K.



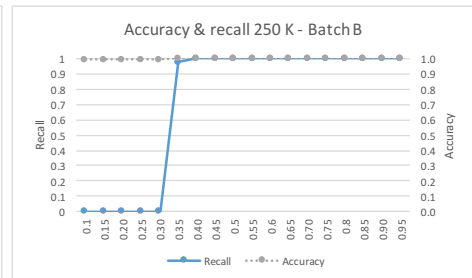
(e) Performance Evolution for 125K.



(f) Accuracy and Recall for 125K.



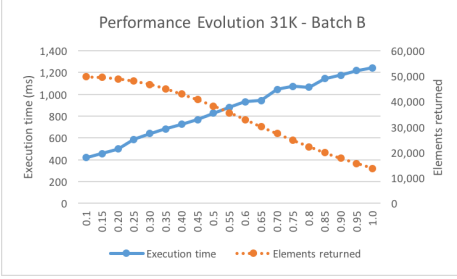
(g) Performance Evolution for 250K.



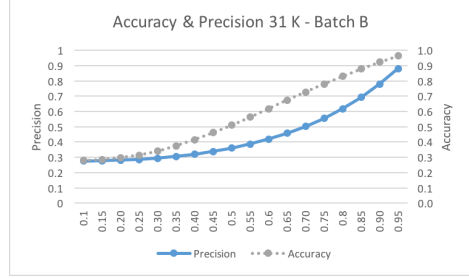
(h) Accuracy and Recall for 250K.

Figure B.8: Q1 Batch B. Accuracy and Recall with Random Approximations.

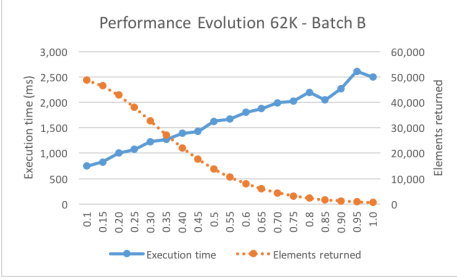
## B.2.2 Q2 - Random approximation



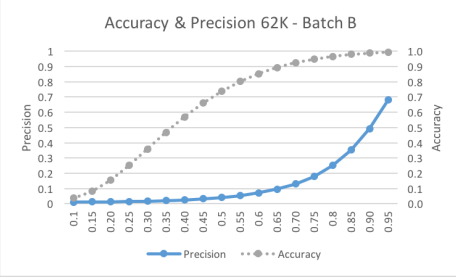
(a) Performance Evolution for 31K.



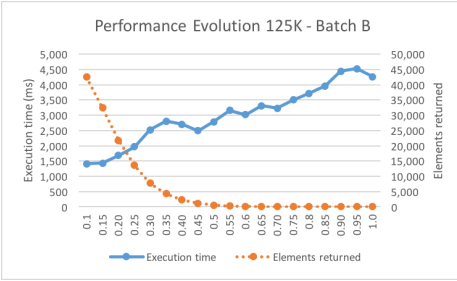
(b) Accuracy and Precision for 31K.



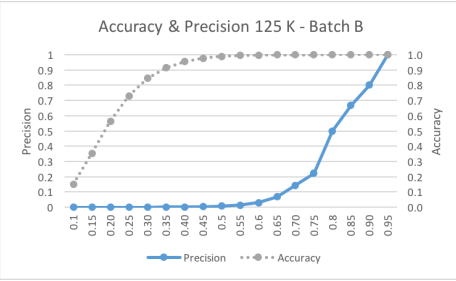
(c) Performance Evolution for 62K.



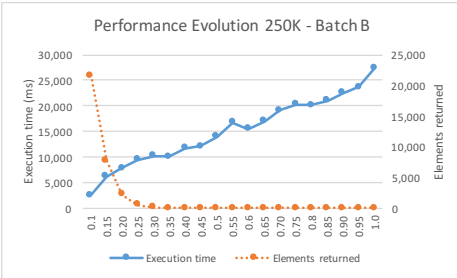
(d) Accuracy and Precision for 62K.



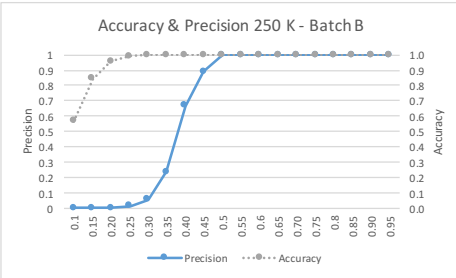
(e) Performance Evolution for 125K.



(f) Accuracy and Precision for 125K.



(g) Performance Evolution for 250K.

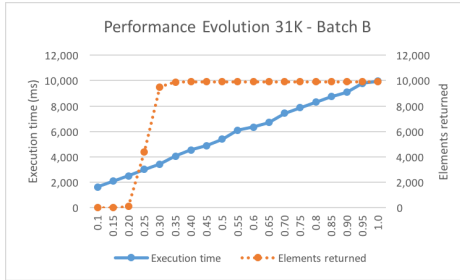


(h) Accuracy and Precision for 250K.

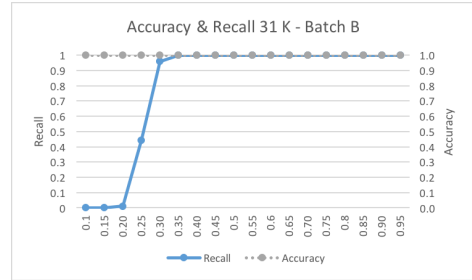
Figure B.9: Q2 Batch B. Accuracy and Precision with Random Approximations.

### B.2.3 Q3 - Random approximation

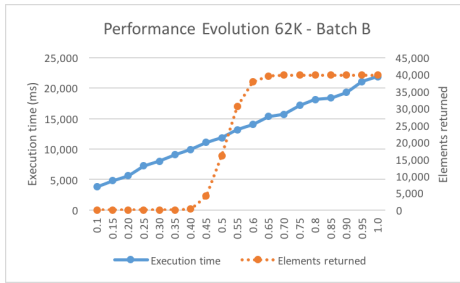
Please note that the number of elements returned for this query with the Pattern Model of 250K is 0. This result is not relevant to get conclusions and therefore it is not shown here.



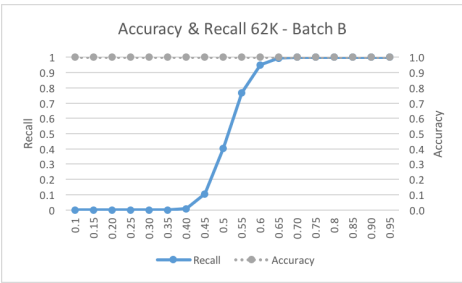
(a) Performance Evolution for 31K.



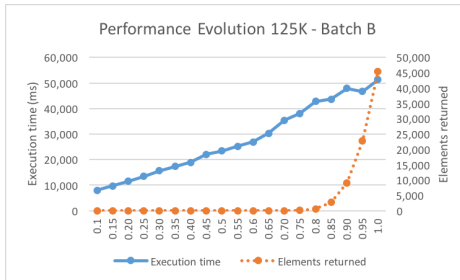
(b) Accuracy and Recall for 31K.



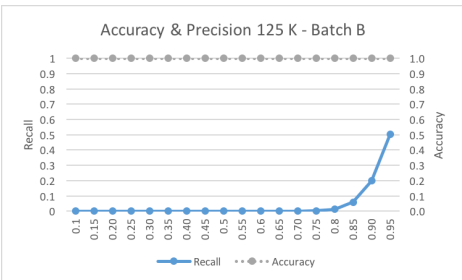
(c) Performance Evolution for 62K.



(d) Accuracy and Recall for 62K.



(e) Performance Evolution for 125K.



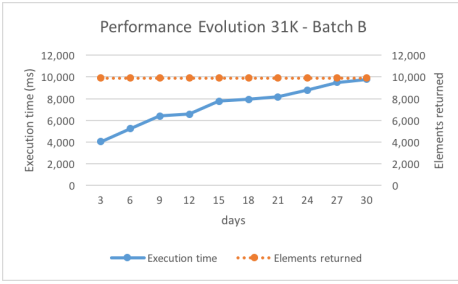
(f) Accuracy and Recall for 125K.

Figure B.10: Q3 Batch B. Accuracy and Recall with Random Approximations.

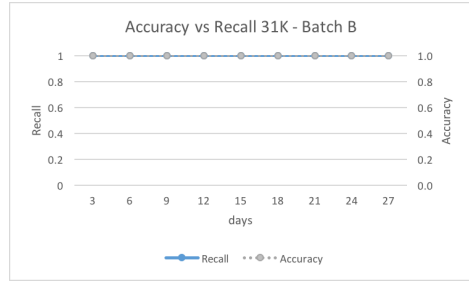
### B.2.4 Q3 - Temporal approximation

Please note that the number of elements returned for this query with the Pattern Model of 250K is 0. This result is not relevant to get conclusions and therefore it is not shown here.

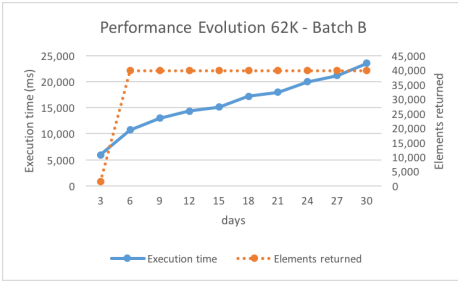




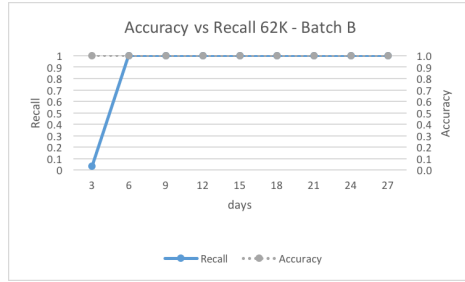
(a) Performance Evolution for 31K.



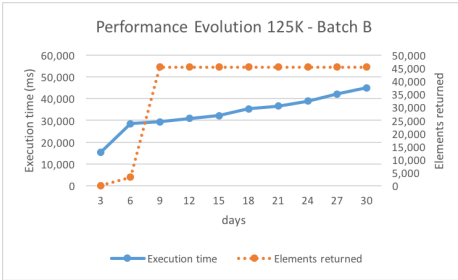
(b) Accuracy and Recall for 31K.



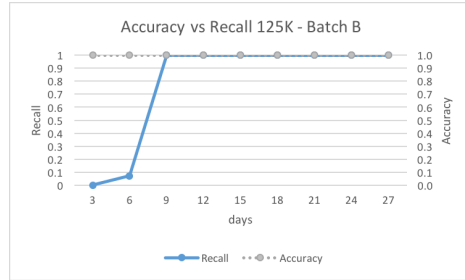
(c) Performance Evolution for 62K.



(d) Accuracy and Recall for 62K.



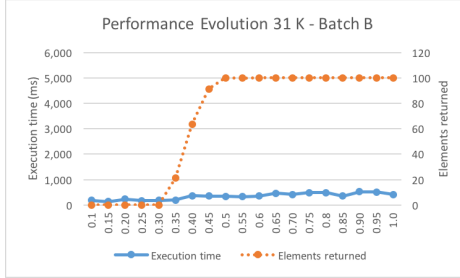
(e) Performance Evolution for 125K.



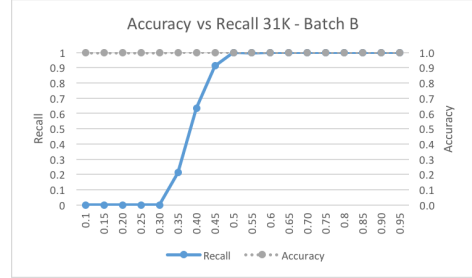
(f) Accuracy and Recall for 125K.

Figure B.11: Q3 Batch B. Accuracy and Recall with Temporal Approximations.

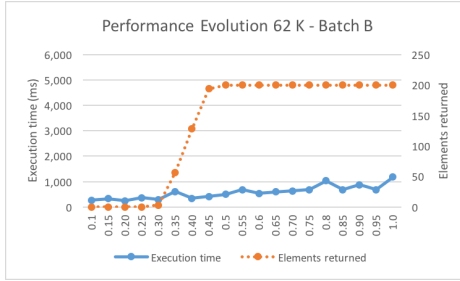
## B.2.5 Q4 - Random approximation



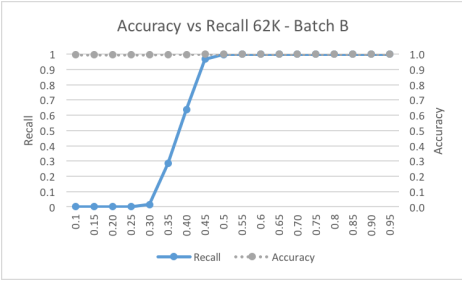
(a) Performance Evolution for 31K.



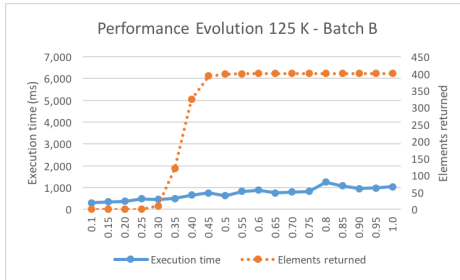
(b) Accuracy and Recall for 31K.



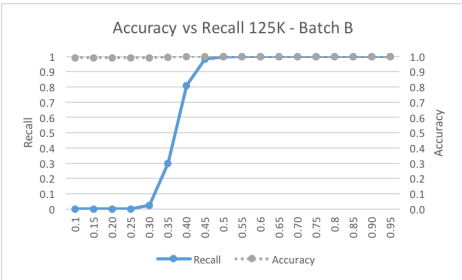
(c) Performance Evolution for 62K.



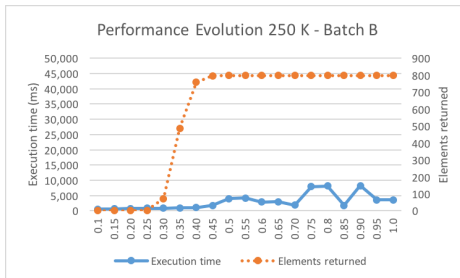
(d) Accuracy and Recall for 62K.



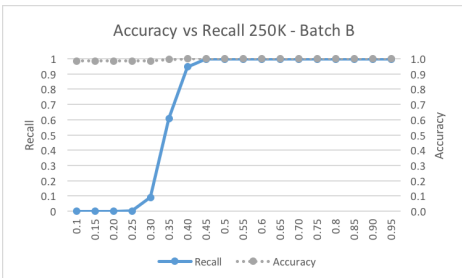
(e) Performance Evolution for 125K.



(f) Accuracy and Recall for 125K.



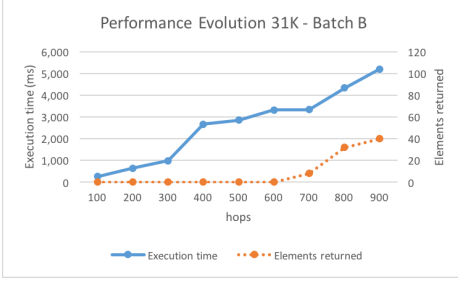
(g) Performance Evolution for 250K.



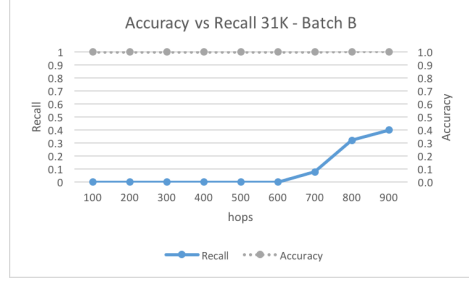
(h) Accuracy and Recall for 250K.

Figure B.12: Q4 Batch B. Accuracy and Recall with Random Approximations.

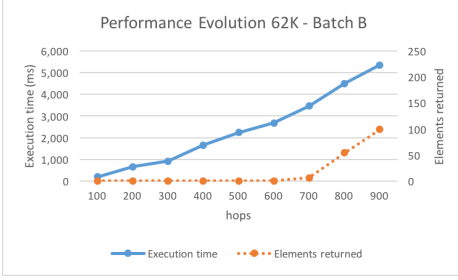
## B.2.6 Q4 - Spatial approximation



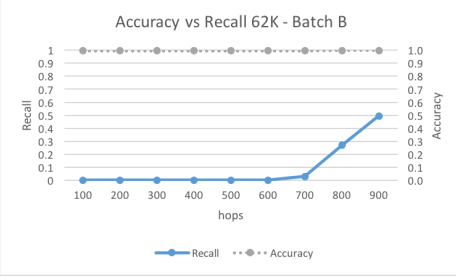
(a) Performance Evolution for 31K.



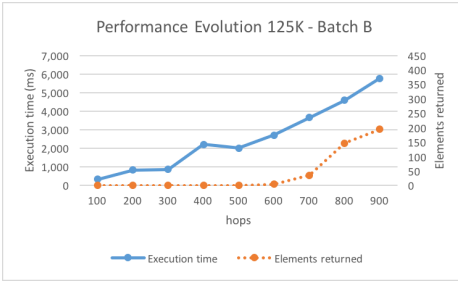
(b) Accuracy and Recall for 31K.



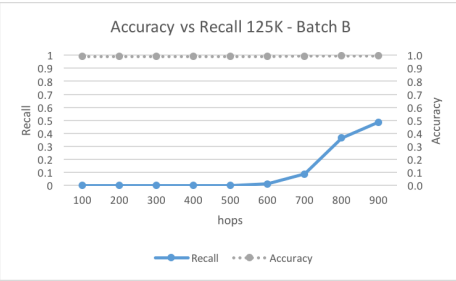
(c) Performance Evolution for 62K.



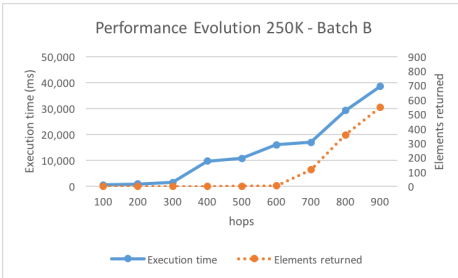
(d) Accuracy and Recall for 62K.



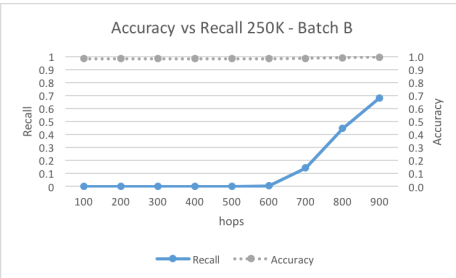
(e) Performance Evolution for 125K.



(f) Accuracy and Recall for 125K.



(g) Performance Evolution for 250K.



(h) Accuracy and Recall for 250K.

Figure B.13: Q4 Batch B. Accuracy and Recall with Spatial Approximations.

B.2.7 Q5 - Spatial approximation

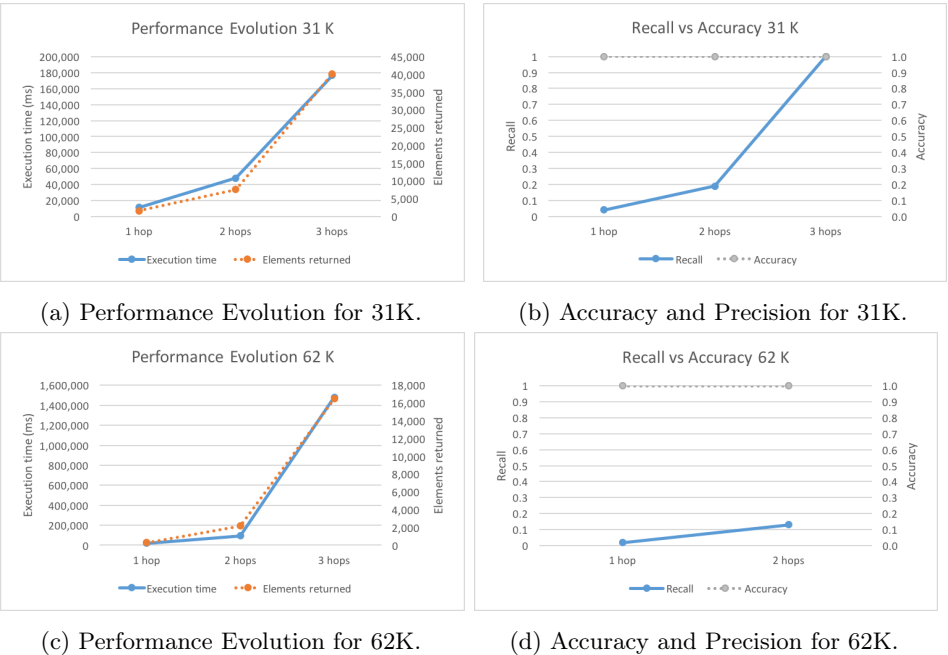


Figure B.14: Q5 Batch B. Accuracy and Precision with Spatial Approximations.



# Appendix C

## SDR Algorithm Execution

---

In this appendix we show several clarifying aspects of the algorithm presented in Chapter 5. First, in Section C.1 we present an example of the functioning of the SDR algorithm with an specific query (ProductPopularity in this case). Second, in Section C.2, we explain how the SDR works when the operators *where*, *not*, *and* and *or* appears in a query. In Section C.3 we show all the execution times of the experiments performed with the incremental version of the SDR algorithm in absolute terms (cf. *Experiments with streams of information* of Section 5.4.4). Finally, in Section C.4 we place some tables and figures referred in Chapter 5 that contains the results of the evaluation, in order to improve the readability of the chapter.

### C.1 *ProductPopularity* with SDR algorithm

To demonstrate how the SDR algorithm works for a specific query, a small graph for Amazon case study is shown in Fig. C.1. In this case, the graph contains two Customers (C1 and C2) and two products (P10 and P20). C1 orders two

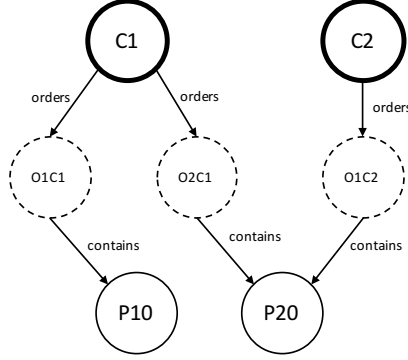


Figure C.1: Graph 1: example for Amazon case

Orders (O1C1 and O2C1), whereas C2 orders one (O1C2). We want to apply the SDR algorithm for this graph with the *ProductPopularity* query showed in Listing 5.2. The updates of the weight for each object as iterations run are displayed in Table C.1. In the following, each function and iteration of the algorithm displayed in Algorithm 1 is explained in detail. Note that when we refer to line numbers, unless otherwise specified, we are referring to the following:

- Text of the section: line numbers that are mentioned in the normal text of this section refer to the lines of **SDRAlgorithm** depicted in Algorithm 1.
- Non-enumerated lists: line numbers that are mentioned in non-enumerated lists refer to the lines of **SDRVertexCentric** function.
- Enumerated lists: line numbers that are mentioned in enumerated lists refer to the lines of functions **WeightInitialisation**, **InWeightPropagation** or **FurWeightPropagation**, depending on the specific case.

First, the SDR algorithm calls the **SDRVertexCentric** function for each object in the graph (line 1). This function starts the initial iteration ( $iteration = 0$ ) and it has the following execution flow:

- First, it establishes *guardCondition* to true (line 4).
- Since *iteration* meets the condition of line 5 ( $iteration == 0$ ), the function selects the last step of the query to be analysed (line 6).

- Then, it calls **WeightInitialisation** function (line 7). Note that at this point  $iteration = 0$  and  $S.size = 2$ . **WeightInitialisation** works as follows:
  1. First, the function checks the type of the step  $s$ . In *ProductPopularity* query, the last step is a *where* step. As a *where* step is a traversal step that has only one statement, the function gets into the *if* clause of line 12 and obtains the subquery contained in this statement (line 13).
  2. Then, it makes a recursive call with this subquery as input data of the **SDRVertexCentric** function (line 15). Note that at this point  $iteration = 0$  and  $S.size = 4$ , since the subquery has 4 steps<sup>1</sup>. This call has the following flow for each iteration:
    - *guardCondition* is established to true (line 4).
    - $iteration$  meets the condition of line 5, so the function selects the last step of the subquery and stores it in  $s$  (line 6).
    - Then, it calls **WeightInitialisation** function, that works as follows:
      - (a) Now, the step  $s$  corresponds with the *has* step (line 5 in Listing 5.2).
      - (b) Same as for the query, the function checks its type. In this case,  $s$  is a property filter, so the function gets into the *if* clause of line 1 and checks if  $v$  matches the filter (line 2). As shown in Figure C.1, only the object P10 matches the filter, so for  $v = P10$  the function gets into the *if* clause of line 2. For the rest of objects the function establishes *guardCondition* to false (line 7).
      - (c) Then, for  $v = P10$ , the function searches the previous step of the subquery that corresponds with a relationship (line 3). As can be viewed in Listing 5.2, this step is the relationship step **contains**.

---

<sup>1</sup>The SDR algorithm adds an initial graph step at the beginning of a traversal subquery. For this reason, a traversal subquery always has one more step than its size, i.e.  $S.size = 4$  in this case.



- (d) Therefore, it counts the number of neighbors that P10 can reach through relationship **contains** (line 4). Since P10 can reach O1C1 through relationship **contains**, *cNeighbors* is equal to 1.
  - (e) As *cNeighbors* is higher than 0, *guardCondition* is established to true (line 5).
  - (f) Once the function finishes the *if-then-else* clause of lines 1 to 18, it checks the value of *guardCondition*. For  $v = \text{P10}$ , this value is true, so the function gets into the *if* clause of line 19 and calculates the weight of P10. Since *weight* is 0 and *cNeighbors* is 1, the new value of *weight* is 1 (line 20). On the contrary, as stated before, for  $v \neq \text{P10}$  *guardCondition* is false so *weight* remains 0. Note how in the second column of Table C.1 the object P10 has *weight* = 1, whereas the remaining objects have *weight* = 0.
  - (g) Finally, it returns the *weight* value (line 22) and the function finishes.
- Then, the **SDRVertexCentric** function increments *iteration* counter (line 16) and the next iteration starts (at this point *iteration* = 1 and S.size = 4).
  - As *iteration* is less than S.size, the **SDRVertexCentric** function stays in the *while* loop of line 3 and sets *guardCondition* to true (line 4).
  - As *iteration* = 1, it gets into the *else* clause of line 8 and selects the same value for *s* than the initial iteration (line 9).
  - Then, it gets into *if* clause of line 10 and it calls **InWeightPropagation** function (line 11). This function works as follows :
    - (a) First, it checks the type of *s*. As stated in the previous iteration (recall that **WeightInitialisation** and **InWeightPropagation** analyse the same step), *s* is a property filter so it gets into *if* clause of line 3.
    - (b) Same as in the **WeightInitialisation** function, it searches

the previous step that corresponds to a relationship and stores it in *pRel* (line 4). This relationship is **contains**.

- (c) Then, *iteration* is incremented (line 5), which means that *iteration* = 2.
  - (d) The algorithm checks if the calculated *weight* in the previous iteration is higher than 0 (line 6). This is true only for *v* = P10, so, in this case, it sends a message through relationship **contains** to O1C1 (line 7).
  - (e) Finally, the **InWeightPropagation** function finishes and it returns the same *weight* calculated in **WeightInitialisation** function (line 10). Note that in columns 2 and 3 of Table C.1 all weights are the same.
- Then, **SDRVertexCentric** increments *iteration* and the new iteration starts, which means that *iteration* = 3 and *S.size* = 4.
  - As *iteration* is smaller or equal to *S.size*, **SDRVertexCentric** stays into *while* loop of line 3 and sets *guardCondition* to true (line 4).
  - *iteration* ≠ 0, so the function gets into *else* clause of line 8 and selects the relationship step **orders** (line 3 in Listing 5.2) for *s* (line 9).
  - Besides, *iteration* ≠ 1, so **SDRVertexCentric** gets into *else* clause of line 12 and it calls **FurWeightPropagation** function, that works as follows:
    - (a) First, it counts the number of messages sent from the previous iteration to *v* (line 1). Note that in the previous iteration only P10 sent a message to O1C1 through the relationship **contains**, so for *v* = O1C1 *cMessages* has value 1, while for the rest it is 0.
    - (b) Therefore, for *v* = O1C1, the function gets into *if* clause of line 2 and checks the type of *s*.
    - (c) As stated before, *s* is the relationship step **orders**, so the function gets into *if* clause of line 3, it counts the number of

neighbors that can be reached for  $v$  through  $s$  and stores this number in  $cNeighbors$ . For  $v = O1C1$ ,  $cNeighbors$  has value 1, since  $O1C1$  can reach  $C1$  through relationship **orders**.

- (d) Then, for this value of  $v$ , *guardCondition* is set to true (line 5) and a message is sent through relationship **orders** to  $C1$  (line 6).
  - (e) Finally, as *guardCondition* is true for every object of the graph, the function updates the value of *weight* for all of them (lines 19-21). However, since  $cNeighbors$  and  $cMessages$  are 0 for  $v \neq O1C1$ , the *weight* value remains the same as in the previous iteration for this case. On the other hand, for  $v = O1C1$ ,  $cMessages = 1$  and  $cNeighbors = 1$ , so *weight* is updated to 2. Updated values for this iteration can be viewed in column 4 of Table C.1.
  - (f) **FurWeightPropagation** returns the updated *weight* and it finishes (line 22).
- Now, **SDRVertexCentric** increments *iteration* (line 16) and the next iteration starts (at this point  $iteration = 4$  and  $S.size = 4$ ).
  - *guardCondition* is set to true (line 4).
  - $iteration \neq 0$ , so **SDRVertexCentric** gets into *else* clause of line 8 and selects the added graph step at the beginning of the subquery for  $s$  (line 9).
  - Besides,  $iteration \neq 1$ , so the **SDRVertexCentric** gets into *else* clause of line 12 and **FurWeightPropagation** starts again:
    - (a) First, it counts the number of messages sent from the previous iteration to  $v$  (line 1). In the previous iteration, only  $O1C1$  sent a message to  $C1$  through the relationship **orders**. For this reason, for  $v = C1$ ,  $cMessages$  has value 1, and 0 for the rest of objects.
    - (b) Then, for  $v = C1$ , the function gets into *if* clause of line 2 and checks the type of  $s$ . However, since  $s$  is a graph step, the algorithm gets out of this *if* clause without any change.
    - (c) Finally, as *guardCondition* is true for every object of the

graph, the algorithm updates the value of *weight* for all of them (lines 19-21). However, since *cNeighbors* and *cMessages* are 0 for  $v \neq C1$ , the *weight* value remains the same as in the previous iteration for this case. On the other hand, for  $v = C1$ , *cMessages* = 1 and *cNeighbors* = 0, so *weight* is updated to 1. Updated values for this iteration can be viewed in column 5 of Table C.1.

- (d) **FurWeightPropagation** returns the updated *weight* and it finishes (line 22).
  - Then, *iteration* is incremented by **SDRVertexCentric** in line 16 and since *iteration* = 5, which is higher than *S.size*, the function escapes the *while* loop of line 3 and returns the value of *weight* (line 18).
- 3. Once the results of the recursive call are obtained, the function computes weights according to the type of traversal (line 16). The computation process for the different types of traversal steps is explained more in detail in Appendix C.2.
- 4. Then, the function escapes the *if* clause of line 12 and checks the *guardCondition* value (line 19).
- 5. Since *guardCondition* remains true, it updates the *weight* value (line 20). However, as *cNeighbors* value is equal to 0 for every object in the graph, the value of *weight* is updated with the result of the recursive call of lines 15 and 16.
- Finally, *iteration* is incremented by **SDRVertexCentric** in line 16 (note that at this point *iteration* = 1 and *S.size* = 2, since the query has 2 steps).
- *guardCondition* is set to true (line 4).
- *iteration* = 1, so **SDRVertexCentric** gets into *else* clause of line 8 and selects the last step of the query for *s* (line 9).
- Then, it gets into *if* clause of line 10 and calls **InWeightPropagation** function (line 11):

1. First, it checks the type of  $s$ . As stated in the previous iteration,  $s$  is a traversal so it gets into *if* clause of line 3.
  2. Then, it searches for the previous step that corresponds to a relationship and stores it in  $pRel$  (line 4). However, since there are no more relationship steps in the query,  $pRel$  does not contain any relationship.
  3. Then,  $iteration$  is incremented (line 5), so that  $iteration = 2$  and  $S.size = 2$ .
  4. The function checks if the calculated  $weight$  in the previous iteration is higher than 0 (line 6). This is true only for P10, O1C1 and C1 so, in this case, the function tries to send a message through  $pRel$  (line 7). But since  $pRel$  does not contain a relationship, no messages are sent.
  5. Finally,  $weight$  value remains the same as in the previous iteration (line 10). Note that weights in columns 5 and 6 of Table C.1 are the same.
- Then,  $iteration$  is incremented by **SDRVertexCentric** and it is equal to 3. In this case,  $iteration$  is higher than  $S.size$ , so **SDRVertexCentric** escapes the *while* loop of line 3, it returns  $weight$  value (line 18) and the execution finishes.

Once **SDRVertexCentric** finishes, the SDR Algorithm obtains a subgraph with the objects with  $weight$  higher than 0, and the relationships among them (lines 2 and 3). In this example, this subgraph only contains C1, O1C1 and P10 objects and the relationships between them. Note that if *ProductPopularity* query is run either over this subgraph or over the complete graph of Figure C.1, the result will be object C1 for both executions.

## C.2 Traversals with SDR algorithm

In this appendix, we explain the strategies to compute the weights for the different types of traversal steps in Algorithm 1. We distinguish four types of traversal steps: *where*, *not*, *and* and *or*. For a better understanding about how the SDR algorithm computes them, we describe several examples applied to the Amazon graph shown in Figure C.1.

### C.2.1 *Where Step*

The *where* step is used to filter objects according to a predicate. This predicate is based on the path history of an object. In this way, an object is selected by the filter if it has the path indicated in the *where* step predicate.

Object/Iteration	It 0				It 1
C1	0	0	0	1	1
C2	0	0	0	0	0
O1C1	0	0	2	2	2
O2C1	0	0	0	0	0
O1C2	0	0	0	0	0
P10	1	1	1	1	1
P20	0	0	0	0	0

Table C.1: Object weights for *ProductPopularity* query with SDR Algorithm

Let us consider the sample query shown in Listing 5.2, which contains a *where* step. In this query, the graph objects that order an *Order* that contains a *Product* with the `idProduct = '10'` are filtered. For this query to be applied to the graph of Figure C.1, the SDR algorithm first obtains the weights of the *where* clause, iterating the steps of the subquery contained in the predicate. The results of the calculated weights for each iteration and each object of the graph are shown in columns 2 to 6 of Table C.1. Once the algorithm finishes the calculation of the *where* step, the resulting weights calculated for this step are assigned to each object of the graph for the next iteration. Note in column 6 of Table C.1 that the weights of all objects are the same as in the last iteration of the computation of the *where* step (column 5). This is because iteration It1 does not modify the weights, since it only sends messages, as explained in Section 5.3.1. After that, the algorithm continues the normal execution updating the calculated weights according to the remaining steps of the query.

### C.2.2 *Not Step*

Same as *where* step, the *not* step is used to filter the objects according to a predicate. However, *not* step removes from the result the objects that satisfy this predicate and returns the rest.

Let us observe again the example shown in Listing 5.2 and suppose we change the *where* step for a *not* step in this query. In this case, the graph objects that

do not order an *Order* that contains a *Product* with the `idProduct = '10'` are filtered. Applying this query to the graph of Figure C.1, the SDR algorithm first traverses the steps of the predicate of the *not* clause. At first, the algorithm calculates the weights in the same way as in the *where* step. However, in the last iteration it performs the following operation with the weight values:

$$weight = \begin{cases} 0 & \text{if } weight > 0 \\ 1 & \text{if } weight \leq 0 \end{cases} + pItWeight$$

Therefore, if the calculated weight is higher than 0, then the algorithm changes it to 0, and the other way around. After this conversion, if the object was relevant to the previous steps of the query, it will have a weight 0 and, therefore, it will be discarded when obtaining the subgraph. To avoid this, the algorithm adds the weight calculated for that object in the penultimate iteration (*pItWeight*). This process is exemplified for the graph of Figure C.1 in column 5 of Table C.2.

Object/Iteration	It 0				It 1
C1	0	0	0	$(1 \rightarrow 0) + 0 = 0$	0
C2	0	0	0	$(0 \rightarrow 1) + 0 = 1$	1
O1C1	0	0	2	$(2 \rightarrow 0) + 2 = 2$	2
O2C1	0	0	0	$(0 \rightarrow 1) + 0 = 1$	1
O1C2	0	0	0	$(0 \rightarrow 1) + 0 = 1$	1
P10	1	1	1	$(1 \rightarrow 0) + 1 = 1$	1
P20	0	0	0	$(0 \rightarrow 1) + 0 = 1$	1

Table C.2: Object weights for *ProductPopularity* query with *not* step with SDR Algorithm

Then, as with the *where* step, the algorithm continues the normal execution updating the calculated weights according to the remaining steps of the query. Note in column 6 of Table C.2 that the weights for each object are the same as the weights of the last iteration of the computation of the *not* step (column 5), since in the It1 only messages are sent to other objects.

### C.2.3 And Step

The *and* step is used to filter objects according to two or more predicates and it ensures that filtered objects meet all predicates. Therefore, since in this case there are more than one predicate, there are more than one subquery where to compute the weights too.

Object/Iteration	It 0	It 1	It 2	It 3
C1	0	0	0	1
C2	0	0	0	1
O1C1	0	0	0	0
O2C1	0	0	2	2
O1C2	0	0	2	2
P10	0	0	0	0
P20	2	2	2	2

Table C.3: Object weights for subquery example with SDR Algorithm

Let us consider Listing 5.4, where *PackagePopularity* query of Amazon case study is shown. In this case, the objects that order an *Order* that contains the *Product* with the `idProduct = '10'` and order an *Order* that contains the *Product* with the `idProduct = '20'` are filtered. Note that this query has two subqueries: the first one is equivalent to the subquery of the *where* step in *ProductPopularity* query, and the second one is similar but with a different property filter step. The weights computed for the second subquery are shown in Table C.3—note that 4 iterations are displayed in the table because it is focused on the subquery. In this way, results for both subqueries with the SDR algorithm are shown in columns 2 to 5 of Tables C.1 and C.3, respectively. For the *and* step, the algorithm computes the weights for both queries separately and performs the following operation to merge them:

$$weight = \sum_{i=1}^n pItWeight_i + \prod_{i=1}^n weight_i$$

If  $n$  is the number of predicates contained in the *and* step,  $weight_i$  is the calculated weight of the subquery of the predicate  $i$ , and  $pItWeight_i$  is the calculated weight of the predicate  $i$  in the penultimate iteration. Results of the weights computed for the *PackagePopularity* query are shown in Table C.4. Note we add  $pItWeight_i$  to avoid that the object has weight 0 if it is relevant to the steps previous to the first one, similar to the situation described in Section C.2.2.

#### C.2.4 Or Step

Similar to the *and* step, the *or* step is used to filter the objects according to two or more predicates. However, in this case, it ensures that the filtered objects meet at least one of the predicates.



Object/Iteration	It 0	It 1
C1	$(0+0) + (1*1) = 1$	1
C2	$(0+0) + (0*1) = 0$	0
O1C1	$(2+0) + (2*0) = 2$	2
O2C1	$(0+2) + (0*2) = 2$	2
O1C2	$(0+2) + (0*2) = 2$	2
P10	$(1+0) + (1*0) = 1$	1
P20	$(0+2) + (0*2) = 2$	2

 Table C.4: Object weights for *PackagePopularity* example with SDR Algorithm

Let us consider we modify in Listing 5.4 the *and* step with an *or* step, obtaining the query *SimProductsPopularity* of Amazon case example. In this case, the objects that order an *Order* that contains the *Product* with the `idProduct = '10'` or order an *Order* that contains the *Product* with the `idProduct = '20'` are filtered. Starting from the results shown in columns 2 to 5 of Table C.1 and Table C.3, the *or* step performs the following merge of subqueries:

$$weight = \sum_{i=1}^n weight_i$$

Being  $n$  the number of predicates contained in the *or* step and  $weight_i$  the calculated weight of the subquery of the predicate  $i$ . Results for *SimProductsPopularity* query over the graph of Figure C.1 are shown in Table C.5. Note that with the simple graph of Figure C.1, all the objects in the graph are assigned weights  $> 0$ . This would not be the case in a real system, where many elements would be discarded, as we describe in Section 5.4.

Object/Iteration	It 0	It 1
C1	$(1+1) = 2$	2
C2	$(0+1) = 1$	1
O1C1	$(2+0) = 2$	2
O2C1	$(0+2) = 2$	2
O1C2	$(0+2) = 2$	2
P10	$(1+0) = 1$	1
P20	$(0+2) = 2$	2

 Table C.5: Object weights for *SimProductsPopularity* example with SDR Algorithm

### C.3 Results for Experiments with SDR algorithm and streams of information

In this section we present three tables where the execution times of all experiments performed with the incremental version of the SDR algorithm are shown. The execution times are exposed in absolute terms and represented in milliseconds. Results of Amazon, Contest and Youtube examples are presented in Tables C.6, C.7 and C.8, respectively. Columns  $T_{CG}$  represent the execution times obtained on the complete graph (without using the SDR algorithm), whereas columns  $T_{SubG}$  represent the executions times obtained where the SDR algorithm is used in the experiments. Some numbers of C.7 are not shown. Please, remember that this is because the queries performed on the complete models take too long and the break-even point was already reached. For this reason, there was no real need to compute them (cf. Section 5.4.4).

Note how the execution times of columns  $T_{SubG}$  decreases respect to the execution times of columns  $T_{CG}$  as the model size increases, i.e. the time gain increases with the model size. In addition, as the value of  $\beta$  increases, the value of columns  $T_{SubG}$  also decreases respect to columns  $T_{CG}$ . This means that the time gain increases as more data arrives to the system. However, columns  $T_{SubG}$  increases respect to the execution times of columns  $T_{CG}$  as  $\alpha$  value increases. This means that the time gain increases as more times the query is executed (cf. RQ3 conclusions of Chapter 5). Finally, note how the disjunctive, conditional, simple, conjunctive and negative query patterns present a higher time gain than queries that follow aggregation pattern. This is because the execution times of columns  $T_{SubG}$  of these patterns are lower with respect to columns  $T_{CG}$  than queries that follow aggregation patterns. In this sense, the time gain of queries that follow aggregation patterns depends on the overload imposed by the aggregation operators and their corresponding filters.

Query Name	Models																
	$\alpha = 5$						$\alpha = 10$										
	2M		4M		8M		15M		2M		4M		8M		15M		
	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	
ProductPopularity (Simple)	50	94582	92057	153935	152346	280902	295247	620625	630398	90607	75239	148420	129145	286252	268805	611320	556458
	100	115979	131761	185291	226912	344510	432360	773134	951594	112459	110517	183420	179479	346120	374600	738884	786192
	150	133707	164534	223723	289564	411627	559805	927335	1273457	135389	134037	215087	234479	410781	470658	881824	1047927
	200	153524	192410	251083	347343	474618	679762	1070594	1613043	153101	163849	248754	283146	475008	580751	1025008	1268507
	250	173291	218072	280248	418404	527617	801081	1216514	1953453	174175	187699	280420	336146	538735	688496	1171322	1555394
ProductPopularityC (Conditional)	50	97911	94744	156179	152461	303865	307534	613037	656779	93116	77081	150733	130601	297787	264202	598781	560813
	100	119771	128844	195005	224628	376811	441668	780670	981734	115976	108311	193268	190171	354613	371545	752777	798436
	150	140849	166611	223416	282621	448111	567981	902735	1274169	132541	137304	229479	242125	429465	482275	897071	1059050
	200	161368	199971	260450	346285	511337	689729	1055094	1596695	155565	159253	268287	292496	493771	574267	1024265	1266137
	250	177966	226570	290930	410494	567612	811047	1197425	1844483	175445	186407	294193	338614	557222	673636	1178049	1532957
AlternativeCustomer (Negative)	50	97068	76353	168377	134496	320639	280400	699135	594487	93894	75029	163790	126888	316045	258448	655965	564130
	100	122567	111540	213585	193342	411731	397172	937919	888591	118439	103239	204478	179031	402238	358750	854324	773438
	150	148939	137840	254482	240660	505078	505402	1181910	1185911	141288	128619	244459	224689	482166	446260	1062539	1027903
	200	173343	165712	295045	300632	596021	613399	1414597	1492454	163764	154670	282552	268213	563961	537801	1272732	1236900
	250	197267	191895	332245	353964	675146	728137	1659493	1869039	186280	176057	322984	316906	643167	623786	1443449	1426903
PackagePopularity (Conjunctive)	50	104203	96679	165368	152412	309912	306457	618535	612001	100718	76369	161058	131772	304970	261236	607523	555448
	100	126796	136405	197174	223731	367769	437586	768418	918244	117365	111939	197477	184319	379431	358150	779778	788580
	150	147881	171875	231170	291585	438639	556888	915852	1265669	141290	135264	238816	246329	448295	481634	922984	1002367
	200	168977	204334	269420	358579	511425	681779	1054222	1649133	158553	100522	275616	292744	519965	571999	1011155	1294478
	250	186032	228219	305488	431908	576078	815457	1202995	1964100	177526	185984	294919	338297	579696	665474	1140163	1519447
SimProductsPopularity (Disjunctive)	50	100296	98262	163630	162145	297592	344906	615393	725169	99454	80689	158273	142321	310278	279730	616013	582598
	100	123752	146012	204114	246551	367569	497881	775496	1064931	119578	118267	194874	196760	377173	395485	775251	902904
	150	145237	189520	231356	336327	435008	652681	884645	1465745	139015	145970	230174	257424	445611	532541	917427	1224391
	200	166864	229278	266011	409746	500625	812498	1037974	1886045	161624	178975	265907	326696	500312	612667	1039793	1416477
	250	183413	263098	303124	498392	566176	949590	1194748	2276864	186677	203539	300909	378602	739860	1210981	1799961	247320
PrefCustomer (Aggregation)	50	100843	75731	175659	134069	315148	242381	641706	492063	92516	67411	158135	116315	297855	220493	621204	469270
	100	127487	99138	214035	171515	387484	320554	800535	692862	116491	89946	195948	151591	368285	294618	773338	638345
	150	151159	124464	251845	210059	461711	396250	947748	181540	138429	111684	231966	191374	435391	367404	911947	818633
	200	174318	145196	289554	256358	537203	476114	1118514	1060461	160503	131655	232000	502329	447320	1061787	985681	1268507
	250	192014	162372	318652	289718	602090	550735	1276552	1253418	181812	150743	300587	270197	568608	517654	1209605	1158584
PrefCustomerSimProducts (Aggregation)	50	90194	71228	170999	132638	297080	241052	643661	487502	86668	66546	163050	118814	308368	238740	622600	456054
	100	115730	94228	211635	173261	373080	324152	806088	682772	113200	88894	202571	162521	381580	315648	782510	644740
	150	138478	117895	248173	211166	450580	402652	971911	879153	135579	112207	239781	202094	453957	387488	928691	833221
	200	160034	138561	284632	261826	531080	480152	1140475	1075808	158819	134504	275238	242272	520292	467561	1086121	1010855
	250	181914	159895	312889	289534	598080	559152	1293368	1268733	181170	153861	311570	280772	592048	538881	1234447	1189421

Table C.6: Incremental results for Amazon case study (ms).

### C.3 Results for Experiments with SDR algorithm and streams of information

Query Name	Models																	
	$\alpha = 5$									$\alpha = 10$								
	1M			4M			9M			12M			1M			4M		
	$\beta$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$
RecentPart (Simple)	50	57086	50482	249035	247677	493521	564701	767268	898172	58147	44583	246468	228794	497453	511788	708101	885194	
	100	88491	80054	368753	405890	748589	936218	1136626	1608534	87558	71248	364854	373119	752465	834711	1068829	1519484	
	150	112776	106621	475755	534779	993407	1295133	1514976	2311699	113420	98392	480510	501739	1010828	1154431	1424241	2126493	
	200	139085	135621	590816	709620	1231939	1649998	1888916	2972176	138456	124265	596595	629781	1269816	1479464	1774530	2728551	
	250	162666	161058	667599	823393	1476080	2006916	2273803	3625746	162480	148837	711801	758473	1525971	1799122	2146670	3320473	
ContestPart (Conditional)	50	64685	54722	273116	284892	539270	628470	836799	1058455	65741	47536	260387	250261	552353	569656	792584	913293	
	100	94557	90755	384707	460563	785558	1059606	1212876	1851564	94423	76979	378322	403684	818426	927646	1147254	1626023	
	150	123814	125295	509116	631663	1027632	1466259	1589616	2662796	122756	105540	490731	548028	1064127	1302333	1500571	2294812	
	200	152355	157168	622285	780515	1268804	1876836	1963133	3451484	148221	132506	604412	689180	1305772	1664729	1852649	2977375	
	250	172392	184972	722056	949912	1509905	2296666	2338789	4267677	173580	159431	717265	826991	1547555	2033330	2204807	3642626	
UnchosenCap (Conjunctive)	50	69049	54913	281421	277407	559017	606365	808453	959344	69347	48882	277774	261283	559033	555967	818913	867771	
	100	100466	88994	399421	456073	824272	1007616	1173156	1715379	98033	77665	393826	419663	808240	900296	1175232	1542825	
	150	129734	121699	516421	616407	1073508	1394155	1529450	2486436	126802	104019	510041	564984	1050621	1245974	1526893	2170254	
	200	158083	151463	633421	772407	1325154	1787779	1885826	3254134	153409	130520	625517	705104	1294813	1588057	1892029	2785591	
	250	183781	181720	750421	920740	1572334	2164643	2244475	3975669	180081	156175	740286	838674	1537425	1915865	2256053	3390726	
FunniestCaption (Aggregation)	10	41136	332155	186432	938040	358136	2346014	517138	8759566	38193	213949	175314	639190	341988	1630642	511418	5868913	
	20	50136	551155	225845	1349169	-	632999	15331814	46270	323742	207209	914890	407533	2386261	607747	8856675	-	
	50	75400	1103502	333755	2766955	-	-	-	-	-	-	-	-	-	-	-	-	
	100	114173	2189979	553741	5170101	-	-	-	-	-	-	-	-	-	-	-	-	
	150	148516	3149790	703845	7927169	-	-	-	-	-	-	-	-	-	-	-	-	
Abandon (Aggregation)	200	181156	4071866	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	250	219915	5179012	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	50	64110	48424	271746	257609	546639	593103	829205	920556	59423	45450	258246	248462	525074	516687	829230	829902	
	100	94298	81430	394128	421386	827180	981110	1239696	1628805	84683	72969	375344	400398	799237	851210	1226048	1440936	
	150	123576	109338	528616	573597	1101269	1334265	1639146	2321023	110198	99856	489303	538913	1044107	1177352	1619180	2026751	
FunniestCaptionU (Aggregation & Conjunctive)	200	150960	139717	643579	705128	1375698	1705218	2038389	2974642	136881	126472	605637	679233	1289729	1511575	2006933	2015556	
	250	180419	168806	779475	854206	1647026	2064524	2439692	3628474	161761	152330	724110	808473	1532903	1832586	2394386	3199692	
	10	42734	136043	185780	260515	343901	522363	533139	1049679	43394	95611	176477	203576	353580	409764	524149	817049	
	20	50734	211043	272712	372971	-	-	610203	1736517	52245	136804	203066	286103	411187	576268	598136	1231294	
	50	75658	407884	283817	745554	-	-	-	-	-	-	-	-	-	-	-	-	
FunniestCaptionU (Aggregation & Conjunctive)	100	111081	772849	411484	1211021	-	-	-	-	-	-	-	-	-	-	-	-	
	150	146901	1076419	522712	1696971	-	-	-	-	-	-	-	-	-	-	-	-	
	200	170939	1469661	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	250	206752	1807561	-	-	-	-	-	-	-	-	-	-	-	-	-	-	

Table C.7: Incremental results for Contest case study (ms).

## Appendix C. SDR Algorithm Execution

Query Name	Models													
	$\alpha = 5$							$\alpha = 10$						
		1M	4M	9M	12M	1M	4M	9M	12M	1M	4M	9M	12M	
	$\beta$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	$T_{SubG}$	$T_{CG}$	
GetAnimalVideos (Conditional)	50	160478	151024	339951	355719	551343	596749	757730	920697	169901	145060	333913	337732	724114
	100	251437	243403	525212	614628	855365	1020609	1178656	1592927	260093	292175	516484	572357	1128889
	150	341257	328462	708897	864694	1153593	1428311	1597962	2232779	348396	316610	697217	796099	1527075
	200	430135	420567	896755	1115371	1462374	1825149	2028702	2870430	435338	402317	877806	1016102	1923395
	250	518536	509171	1075018	1357568	1750257	2232831	2430517	3496813	524031	486877	1057314	1235370	2321977
NotPresent (Negative)	50	179943	136610	365196	327602	558188	575189	776919	827970	177255	146564	360515	329827	748057
	100	276293	224532	563001	550363	866723	974517	1220634	1380348	269992	237699	546541	555899	1158239
	150	372947	305832	758548	765229	1155284	1353803	1663706	1903882	360902	324780	731185	771814	1565789
	200	466442	385819	952131	981728	1441829	1708615	2102329	2420291	451102	411319	913989	993444	1971174
	250	560899	465194	1146298	1195737	1729432	2066789	2545586	2944246	541672	498219	1083802	1210209	2380094
AnimalPerson (Conjunctive)	50	182173	147167	353107	354814	556884	594664	745489	919398	183873	137045	359683	328866	790934
	100	273160	241997	550671	612610	852920	1019406	1165088	1608660	275955	221243	553562	558083	1207923
	150	363608	333103	746511	857663	1145859	1440244	1577528	2246671	365468	306850	747897	781565	1620163
	200	454527	427819	939787	1098694	1435693	1857702	1992254	2908602	455032	395812	941483	999644	2028310
	250	545788	515771	1132095	1328547	1727115	2264884	2413026	3553216	544144	476185	1134503	1210875	2436655
PresentSoon (Conjunctive)	50	169930	136743	356534	351968	562544	578349	807998	902938	170314	146048	371142	344294	800141
	100	262583	230832	542881	590477	843594	979087	1214888	1570942	258269	234071	561939	581804	1207906
	150	353010	321449	728358	833739	1125350	1361926	1614583	2195491	346408	318433	751016	815434	1615512
	200	443102	412956	911948	1064984	1406929	1750513	2010722	2858950	431432	403691	939688	1045190	2017411
	250	531395	519177	1099003	1300461	1689538	2128997	2411279	3468374	517188	487878	1128802	1268845	2424486
Pets (Disjunctive)	50	182190	192586	382661	431639	575667	711542	806525	1056724	175111	171466	388845	388510	807300
	100	275713	330336	579650	746875	867966	1272093	1243505	1840702	265752	280716	572848	653852	1219504
	150	367341	467490	776762	1058480	1161325	1805957	1671150	2631451	359087	392896	755316	916304	1626860
	200	457045	609013	969836	1365272	1453549	2324521	2097777	3403946	449539	494704	938705	1170456	2030647
	250	548030	739279	1161644	1665808	1748329	2839486	2526527	4190922	541563	601612	1122381	1422895	2396798
In Cast (Aggregation)	50	181810	141067	383409	361684	596940	558123	806608	841923	180088	137440	372906	337400	791789
	100	275610	235005	604184	602986	922234	961807	1255719	1455437	271789	228687	570467	569561	1226164
	150	366352	324778	823124	842514	1234781	1361884	1701410	2055581	362861	316072	767186	792992	1661492
	200	457368	420720	1040167	1080807	1543214	1730908	2142503	2625442	453822	401108	962330	1014089	2097217
	250	548596	510517	1258630	1314109	1850993	2103743	2584508	3187884	542425	484389	1157606	1238524	2529484

Table C.8: Incremental results for Youtube case study (ms).

## C.4 Additional charts and tables displaying experiments results

To improve the readability of Chapter 5, this appendix contains some of the tables and figures that show the results of the evaluations. Specifically, Figures C.2 and C.3 show the execution time and memory consumption of the experiments with static information of *Contest* and *YouTube* case studies, respectively. Then, Tables C.9 and C.10 show the gain results of the experiments with dynamic information of the same case studies.

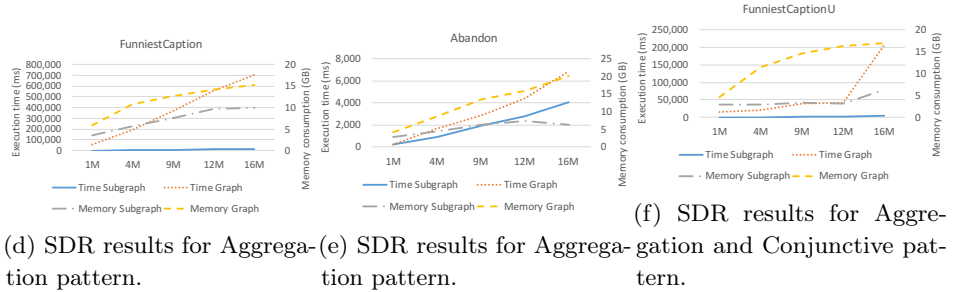
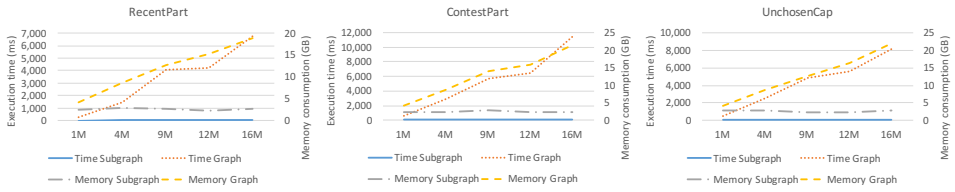


Figure C.2: Performance results for SDR algorithm in Contest example queries.

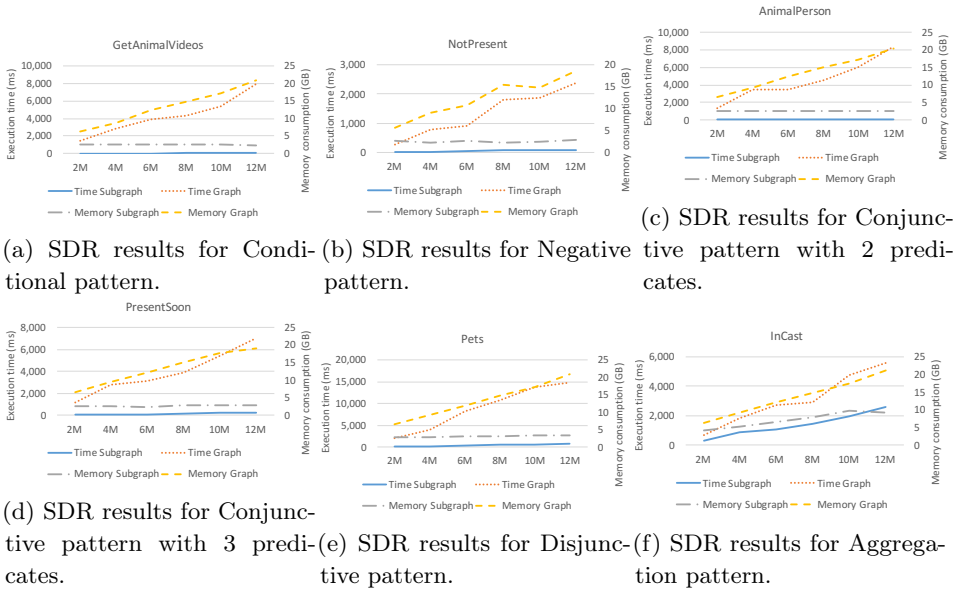


Figure C.3: Performance results for SDR algorithm in YouTube example queries.

## C.4 Additional charts and tables displaying experiments results

Query Name		Models							
		$\alpha = 5$				$\alpha = 10$			
	$\beta$	1M	4M	9M	12M	1M	4M	9M	12M
RecentPart (Simple)	50	-0.1308	-0.0055	<b>0.1260</b>	<b>0.1457</b>	-0.3042	-0.0773	<b>0.0280</b>	<b>0.2001</b>
	100	-0.1054	<b>0.0915</b>	0.2004	0.2934	-0.2289	<b>0.0222</b>	0.0985	0.2966
	150	-0.0577	0.1104	0.2330	0.3446	-0.1527	0.0423	0.1244	0.3302
	200	-0.0255	0.1674	0.2534	0.3645	-0.1142	0.0527	0.1417	0.3496
	250	-0.0100	0.1892	0.2645	0.3729	-0.0917	0.0615	0.1518	0.3535
ContestPart (Conditional)	50	-0.1821	<b>0.0413</b>	<b>0.1419</b>	<b>0.2094</b>	-0.3830	-0.0405	<b>0.0304</b>	<b>0.1322</b>
	100	-0.0419	0.1647	0.2586	0.3449	-0.2266	<b>0.0628</b>	0.1177	0.2944
	150	<b>0.0118</b>	0.1940	0.2991	0.4030	-0.1631	0.1046	0.1829	0.3461
	200	0.0306	0.2027	0.3240	0.4312	-0.1186	0.1230	0.2156	0.3778
	250	0.0680	0.2399	0.3426	0.4520	-0.0887	0.1327	0.2389	0.3947
UnchosenCap (Conjunctive)	50	-0.2574	-0.0145	<b>0.0781</b>	<b>0.1573</b>	-0.4187	-0.0631	-0.0055	<b>0.0563</b>
	100	-0.1289	<b>0.1242</b>	0.1820	0.3161	-0.2622	<b>0.0616</b>	<b>0.1023</b>	0.2383
	150	-0.0660	0.1622	0.2300	0.3849	-0.2190	0.0972	0.1568	0.2964
	200	-0.0437	0.1799	0.2588	0.4205	-0.1754	0.1129	0.1847	0.3208
	250	-0.0113	0.1850	0.2736	0.4354	-0.1531	0.1173	0.1975	0.3346
FunniestCaption (Aggregation)	0	<b>0.7680</b>	<b>0.6378</b>	<b>0.6803</b>	<b>0.8520</b>	<b>0.7680</b>	<b>0.6378</b>	<b>0.6803</b>	<b>0.8520</b>
	10	0.8762	0.8013	0.8473	0.9410	0.8215	0.7257	0.7903	0.9129
	20	0.9090	0.8326	-	-	0.8571	0.7735	0.8292	0.9314
	50	0.9317	0.8794	-	-	-	-	-	-
	100	0.9479	0.8929	-	-	-	-	-	-
Abandon (Aggregation)	50	-0.3239	-0.0549	<b>0.0783</b>	<b>0.0992</b>	-0.3074	-0.0394	-0.0162	<b>0.0008</b>
	100	-0.1580	<b>0.0647</b>	0.1569	0.2389	-0.1605	<b>0.0626</b>	<b>0.0611</b>	0.1491
	150	-0.1302	0.0784	0.1746	0.2938	-0.1036	0.0921	0.1132	0.2011
	200	-0.0805	0.0873	0.1932	0.3147	-0.0823	0.1084	0.1468	0.2327
	250	-0.0688	0.0875	0.2022	0.3276	-0.0619	0.1043	0.1635	0.2517
FunniestCaptionU (Aggregation and Conjunctive)	0	<b>0.3517</b>	-0.2278	-0.1412	-0.0071	<b>0.3517</b>	-0.2278	-0.1412	-0.0071
	10	0.6859	<b>0.2869</b>	<b>0.3416</b>	<b>0.4921</b>	0.5461	<b>0.1331</b>	<b>0.1371</b>	<b>0.3585</b>
	20	0.7596	0.4458	-	-	0.6181	0.2902	0.2865	0.5142
	50	0.8145	0.6193	-	-	-	-	-	-
	100	0.8563	0.6602	-	-	-	-	-	-

Table C.9: Ratio Incremental gain results for Contest case study.



## Appendix C. SDR Algorithm Execution

Query Name		Models							
		$\alpha = 5$				$\alpha = 10$			
	$\beta$	2M	4M	6M	8M	2M	4M	6M	8M
GetAnimalVideos (Conditional)	50	-0.0626	<b>0.0443</b>	<b>0.0761</b>	<b>0.1770</b>	-0.1712	<b>0.0113</b>	<b>0.0317</b>	<b>0.1305</b>
	100	-0.0330	0.1455	0.1619	0.2601	-0.1202	0.0976	0.1033	0.2007
	150	-0.0390	0.1802	0.1923	0.2843	-0.1004	0.1242	0.1272	0.2213
	200	-0.0228	0.1960	0.1988	0.2932	-0.0821	0.1361	0.1432	0.2260
	250	-0.0184	0.2081	0.2161	0.3049	-0.0763	0.1441	0.1561	0.2301
NotPresent (Negative)	50	-0.3172	-0.1148	<b>0.0296</b>	<b>0.0617</b>	-0.2094	-0.0930	-0.0118	<b>0.1231</b>
	100	-0.2305	-0.0230	0.1106	0.1157	-0.1359	<b>0.0168</b>	<b>0.0790</b>	0.2155
	150	-0.2195	<b>0.0087</b>	0.1466	0.1262	-0.1112	0.0526	0.1095	0.2374
	200	-0.2090	0.0301	0.1561	0.1314	-0.0967	0.0800	0.1141	0.2540
	250	-0.2057	0.0413	0.1632	0.1354	-0.0872	0.0962	0.1225	0.2609
AnimalPerson (Conjunctive)	50	-0.2379	<b>0.0048</b>	<b>0.0635</b>	<b>0.1892</b>	-0.3417	-0.0806	-0.0405	<b>0.0653</b>
	100	-0.1288	0.1011	0.1633	0.2757	-0.2473	<b>0.0081</b>	<b>0.0563</b>	0.1768
	150	-0.0918	0.1296	0.2044	0.2978	-0.1910	0.0431	0.0887	0.2154
	200	-0.0624	0.1446	0.2272	0.3150	-0.1496	0.0582	0.1050	0.2322
	250	-0.0582	0.1479	0.2374	0.3209	-0.1427	0.0631	0.1215	0.2453
PresentSoon (Conjunctive)	50	-0.2427	-0.0130	<b>0.0273</b>	<b>0.1051</b>	-0.1661	-0.0780	-0.0275	<b>0.0162</b>
	100	-0.1376	<b>0.0806</b>	0.1389	0.2266	-0.1034	<b>0.0341</b>	<b>0.0728</b>	0.1608
	150	-0.0982	0.1264	0.1737	0.2646	-0.0879	0.0790	0.1092	0.2064
	200	-0.0730	0.1437	0.1963	0.2967	-0.0687	0.1009	0.1285	0.2228
	250	-0.0235	0.1549	0.2064	0.3048	-0.0601	0.1104	0.1435	0.2277
Pets (Disjunctive)	50	<b>0.0540</b>	<b>0.1135</b>	<b>0.1910</b>	<b>0.2368</b>	-0.0213	-0.0009	<b>0.1418</b>	<b>0.1334</b>
	100	0.1654	0.2239	0.3177	0.3244	<b>0.0533</b>	<b>0.1239</b>	0.2193	0.2351
	150	0.2142	0.2662	0.3569	0.3649	0.861	0.1757	0.2585	0.2761
	200	0.2495	0.2896	0.3747	0.3837	0.0913	0.1980	0.2812	0.3027
	250	0.2587	0.3027	0.3843	0.3971	0.0998	0.2112	0.2961	0.3200
InCast (Aggregation)	50	-0.2888	-0.0601	-0.0695	<b>0.0419</b>	-0.3103	-0.1052	-0.0306	<b>0.0649</b>
	100	-0.1728	-0.0020	<b>0.0411</b>	0.1372	-0.1885	-0.0016	<b>0.0379</b>	0.1518
	150	-0.1280	<b>0.0230</b>	0.0933	0.1723	-0.1480	<b>0.0325</b>	0.0706	0.1801
	200	-0.0871	0.0376	0.1084	0.1839	-0.1314	0.0510	0.0871	0.1822
	250	-0.0746	0.0422	0.1201	0.1893	-0.1198	0.0653	0.0955	0.1853

Table C.10: Ratio Incremental gain results for Youtube case study.

# Appendix D

## Experiments Replicability

---

This appendix presents the configuration steps that are necessary to run the experiments conducted in this thesis. In Section D.1, we explain the configuration for experiments' replicability packages of Chapter 4. In addition, the configuration for experiments' replicability of Chapter 5 are exposed in Section D.2.

### D.1 Online Techniques experiments

This section covers all the configurations steps that are necessary to run the experiments with random, temporal and spatial approximations presented in Chapter 4. All the packages and source datasets are available on our Git repository [16]. These experiments must be run in an Eclipse environment by following the steps presented below:

1. Import Java projects into a workspace.
2. Create a folder named 'model' in *approximateTransformation/src/main/resources*.

3. Copy the source models files into the created folder. These files must be in *.graphml* format.
4. Open *config.properties* file located in *approximateTransformation/src/main/resources* that contains several configuration parameters. The configuration of this file is divided into six parts:
  - Configuration parameters: indicate the name of the source model to be loaded in the property 'file'.
  - Q1 - CreateAdCampaign parameters: set 'q1' value to true to execute **CreateAdCampaign** query. Besides, set a decimal value for 'prob1' from 0 to 1 to indicate the probability of the random approximation.
  - Q2 - UnpopularStock parameters: set 'q2' value to true to execute **UnpopularStock** query. Besides, set decimal value for 'prob2' from 0 to 1 to indicate the probability for the random approximation.
  - Q3 - RelatedProducts parameters: set 'q3Random' value to true and a decimal value for 'prob3' from 0 to 1 to test **RelatedProducts** query with random approximation. On the contrary, set 'q3Temporal' value to true and an entire value for n'window3' from 0 to 30 to test the temporal approximation.
  - Q4 - OlympicGamesTrending parameters: set 'q4Random' value to true and a decimal value for 'prob4' from 0 to 1 to test **OlympicGamesTrending** query with random approximation. On the contrary, set 'q4Spatial' value to true and select an entire value for 'hops4' from 100 to 900 to indicate the number of hops in order to test the spatial approximation.
  - Q5 - RecommendsPack parameters: set 'q5' value to true to execute **RecommendsPack** query. Select an entire value for 'hops5' from 1 to 3 to indicate the hops of the spatial approximation.
5. Once the configuration parameters are set, run the file *ApproximateTransformationApp.java* to start the experiment.

## D.2 Offline techniques experiments

This section presents the configuration steps that are necessary to run the experiments with the SDR algorithm exposed in Chapter 5. We have classified the configuration of these experiments in four parts. First, in Section D.2.1 we present the common configuration steps for all the experiments. Second, in Section D.2.2 we expose how to obtain a subgraph from a source graph using the SDR algorithm. Third, in Section D.2.3 we explain how to run a query over a graph or a subgraph stored in a *.graphml* file. Finally, in Section D.2.4 we present the steps to run the incremental algorithm over a graph. All the packages and source datasets are available on our Git repository [17]. Note that all the experiments are designed to be run in an Eclipse environment.

### D.2.1 Configuration and execution

In order to run the experiments for each case study, it is necessary to follow some previous steps:

1. Import Java projects into a workspace.
2. Copy the source models into the main folder of the project (folders *AmazonCase*, *ContentCase* or *YoutubeCase* depending on the case study).
3. Copy the *yt\_bb\_detection\_train.csv* file into the folder *YoutubeCase/src/main/resources*.

Note that the repository has one artifact for each case study and each artifact contains four runnable files in turn:

- *<CaseStudy>SubgraphApp.java*: it is used to obtain a subgraph from a source graph contained in a *.graphml* file using the SDR algorithm.
- *<CaseStudy>App.java*: it is used to run a query over a graph or subgraph contained in a *.graphml* file.
- *<CaseStudy>IncApp.java*: it is used to run the incremental SDR algorithm using a graph stored in a *.graphml* file with a specific value of  $\alpha$  and  $\beta$ .

- *<CaseStudy>DecApp.java*: it is also used to run the incremental SDR algorithm in a graph stored in a *.graphml* file. However, in this case  $\beta$  value represents the number of objects (orders, answers or segments for Amazon, Contest and Youtube case studies respectively) to be removed from the graph. Therefore, this file does not increase the graph size but decrease it.

Note that *<CaseStudy>* must be replaced by *AmazonCase*, *ContestCase* or *YoutubeCase* depending on the case study.

### D.2.2 Obtaining a subgraph

In order to obtain a subgraph from a graph stored in a *.graphml* file, it is necessary to follow the following steps:

1. Open *config.properties* file located in *<CaseStudy>/src/main/resources*. This file contains the configuration parameters to run the experiments. In this case, it is necessary to modify the following properties:
  - Write the source model file in the property ‘file’.
  - Change the property ‘nameWeights’ with a descriptive label. This property will be used to set the name of the *.graphml* file where the resulting subgraph will be stored. We recommend to set this property according to the name of the source model. In this way, the name of the subgraph file will have the following structure: *<Query-Name><nameWeights>.graphml*.
  - Set the property ‘query’ to indicate the query of the case study to be run. Notice that Amazon case study allows values from 1 to 7 whereas Contest and Youtube cases allow values from 1 to 6.
  - The rest of properties remain the same.
2. Once the configuration is selected, it is recommended to set the Java memory heap to 10G before running the file *<CaseStudy>SubgraphApp.java*.
3. After a few seconds, the program will create two files in the main folder of the project: (i) a *.graphml* file with the resulting subgraph and (ii) a *.log* file that contains the execution time (in milliseconds) taken for the SDR algorithm to calculate the subgraph.

### D.2.3 Running a query over a graph or a subgraph

In order to run a query over a graph or a subgraph stored in a *.graphml* file, it is necessary has to follow the steps presented below:

1. Open file *config.properties* located in *<CaseStudy>/src/main/resources*. In this case, the properties to be modified are the following:
  - Indicate the name of the *.graphml* file to be loaded in the property ‘file’. According to the experiment, this file may contain a graph or subgraph.
  - Change the property ‘nameWeights’ with a descriptive label. This property will be used to set the name of the *.log* file that contains the execution time results of the experiment. We recommend to set this property according to the name of the loaded *.graphml* file. The file name will have the following structure: *My-Log<CaseStudyName>File<nameWeights>.log*.
  - Change the property ‘query’ to indicate the number of query of the case study to be run. Notice that Amazon case study allows values from 1 to 7 whereas Contest and Youtube cases allow values from 1 to 6.
  - The rest of properties remain the same.
2. Once the configuration is selected, it is recommended to set the Java memory heap to 10G before running *<CaseStudy>App.java* file.
3. After a few seconds, the program will create the *.log* file with the execution times of six runs of the query over the *.graphml* file, previously indicated in the property ‘file’.

### D.2.4 Running the incremental SDR algorithm

The following steps are necessary to run the incremental SDR algorithm:

1. Open file *config.properties* located in *<CaseStudy>/src/main/resources* and modify the following properties:

- Indicate the name of the *.graphml* file to be loaded in the property ‘file’. According to the experiment, this file should contain a complete graph.
  - Change the property ‘nameWeights’ with a descriptive label. This property will be used to set the name of the *.log* file with the results of the experiment. We recommend to set this property according to the name of the source model file.
  - Change the property ‘query’ to indicate the query of the case study to be run. Notice that Amazon case study allows values from 1 to 7 whereas Contest and Youtube cases allow values from 1 to 6.
  - Change the property ‘records’ to select a  $\beta$  value.
  - Change the property ‘recordsQuery’ to select an  $\alpha$  value.
  - Set the property ‘incremental’ to ‘true’ to test the *SubG* execution. Otherwise, set this property to ‘false’ to test the CG execution.
2. Once the configuration is selected, it is recommended to set the Java memory heap to 10G before running *<CaseStudy>IncApp.java* or *<CaseStudy>DecApp.java* files, depending on whether the experiments pretends to add or delete records from the graph.
  3. After a few seconds, the program will start to show the results of each query execution in the console and the execution time of the experiment. When the program finishes, it will create a *.log* file with this information. In this case, the *.log* file name will have the following structure: *MyLog<CaseStudyName>FileIncremental<nameWeights>-<query>-<records>-<recordsQuery>.log*.

# Apéndice E

## Resumen

---

Actualmente una gran cantidad de datos se genera de forma diaria procedente de distintas fuentes, como son las redes sociales, las páginas de comercio electrónico o los buscadores web, entre otras. Un ejemplo concreto de estas fuentes es Twitter, donde las estadísticas calculan unos 6000 tweets por segundo [63]. Cada año, esta información crece de forma exponencial. De hecho, en 2016 se registraron alrededor de 6,5 zettabytes de datos en los centros de datos, mientras se estima que esta cifra subirá a 44 zettabytes en 2021. El drástico incremento en la cantidad de información producida por estas fuentes requiere un procesamiento eficiente de los flujos de datos en tiempo real para la toma de decisiones y la detección de situaciones de interés que, a su vez, requieren respuestas instantáneas. Un ejemplo de la importancia del procesamiento eficiente de grandes cantidades de flujos de información viene dado por el análisis realizado por el banco BBVA sobre el impacto económico del *Mobile World Congress* [19] de Barcelona en 2012. En este estudio se analizaron todas las transacciones con tarjetas de crédito efectuadas durante dos semanas. Otro ejemplo es la necesidad de analizar en tiempo real los flujos de datos procedentes de las redes sociales o weblogs para detectar posibles ataques terroristas [97, 130].



Sin embargo, debido a su tamaño y complejidad, las herramientas tradicionales no pueden procesar este tipo de datos. Por lo tanto, se hace necesario utilizar software y hardware cuya velocidad de procesamiento y capacidad de almacenamiento sean lo suficientemente potentes para administrar esta información.

Existen diversas propuestas para abordar el procesamiento de estas cantidades de información, que se basan en el hecho de que la mayoría de los datos que son procesados no son significativamente relevantes para la toma de decisiones, especialmente cuando se trabaja con grandes volúmenes de datos. En este sentido, los sistemas de procesamiento de flujos de datos se están volviendo populares, como por ejemplo Apache Spark [70] o Apache Kafka [68]. En la misma línea, el procesamiento de eventos complejos (CEP, por el término en inglés Complex Event Processing) es capaz de procesar y analizar flujos de información representados como una secuencia de eventos simples para obtener conclusiones de ellos, que son representadas como eventos complejos [38, 49, 77, 78]. Existen varios motores CEP y lenguajes de procesamiento de eventos (EPL), como el lenguaje Esper [48]. Estas tecnologías consideran que solo los datos más recientes son relevantes a la hora de obtener resultados. De esta manera, el procesamiento de la información no se lleva a cabo en el conjunto completo de datos, sino en un subconjunto que viene determinado por marcas de tiempo. De esta forma, los datos más antiguos son descartados, ya que se considera que no brindan información de interés. Este tipo de enfoque es muy útil en los casos en los que los eventos procedentes de las fuentes de información no están relacionados entre sí.

Sin embargo, en los sistemas reales la información está normalmente compuesta por datos conectados entre sí, formando estructuras de grafos. A modo de ejemplo, en Twitter los tweets son publicados por los usuarios que, a su vez, son seguidos por otros usuarios que también siguen a otros usuarios. Con respecto a estas estructuras de grafos, podemos distinguir dos tipos de información: persistente y transitoria. La primera se refiere a los datos almacenados en el sistema de forma persistente (por ejemplo, los usuarios, los productos o las tiendas). Por otra parte, la información transitoria hace referencia a los datos que se almacenan temporalmente (por ejemplo, los tweets, los pedidos o las transacciones bancarias) y se descartan después de un período de tiempo—es decir, la información transitoria *expira* con el tiempo. Las interconexiones entre estos datos también deben ser procesadas, lo que inevitablemente implica una disminución en el rendimiento del sistema [111].

Esto quiere decir que no basta con considerar únicamente técnicas basadas en la marca de tiempo de los datos a la hora de buscar mecanismos que seleccionen solo una parte de la información a procesar para la mejora del rendimiento [38, 44, 77]. Por el contrario, es necesario diseñar mecanismos que también seleccionen el subconjunto de la información con respecto a distintas características, como son las conexiones, el estado y la topología de la red. En la mayoría de los casos, será necesario descartar parte de la información para mejorar el rendimiento, es decir, realizar una aproximación de los datos. En consecuencia, la precisión de nuestros resultados podría verse comprometida. Sin embargo, muchas aplicaciones no necesitan resultados extremadamente precisos, ya que gestionan información no crítica, como los sistemas de recomendación en Facebook, Netflix o Amazon. En estos casos, el objetivo es encontrar un equilibrio adecuado entre el rendimiento de nuestras consultas y la precisión de sus resultados.

En la literatura, se pueden encontrar diferentes trabajos en esta línea. Por ejemplo, en un trabajo anterior [121] se introdujo el concepto de *Approximate Model Transformations* (AMT) para encontrar un equilibrio adecuado entre rendimiento y precisión de los resultados, en el contexto de las transformaciones de modelos. Para ello, se aplicaron técnicas de muestreo a un ejemplo de red de sensores inalámbricos con el objetivo de mostrar los efectos de seleccionar ciertos subconjuntos de los elementos. Sin embargo, en este caso, la información procesada no estaba compuesta por datos interconectados, sino que los elementos eran independientes entre sí. Otros trabajos emplean el procesamiento de consultas aproximadas (AQP, por el término inglés *Approximate Query Processing*) [32, 53, 75, 87], que pretenden obtener una respuesta aproximada que sea lo suficientemente precisa para extraer resultados válidos, pero mejorando el rendimiento. Sin embargo, en la mayoría de estos trabajos no se consideran flujos de datos ni información estructurada en grafos. Por esta razón, el procesamiento de flujos de datos estructurados en forma de grafos es aún un problema de investigación a abordar.

## E.1 Motivación y objetivos

Según lo expuesto al comienzo de este apéndice, esta tesis tiene como objetivo abordar el problema de la mejora del rendimiento en el procesamiento de grandes cantidades de información compuesta por datos estructurados. Para mejorar este

rendimiento, se debe reducir la cantidad de información a procesar, lo que pone en peligro la precisión de los resultados. Para abordar dicho problema, consideramos información estructurada en grafos, es decir, aquella que está compuesta por datos interconectados entre sí. Además, dado que la llegada de nueva información es constante, cuando hablamos de este tipo de información nos referimos a “flujos de datos estructurados en forma de grafos”. De esta forma, la principal pregunta de investigación de esta tesis es:

**RQ** *¿Es posible obtener una buena (u óptima) compensación entre el rendimiento y la pérdida de precisión cuando procesamos grandes cantidades de información?*

De esta pregunta de investigación se han derivado algunos objetivos que se exponen a continuación.

### E.1.1 Objetivos generales

Para dar respuesta a la pregunta de investigación de esta tesis, se derivaron tres objetivos generales con respecto a las técnicas para descartar la información y los errores que pueden surgir a partir de ellas.

- En primer lugar, nuestro objetivo es diseñar un mecanismo para seleccionar los datos relevantes que son necesarios para una consulta. Para ello, la información debe filtrarse de forma temporal y espacial.
- Para obtener resultados válidos, pretendemos definir los tipos de errores que pueden surgir al seleccionar solo una parte de la información origen a procesar y el significado de dichos errores.
- Una vez definidos los errores, queremos calcularlos y estudiarlos de acuerdo a varios parámetros como son la cantidad de información origen o la cantidad de información seleccionada para procesar.

### E.1.2 Objetivos específicos

Además, también se derivaron los siguientes objetivos específicos de la pregunta de investigación de esta tesis:

- Se necesita encontrar una plataforma de procesamiento que cumpla con nuestros requisitos a la hora de desarrollar un método eficiente para seleccionar solo un subconjunto de la información a procesar.
- Además, se necesita encontrar o desarrollar un lenguaje de consulta con una sintaxis simple y clara que permita realizar las consultas sobre los datos.
- Una vez desarrollado el método, se desea probar en diferentes casos de uso y diferentes tamaños de modelo que representen la información a procesar.

## E.2 Contribuciones

Tres aportaciones principales se han obtenido a lo largo del proceso de investigación de esta tesis, que se pueden resumir de la siguiente manera:

1. Un estudio comparativo entre distintas plataformas de procesamiento y los lenguajes específicos de dominio (DSL, por el término en inglés Domain Specific Language) más comunes que se utilizan para manejar grandes cantidades de datos. En este estudio, se tienen en cuenta el rendimiento de las consultas y la complejidad de su sintaxis. Medimos el rendimiento en términos de tiempo de ejecución, mientras que la complejidad de la sintaxis se mide en términos de número de caracteres, operadores y variables internas. Este estudio se explica con más detalle en la sección E.3.
2. Tres métodos en línea con el procesamiento para descartar información considerada no relevante para una consulta determinada. Estos métodos se basan en técnicas de procesamiento de consultas aproximadas (AQP) y seleccionan la información de acuerdo a rangos temporales y espaciales y parámetros aleatorios. Para medir la pérdida de precisión derivada de estos métodos definimos el error de transformación, que se da en términos de exactitud, exhaustividad y precisión. Un resumen de estos métodos se expone en la sección E.4.
3. Un algoritmo, llamado algoritmo SDR, basado en técnicas de preprocesamiento de AQP, que selecciona un subconjunto de la información de origen que es considerada relevante de acuerdo con los patrones que se pueden

encontrar en una consulta determinada. Nuestros experimentos empíricos muestran que la precisión de los resultados no se ve afectada al aplicar este algoritmo, ya que considera todos los datos que son importantes para el procesamiento. Por este motivo, no es necesario calcular los errores producidos al ejecutar dicho algoritmo. Un resumen de este algoritmo se expone en la sección E.5.

### E.3 Comparación y evaluación del rendimiento de las plataformas de procesamiento

En esta sección resumimos el estudio comparativo de siete de las plataformas más populares para el procesamiento de grandes cantidades de datos, además de los lenguajes de consultas empleados para trabajar con ellas.

#### E.3.1 Plataformas de procesamiento y lenguajes de consulta

Como se expone al comienzo de este apéndice, el tratamiento de flujos de información implica el procesamiento de los datos en tiempo real, lo que requiere de una baja latencia. Es por esto que, con el objetivo de seleccionar la tecnología que más se adapte a nuestros requisitos, se han analizado siete plataformas diseñadas para trabajar con grandes cantidades de datos. Estas plataformas incluyen cinco bases de datos de grafos (Neo4j [89], JanusGraph [64], OrientDB [29], TinkerGraph [118] y Memgraph [83]), una base de datos SQL distribuida (CrateDB [35]) y un paquete para trabajar con grafos distribuidos proporcionado por Apache Spark (GraphFrames [106]). Sus características principales pueden observarse en la Tabla E.1.

Por otro lado, cuatro lenguajes de consulta empleados con estas plataformas se han analizado con el objetivo de escoger el que posea una sintaxis más clara para trabajar con grafos. Estos lenguajes son Gremlin [6], Cypher [88], SQL y el DSL diseñado para trabajar con GraphFrames [107].

El objetivo de este estudio es encontrar la mejor combinación de plataforma de procesamiento y lenguaje de consulta que se adapte a los siguientes requisitos: (i) permita realizar consultas y actualizar la información lo más rápido posible para dar respuestas en tiempo real, (ii) pueda hacer frente a información estructurada

Platform	Distributed	In-memory	Disk	Updatable	Query languages
Neo4j	No	No	Sí	Sí	Cypher
JanusGraph	Sí	Sí	Sí	Sí	Gremlin
OrientDB	Sí	Sí	Sí	Sí	Gremlin, SQL
TinkerGraph	No	Sí	Sí	Sí	Cypher, Gremlin
Memgraph	Sí	Sí	Sí	Sí	Cypher
CrateDB	Sí	No	Sí	Sí	SQL
GraphFrames	Sí	Sí	No	No	GraphFrames DSL

Cuadro E.1: Plataformas de procesamiento utilizadas en los experimentos del estudio del rendimiento

en forma de grafos, y (iii) con un lenguaje que proporcione una sintaxis clara para poder estudiar el tipo de consulta a ejecutar sobre los datos. Para ello, se han probado las tecnologías propuestas en dos casos de estudio con información estructurada en grafos. En primer lugar, en el dominio de las redes sociales, un sistema que trabaja con información de Twitter y Flickr y la relaciona por medio de una clase `Hashtag`. Después, un caso de estudio extraído de un trabajo de Szárnyas et al. [111], que modela un sistema de seguridad de un ferrocarril. Ambos metamodelos están representados en las Figuras E.1 y E.2, respectivamente. Cabe resaltar que las consultas elegidas para estos ejemplos tienen como resultado la modificación del grafo origen, por medio de la creación, actualización o eliminación de elementos. Además, los experimentos se realizan para distintos tamaños de modelo comprendidos entre 3K y 14 millones de elementos.

#### E.3.2 Métodos de medición

Para realizar el análisis en los casos de estudio propuestos, se han realizado cuatro conjuntos de experimentos. Estos experimentos se han diseñado de acuerdo a dos características:

- Ejecuciones en paralelo vs individuales. Para cada caso de estudio y plataforma de procesamiento, se han diseñado dos tipos de ejecuciones para las consultas. En primer lugar, una ejecución individual sin que ninguna otra consulta esté ejecutándose al mismo tiempo en el sistema. En segundo lugar, se han ejecutado todas las consultas a la vez en paralelo.

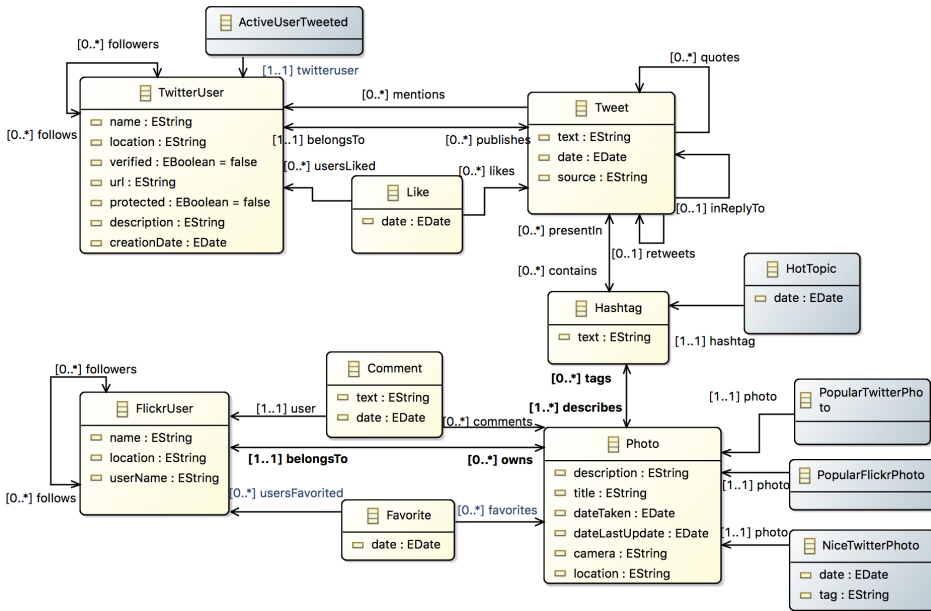


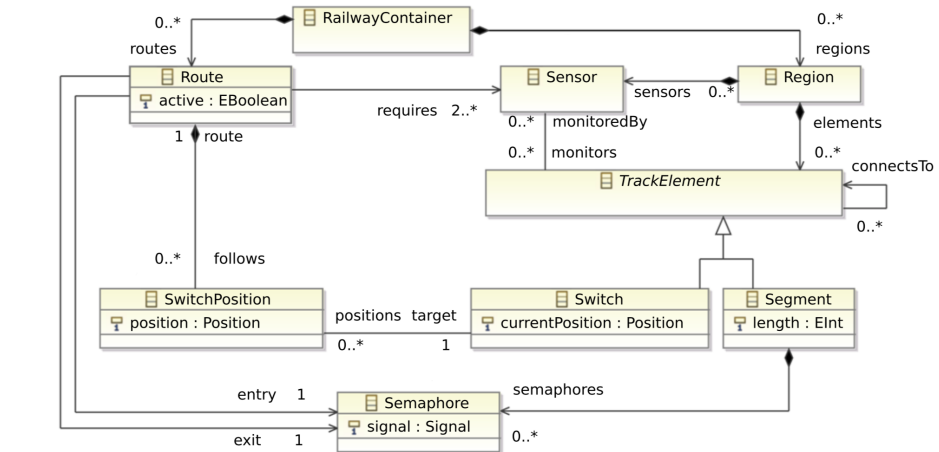
Figura E.1: Metamodelo conjunto de Twitter y Flickr.

- Consultas con y sin efecto. Dado que las consultas tienen como consecuencia la modificación del grafo inicial, se pueden ejecutar de dos formas diferentes. En primer lugar, se ejecutaron las consultas devolviendo los elementos filtrados sin ningún efecto sobre el grafo, es decir, eliminando la parte de la consulta que se encarga de modificar el grafo. En segundo lugar, se ejecutaron generando sus respectivos efectos sobre grafo.

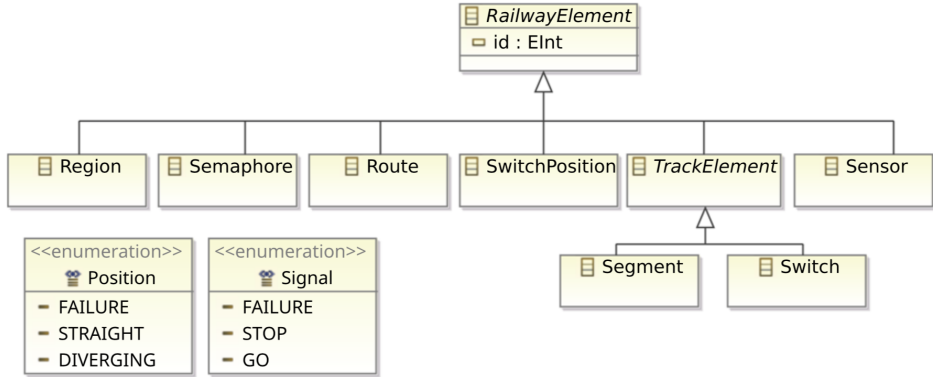
De esta manera, pueden distinguirse cuatro conjuntos de experimentos: (i) ejecuciones individuales de consultas sin efecto, (ii) ejecuciones individuales de consultas con efecto, (iii) ejecuciones paralelas de consultas sin efecto y (iv) ejecuciones paralelas de consultas con efecto sobre el grafo.

### E.3.3 Parámetros de estudio

Para estudiar la latencia de las plataformas de procesamiento, tanto en el caso de consultas con efecto sobre el grafo como consultas que únicamente devuelven información, se realizaron medidas del tiempo de ejecución en milisegundos en todos los experimentos llevados a cabo. Además, se estudió la expresividad de los



(a) Jerarquía de contención y referencias



(b) Relaciones de supertipo

Figura E.2: Metamodelo de TrainBenchmark [111]

lenguajes en las consultas de acuerdo a tres parámetros, como son el número de caracteres, operadores y variables internas. En esta sección se expone una visión general de los resultados.

### Latencia de consulta

Nuestra hipótesis es que todas las tecnologías permiten obtener tiempos de ejecución que las hacen adecuadas para el procesamiento en tiempo real. Sin embargo, nos interesa conocer un valor medio aproximado de los tiempos de ejecución de las consultas en modelos grandes con varios millones de elementos, con



el fin de comparar todas las tecnologías. En la Tabla E.2 se muestra un resumen de los tiempos de ejecución para cada plataforma de procesamiento con respecto al tamaño del modelo. Estos valores se han obtenido calculando el promedio de los tiempos de ejecución de todas las consultas para cada tamaño de modelo en ambos casos de estudio. En esta tabla, las filas 3 a 9 y 19 a 25 muestran los tiempos de ejecución promedio para los ejemplos de TwitterFlickr y TrainBenchmark con ejecuciones individuales, mientras que las filas 10 a 16 y 26 a 32 muestran los tiempos de ejecución promedio para ejemplos de TwitterFlickr y TrainBenchmark con ejecuciones paralelas, respectivamente. Algunos valores no se muestran en esta tabla debido a que estos experimentos provocaron errores de desbordamiento de memoria a causa del tamaño del modelo origen.

De los resultados obtenidos observamos que CrateDB presenta los tiempos de ejecución más bajos para casi todas las ejecuciones individuales de las consultas del ejemplo de TwitterFlickr. Sin embargo, también presenta los tiempos de ejecución más altos en el caso de TrainBenchmark. Esto es debido a que las consultas de TrainBenchmark contienen un mayor número de accesos a las relaciones del grafo, mientras que en el caso de TwitterFlickr las consultas se componen principalmente de operadores de agregación. De esta forma, dado que CrateDB emplea SQL, que proporciona funciones optimizadas para filtrar por operadores de agregación, CrateDB obtiene un mayor rendimiento en las consultas de TwitterFlickr. Además, en la Tabla E.2 se puede observar que las ejecuciones paralelas tienen un peor promedio del tiempo de ejecución para CrateDB que para el resto de plataformas. Esto es debido a que existe una gran interferencia entre consultas cuando se ejecutan de forma simultánea en el sistema.

Por otro lado, se puede observar que Memgraph y TinkerGraph presentan tiempos de ejecución más bajos que Neo4j y JanusGraph en la mayoría de los casos. Esto se debe a que su implementación es en memoria, a diferencia de Neo4j y JanusGraph que almacenan la información en disco. Sin embargo, se puede observar que OrientDB, a pesar de ser en memoria, presenta tiempos de ejecución mayores que JanusGraph y Neo4j en general. Esto es debido a que el rendimiento de OrientDB empeora a medida que se utiliza un mayor número de operadores de agregación en las consultas. Para aclarar esto, si observamos en la Tabla E.2 los resultados del ejemplo de TwitterFlickr, podemos ver que OrientDB presenta los mayores promedios de tiempo de ejecución y una curva de crecimiento más

### E.3 Comparación y evaluación del rendimiento de las plataformas de procesamiento

Experiment	Tech	Models					
		2M	2M5	3M	4M	6M5	14M
TF Individual	TinkerGraph	3,061	3,591	5,044	7,193	10,608	22,468
	Neo4j	25,422	15,922	21,257	27,667	34,787	263,737
	JanusGraph	36,182	40,652	52,754	76,643	122,244	-
	OrientDB	133,314	171,452	267,617	478,438	701,625	1,402,423
	CrateDB	448	575	645	1,039	1,446	3,050
	Memgraph	421	1,232	1,177	2,842	4,623	10,492
	GraphFrames	17,233	17,466	18,567	21,881	26,645	39,429
TF Paralelo	TinkerGraph	3,499	3,994	4,747	6,910	10,969	22,550
	Neo4j	26,595	43,230	20,702	28,393	36,867	373,875
	JanusGraph	55,906	58,372	80,972	103,398	189,243	-
	OrientDB	169,732	191,531	398,669	-	-	-
	CrateDB	2,316	2,531	3,260	4,755	5,930	11,556
	Memgraph	428	1,298	1,231	3,052	5,226	11,572
	GraphFrames	71,467	76,936	81,621	92,317	113,373	169,070
Experiment	Tech	Models					
		420K	820K	1M5	3M	6M5	13M
TB Individual	TinkerGraph	151	316	584	1,270	2,575	4,821
	Neo4j	4,930	9,737	4,158	7,885	10,742	21,692
	JanusGraph	6,833	12,558	37,598	72,843	142,712	-
	OrientDB	3,090	5,383	11,265	34,434	113,140	-
	CrateDB	9,507	56,867	196,715	803,027	3,193,544	-
	Memgraph	829	1,589	3,212	6,502	13,523	26,886
	GraphFrames	6,811	7,824	10,242	66,310	-	-
TB Paralelo	TinkerGraph	185	375	681	1,741	4,624	8,166
	Neo4j	5,943	11,689	5,307	9,663	13,252	25,175
	JanusGraph	9,406	20,510	51,213	115,522	277,571	-
	OrientDB	3,572	9,245	17,569	52,835	82,502	-
	CrateDB	29,840	213,914	1,060,149	3,285,652	14,330,287	-
	Memgraph	960	1,893	3,732	7,592	15,572	33,241
	GraphFrames	30,417	38,632	52,880	376,672	-	-

Cuadro E.2: Promedios de tiempo de ejecución (ms) según el tamaño de modelo

pronunciada para el ejemplo de TrainBenchmark que el resto de tecnologías, a excepción de CrateDB. Esto significa que la penalización que proviene de las consultas con operadores de agregación hace que incluso las implementaciones en disco superen a OrientDB. Con respecto a las otras dos implementaciones de bases de datos de grafos en memoria, Memgraph supera a TinkerGraph para el ejemplo de TwitterFlickr, mientras que TinkerGraph supera a Memgraph para el ejemplo de TrainBenchmark. Finalmente, podemos observar que las consultas de Neo4j son más rápidas que las consultas de JanusGraph, lo que significa que Neo4j supera a JanusGraph. Además, observemos cómo los tiempos de ejecución obtenidos con Neo4j suelen ser más bajos que los tiempos de ejecución obtenidos con GraphFrames para ejecuciones individuales y en paralelo. Por lo tanto, Neo4j supera a GraphFrames y JanusGraph en la mayoría de los casos.

### Latencia de actualización

En este caso, nuestra hipótesis es que la modificación del grafo como consecuencia de la consulta no implica una gran sobrecarga con respecto a todo el tiempo de ejecución de la misma, lo que significa que las tecnologías estudiadas podrían utilizarse para el procesamiento en tiempo real. La Tabla E.3 muestra un resumen de los tiempos de ejecución para cada plataforma de procesamiento con respecto al tamaño del modelo cuando las consultas modifican el grafo. Nótese que para algunos tamaños de modelo y casos de estudio, algunos valores no se muestran en la tabla. Esto se debe a que los experimentos en paralelo presentaban problemas de concurrencia al modificar el grafo con varias consultas en paralelo (por ejemplo, en el caso de las consultas con Neo4j y Memgraph). Además, no hemos ejecutado ningún experimento con consultas de OrientDB en paralelo para este caso, ya que esta tecnología no admite multi-threading cuando se usa el lenguaje Gremlin [28]. Con respecto a las consultas en experimentos individuales, algunas de ellas tardaban demasiado tiempo y la línea de tendencia ya puede inferirse con los resultados presentados, por lo que no es necesario terminar de calcular estos experimentos.

En primer lugar, se puede observar cómo Neo4j y JanusGraph presentan los tiempos de ejecución más altos para todos los casos del ejemplo de TwitterFlickr, excepto OrientDB. La razón es que almacenan el grafo en el disco y acceder al disco es más costoso que acceder a la memoria. En cuanto al ejemplo de

### E.3 Comparación y evaluación del rendimiento de las plataformas de procesamiento

Experiment	Tech	Models					
		2M	2M5	3M	4M	6M5	14M
TF Individual	TinkerGraph	3,186	4,227	4,768	7,260	12,175	25,162
	Neo4j	25,870	40,453	20,559	29,667	38,319	370,804
	JanusGraph	38,913	49,455	53,177	81,612	117,246	-
	OrientDB	135,967	174,957	230,198	448,458	664,727	1,636,927
	CrateDB	516	720	773	1,268	1,609	3,443
	Memgraph	413	855	1,009	1,904	2,954	7,991
TF Paralelo	TinkerGraph	3,378	4,150	4,845	7,201	11,418	25,574
	Neo4j	25,768	42,180	22,433	-	-	-
	JanusGraph	51,585	67,487	77,478	135,398	218,998	-
	OrientDB	-	-	-	-	-	-
	CrateDB	515	710	790	1,232	1,825	3,535
	Memgraph	439	957	-	-	-	-
Experiment	Tech	Models					
		420K	820K	1M5	3M	6M5	13M
TB Individual	TinkerGraph	152	290	615	1,187	2,632	5,253
	Neo4j	88,410	397,606	1,637,751	-	-	-
	JanusGraph	7,442	14,074	32,190	61,800	128,325	-
	OrientDB	2,684	5,666	14,177	42,713	146,364	1,186,161
	CrateDB	13,447	61,167	226,460	936,874	-	-
	Memgraph	140,121	588,041	-	-	-	-
TB Paralelo	TinkerGraph	150	275	685	-	4,566	8,098
	Neo4j	99,823	405,467	-	-	-	-
	JanusGraph	11,367	17,678	56,544	112,923	272,512	-
	OrientDB	-	-	-	-	-	-
	CrateDB	35,389	106,603	227,880	954,774	-	-
	Memgraph	-	-	-	-	-	-

Cuadro E.3: Promedios de tiempo de ejecución (ms) según el tamaño del modelo (con efecto sobre el grafo)

TrainBenchmark, CrateDB presenta los tiempos de ejecución más altos, ya que no está diseñado para trabajar con grafos y las consultas de TrainBenchmark están compuestas principalmente por filtros que atraviesan relaciones del grafo, como comentamos anteriormente. Sin embargo, podemos observar que Neo4j y OrientDB obtienen el mayor tiempo promedio de ejecución en ambos casos de estudio y presentan problemas de concurrencia al modificar el grafo con consultas paralelas. En este sentido, Memgraph también presenta problemas de concurrencia para ambos casos de estudio al modificar el grafo en paralelo. Finalmente, los resultados de TinkerGraph y JanusGraph muestran que las consultas con efecto sobre el grafo se comportan de manera similar a las consultas sin efecto (compárense Tablas E.2 y E.3). Sin embargo, los tiempos de ejecución de las consultas TinkerGraph son más rápidos que las consultas de JanusGraph.

En general, podemos ver que los promedios de tiempo de ejecución del ejemplo de TwitterFlickr son mucho más bajos que los promedios de tiempos de ejecución del ejemplo de TrainBenchmark para los experimentos de Neo4j y Memgraph. La razón es que el efecto que tienen las consultas de TwitterFlickr sobre el grafo es la agregación de nuevos objetos y relaciones, mientras que las consultas de TrainBenchmark generalmente actualizan la información ya existente. Por esta razón, podemos afirmar que Neo4j y Memgraph obtienen un mejor rendimiento al crear nuevos elementos que al actualizar la información existente. Comparando las Tablas E.2 y E.3, se puede ver que CrateDB, TinkerGraph y JanusGraph—y también las consultas de OrientDB en ejecución individual—no presentan una alta disminución de rendimiento al actualizar el grafo con respecto a la obtención de los resultados con consultas sin efecto. Sin embargo, TinkerGraph obtiene menores tiempos de ejecución.

### Expresividad del lenguaje

Con respecto a la expresividad del lenguaje la hipótesis es que los DSL para bases de datos de grafos proporcionan una mayor expresividad que el resto de DSL estudiados. Sin embargo, nos interesa estudiar qué lenguaje permite escribir consultas de forma más sencilla. Para ello medimos el número de operadores, caracteres y variables internas de las consultas. Un ejemplo de estos resultados para el caso de TwitterFlickr se muestra en la Tabla E.4. Esta se divide en cuatro

columnas. La primera columna indica la plataforma utilizada para ejecutar la consulta, la segunda columna muestra el nombre de la consulta, la tercera columna representa las características de las consultas con efecto sobre el grafo y la cuarta columna representa las características de las consultas sin efecto. Las columnas tercera y cuarta se dividen, a su vez, en tres subcolumnas, que muestran el número de operadores, caracteres y variables internas, respectivamente. Además, las filas en negrita representan el promedio de estos parámetros para cada plataforma. Cabe resaltar que las consultas de JanusGraph, TinkerGraph y OrientDB tienen la misma sintaxis ya que usan el lenguaje Gremlin. Además, aunque Neo4j y Memgraph usan Cypher como DSL, Memgraph no permite todos los métodos incluidos en el lenguaje Cypher, por ello diferenciamos entre Cypher para Memgraph y para Neo4j.

En los resultados se puede observar que Gremlin reduce significativamente el número de operadores y caracteres al consultar el grafo con respecto a la actualización del mismo como resultado de una consulta. Cypher no obtiene cambios importantes al consultar el grafo con respecto a su actualización y presenta resultados similares para las plataformas Neo4j y Memgraph.

Por otro lado, incluso teniendo en cuenta que GraphFrames DSL está diseñado para trabajar con grafos, proporciona una sintaxis más compleja que Gremlin y Cypher. Además, esta plataforma no permite la actualización del grafo. Dado que SQL no está diseñado para trabajar con grafos, proporciona menor intuitividad que Cypher y Gremlin. Luego, Gremlin y Cypher permiten consultas sobre el grafo de una manera más sencilla que SQL o GraphFrames DSL, ya que presentan un menor número de caracteres, variables y operadores. Por lo tanto, los DSL para bases de datos de grafos proporcionan más intuición y expresividad que el resto de DSL.

## E.4 Mejora del rendimiento empleando técnicas en línea con el procesamiento

En esta sección se resumen las técnicas desarrolladas para obtener una mejora de rendimiento en el procesamiento de flujos de datos estructurados, empleando técnicas AQP que se ejecutan en línea con dicho procesamiento. También se resumen los tipos de error derivados de estas técnicas en términos de precisión, exhaustividad

Tech	Query	Efecto			Sin Efecto		
		Op	Char	Var	Op	Char	Var
TinkerGraph, JanusGraph and OrientDB (Gremlin)	HotTopic	15	202	2	8	89	0
	PopularTwitterPhoto	24	328	3	13	144	0
	PopularFlickrPhoto	21	284	2	14	149	0
	NiceTwitterPhoto	46	593	5	35	412	2
	ActiveUserTweeted	34	446	3	23	258	0
	<b>AVG</b>	<b>28</b>	<b>370.6</b>	<b>3</b>	<b>18.6</b>	<b>210.4</b>	<b>0.4</b>
Neo4j (Cypher)	HotTopic	12	222	6	11	177	6
	PopularTwitterPhoto	15	275	6	14	199	6
	PopularFlickrPhoto	16	281	5	15	222	5
	NiceTwitterPhoto	28	538	11	27	465	11
	ActiveUserTweeted	29	535	8	28	463	8
	<b>AVG</b>	<b>20</b>	<b>370.2</b>	<b>7.2</b>	<b>19</b>	<b>305.2</b>	<b>7.2</b>
CrateDB (SQL)	HotTopic	17	476	8	16	418	8
	PopularTwitterPhoto	12	373	3	11	294	3
	PopularFlickrPhoto	14	360	3	13	292	3
	NiceTwitterPhoto	29	731	8	28	655	8
	ActiveUserTweeted	31	651	6	30	563	6
	<b>AVG</b>	<b>20.6</b>	<b>518.2</b>	<b>5.6</b>	<b>19.6</b>	<b>444.4</b>	<b>5.6</b>
Memgraph (Cypher)	HotTopic	12	224	6	11	178	6
	PopularTwitterPhoto	15	276	6	14	200	6
	PopularFlickrPhoto	16	281	5	15	222	5
	NiceTwitterPhoto	28	525	11	27	452	11
	ActiveUserTweeted	29	535	8	28	463	8
	<b>AVG</b>	<b>20</b>	<b>368.2</b>	<b>7.2</b>	<b>19</b>	<b>303</b>	<b>7.2</b>
GraphFrames	HotTopic	-	-	-	11	151	4
	PopularTwitterPhoto	-	-	-	38	745	10
	PopularFlickrPhoto	-	-	-	30	480	8
	NiceTwitterPhoto	-	-	-	74	1,517	18
	ActiveUserTweeted	-	-	-	62	1,273	15
	<b>AVG</b>	-	-	-	<b>43</b>	<b>833.2</b>	<b>11</b>

Cuadro E.4: Resumen de las características de cada DSL para el caso de estudio de TwitterFlickr

y exactitud.

### E.4.1 Técnicas AQP en línea

El objetivo de esta tesis es seleccionar el subconjunto de datos relevantes que permita obtener resultados válidos con las consultas realizadas. De esta manera, aplicar AQP a nuestros datos puede resultar en una pérdida de precisión, ya que no todos los elementos y conexiones serán considerados en la aproximación. Por el contrario, tratar de tener en cuenta toda la información relevante puede resultar en un tiempo de respuesta inaceptable o en la necesidad de contar con más recursos (por ejemplo, memoria) de los que se dispone. Por eso, el objetivo es encontrar un equilibrio adecuado entre el rendimiento de las consultas y la precisión de sus resultados. Para ello, se necesita responder a dos preguntas: (a) ¿cómo seleccionar el subconjunto de datos que es relevante para una consulta determinada?; y (b) ¿cómo estimar el error que cometemos al descartar algunos de los datos de entrada? Este problema puede aplicarse en aquellos sistemas que tratan con grandes cantidades de datos y no necesitan resultados extremadamente precisos. Un ejemplo de esto es el sistema de recomendación de Amazon, cuyo metamodelo se presenta de forma simplificada en la Figura E.3.

Con respecto al procesamiento de consultas aproximadas, se utilizan dos tipos de técnicas de AQP. Estas son las técnicas en línea con el procesamiento y las técnicas de preprocesamiento. Las primeras se realizan al momento de procesar los datos, mientras que las segundas realizan un cálculo inicial antes de la ejecución de las consultas y almacenan esta información para ser utilizada en el procesamiento. En esta sección abordamos las técnicas AQP en línea con el procesamiento. En concreto exploramos tres posibles aproximaciones, que son las aproximaciones aleatorias, temporales y espaciales.

#### Aproximaciones espaciales

En nuestro enfoque, los datos estructurados en grafos implican que los objetos están vinculados entre sí a través de diferentes tipos de relaciones. De esta manera, podemos navegar por un modelo comenzando en un objeto y atravesando las relaciones existentes. Para aclarar esto, definimos el concepto de *salto*. Un salto es la navegación de un objeto a otro por la relación que los vincula. Por ejemplo,



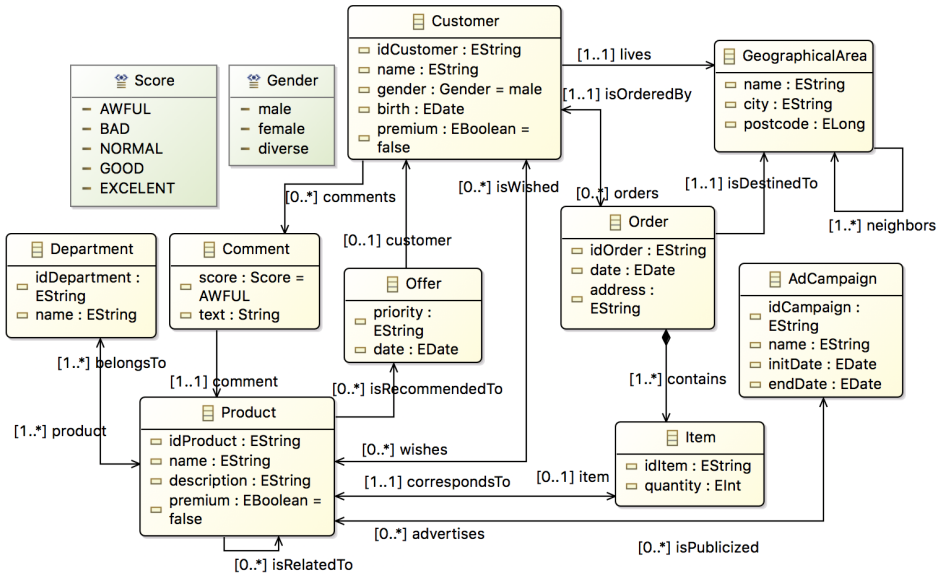


Figura E.3: Metamodelo de ejemplo de Amazon.

observando el metamodelo de la Figura E.3, a partir de un pedido podemos determinar el área geográfica a la que está destinado en un salto, navegando por la relación *isDestinedTo*. Además, los objetos se pueden conectar a otros objetos del mismo tipo. Por ejemplo, desde un área geográfica podemos llegar, en un salto, a todas sus áreas geográficas vecinas, a través de la relación de *neighbors*. De esta forma, en dos saltos, podemos llegar a todas las áreas geográficas que sean vecinas de sus vecinas, etc. A partir de esto, podemos obtener ventanas espaciales partiendo de un objeto y considerando otros objetos alcanzables en  $n$  saltos.

### Aproximaciones temporales

Dado que los datos entrantes se etiquetan generalmente con una marca de tiempo, que indica el momento en el que ocurren, y los flujos de datos pueden considerarse infinitos (tengamos en cuenta, por ejemplo, toda la información almacenada en Facebook durante su vida útil), podemos crear ventanas temporales filtradas por la marca de tiempo de los datos. Generalmente, la consulta determinará una ventana temporal, ya que las consultas se suelen centrar en un intervalo de tiempo específico. La idea es reducir el modelo completo seleccionando un subconjunto del

modelo que viene determinado por la ventana temporal. Sin embargo, podemos estar interesados en aplicar una aproximación temporal adicional al modelo origen para obtener un modelo aproximado. De esta manera, partiendo de una ventana temporal  $(t_i, t_e)$  de tamaño  $N$  donde  $t_i$  es el tiempo inicial y  $t_e$  es el tiempo final de la ventana, es posible procesar solo un subperíodo de tiempo  $(t_{ai}, t_{ae})$  con tamaño  $n$  donde  $(t_{ai}, t_{ae}) \subseteq (t_i, t_e)$  y  $n < N$ .

### Aproximaciones aleatorias

Otra forma de obtener modelos aproximados es aplicando técnicas de muestreo aleatorio, lo que implica que la decisión sobre qué elementos del modelo completo conformarán el modelo aproximado se toma al azar. Por ejemplo, podemos asignar una probabilidad a cada elemento del modelo para que se incluya en el modelo aproximado. Además de esto, también se pueden hacer aproximaciones por tipo de elemento (por ejemplo, se podría determinar que solo el 30 % de los pedidos se incluyan en el modelo aproximado). En el caso de esta tesis, se ha implementado la primera propuesta. Por supuesto, las aproximaciones aleatorias se pueden combinar tanto con aproximaciones temporales como con aproximaciones espaciales.

### E.4.2 Medidas de precisión

Dado que estamos sacrificando precisión para obtener rendimiento, un aspecto muy importante de nuestro enfoque es poder medir ambos. Consideramos el rendimiento en términos de tiempo de ejecución de las consultas. Con respecto a la precisión, empleamos los términos de *precisión*, *exhaustividad* y *exactitud* [81]. Estas tres medidas se definen mediante fórmulas que incluyen los conceptos de Verdaderos Positivos (TP, por el término en inglés True Positive), Falsos Positivos (FP), Falsos Negativos (FN) y Verdaderos Negativos (TN, por el término en inglés True Negative). Según esto, las tres medidas de precisión se pueden calcular de la siguiente manera [81]:

- Exactitud: en nuestro contexto, describe el efecto de FP y FN cuando se ejecutan consultas en el modelo aproximado. Se calcula de la siguiente manera:  $Exactitud = (TP + TN) / (TP + TN + FN + FP)$ .

- **Precisión:** esta medida es útil para determinar qué tan preciso es el modelo cuando los FP son costosos. Por ejemplo, en la detección de correo no deseado, un FP puede provocar la pérdida de información importante cuando un correo electrónico que no es spam se identifica como spam. Se calcula de la siguiente manera:  $Precisin = TP/(TP + FP)$ .
- **Exhaustividad:** esta medida calcula la precisión del modelo cuando los FN son costosos. Por ejemplo, un FN en la detección de enfermedades puede tener consecuencias catastróficas en la vida del paciente. Se calcula de la siguiente manera:  $Exhaustividad = TP/(TP + FN)$ .

### E.4.3 Evolución del rendimiento y precisión con modelos aproximados

Lógicamente, ejecutar las consultas en los modelos aproximados es más rápido que ejecutarlas en el modelo completo. Sin embargo, estamos interesados en comprobar cómo es la ganancia de rendimiento (entendido en términos de tiempo de ejecución), dependiendo del tamaño del modelo completo y la distribución de los datos. Esto conlleva, además, el cálculo de los tres parámetros elegidos para medir el error de los resultados, explicados previamente. Para ello, hemos llevado a cabo una serie de experimentos empleando Gremlin sobre distintos tamaños de modelos del ejemplo mostrado en la Figura E.3 (entre 2 y 16 millones elementos) con los tres tipos de aproximaciones. Además, hemos agrupado los modelos fuente en dos lotes. En primer lugar, un lote A en el que los datos se distribuyen uniformemente a lo largo un mes. En segundo lugar, un lote B donde los datos se centran principalmente en la primera semana.

En todos los experimentos se ha medido el tiempo de ejecución y el número de elementos devueltos por la consulta sobre distintos tamaños de modelos aproximados, cada uno de ellos de mayor tamaño que el anterior hasta alcanzar el tamaño del modelo completo. Para las aproximaciones aleatorias consideramos tamaños de modelo aproximado desde el 10 % al 100 % del total con incrementos de 10 % entre uno y otro. En el caso de las aproximaciones temporales consideramos tamaños del modelo que comprenden de 3 a 30 días con incrementos de 3 días entre uno y otro. Finalmente, las aproximaciones espaciales comprenden modelos aproximados con ventanas espaciales de 100 a 900 saltos con incrementos de 100 saltos entre uno

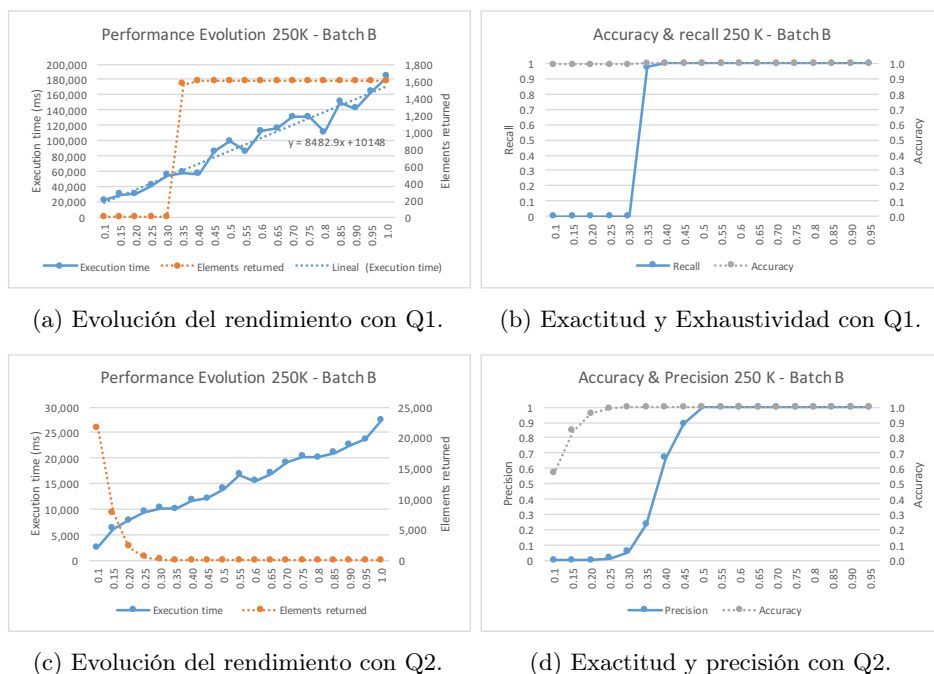


Figura E.4: Exactitud, Precisión y Exhaustividad con aproximaciones aleatorias.

y otro. En las figuras E.4, E.5, E.6 y E.7 se pueden observar algunos resultados comparativos del tiempo de ejecución (gráficas a y c) y el error en términos de precisión, exactitud y exhaustividad (gráficas b y d) obtenidos con los tres tipos de aproximaciones. Para todas las gráficas se representan los siguientes parámetros: modelo y lote (título de la gráfica) y tamaño del modelo aproximado considerados de acuerdo a la aproximación empleada (eje X). Además las gráficas a y c de todas las figuras representan el tiempo de ejecución con una línea azul (eje Y izquierdo) y el número de elementos devueltos por la consulta con una línea naranja (eje Y derecho). Finalmente, en las gráficas b y d viene representado el valor de precisión o exhaustividad con una línea azul (eje Y izquierdo) y el valor de exactitud en gris (eje Y derecho).

En todas las gráficas se puede observar que cuanto menor es el modelo aproximado considerado, más rápido es el tiempo de ejecución. Esto significa que el tiempo que emplea el motor Gremlin para filtrar los datos que componen nuestros modelos aproximados compensa, ya que el motor ejecuta las consultas más rápido con modelos más pequeños.

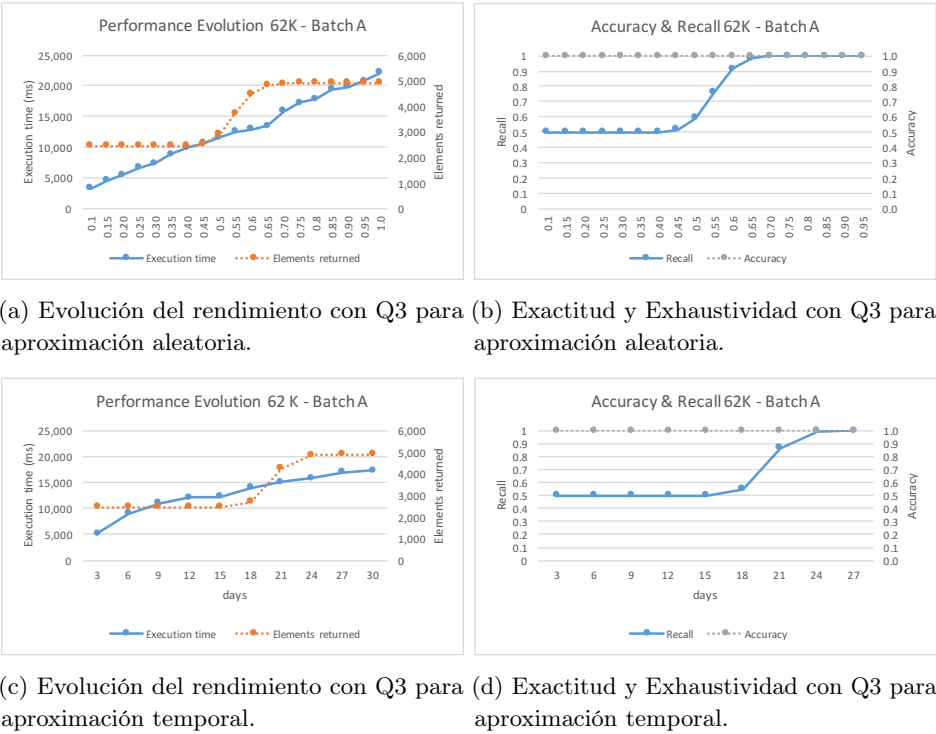
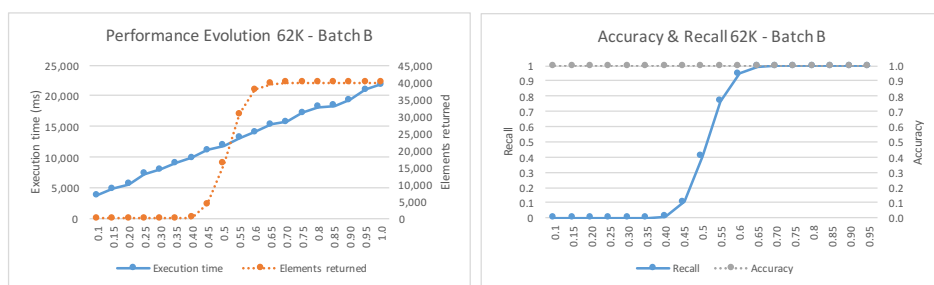


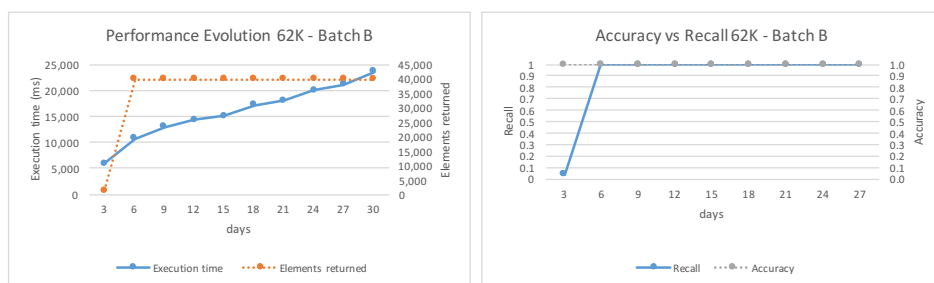
Figura E.5: Comparación entre aproximaciones temporales y aleatorias con datos uniformemente distribuidos.

Por otro lado, considerando el tipo de aproximación, vemos que en las aproximaciones aleatorias el tiempo de ejecución aumenta linealmente a medida que crece el tamaño del modelo aproximado independientemente de si la consulta se ejecuta en el lote A o B (obsérvense como ejemplo las figuras E.5a y E.6a). Con respecto a las aproximaciones temporales podemos ver que los tiempos de ejecución no presentan mucha variación de acuerdo a la distribución de los datos (figuras E.5c y E.6c), y, además, ese tiempo de ejecución también crece linealmente. Sin embargo, en las aproximaciones espaciales podemos ver que el tiempo de ejecución no crece linealmente, sino que lo hace más rápido (obsérvense por ejemplo la Figura E.7c). Esto es razonable, ya que un aumento lineal en el número de saltos implica un aumento exponencial de los datos considerados en el modelo aproximado.

Tengamos en cuenta ahora los resultados de número de elementos devueltos por la consulta, precisión y exhaustividad de las gráficas. Vemos que a medida que aumenta el tamaño de los datos considerado en el modelo aproximado, representado



(a) Evolución del rendimiento con Q3 para aproximación aleatoria. (b) Exactitud y exhaustividad con Q3 para aproximación aleatoria.



(c) Evolución del rendimiento con Q3 para aproximación temporal. (d) Exactitud y exhaustividad con Q3 para aproximación temporal.

Figura E.6: Comparación entre aproximaciones temporales y aleatorias con un foco temporal en los datos.

en el eje horizontal, el número de elementos devueltos por las consultas (línea naranja de las gráficas) comienza a estabilizarse. De esta forma, en el punto de estabilización podríamos considerar que hemos alcanzado la cantidad de datos necesaria para que un modelo aproximado sea óptimo, ya que los valores de precisión y exhaustividad en cada caso son cercanos a 1.

Finalmente, observamos que en la mayoría de los casos los valores de exactitud están muy cerca de 1. Esto se debe a la influencia de los TN en la ecuación de exactitud. En consecuencia, el valor de exactitud no es lo suficientemente descriptivo para representar la desviación producida al ejecutar la consulta en el modelo aproximado frente a ejecutarla en el modelo completo.

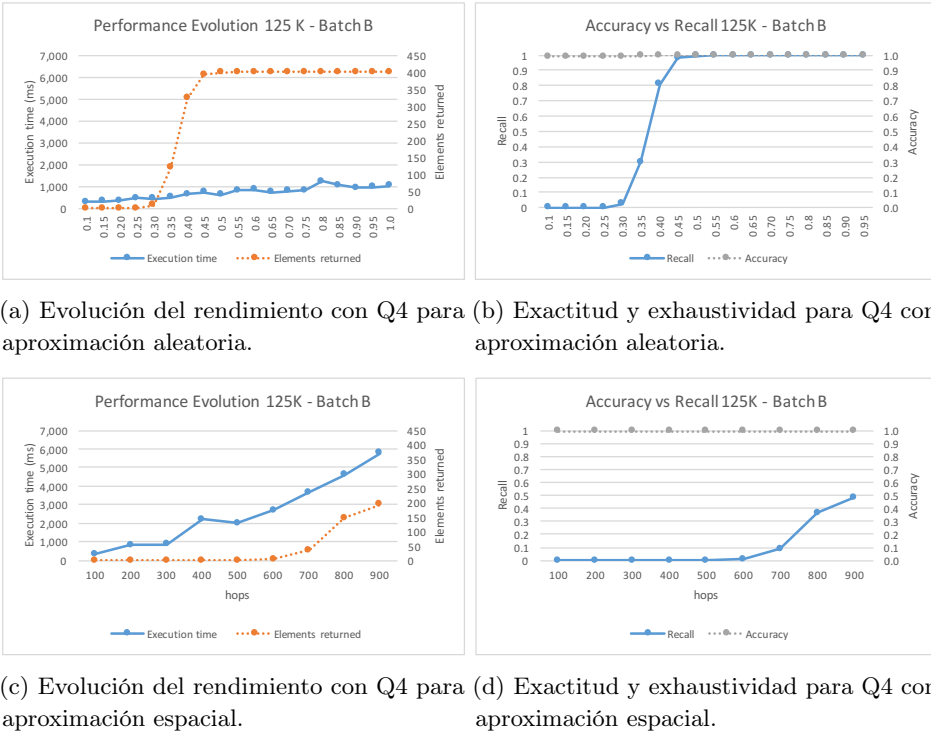


Figura E.7: Comparación entre aproximaciones espaciales y aleatorias.

## E.5 Mejora del rendimiento empleando técnicas de preprocesamiento

En esta sección resumimos la última contribución de esta tesis, que consiste en un algoritmo de preprocesamiento, llamado algoritmo SDR (por sus siglas en inglés Source DataSet Reduction). Este algoritmo es capaz de manejar datos persistentes y transitorios al mismo tiempo y tiene como objetivo mejorar el rendimiento de las consultas reduciendo el conjunto de datos a procesar. Sin embargo, a diferencia de las técnicas convencionales de AQP, reduce el conjunto de datos sin comprometer la precisión de los resultados, es decir, puede producir respuestas completamente correctas. Para ello, obtiene un subgrafo del conjunto de datos completo (es decir, del modelo) con los elementos que son relevantes para una consulta determinada. De esta manera podemos lograr aceleraciones de más de 100x para algunos tipos de consultas, incluso en sistemas ya optimizados. El algoritmo se ejecuta una primera vez antes de que se realice la consulta para obtener un subgrafo inicial. Después,

una versión incremental del algoritmo actualiza el subgrafo conforme se modifica el grafo original. En la figura E.8 se muestra una visión general de nuestra propuesta y todos sus componentes.

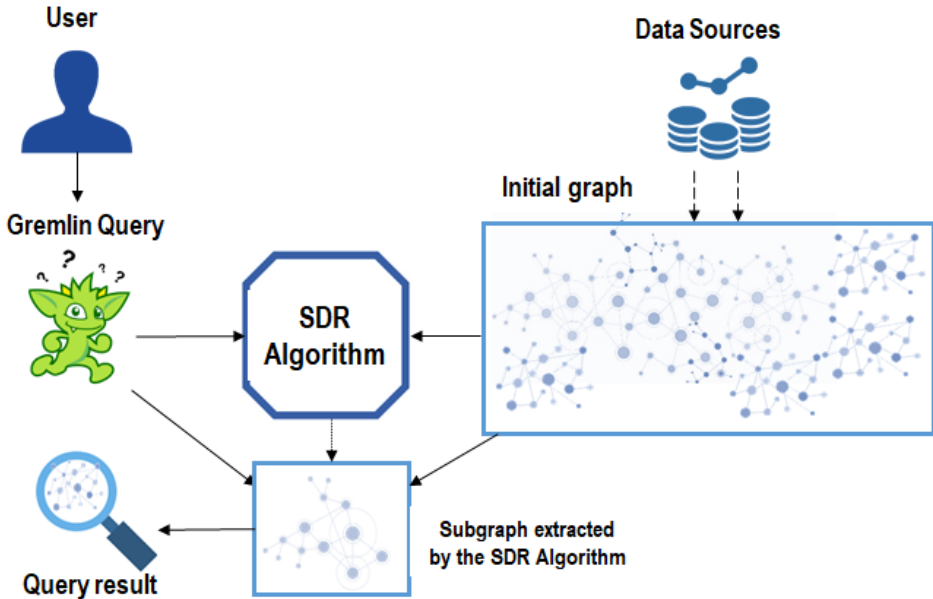


Figura E.8: Overall view of queries using the SDR algorithm.

### E.5.1 Clasificación de las queries

La estrategia seguida por el algoritmo a la hora de reducir el conjunto de datos de origen depende del tipo de consulta. Por este motivo, hemos definido una clasificación que tiene en cuenta los operadores y cláusulas que constituyen la consulta. A continuación, describimos los patrones más habituales que pueden encontrarse en las consultas sobre grafos y que son relevantes para nuestro algoritmo. Estos patrones se tratan de forma atómica, es decir, omitiendo cualquier otro patrón que pueda estar presente.

- Patrón simple: las consultas que siguen un patrón simple obtienen la información empleando solo las relaciones entrantes y salientes de los objetos y filtros de propiedad. Por relación entrante y saliente nos referimos a una



navegación a través de una relación, mientras que un filtro de propiedad se utiliza para obtener uno o más elementos del grafo según el valor de una propiedad del objeto o la relación.

- Patrón de condición: las consultas que siguen un patrón de condicion seleccionan la información mediante una cláusula **where**, que implica una subconsulta con la condición definida dentro de la cláusula.
- Patrón de Negación: las consultas que siguen el patrón de negación obtienen la información mediante una condición **not**.
- Patrón conjuntivo: las consultas que siguen el patrón conjuntivo seleccionan la información con una cláusula **and** que contiene dos o más predicados. La consulta selecciona aquellos elementos que cumplen todos los predicados.
- Patrón disyuntivo: las consultas que siguen el patrón disyuntivo seleccionan la información con una cláusula **or** que contiene dos o más predicados. La consulta selecciona aquellos elementos que cumplen al menos uno de los predicados.
- Patrón de agregación: las consultas que siguen el patrón de agregación en primer lugar agrupan la información con operadores de agregación y luego la filtran con un operador relacional.

### E.5.2 Algoritmo SDR

Una vez definidos los tipos de patrones que se pueden encontrar en una consulta, podemos explicar el funcionamiento del algoritmo SDR. Este algoritmo está inspirado en el algoritmo PageRank de Google [96], que obtiene un ranking de las páginas web más relevantes según el número de páginas que apuntan hacia ellas y sus respectivos pesos. Estos pesos representan la probabilidad de que una persona que haga clic aleatoriamente en enlaces web llegue a esta página. De manera similar, el algoritmo SDR analiza una consulta para asignar un peso a todos los objetos del grafo de acuerdo con su relevancia para la consulta. Como resultado devuelve un subgrafo con los objetos que tienen un peso mayor que 0 y las relaciones entre ellos. Estos elementos son considerados relevantes para la consulta.

Una consulta se compone de diferentes cláusulas, operaciones y filtros, que en el contexto de este trabajo llamaremos *pasos*. Es decir, consideramos que un paso es cualquier tipo de cláusula, filtro u operación que se aplica a los elementos de un modelo según lo especificado por una consulta. De acuerdo con los patrones de consulta presentados en la Sección E.5.1, consideramos ocho tipos de pasos: filtros de tipo de elemento, filtros de propiedad, relaciones, operación *and*, operación *or*, operación *not*, operación *where* y agregaciones. Un paso puede, a su vez, dividirse en subpasos.

El algoritmo se ejecuta en paralelo en cada objeto, por medio del uso de VertexProgram [120] de Tinkerpop, y ejecuta varias iteraciones. De esta forma, en cada iteración se analiza un paso de la consulta y el objeto envía un mensaje a través de las relaciones relevantes relacionadas con dicho paso y cuenta la cantidad de mensajes que sus vecinos le enviaron en la iteración anterior. El peso se calcula utilizando el número de mensajes recibidos y enviados. El flujo completo del algoritmo SDR se muestra en el Algoritmo 4. Las entradas del algoritmo son la consulta  $Q$  y el grafo  $G$ ; el resultado es el subgrafo con los objetos que son relevantes para  $Q$ . Como se indicó anteriormente, el algoritmo SDR recorre los pasos de la consulta en varias iteraciones. Para lograr esto, la función  $\text{SDRVertexCentric}(Q, v)$  se ejecuta en cada objeto en paralelo.

Al igual que el algoritmo PageRank de Google, las dos primeras iteraciones del algoritmo SDR son ligeramente diferentes al resto. El algoritmo SDR usa la iteración inicial (función  $\text{WeightInitialisation}(s, v)$ ) para calcular un peso inicial de aquellos objetos que son relevantes para el primer paso de la consulta. Para calcular este peso inicial, el algoritmo cuenta el número de relaciones que tiene  $v$  con los objetos que son relevantes para el paso. Luego, en la segunda iteración (función  $\text{InWeightPropagation}(s, v, \text{weight})$ ), los objetos informan, a través de esas relaciones, a sus objetos vecinos sobre su peso actual. Las iteraciones restantes (función  $\text{FurWeightPropagation}(s, v, \text{weight})$ ) calcularán los pesos de los objetos de acuerdo con su relevancia para la consulta y las relaciones con los otros objetos relevantes.

---

**Algorithm 4:** The main SDR algorithm

---

**Data:** A query  $Q$  and a Graph  $G(V, E)$

**Result:** A subgraph  $SG(V_{SG}, E_{SG})$

- 1:  $v.weight = \mathbf{SDRVertexCentric}(Q, v) \forall v \in V$
  - 2:  $ListSGIds$  add  $\{v_w.id, v_w.weight\} \forall v_w \in V$  where  $v_w.weight \neq 0$
  - 3: **return**  $SG = G - \{v_d \in V \text{ where } v_d.id \notin ListSGIds\}$
- 

**Function**  $\mathbf{SDRVertexCentric}(Q, v)$

- 1: Obtain the set  $S$  of steps of  $Q$
  - 2:  $iteration = 0, weight = 0$
  - 3: **while**  $iteration \leq S.size$  **do**
  - 4:    $guardCondition = true$
  - 5:   **if**  $iteration == 0$  **then**
  - 6:      $s = S.get(S.size - 1)$
  - 7:      $weight = \mathbf{WeightInitialisation}(s, v)$
  - 8:   **else**
  - 9:     Select  $s = S.get(S.size - iteration)$
  - 10:    **if**  $iteration == 1$  **then**
  - 11:      $weight = \mathbf{InWeightPropagation}(s, v, weight)$
  - 12:    **else**
  - 13:      $weight = \mathbf{FurWeightPropagation}(s, v, weight)$
  - 14:    **end if**
  - 15:   **end if**
  - 16:    $iteration++$
  - 17: **end while**
  - 18: **return**  $weight$
-

---



---

Function `WeightInitialisation( $s, v$ )`

```

1: if  $s$  is property filter then
2:   if  $v$  matches the filter then
3:      $pRel$  = previous relationship step of  $s$ 
4:      $cNeighbors$  = No. neighbors of  $v$  through  $pRel$ 
5:      $guardCondition$  =  $cNeighbors > 0$ ?
6:   else
7:      $guardCondition$  = false
8:   end if
9: else if  $s$  is a relationship then
10:   $cNeighbors$  = No. neighbors of  $v$  through  $s$ 
11:   $guardCondition$  =  $cNeighbors > 0$ ?
12: else if  $s$  is a TraversalParent filter then
13:  Obtain subqueries  $SQ$  from  $s$ 
14:  for  $q : SQ$  do
15:     $weightsSQ$  = SDRVertexCentric( $q, v$ ),  $q \in SQ$ 
16:     $weight$  = TraversalParentType( $weightsSQ$ )
17:  end for
18: end if
19: if  $guardCondition$  then
20:   $weight$  =  $weight + cNeighbors$ 
21: end if
22: return  $weight$ 

```

---

Function `InWeightPropagation( $s, v, weight$ )`

```

1: if  $s$  is relationship and  $weight > 0$  then
2:  Send messages through  $s$ 
3: else if  $s$  is property filter or TraversalParent then
4:   $pRel$  = previous relationship of  $s$ 
5:   $iteration++$ 
6:  if  $weight > 0$  then
7:    Send messages through  $pRel$ 
8:  end if
9: end if
10: return  $weight$ 

```

---

---



---

Function *FurWeightPropagation*(*s*, *v*, *weight*)

---

```

1: cMessages = sum(received messages)
2: if cMessages > 0 then
3:   if s is relationship then
4:     cNeighbors = No. neighbors of v through s
5:     guardCondition = cNeighbors > 0?
6:     Send messages through s
7:   else if s is a property filter then
8:     pRel = previous relationship of s
9:     iteration ++
10:    if v match the filters then
11:      cNeighbors = No. neighbors of v thru pRel
12:      guardCondition = cNeighbors > 0?
13:      Send messages through pRel
14:    else
15:      guardCondition = false
16:    end if
17:  end if
18: end if
19: if guardCondition then
20:   weight = weight + cNeighbors + cMessages
21: end if
22: return weight

```

---

### E.5.3 Algoritmo SDR incremental

Nuestro enfoque está diseñado para sistemas dinámicos que se actualizan constantemente con nueva información. Ejecutar el algoritmo SDR principal en todos los objetos cada vez que cambia el grafo es demasiado costoso en términos de tiempo y memoria. Por esta razón, hemos desarrollado el llamado *algoritmo SDR incremental* que actualiza los pesos de los objetos del grafo cuando se agregan nuevos elementos o se actualizan o descartan elementos existentes. De esta manera, el algoritmo principal SDR debe ejecutarse solo una vez al inicio y luego actualizarse cada vez que cambia la información del grafo. El algoritmo SDR incremental se muestra en Algoritmo 5. Cabe resaltar que el algoritmo SDR incremental solo actualiza el peso de los objetos que llegan o se modifican en el sistema, ya que los

objetos eliminados no necesitan actualizar sus pesos. También actualiza el peso de los objetos que pueden verse afectados debido a un cambio en la estructura del grafo, es decir, los objetos a los que se puede acceder desde los objetos agregados, actualizados o eliminados a través de las relaciones de la consulta.

---

**Algorithm 5:** The Incremental SDR algorithm

---

**Data:** A set of objects  $V_n$ , a query  $Q$  and a Graph  $G(V, E)$

**Result:** A subgraph  $SG(V_{SG}, E_{SG})$

```

1: Obtain steps  $S$  from  $Q$ 
2: Initialise an empty subgraph  $SG_i(V_i, E_i)$ 
3: for  $s : S$  do
4:   if  $s$  represents a relationship then
5:      $SG_i = SG_i \cup \text{createSubGraph}(s, V_n)$ 
6:   else if  $s$  represents a TraversalParent then
7:     Obtain subqueries  $SQ$  from  $s$ 
8:     for  $q : SQ$  do
9:       Obtain steps  $S_{SQ}$  from  $q$ 
10:      for  $s_{SQ} : S_{SQ}$  do
11:        if  $s_{SQ}$  represents a relationship then
12:           $SG_i = SG_i \cup \text{createSubGraph}(s_{SQ}, V_n)$ 
13:        end if
14:      end for
15:    end for
16:  end if
17: end for
18:  $v_i.weight = \text{SDRVertexCentric}(Q, v_i) \forall v_i \in V_i$ 
19:  $ListWeights = \text{get weight and id from } SG_i$ 
20: Update  $ListSGIds$  with  $ListWeights$ 
21: return  $SG = G - \{v_d \in V \text{ where } v_d.id \notin ListSGIds\}$ 

```

---

Por lo general, se agregará, descartará o modificará más de un objeto en el grafo al mismo tiempo, porque los eventos generalmente llegan en lotes. Por tanto, las entradas del Algoritmo 5 son un conjunto de objetos  $V_n$ , la consulta  $Q$  y el grafo  $G(V, E)$ . El conjunto  $V_n$  contiene aquellos objetos que se agregan o actualizan en el grafo más el conjunto de vecinos de los objetos eliminados, en caso de haberlos. Como salida, el algoritmo devuelve el subgrafo actualizado  $SG$  para ser consultado (línea 21).

El hecho de que el algoritmo SDR incremental solo tenga que actualizar los pesos

---

**Algorithm 6:** Function createSubGraph

---

**Data:** A step  $s$  and an a set of objects  $V_n$

**Result:** A subgraph  $SG(V_{SG}, E_{SG})$

- 1: Initialise an empty subgraph  $SG(V_{SG}, E_{SG})$
  - 2:  $next_r = s \cup \text{forward relationships of } s$
  - 3:  $previous_r = \text{backward relationships of } s$
  - 4: **for**  $n : next_r \cup p : previous_r$  **do**
  - 5:    $SG = SG \cup \text{neighbors of } V_n \text{ through } n \text{ and } p$
  - 6: **end for**
  - 7: **return**  $SG$
- 

de los vecinos de los objetos recién añadidos, actualizados o descartados del grafo no representa una sobrecarga de rendimiento significativa, ya que la complejidad del algoritmo es  $\mathcal{O}(v \cdot r \cdot n)$ , donde  $v$  es el tamaño de  $V_n$  (es decir, el número de elementos nuevos, actualizados o de vecinos de los elementos descartados),  $r$  es el número de relaciones de  $s$ , y  $n$  es el número de vecinos de  $V_n$  a través de  $s$ . Dado que estos números son normalmente pequeños, el tiempo de ejecución de este algoritmo no es significativo en comparación con la ejecución de la consulta. Además, este algoritmo incremental se ejecuta en paralelo con las consultas, por lo que no afecta a su rendimiento.

#### E.5.4 Mejora del rendimiento con algoritmo SDR

Con el objetivo de demostrar que el algoritmo SDR efectivamente mejora el rendimiento de las consultas sobre flujos de datos estructurados en forma de grafos, se ha probado en tres casos de estudio. El primero de ellos es el ejemplo de Amazon de la Figura E.3, compuesto por datos sintéticos, y los otros dos procedentes de datasets reales, cuyos metamodelos se representan en las figuras E.9 y E.10. Con cada uno de ellos hemos desarrollado consultas que consideran todos los tipos de patrones expuestos en la sección E.5.1. Estas consultas se han probado en varios tamaños de modelo empleando dos tipos de sistemas, uno con información estática (que representaría una captura de los datos de un sistema dinámico en un instante determinado) y otro con información dinámica.

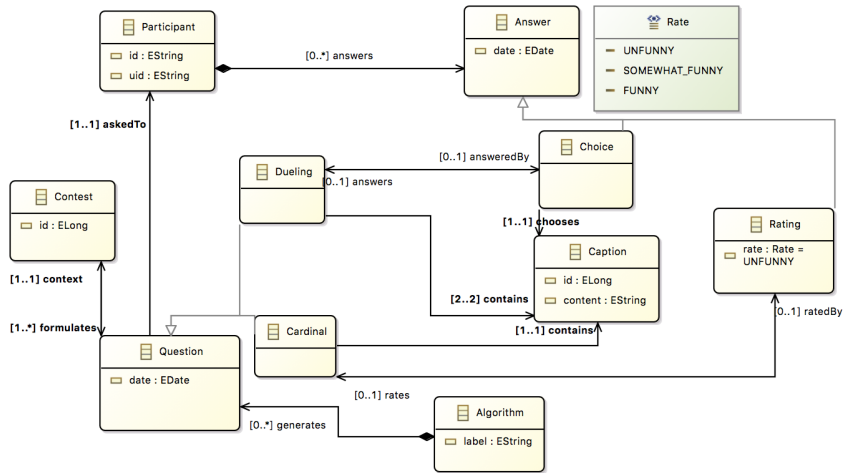


Figura E.9: Metamodelo del NY Caption Contest

### Sistema con información estática

Para el sistema estático se han medido tres parámetros: (i) tiempo de ejecución, (ii) memoria consumida y (iii) número de elementos en el grafo y en el subgrafo. Como ejemplo, en la Figura E.11 se representan los resultados para el tiempo de ejecución y la memoria consumida en el caso de estudio de Amazon. Como se puede observar, en todos los casos ambos parámetros son menores cuando la consulta se ejecuta sobre el subgrafo que cuando se ejecuta sobre el grafo. Aunque la mejora de rendimiento de las consultas que siguen patrones de agregación (figuras E.11f y E.11g) es menor que la del resto de consultas. Esto es debido a que el rendimiento para este tipo de patrones depende en gran medida del tiempo y la memoria necesarios para resolver los filtros y operadores de agregación de las consultas. Además, se puede observar que las consultas que contienen un patrón de negación (figura E.11c) también presentan una mejoría de rendimiento menor que las consultas que presentan patrones simples, condicionales, de conjunción o de disyunción. Esto es debido a que cuantos más elementos cumplan la condición impuesta por la cláusula **not** menor será el tamaño del subgrafo y mayor mejora del rendimiento, es decir, que la mejora del rendimiento es directamente proporcional al número de elementos que cumplen la condición. De esta forma, según los resultados obtenidos en la figura E.11c, podemos concluir que el subgrafo obtenido para este ejemplo contiene más elementos que los subgrafos obtenidos para el resto de



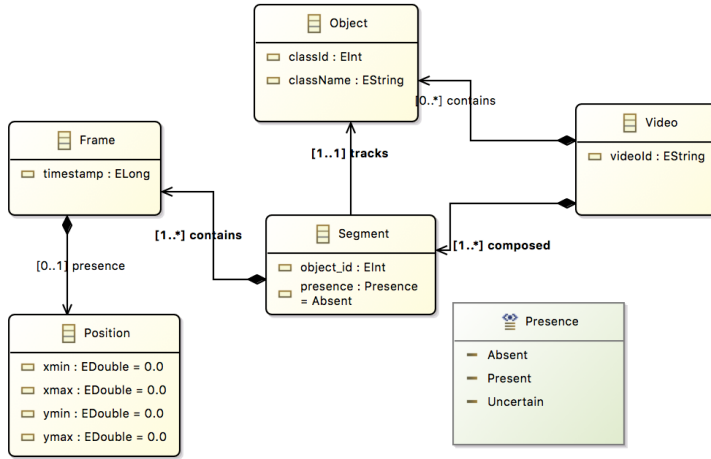


Figura E.10: Metamodelo del ejemplo de Youtube

consultas.

Con respecto a los tamaños del grafo y del subgrafo, en la Tabla E.5 se muestra, para todos los casos de estudio, en cuánto se reduce el tamaño del subgrafo obtenido con el algoritmo SDR en comparación con el tamaño del grafo origen. Este valor es calculado como del ratio de elementos del grafo que se encuentran en el subgrafo. De esta forma, a mayor valor del ratio menor será el número de elementos del grafo que se encuentran en el subgrafo. En la tabla, podemos observar que las consultas que siguen patrones simples, condicionales, conjuntivos y disyuntivos logran una reducción de más del 90 % en todos los casos y cerca del 100 % en muchos de ellos. Esto sugiere que, en estos casos, el algoritmo SDR obtiene un subgrafo que está cerca del subgrafo mínimo requerido para ejecutar la consulta. Por el contrario, los resultados no son tan buenos para las consultas que siguen el patrón de agregación. Esto se debe al hecho de que la implementación del algoritmo no considera los pasos de agregación para obtener el subgrafo, por lo que la reducción del tamaño depende únicamente de los filtros de consulta que se encuentran antes del operador de agregación en la consulta. De esta forma, cuanto más restrictivos sean estos filtros menor será el tamaño del subgrafo. Además, como hemos comentado anteriormente, la reducción lograda en el caso de consultas que siguen un patrón negativo depende directamente del número de elementos que cumplen el predicado de la cláusula **not**.

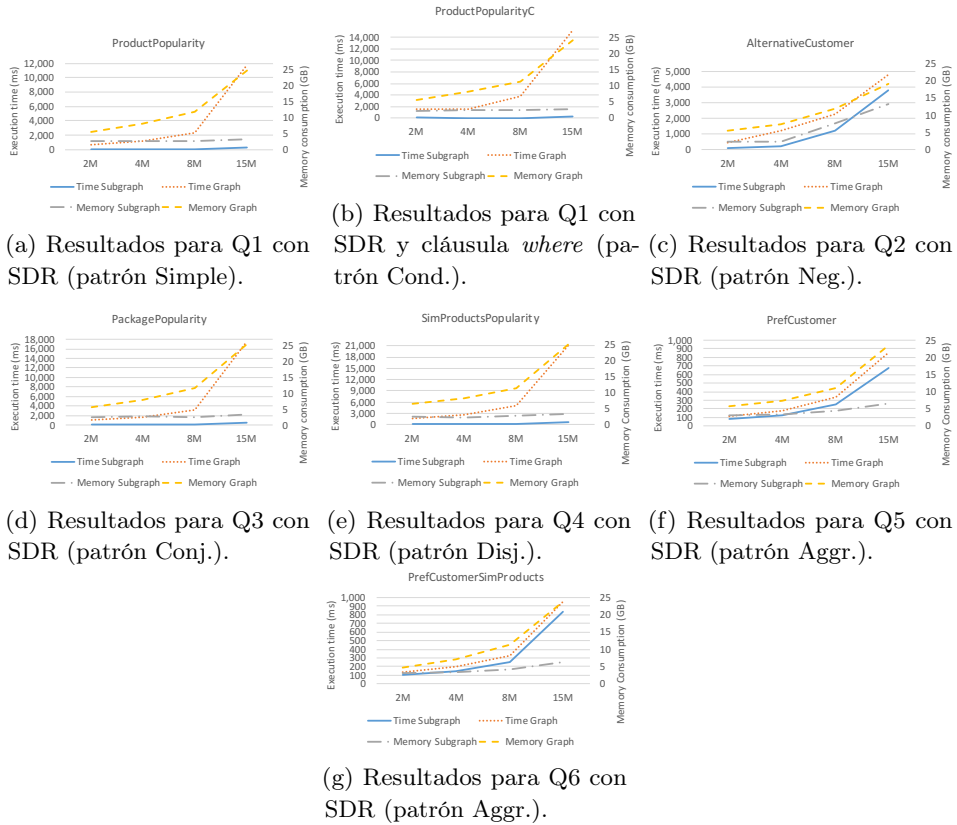


Figura E.11: Performance results of the SDR algorithm for the Amazon queries.

Esto explica que el valor de la ganancia de reducción sea menor en este tipo de consultas que en el resto.

### Sistema con información dinámica

Con respecto al sistema dinámico, se han realizado dos tipos de experimentos. El primero consiste en ejecutar la consulta de forma periódica sobre el grafo completo, más concretamente cada vez que se añade un número determinado de registros. Entendiéndose como registro un conjunto de elementos que están relacionados entre sí. El segundo realiza una ejecución inicial del algoritmo SDR principal para calcular un subgrafo inicial y, a partir de ahí, el algoritmo incremental se encarga de actualizar dicho subgrafo a medida que se modifica el grafo original y la consulta se ejecuta de forma periódica en dicho subgrafo (cada vez que se añade un número

Caso de estudio	Nombre de la consulta	Patrón	Modelos					
			2M	4M	8M	15M		
Amazon	ProductPopularity	Simple	0.9912	0.9949	0.9973	0.9926		
	ProductPopularityC	Cond.	0.9912	0.9949	0.9973	0.9926		
	AlternativeCustomer	Neg.	0.4739	0.5140	0.4423	0.5206		
	PackagePopularity	Conj.	0.9861	0.9921	0.9959	0.9880		
	SimProductsPopularity	Disj.	0.9817	0.9895	0.9945	0.9859		
	PrefCustomer	Aggr.	0.9039	0.8902	0.8815	0.8757		
	PrefCustomerSimProducts	Aggr.	0.8970	0.8858	0.8790	0.8734		
			1M	4M	9M	12M	16M	
Contest	RecentPart	Simple	0.9663	0.9806	0.9898	0.9926	0.9942	
	ContestPart	Cond.	0.9226	0.9803	0.9896	0.9924	0.9941	
	UnchosenCap	Conj.	0.9086	0.9668	0.9825	0.9872	0.9901	
	FunniestCaption	Aggr.	0.8427	0.7657	0.7444	0.7429	0.7435	
	Abandon	Aggr.	0.7721	0.7564	0.7525	0.7513	0.7506	
	FunniestCaptionU	Aggr.&Conj.	0.8658	0.9548	0.9584	0.9603	0.8634	
			2M	4M	6M	8M	10M	12M
YouTube	GetAnimalVideos	Cond.	0.9951	0.9951	0.9951	0.9951	0.9951	0.9951
	NotPresent	Neg.	0.9688	0.9683	0.9683	0.9685	0.9685	0.9685
	AnimalPerson	Conj.	0.9946	0.9945	0.9945	0.9945	0.9945	0.9945
	PresentSoon	Conj.	0.9815	0.9817	0.9817	0.9817	0.9817	0.9816
	Pets	Disj.	0.9588	0.9582	0.9574	0.9573	0.9574	0.9578
	InCast	Aggr.	0.5456	0.5460	0.5464	0.5462	0.5464	0.5464

Cuadro E.5: Ratio de la ganancia de elementos cuando se usa el algoritmo SDR.

determinado de registros, como en el caso anterior). En ambos sistemas, se ha medido el tiempo de ejecución para el experimento completo. Nótese que, en el caso en que se emplea el algoritmo SDR, se ha considerado el tiempo que tarda la ejecución inicial del algoritmo. Con estos valores hemos calculado la ganancia del tiempo de ejecución sobre el subgrafo con respecto al tiempo de ejecución sobre el grafo para ver en qué momento se ve compensada la ejecución inicial del algoritmo SDR. Los resultados para el caso de estudio de Amazon se muestran en la tabla E.6, donde  $\beta$  es el número total de registros agregados por experimento, mientras que  $\alpha$  representa el tamaño del lote de registros nuevos que han llegado al sistema cada vez que se ejecuta la consulta. Esto significa que, para un valor constante de  $\alpha$ , cuanto mayor es el valor de  $\beta$ , mayor es el número de veces que se ejecuta la consulta en cada experimento. En las tablas se resalta en negrita el punto en el que la ganancia de tiempo se vuelve positiva, que depende del valor de  $\beta$ , es decir, el punto en el que la ejecución inicial del algoritmo SDR se ve compensada. Como se puede observar la ganancia de tiempo, en otras palabras la rapidez con que se alcanza el punto de equilibrio, es directamente proporcional al valor de  $\beta$  e inversamente proporcional al valor de  $\alpha$ . Además, dicha ganancia también aumenta

con el tamaño del modelo.

Query Name		Models							
		$\alpha = 5$				$\alpha = 10$			
	$\beta$	2M	4M	8M	15M	2M	4M	8M	15M
ProductPopularity (Simple)	50	-0.0274	-0.0104	<b>0.0486</b>	<b>0.0155</b>	-0.2043	-0.1492	-0.0649	-0.0986
	100	<b>0.1198</b>	<b>0.1834</b>	0.2032	0.1875	-0.0176	-0.0220	<b>0.0760</b>	<b>0.0602</b>
	150	0.1874	0.2274	0.2647	0.2718	-0.0101	<b>0.0827</b>	0.1272	0.1585
	200	0.2021	0.2771	0.3018	0.3363	<b>0.0656</b>	0.1215	0.1821	0.1920
	250	0.2053	0.3302	0.3414	0.3772	0.0721	0.1658	0.2175	0.2469
ProductPopularityC (Conditional)	50	-0.0334	-0.0244	<b>0.0119</b>	<b>0.0666</b>	-0.2080	-0.1541	-0.1271	-0.0677
	100	<b>0.0704</b>	<b>0.1319</b>	0.1468	0.2048	-0.0708	-0.0163	<b>0.0456</b>	<b>0.0572</b>
	150	0.1546	0.2095	0.2110	0.2915	<b>0.0347</b>	<b>0.0522</b>	0.1095	0.1537
	200	0.1930	0.2479	0.2586	0.3392	0.0232	0.0828	0.1402	0.1910
	250	0.2145	0.2913	0.3001	0.3508	0.0588	0.1312	0.1728	0.2315
AlternativeCustomer (Negative)	50	-0.2713	-0.2519	-0.1435	-0.1760	-0.2514	-0.2908	-0.2229	-0.1628
	100	-0.0989	-0.1047	-0.0367	-0.0555	-0.1472	-0.1421	-0.1212	-0.0990
	150	-0.0805	-0.0218	<b>0.0006</b>	<b>0.0034</b>	-0.0985	-0.0880	-0.0805	-0.0337
	200	-0.0461	<b>0.0186</b>	0.0283	0.0522	-0.0588	-0.0535	-0.0486	-0.0290
	250	-0.0280	0.0614	0.0728	0.1121	-0.0581	-0.0192	-0.0311	-0.0116
PackagePopularity (Conjunctive)	50	-0.0778	-0.0850	-0.0113	-0.0107	-0.3188	-0.2222	-0.1674	-0.0938
	100	<b>0.0704</b>	<b>0.1187</b>	<b>0.1596</b>	<b>0.1632</b>	-0.0485	-0.0714	-0.0594	<b>0.0112</b>
	150	0.1396	0.2072	0.2123	0.2764	-0.0445	<b>0.0305</b>	<b>0.0692</b>	0.0792
	200	0.1730	0.2486	0.2499	0.3607	<b>0.0123</b>	0.0585	0.0910	0.2189
	250	0.1849	0.2927	0.2936	0.3875	0.0455	0.1282	0.1289	0.2496
SimProductsPopularity (Disjunctive)	50	-0.0207	-0.0092	<b>0.1372</b>	<b>0.1514</b>	-0.2326	-0.1121	-0.1092	-0.0574
	100	<b>0.1525</b>	<b>0.1721</b>	0.2617	0.2718	-0.0111	<b>0.0096</b>	<b>0.0463</b>	<b>0.1414</b>
	150	0.2337	0.3121	0.3335	0.3965	<b>0.0477</b>	0.1059	0.1632	0.2507
	200	0.2722	0.3508	0.3838	0.4497	0.0969	0.1861	0.1834	0.2659
	250	0.3029	0.3918	0.4038	0.4753	0.0828	0.2052	0.2215	0.3278
PrefCustomer (Aggregation)	50	-0.3316	-0.3102	-0.3002	-0.3041	-0.3724	-0.3595	-0.3509	-0.3238
	100	-0.2860	-0.2479	-0.2088	-0.1554	-0.2951	-0.2926	-0.2500	-0.2115
	150	-0.2145	-0.1989	-0.1652	-0.0751	-0.2395	-0.2121	-0.1850	-0.1140
	200	-0.2006	-0.1295	-0.1283	-0.0547	-0.2191	-0.1526	-0.1230	-0.0772
	250	-0.1826	-0.0999	-0.0932	-0.0185	-0.2061	-0.1125	-0.0984	-0.0440
PrefCustomerSimProducts (Aggregation)	50	-0.2663	-0.2892	-0.2294	-0.3203	-0.3024	-0.3723	-0.2916	-0.3652
	100	-0.2282	-0.2215	-0.1509	-0.1806	-0.2734	-0.2464	-0.2089	-0.2137
	150	-0.1746	-0.1753	-0.1190	-0.1055	-0.2083	-0.1865	-0.1715	-0.1146
	200	-0.1550	-0.0871	-0.1061	-0.0601	-0.1808	-0.1361	-0.1128	-0.0745
	250	-0.1377	-0.0807	-0.0696	-0.0194	-0.1775	-0.1097	-0.0987	-0.0379

Cuadro E.6: Gain ratio when using the incremental algorithm in the Amazon case study.

Considerando los diferentes patrones de consulta, podemos ver que las consultas que contienen patrones disyuntivos logran la mayor ganancia, seguidas de las consultas que contienen patrones simples y las consultas que contienen patrones condicionales. Por otro lado, las consultas que contienen patrones conjuntivos tienen una ganancia mayor que las consultas con patrones negativos y con patrones de agregación.



# Apéndice F

## Conclusiones y Contribuciones

---

En este apéndice se exponen, en castellano, las conclusiones y contribuciones principales derivadas tras la realización de esta tesis doctoral.

Como se ha mencionado a lo largo de este documento, uno de los principales retos al trabajar con aplicaciones de generación y transmisión de datos es conseguir una baja latencia en el procesamiento, lo que implica obtener respuestas rápidamente. Es precisamente esta característica la que motivó la primera contribución de esta tesis, presentada en el Capítulo 3. En dicho capítulo presentamos un estudio comparativo de 7 plataformas de procesamiento que son habitualmente usadas para trabajar con grandes volúmenes de datos. Estas plataformas son TinkerGraph, Neo4j, CrateDB, Memgraph, GraphFrames, OrientDB y JanusGraph. Además, se compararon 4 lenguajes específicos de dominio para escribir las consultas, que son Gremlin, Cypher, SQL y el DSL de GraphFrames. Nuestro objetivo era obtener la mejor combinación de lenguaje y plataforma de procesamiento que se adaptara a los siguientes requisitos: (i) permitiera realizar consultas y actualizar la información lo más rápido posible para dar respuestas en tiempo real, (ii) pudiera hacer frente a información estructurada en forma de grafos, y (iii) con un lenguaje que

proporcionara una sintaxis clara para poder estudiar el tipo de consulta a ejecutar sobre los datos.

Todas las tecnologías se evaluaron mediante dos casos de estudio con información estructurada en forma de grafos. En los experimentos se comparó el rendimiento, en términos de tiempo de ejecución, y la complejidad del lenguaje, en términos de número de caracteres, operadores y variables internas. Los resultados mostraron que las bases de datos de grafos son las tecnologías más eficientes para trabajar con este tipo de información. Además, los lenguajes utilizados con estas bases de datos presentan la sintaxis más simple. Por lo tanto, concluimos que la combinación más adecuada para nuestros requisitos era TinkerGraph y Gremlin.

La segunda contribución principal se presentó en el Capítulo 4. Esta contribución aborda el procesamiento de consultas aproximadas de forma online cuando se trabaja con flujos de información estructurados en grafos. Se propusieron tres técnicas para mejorar el rendimiento al consultar modelos grandes, denominadas aproximaciones temporales, espaciales y aleatorias. Las aproximaciones temporales y espaciales seleccionan un subconjunto de la fuente de información mediante la reducción de los rangos temporal y espacial, respectivamente, mientras que las aproximaciones aleatorias agregan una probabilidad a cada elemento del grafo para ser incluido en el subconjunto. Para encontrar el equilibrio correcto entre la pérdida de precisión derivada de estas aproximaciones y la ganancia de rendimiento, propusimos un método para medir la precisión. Este método se basa en los términos de exactitud, exhaustividad y precisión.

Se utilizaron dos distribuciones de datos diferentes en los experimentos del Capítulo 4, con el fin de analizar cómo esta característica afecta a las aproximaciones. Los resultados concluyeron que es posible mejorar el rendimiento empleando aproximaciones y se puede adquirir un valor de precisión óptimo cuando se considera solo una parte del modelo fuente. Sin embargo, las aproximaciones temporales son la opción más conveniente cuando los datos presentan un enfoque temporal. Además, las aproximaciones aleatorias mostraron un rendimiento similar independientemente de la distribución de datos y sus experimentos demostraron que son la mejor opción cuando una consulta no contiene filtrado temporal o espacial. Finalmente, los resultados de las aproximaciones espaciales mostraron que son muy costosos en términos de tiempo de ejecución y solo se amortizan cuando no hay otra opción viable.

---

La última contribución se presentó en el Capítulo 5. Este capítulo aborda el procesamiento de consultas aproximadas offline para mejorar el rendimiento de las aplicaciones de transmisión de datos. Para lograrlo, diseñamos y desarrollamos el algoritmo de Reducción del Conjunto Origen de Datos (SDR, por sus siglas en inglés *Source Dataset Reduction*). Este algoritmo optimiza el rendimiento de las consultas sobre flujos de datos estructurados en grafos, al seleccionar un subgrafo del modelo de origen. Este subgrafo contiene los datos que son relevantes para la consulta y, por lo tanto, la consulta se puede ejecutar de manera más eficiente. Dado que los elementos contenidos en el subgrafo dependen de la estructura de la consulta, se identificaron seis patrones que se pueden encontrar en las consultas sobre grafos. Estos patrones son: (i) simples, (ii) condicionales, (iii) de conjunción, (iv) de disyunción, (v) negativos y (vi) de agregación. Dado que nuestro enfoque está diseñado para trabajar con flujos de información, desarrollamos una versión incremental del algoritmo, llamada *Incremental SDR*. Este algoritmo actualiza el subgrafo a medida que llega nueva información al sistema.

Se utilizaron tres casos de estudio para validar nuestra propuesta. Los resultados mostraron que al consultar el subgrafo obtenido con el algoritmo SDR en lugar de todo el grafo se logra una mejora del rendimiento para todos los patrones de consulta. De hecho, algunos patrones solo necesitan consultar un subgrafo que contiene únicamente el 1 % de los elementos del grafo original. En concreto, los patrones de consulta en los que la ganancia de tiempo es mayor son, por este orden, disyuntivo, condicional, simple, conjuntivo y negativo. Sin embargo, las consultas que siguen patrones de agregación se comportan de forma ligeramente diferente al resto, ya que dependen de los filtros y operadores de agregación que contienen. Además, también demostramos que las mejoras de rendimiento aumentan con el tamaño del grafo original, así como con el número de veces que se ejecuta la consulta.

Por lo tanto, nuestro enfoque ha demostrado que es posible obtener la mejor compensación entre la exactitud de los resultados y el rendimiento del procesamiento. Por esta razón, consideramos que esta tesis proporciona una respuesta a la pregunta de investigación propuesta en la Sección 1.1 (*¿Podemos obtener una buena (u óptima) compensación entre rendimiento y pérdida de precisión al procesar cantidades muy grandes de información?*) y también logra los objetivos esperados.