Improving query performance on dynamic graphs

Gala Barquero, Javier Troya & Antonio Vallecillo

Software and Systems Modeling

ISSN 1619-1366
Volume 20
Number 4

Softw Syst Model (2021) 20:1011-1041
DOI 10.1007/s10270-020-00832-3

Springer

Software and Systems Modeling (2021) 20:1011–1041
https://doi.org/10.1007/s10270-020-00832-3

REGULAR PAPER

Improving query performance on dynamic graphs

Gala Barquero[1] · Javier Troya[2] · Antonio Vallecillo[1]

Received: 30 January 2020 / Revised: 23 September 2020 / Accepted: 30 September 2020 / Published online: 2 November 2020

Abstract
Querying large models efficiently often imposes high demands on system resources such as memory, processing time, disk access or network latency. The situation becomes more complicated when data are highly interconnected, e.g. in the form of graph structures, and when data sources are heterogeneous, partly coming from dynamic systems and partly stored in databases. These situations are now common in many existing social networking applications and geo-location systems, which require specialized and efficient query algorithms in order to make informed decisions on time. In this paper, we propose an algorithm to improve the memory consumption and time performance of this type of queries by reducing the amount of elements to be processed, focusing only on the information that is relevant to the query but without compromising the accuracy of its results. To this end, the reduced subset of data is selected depending on the type of query and its constituent filters. Three case studies are used to evaluate the performance of our proposal, obtaining significant speedups in all cases.

Keywords Data stream processing · Dynamic graphs · Performance optimization · Precomputing systems · Data queries

1 Introduction

Nowadays, many information processing systems need to handle the data flows that are constantly generated from different sources, such as social networks, geo-location systems or e-commerce applications [16,17]. Companies and organizations use this information to make informed decisions or detect situations of interest in real time. For example, the Spanish BBVA bank studied the economic impact of Barcelona's 2012 *Mobile World Congress* by analysing all credit card transactions during two weeks [13]. Another example of the need to process data flows to extract useful information is the detection of possible terrorist attacks analysing social networks and Web access logs [50,71]. In these cases, an efficient query system needs to process the information as fast as possible to identify those situations of interest without delay. However, the large volume of data to be efficiently processed represents a significant challenge to current information processing systems.

To speed up the processing of the information, these systems usually apply different mechanisms, such as storing the information in clusters to be processed in parallel [35,49,67], or selecting a subset of the data that is finally queried [10,24,26,61]. These approaches usually 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 [55].

In addition, the information handled by many current software applications is no longer exclusively static or dynamic, but is composed of both *persistent* data residing on disks and *transient* information events flowing in data streams. Persistent data are permanently stored in the system and not frequently modified, for example, the users in a social network or the products in an e-commerce website. In turn, the *transient* information is temporarily stored in the system and expires after some period of time, e.g. user tweets or temporary product offers. The way to effectively deal with these

Communicated by Daniel Varro.

✉ Javier Troya
jtroya@us.es
http://www.lsi.us.es/~jtroya

Gala Barquero
gala@lcc.uma.es

Antonio Vallecillo
av@lcc.uma.es

[1] ITIS Software, Universidad de Málaga, Málaga, Spain

[2] SCORE Lab, I3US Institute, Universidad de Sevilla, Seville, Spain

two types of information in the same system, especially when they are highly related, is still an open issue.

The proposal presented here derives from three previous works. First, in [61] the concept of *Approximate Model Transformations* (AMT) was introduced to query streams of independent events. The idea was to trade accuracy for performance, using sampling techniques to reduce the source datasets. Then, in [9] we proposed a solution that works with graph-structured systems composed of persistent data and streams of transient information, based on CEP concepts and languages. However, no techniques were proposed to improve the performance of the queries. Finally, in [10] we extended that work, proposing different mechanisms for reducing the source input data so that performance could be improved at the cost of sacrificing correctness, i.e. obtaining approximate results. A proposal for the estimation of the accuracy of the results was also presented.

This paper presents a new approach for querying graph-structured information flows that is able to deal with both persistent and transient data at the same time and that aims at improving the performance of the queries by reducing the dataset to be processed without compromising the accuracy of the results. For this, we use models to represent the datasets, and a new algorithm has been developed, called Source Dataset Reduction (SDR) that obtains a subgraph of the complete dataset (i.e. the model) with the elements that are relevant to a given query. In this way, we are able to achieve speedups of more than 100× 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 used in Sect. 3.1 to illustrate our proposal and 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 [52,57]. Our solution has been implemented using the TinkerGraph in-memory graph database [58], since its execution time is lower than other similar solutions. Gremlin has been used as the query language because it presents some relevant benefits over other graph query languages (cf. Sect. 2.3 and [29]).

The structure of this paper is as follows. After this introduction, Sect. 2 presents the basic technologies used in the paper and discusses the reasons that justify their selection. Section 3 presents a case study that is used to illustrate our proposal and 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, Sect. 4 presents the algorithm, which is evaluated in Sect. 5, and the results are presented in Sect. 6. Finally, Sect. 7 relates our work to other similar proposals, while Sect. 8 concludes with an outline on future work. In addition, two appendixes explain the proposed algorithm in detail (A and B) and a third appendix shows several figures and tables with the results of our experiments.

2 Background

This section presents the information required for setting the context of our approach and explaining the technology used to implement it.

2.1 Streaming data processing and dynamic graphs

Data streaming applications were created to handle and process large amounts of information coming from external sources that generate flows of data at high speeds (often gigabytes per second). In order to obtain instantaneous responses when processing these data, high-performance algorithms are required, which impose stringent requirements in terms of resource consumption (execution time and memory). Common approaches to increase performance and decrease resource requirements include selecting a subset of the information to be queried [10,24,26,61], e.g. in Complex Event Processing systems [22,23,38,39], or dividing the data in subsets that can be processed in parallel [35,49,67], such as in *Apache Kafka* [33] and *Apache Spark* [4] platforms.

Generally, the information processed by these technologies is represented as sequences of simple events without any strong relations between them, apart from their partial order. However, many sources generate information where data are organized into complex structures. In these cases, the data elements are connected among themselves conforming trees or graphs. For example, in social networks, users are connected to other users and can publish pictures or posts that can in turn reference other users, pictures or posts. 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* [16,17]. 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.

2.2 Graph processing systems

Information models are normally represented by graph structures, which are composed of nodes and edges. Nodes in a graph represent *objects* in the model, and edges represent their *relationships*. By *elements*, we will refer to objects or relationships, indistinctly. Node properties represent object *attributes*.

Various technologies support efficient graph processing. For instance, Apache Spark provides a component for graph-parallel computation called *GraphX* [27]. It uses RDDs to perform graph-related operations with a set of basic operators (e.g. *subgraph*, *joinVertices* or *groupEdges*) and operators for graph algorithms or analytic tasks, e.g. PageRank [48]. The absence of a domain-specific language for developing GraphX code was initially an important limitation, because this leads to the use of very complex patterns to define information queries [9]. To overcome this drawback, Apache Spark developed *GraphFrames* [5], which allow users to operate with graphs using *Spark DataFrames*. A DataFrame is a distributed collection of data organized in columns labelled by names. It provides the benefits of RDDs and Spark SQL. GraphFrames enable users to perform the same operations as GraphX, but in a more intuitive way because it uses DataFrames for handling the data. However, these technologies do not provide efficient methods for data updates since they were mainly designed to perform queries, and not for storing data.

As an alternative, graph databases represent an effective solution to store, update and perform queries over very large graph-structured datasets when the information is organized in terms of objects and relationships [68,70]. The queries are normally developed with graph-query languages that provide an intuitive and fast way to access the information stored in the database. Some of the most popular graph-query languages are *Gremlin* [6], *Cypher* [45] and *SPARQL* [66]. Several graph databases store the data on disk, such as *Neo4j* [44], *JanusGraph* [30] or *OrientDB* [18], while others implement in-memory graph databases, such as *TinkerGraph* [58], *Memgraph* [41] and in-memory implementations of JanusGraph and OrientDB. Evidently, in-memory graph databases perform queries faster than those that store information on disk. In the following section, we justify the technology used in our approach.

2.3 Rationale for the chosen technology

In order to choose the most suitable technology to implement our proposal, we analysed the benefits and limitations of the solutions mentioned above.

First, our proposal uses 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 GraphX due to the complexity of the query expressions and the absence of clear syntax patterns for writing them. Second, our approach has to deal with systems that are constantly updated as new information arrives, and whose data can be modified as consequence of the queries. Thus, we discarded GraphFrame because it does not currently support efficient graph updates.

As mentioned earlier in Sect. 2.1, one of the main requirements when working with data flows is the need to process data in real time. In order to find the most suitable technology that fulfils these requirements, we compared different graph databases, namely Neo4j, Memgraph, TinkerGraph, JanusGraph, OrientDB,[1] and CrateDB[2] using two cases studies taken from previous works [9,55]. For the comparison, we measured the execution times of each database with different types of queries and the expressiveness of the languages used to write them. A technical report with the results is available on our website [12]. We found that the requisite for real-time processing is too restrictive for solutions that need to access disk even when they use indexing techniques. For this reason, we concluded that in-memory solutions are the most suitable choice, and hence, we decided to discard Neo4j, CrateDB and JanusGraph. Among the existing in-memory graph databases, we analysed TinkerGraph [58], Memgraph [41] and OrientDB [18]. However, OrientDB showed higher execution time averages than disk implementations. Then, we also decided to discard it. Regarding TinkerGraph and Memgraph, even when both technologies are suitable for implementing real-time applications, the results of our study showed that TinkerGraph is faster than Memgraph in most cases. In addition, Memgraph showed concurrency problems when creating new elements in parallel with models of around 4 million elements.

Regarding the pros and cons of the query languages provided by each solution, some studies (e.g. [29]), as well as our own analyses, show that although Cypher is usually easier to learn, the implementation of queries that imply *vicinity*—a concept that is present in graph-based structures [9]—is easier and more efficient with Gremlin. Besides, although the performance of TinkerGraph and Memgraph engines was similar, the more complete support of Gremlin by TinkerGraph, against the lack of full support of Cypher by Memgraph, i.e. the differences between Neo4j's Cypher and Memgraph's openCypher implementation [37], made us decide for TinkerGraph and Gremlin to implement our approach. Note, however, that the algorithm presented in this

[1] Note that we used the in-memory implementation for OrientDB and the BerkeleyDB backend [28] for JanusGraph.

[2] Even if CrateDB is not a graph database, we included it in our study because of its high scalability.

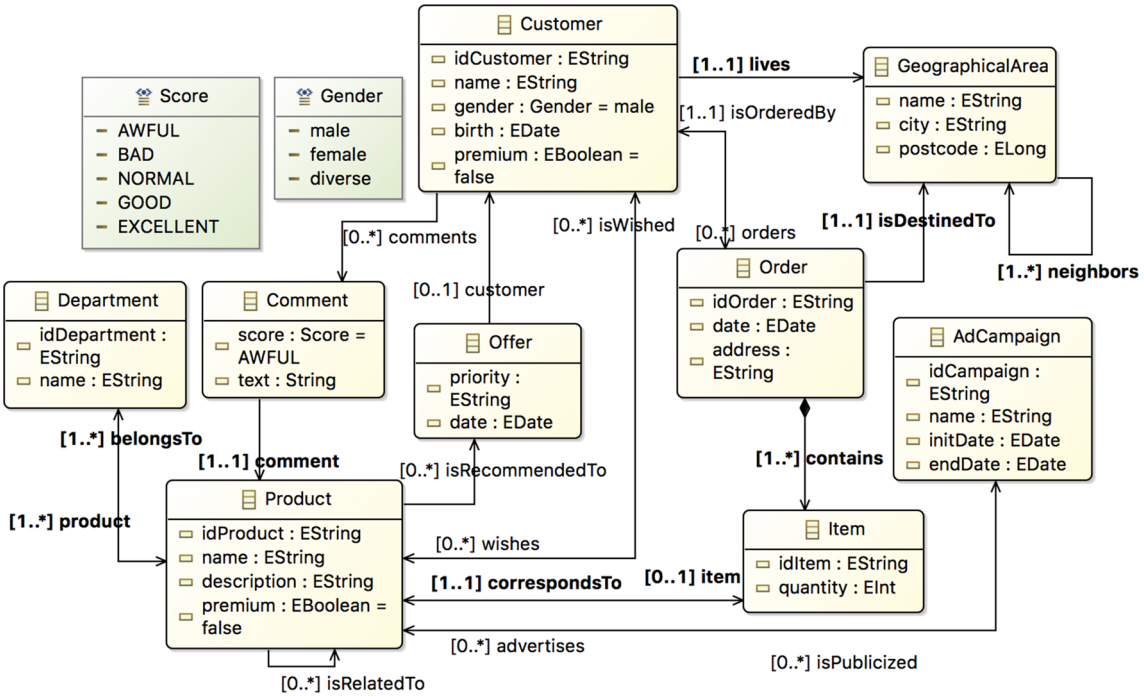


Fig. 1 The Amazon example metamodel

paper is technology-independent and therefore it could be implemented using other technologies.

3 A classification of queries

This section introduces a case study that is used to illustrate our proposal and describes a classification of query patterns that we have defined for the development of our algorithm.

3.1 Running example

To explain our approach, we will use the Amazon ordering service already presented in [10]. It is a simplification of the real Amazon ordering service but focusing on the features that are relevant to our proposal.

The metamodel for this case study is depicted in Fig. 1. It defines nine types of objects and different kinds of relationships among them. It models a system where `customers` can place `orders` on `items` of certain `products`. `Customers` can also `comment` on the `products`, and the system may create `offers` of `products` to specific `customers` and also create marketing `campaigns` to advertise some `products`. Finally, `products` belong to `departments`, and `customers` are located in `geographical areas`, where they live.

We have designed six queries that represent different features of interest to our proposal, realizing the query patterns that we have identified (see Sect. 3.2).

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.

3.2 Classification of queries

To reduce the source dataset 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. This classification takes 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 individually treated, i.e. they are described omitting any other pattern that could also be present in the query.

3.2.1 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 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

Listing 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 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 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))
6                         // property filter step
7 ) // "not" step ends.
8 // returns the customers of the first step:
9  .select("customers")
```

labelled as a variable `customers` that have ordered the products filtered in lines 2–4.

3.2.2 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 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.

3.2.3 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 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).

3.2.4 Conjunctive pattern

Queries that follow the *conjunctive pattern* select the information with an `and` clause that contains two or more

Listing 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
```

Listing 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   )
```

predicates. The query selects those elements that satisfy all predicates.

To illustrate an example of a query that follows this pattern, Listing 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).

3.2.5 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 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'`.

3.2.6 Aggregation pattern

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

An example of a query that follows this pattern is presented in Listing 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).

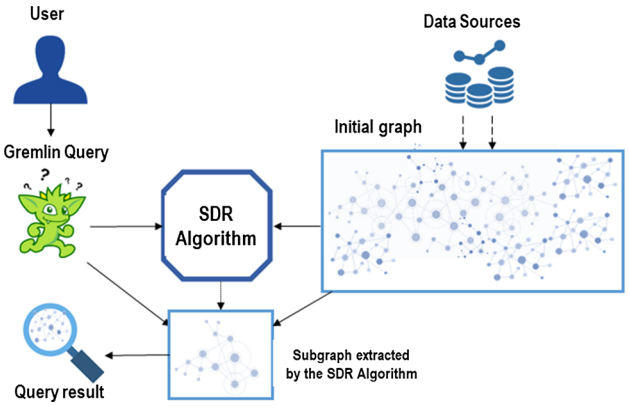


Fig. 2 Overall view of queries using the SDR algorithm

Then, it selects the customers who ordered this product at least 3 times (line 7).

4 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. We have called it *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 (Sect. 4.1). The second version incrementally updates that subgraph when new elements arrive to the system, or the persistent information changes (Sect. 4.2).

An overall view of our proposal and all its components are depicted in Fig. 2. The implementation of the SDR algorithms and all artefacts used in their evaluation are available from [11].

4.1 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 [48]. 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

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 neighbour pages in the previous iteration.

In a similar way, the SDR algorithm analyses 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. 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 Sect. 3.2, 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 into sub-steps.

Let us illustrate the steps of a query with the *ProductPopularity* query of the Amazon case study, shown in Listing 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 [60], which implements the vertex-centric programming model [34,40]. 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 neighbours 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.

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 neighbouring 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 neighbouring 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 A'.

4.1.1 Iteration 0: weight initialization

When the algorithm starts, it calls the function `SDRVertex Centric`(Q, v) that will run over all objects in parallel (line 1). Then, this function retrieves (line 6 of

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: $v.weight$ = **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 = $G - \{v_d \in V$ where $v_d.id \notin ListSGIds\}$

`Function SDRVertexCentric(`Q, v`)`
1: Obtain the set S of steps of Q
2: *iteration* = 0, *weight* = 0
3: **while** *iteration* <= S.size **do**
4: *guardCondition* = true
5: **if** *iteration* == 0 **then**
6: s = S.get(S.size – 1)
7: weight = **WeightInitialisation**(s, v)
8: **else**
9: Select s = S.get(S.size – *iteration*)
10: **if** *iteration* == 1 **then**
11: $weight$ = **InWeightPropagation**(s, v, $weight$)
12: **else**
13: $weight$ = **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

`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 neighbours 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 neighbours 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**[3] 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 B'.

[3] 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 neighbours 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`).

4.1.2 Iteration 1: initial weight propagation

After `WeightInitialisation`, all objects have a weight but they are not aware of their neighbours' weights. This is the goal of the `InWeightPropagation`(s, v, $weight$) function, which proceeds with the same step s (lines 9–12 of `SDRVertexCentric`). The behaviour 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 neighbours (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 analysed 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`).

4.1.3 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 neighbours through s, it sends messages to its neighbours; 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 19–21. Once `SDRVertexCentric` is executed on the objects of the initial model, the subgraph composed of the nonzero 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 nonzero 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.

4.2 The incremental SDR algorithm

Our approach is designed for dynamic systems that are constantly updated with new information. Executing the main SDR algorithm (Sect. 4.1) 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.

When new elements are added, updated or removed from the graph, we analyse the query and obtain a list of the neighbours 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 sys-

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: $SG_i = SG_i \cup$ **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: $SG_i = SG_i \cup$ **createSubGraph**(s_{SQ}, V_n)
13: **end if**
14: **end for**
15: **end for**
16: **end if**
17: **end for**
18: $v_i.weight =$ **SDRVertexCentric**$(Q, v_i) \forall\ v_i \in V_i$
19: *ListWeights* = get *weight* and *id* from SG_i
20: Update *ListSGIds* with *ListWeights*
21: **return** SG = $G - \{v_d \in V$ where $v_d.id \notin ListSGIds\}$

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 $\cup$ *forward relationships of* s
3: $previous_r$ = *backward relationships of* s
4: **for** $n : next_r \cup p : previous_r$ **do**
5: $SG = SG \cup$ *neighbors of* V_n *through* n *and* p
6: **end for**
7: **return** SG

tem, 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.

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 neighbours 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 analysed to obtain the updated weights of the objects of SG_i, and the list *ListSGIds* (cf. Sect. 4.1.3) 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 neighbours 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 neighbours of V_n that can be reached through them (lines 4–6) and returns the subgraph composed by the set V_n and their neighbours. 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 analyse 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.

The fact that the incremental SDR algorithm has to update only the weights of the neighbours 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 neighbours of discarded elements), r is the number of relationships of s, and n is the number of neighbours 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 Evaluation

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 analyse 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 analyse the break-even point, i.e. how many queries are needed to amortize such initial costs, making our approach worthwhile.

5.1 Experimental setup

5.1.1 Case studies

In order to evaluate our proposal, we have performed our experiments in three case studies:

1. **Amazon**: It was described in Sect. 3.1.
2. **NY Caption Contest**: this case study is extracted from the New Yorker caption contest dataset [57]. 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:
   - Dueling questions: two captions are shown for the same cartoon and users have to choose the funniest one.
   - Cardinal questions: a cartoon with a caption is displayed and users have to score how funny they are by selecting either 'unfunny', 'somewhat funny' or 'funny'.
3. **YouTube videos**: this case study uses the YouTube-BoundingBoxes dataset [52], which consists of approximately 380,000 video segments of 15–20 s 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.

Table 1 Summary of the models used in the experiments

Case study	Name	Objects	Relationships
Amazon	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
Contest	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
YouTube	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

The metamodels and queries of all case studies are described in detail on our project's website [12].

5.1.2 Source models

Our experiments have been run on models of different sizes in order to analyse the performance of our approach. The number of objects and relationships for each model are shown in Table 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.

5.1.3 Queries

As described in Sect. 3.2, 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 2 summarizes all the queries we have used. They are fully described and implemented on our Git repository [11]. Note that the objects of queries that involve a specific object

Table 2 Summary of the queries used in the experiments

Case study	Query name	Query pattern	Description
Amazon	ProductPopularity	Simple	Description of **Q1** given in Sect. 3.1
	ProductPopularityC	Conditional	Description of **Q1** given in Sect. 3.1
	AlternativeCustomer	Neg	Description of **Q2** given in Sect. 3.1
	PackagePopularity	Conjunctive	Description of **Q3** given in Sect. 3.1
	SimProductsPopularity	Disjunctive	Description of **Q4** given in Sect. 3.1
	PrefCustomer	Aggregation	Description of **Q5** given in Sect. 3.1
	PrefCustomerSimProducts	Aggregation	Description of **Q6** given in Sect. 3.1
Contest	RecentPart	Simple	Participants who have answered at least one question in the last month
	ContestPart	Conditional	Participants who have answered at least one question in a specific contest
	UnchosenCap	Conjunctive	Counts how many times a caption appeared in a dueling contest question and was not chosen
	FunniestCaption	Aggregation	Gets the highest scored caption in a cardinal contest
	Abandon	Aggregation	Participants who answered only one question
	FunniestCaptionU	Agg and Conj	Gets the highest scored caption in a cardinal contest
YouTube	GetAnimalVideos	Conditional	Get all videos that contain an animal
	NotPresent	Neg	Segments where the object is not present
	AnimalPerson	Conjunctive	Videos that contain at least one animal and one person
	PresentSoon	Conjunctive	Videos where the object is present during the first 3 s
	Pets	Disjunctive	Gets all frames that contain a cat or a dog
	InCast	Aggregation	Videos where the object is present in at least 10 segments

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

5.1.4 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 [58], 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.

5.2 Experiments and data collection

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 increment in the volume of the graph, i.e. they are the most costly operation when working with streams. Then, the results reflect the behaviour of our approach in the worst-case scenario. This way, we aim to evaluate both our SDR algorithm (Sect. 4.1) and its incremental version (Sect. 4.2). Both sets of experiments are described next.

5.2.1 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 1 and 2, respectively. Table 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

Table 3 Elements savings ratio when running the SDR algorithm

Case study	Query name	Pattern	Models					
Amazon			2M	4M	8M	15M		
	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	44.23	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		
Contest			1M	4M	9M	12M	16M	
	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	
YouTube			2M	4M	6M	8M	10M	12M
	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

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 3, 5 and 6 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 labelled 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 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.

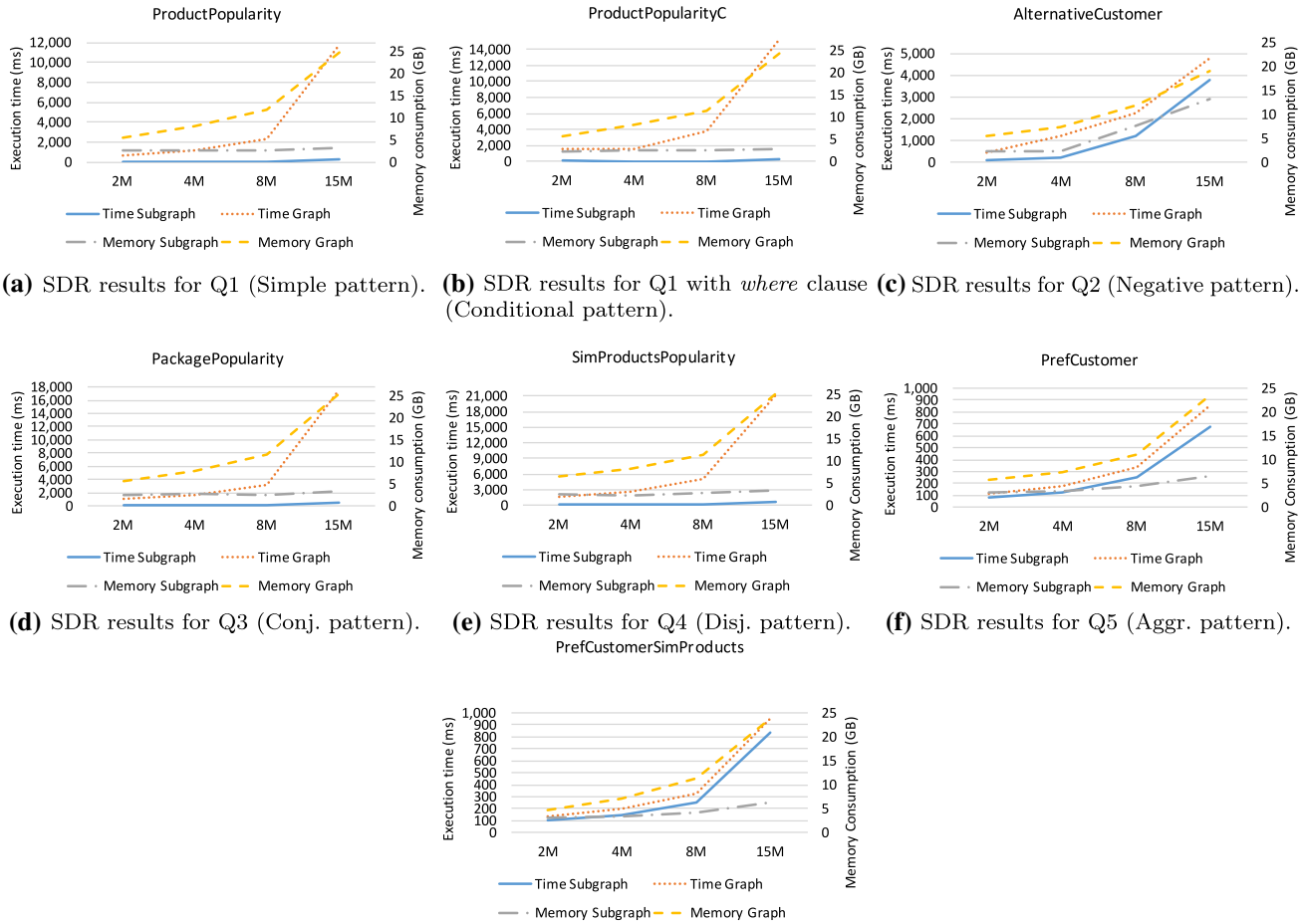
Table 4 summarizes in tabular format the information displayed in Figures 3, 5 and 6 with the times (in ms) of the queries when executed on the complete graph (T_g), on the reduced subgraph as calculated by the SDR algorithm (T_{sg}), and the corresponding speedups (S). Note that we are able to obtain the results below 1 second in most cases, when the queries on the complete graph took much longer. Recall that the results shown in Figures 3, 5 and 6 and Table 4 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.

5.2.2 Experiments with streams of information

The second set of experiments is devoted to analyse our approach when dealing with dynamic graphs. For this, we need to mimic the arrival of new information. In particular,

Table 4 Execution times (ms) of queries with the complete graph (T_g), the subgraph (T_{sg}), and speedups (S)

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	1121	36	31.14	2333	37	63.05	11,793	291	40.53
ProductPopularityC	Cond.	1495	40	37.38	1607	35	45.91	3855	31	124.35	15,178	327	46.46
AlternativeCustomer	Neg.	439	82	5.35	1182	239	4.95	2258	1194	1.89	4785	3788	1.26
PackagePopularity	Conj.	1027	75	13.69	1641	53	30.96	3152	67	0.4704	17,561	505	34.77
SimProductsPopularity	Disj.	1739	105	16.56	2556	83	30.80	4985	104	0.4793	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
		1M			4M			9M			12M		
Contest Case		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	1470	78	18.85	4047	78	51.88	4250	74	57.43
ContestPart	Cond.	610	88	6.93	2954	81	36.47	5642	64	88.16	6479	86	75.34
UnchosenCap	Conj.	497	62	8.02	2562	123	20.83	4832	94	0.5140	5613	92	61.01
FunniestCaption	Aggr.	56,377	499	112.98	197,117	4045	48.73	375,390	8641	43.44	556,320	12,167	45.72
Abandon	Aggr.	245	189	1.30	1695	893	1.90	2844	1939	1.47	4365	2808	1.55
FunniestCaptionU	Aggr. & Conj.	15,535	312	49.79	21,617	314	68.84	39,343	669	58.81	40,790	757	53.88
		2M			4M			6M			8M		
YouTube Case		T_g	T_{sg}	S	T_g	T_{sg}	S	T_g	T_{sg}	S	T_g	T_{sg}	S
GetAnimalVideos	Cond.	1485	15	99.00	2851	21	135.76	3931	22	178.68	4283	35	122.37
NotPresent	Neg.	280	17	16.47	795	30	26.50	929	49	18.96	1818	72	25.25
AnimalPerson	Conj.	1400	29	48.29	3470	39	88.97	3432	49	70.04	4513	54	83.57
PresentSoon	Conj.	1200	46	26.09	2743	87	31.53	3122	95	32.86	3849	109	35.31
Pets	Disj.	2,166	138	15.70	4075	256	15.92	8199	379	21.63	10,696	552	19.38
InCast	Aggr.	688	289	2.38	1817	864	2.10	2745	1081	2.54	2883	1444	2.00



(a) SDR results for Q1 (Simple pattern). **(b)** SDR results for Q1 with *where* clause (Conditional pattern). **(c)** SDR results for Q2 (Negative pattern).

(d) SDR results for Q3 (Conj. pattern). **(e)** SDR results for Q4 (Disj. pattern). **(f)** SDR results for Q5 (Aggr. pattern).

(g) SDR results for Q6 (Aggr. pattern).

Fig. 3 Performance results of the SDR algorithm for the Amazon queries

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, but the time taken by the incremental algorithm is not 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 and Tables 12 and 13 of 'Appendix C'. Queries are executed every time α 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 β new records have arrived. For instance, if $\alpha = 5$ and $\beta = 100$, it means that the query is executed every time five 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

Table 5 Gain ratio when using the incremental algorithm in the Amazon case study

Query Name		Models							
		$\alpha = 5$				$\alpha = 10$			
	β	2M	4M	8M	15M	2M	4M	8M	15M
ProductPopularity (Simple)	50	−0.0274	−0.0104	**0.0486**	**0.0155**	−0.2043	−0.1492	−0.0649	−0.0986
	100	**0.1198**	**0.1834**	0.2032	0.1875	−0.0176	−0.0220	**0.0760**	**0.0602**
	150	0.1874	0.2274	0.2647	0.2718	−0.0101	**0.0827**	0.1272	.1585
	200	0.2021	0.2771	0.3018	0.3363	**0.0656**	0.1215	0.1821	.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	**0.0119**	**0.0666**	−0.2080	−0.1541	−0.1271	−0.0677
	100	**0.0704**	**0.1319**	0.1468	0.2048	−0.0708	−0.0163	**0.0456**	**0.0572**
	150	0.1546	0.2095	0.2110	0.2915	**0.0347**	**0.0522**	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	**0.0006**	**0.0034**	−0.0985	−0.0880	−0.0805	−0.0337
	200	−0.0461	**0.0186**	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	**0.0704**	**0.1187**	**0.1596**	**0.1632**	−0.0485	−0.0714	−0.0594	**0.0112**
	150	0.1396	0.2072	0.2123	0.2764	−0.0445	**0.0305**	**0.0692**	0.0792
	200	0.1730	0.2486	0.2499	0.3607	**0.0123**	.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	**0.1372**	**0.1514**	−0.2326	−0.1121	−0.1092	−0.0574
	100	**0.1525**	**0.1721**	0.2617	0.2718	−0.0111	**0.0096**	**0.0463**	**0.1414**
	150	0.2337	0.3121	0.3335	0.3965	**0.0477**	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

approach (*SubG*) is slower than *CG*. The execution times in absolute terms for all experiments are shown on our project's website [12]. As we can see in the tables, the results are organized by values of α and β, queries and size of models. For each value of α and each model, the results are to be read vertically. Values in bold represent the first value of β where the execution time of *SubG* is faster than *CG*. For instance, consider the values for the query *ProductPopularity* in Table 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.

Note that some numbers are not shown in Table 12. This is because the queries performed on *CG* take too long and the break-even point has already been reached.

Table 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.

5.3 Functional correctness

In order to test the proposed algorithms and to check that their behaviour 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.

6 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.

6.1 RQ1: graph size reduction

To answer the first research question, Table 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. Sect. 4.1). 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).

6.2 RQ2: performance improvement

Figures 3, 5 and 6 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 3a and 5a), conditional (Figures 3b, 5b and 6a), conjunctive (Figures 3d, 5c, 6c and 6d) and disjunctive (Figures 3e and 6e) 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 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 for an example). Figures 3f, 3g, 5e and 6f show situations where the performance using the graph and subgraph is practically the same, because these steps are very costly (in Table 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 5d and 5f, mainly because the aggregation filter is solved faster. In these queries, the speedup is above 40 in all cases (Table 4).

Table 6 Number of query executions needed to obtain a positive gain for each query

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

Figures 3c and 6b show the results for queries that follow a negative pattern. Again, the more elements matching the pattern, the better the performance improvement.

6.3 RQ3: execution time gains with data streams

Tables 5, 12 and 13 show the results of running the algorithms when dealing with data streaming applications (cf. Sect. 5.2.2), where queries are executed while new data is constantly arriving and being added to the model. These results are summarized in Table 6. Recall that β is the total number of records added per experiment, while α represents the size of the new records batch that have arrived at the system each time the query is run. This means that, for a constant value of α, the higher the value of β, the higher number of times the query of each experiment is executed.

To analyse 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 β, is highlighted in bold. This is what we call the break-even point. In Tables 5, 12 and 13, the break-even point is shown as gain ratio, while Table 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 β and inversely proportional to the value of α. Another hypothesis is that time gain also increases with the model size. Tables 5, 12 and 13 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 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, disjunctive queries achieve the highest gain (see *SimProductsPopularity* in Table 5 and *Pets* in Table 13), followed by simple and conditional queries, which have a similar gain (see *ProductPopularity* and *ProductPopularityC* in Table 5, *RecentPart* and *ContestPart* in Table 12 and *GetAnimalVideos* in Table 13), where conditional queries have a slightly higher gain than simple queries. Then, conjunctive queries (see *PackagePopularity* in Table 5, *UnchosenCap* in Table 12, and *AnimalPerson* and *PresentSoon* in Table 13) have a higher gain than negative queries (*AlternativeCustomer* in Table 5 and *NotPresent* in Table 13). Regarding aggregation queries, they present very different gain values in the three case studies. For example, observe how the *FunniestCaption* query in Table 12 has a gain higher than 70% for all α and β values, whereas *PrefCustomer* in

Table 5 does not present any positive gain for any α and β 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 the 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.

6.4 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. Sect. 2.3) have some support to implement them. Some examples are the indexing of objects and relationships from TinkerGraph [59], the indexing of labels and properties from Neo4j [46], or Memgraph label and label-property indices [42].

According to the classification presented in Sect. 3.2, 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 [43], this is still an open issue to be addressed [65]. For this reason, in the present paper, we have addressed the improvement in query performance on graphs from a different perspective that does not use indices. Nevertheless, our work does 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 paper, so we consider them as future work.

6.5 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. [69].

6.5.1 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 the results obtained by the different metrics are consistent when drawing the conclusions, we consider the mono-method bias threat neutralized.

6.5.2 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 [11,12].

6.5.3 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 analyse the behaviour 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 dataset might have also affected the validity of our results. In order to mitigate this threat, we have analysed 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, 12 and 13).

6.5.4 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 datasets, 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 Sect. 2.3, 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.

7 Related work

Our work mainly derives from three of our previous works. First, in [61] we introduced the concept of *Approximate Model Transformations* (AMT) to query streams of independent events. The idea was to trade accuracy for performance, using sampling techniques to reduce the source datasets. However, we did not consider graph-structured data but unrelated events Then, in [9] we used CEP concepts and languages to be able to deal with graph-structured systems composed of persistent data and streams of transient information. However, we did not propose any solution to speed-up the queries. Finally, in [10] we extended that work, proposing different mechanisms for reducing the source data to improve performance at the expense of sacrificing correctness, i.e. obtaining approximate results. A proposal for the estimation of the resulting error of such approximations in terms of precision, recall and accuracy was also presented. The present work aims at improving performance without sacrificing correctness, by using a precomputation step that takes into account the syntax of the query to reduce the source graph.

Two surveys about approximate query processing mention a precomputation step in order to select important information for the query before it is executed [21,36]. 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 [1,2,7,19,20]. 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. [26], 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 our approach, 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. [25], 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 [62,63]. 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 uptaded.

Some other works that deal with the arrival of new information propose incremental transformations, where the input model changes over time [14,31,32,51,54,64]. They present partial and incremental model transformations using EMF-IncQuery and EMF-IncQuery-D frameworks [14,54,64], an incremental algorithm for ATL [32], a framework for the instant and incremental transformation of changes among models [31], and a partial evaluator prototype called Qvt-Mix [51]. 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. [15] 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. The 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 [56] and OrientDB's *LiveQuery* [47]. 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. [8] 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. [3] 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 behaviour and performance of the proposed algorithm.

Finally, our algorithm was developed considering the rationale behind Google's PageRank algorithm [48]. 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 [53] 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.

8 Conclusions and future work

In this paper, we have designed and developed an algorithm that implements an offline technique to optimize the performance of queries on dynamic graphs. This algorithm identifies a subgraph of the original model that contains the data relevant to the query, and on which the query can be more efficiently executed. Inspired by Google's PageRank algorithm, it does so by assigning a weight to all objects in the graph according to their relevance for the query. The algorithm returns the subgraph with the objects with a weight greater than 0 and the relationships among them, which corresponds to the subgraph that contains all the elements that are relevant to the query. Furthermore, as new information arrives and is added to the system, the subgraph is updated using another algorithm, the so-called incremental SDR.

We have also defined six types of patterns that can be found in queries over graph-structured data according to their structure and used them to improve the performance of the algorithms.

Our experiments show that, by querying the subgraph obtained by the SDR algorithm instead of the complete source graph, an improvement in the performance of all query patterns is achieved. We have also shown that these improvements increase with the original graph size, as well as with the number of times the query is executed.

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 as part of our future work. We also plan to study how to improve the performance of negative 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.

Our work can be extended towards several directions, which may constitute interesting research lines. First, in this paper we have considered queries that follow mostly one pattern, in order to characterize their behaviour. The analysis of more complex queries with more patterns could also be of interest. Furthermore, since the SDR algorithm is technology-independent, we could implement this algorithm with a different technology and compare the results. Another interesting research line has to do with the penalty introduced in the latest update of the query subgraph when

executing the incremental algorithm. In particular, it is worth studying whether such penalty is influenced by (i) the execution frequency of the incremental algorithm, (ii) the number of objects contained in the batch and (iii) the type of query.

Finally, we also aim to consider other techniques that are typically used to improve the performance of queries, such as indexing or approximation techniques, and how they can be combined with our proposal. For example, indexing could be applied to the entire graph in order to obtain the subgraph calculated with the SDR algorithm faster, while approximation techniques could be applied to the subgraph obtained with the SDR algorithm. The combination with approximation techniques could also be of interest, e.g. by selecting only those elements whose relevance to the query, as indicated by their weights, was above a certain threshold. In addition, spatial and temporal windows [10] as well as random techniques could be also applied to reduce the subgraph size. In these cases, the improvement on performance would be in detriment of the accuracy of the results.

Verifiability

For the sake of verifiability, our prototype as well as all artifacts of the experiments and their descriptions are available on our project website [12] and Github [11].

Acknowledgements This work is partially supported by the European Commission (FEDER) and the Spanish Government under projects APOLO (US-1264651), HORATIO (RTI2018-101204-B-C21), EKIPMENT-PLUS (P18-FR-2895) and COSCA (PGC2018-094905-B-I00).

A Appendix: *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. 4. In this case, the graph contains two customers (C1 and C2) and two products (P10 and P20). C1 orders two 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 2. The updates of the weight for each object as iterations run are displayed in Table 7. 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[4]. 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 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 4, 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

[4] 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.

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 2, this step is the relationship step `contains`.
(d) Therefore, it counts the number of neighbours 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 = 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 $v \neq P10$ *guardCondition* is false so *weight* remains 0. Note how in the second column of Table 7 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 7 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).
– $iteration \neq 0$, so the function gets into *else* clause of line 8 and selects the relationship step `orders` (line 3 in Listing 2) for s (line 9).
– Besides, $iteration \neq 1$, 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 neighbours 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 7.

(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 7.
  (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 B'.
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 7 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 4, the result will be object C1 for both executions.

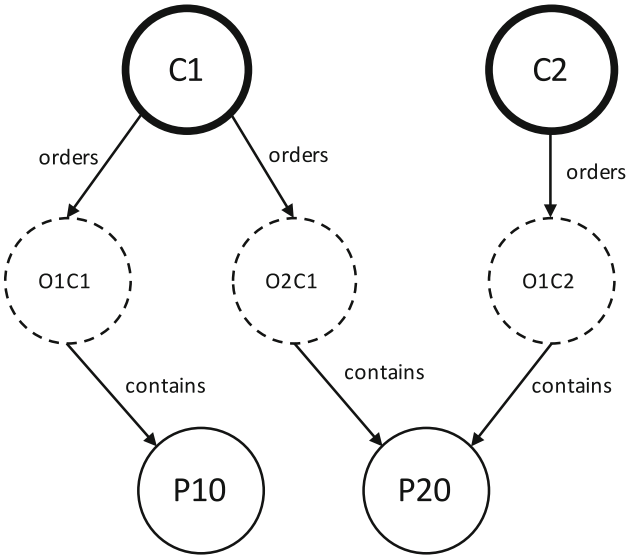


Fig. 4 Graph 1: example for Amazon case

B Appendix: 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 4.

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.

Let us consider the sample query shown in Listing 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 4, 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 7. 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 7 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 Sect. 4.1. After that, the algorithm continues the normal execution updating the calculated weights according to the remaining steps of the query.

Table 7 Object weights for *ProductPopularity* query with SDR algorithm

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

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 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 4, 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:

$$\text{weight} = \begin{cases} 0 \text{ if weight} > 0 \\ 1 \text{ if weight} \leq 0 \end{cases} + \text{pItWeight}$$

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 4 in column 5 of Table 8.

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 8 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.

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

Table 8 Object weights for *ProductPopularity* query with *not* step with SDR algorithm

Object/iteration	It 0				It 1
C1	0	0	0	(1 → 0) + 0 = 0	0
C2	0	0	0	(0 → 1) + 0 = 1	1
O1C1	0	0	2	(2 → 0) + 2 = 2	2
O2C1	0	0	0	(0 → 1) + 0 = 1	1
O1C2	0	0	0	(0 → 1) + 0 = 1	1
P10	1	1	1	(1 → 0) + 1 = 1	1
P20	0	0	0	(0 → 1) + 0 = 1	1

Table 9 Object weights for subquery example with SDR algorithm

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

one predicate, there are more than one subquery where to compute the weights too.

Let us consider Listing 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 9—note that four iterations are displayed in the table because it is focused on the subquery. In this way, the results for both subqueries with the SDR algorithm are shown in columns 2–5 of Tables 7 and 9, respectively. For the *and* step, the algorithm computes the weights for both queries separately and performs the following operation to merge them:

$$weight = \sum_{i=1}^{n} \text{pItWeight}_i + \prod_{i=1}^{n} \text{weight}_i$$

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. The results of the weights computed for the *PackagePopularity* query are shown in Table 10. 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 'Appendix B.2'.

Table 10 Object weights for *PackagePopularity* example with SDR algorithm

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 11 Object weights for *SimProductsPopularity* example with SDR algorithm

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

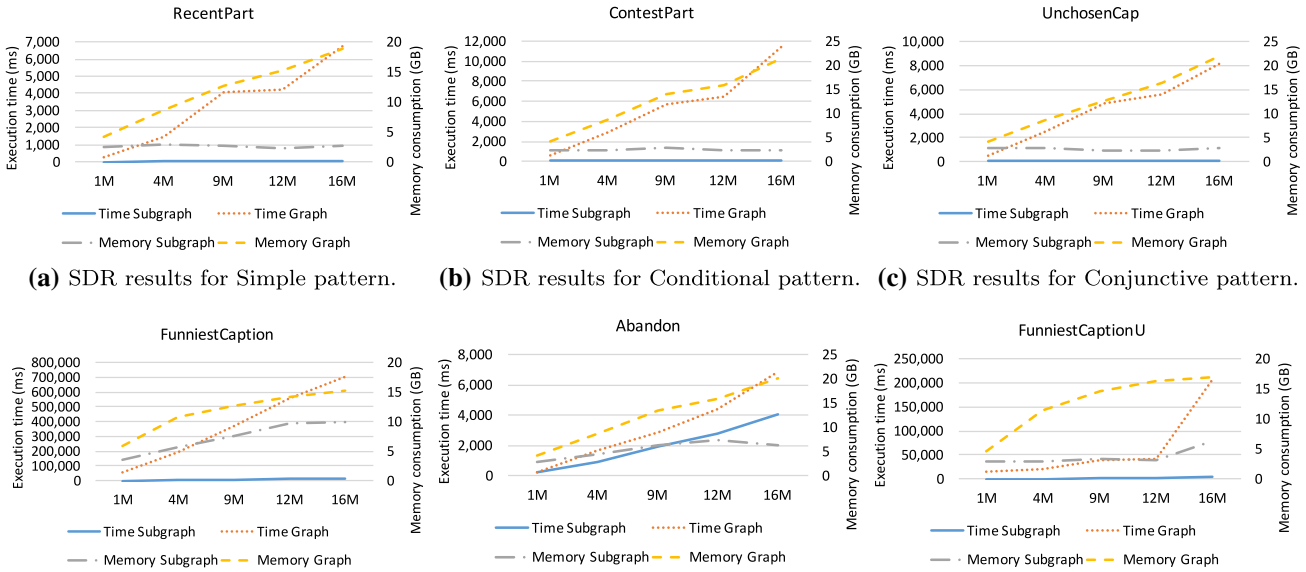
Or Step

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.

Let us consider we modify in Listing 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 7 and Table 9, the *or* step performs the following merge of subqueries:

$$\text{weight} = \sum_{i=1}^{n} \text{weight}_i$$

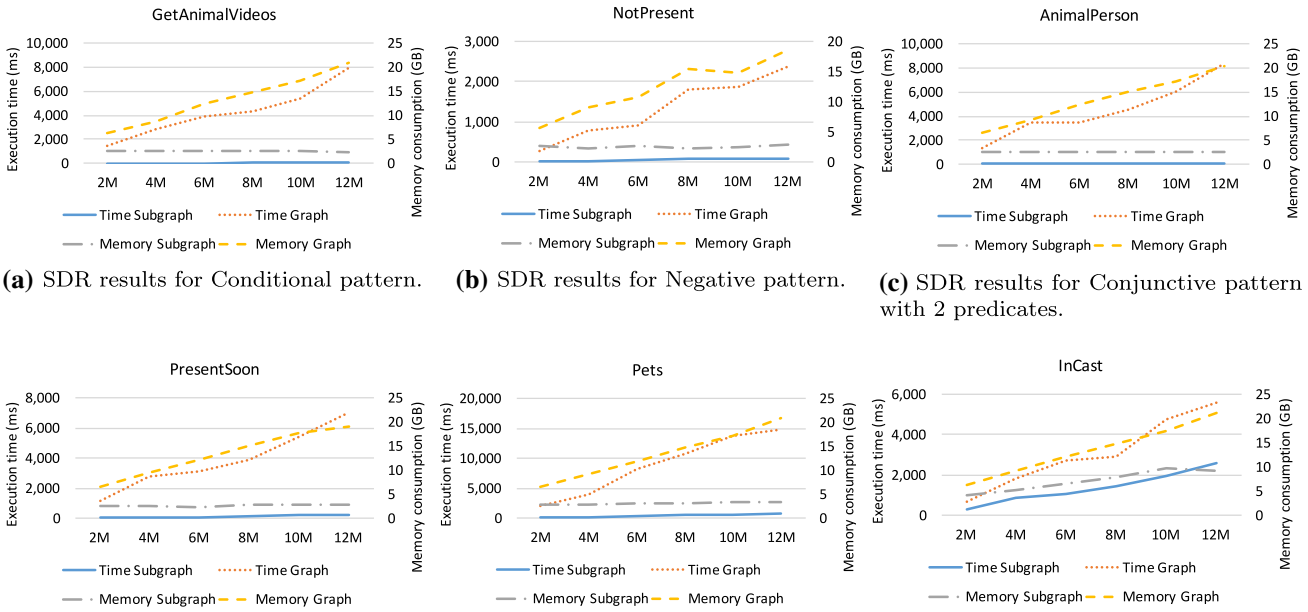
Being n the number of predicates contained in the *or* step and $weight_i$ the calculated weight of the subquery of the predicate i. The results for *SimProductsPopularity* query over the graph of Fig. 4 are shown in Table 11. Note that with the simple graph of Figure 4, all the objects in the graph are



(a) SDR results for Simple pattern. **(b)** SDR results for Conditional pattern. **(c)** SDR results for Conjunctive pattern.

(d) SDR results for Aggregation pattern. **(e)** SDR results for Aggregation pattern. **(f)** SDR results for Aggregation and Conjunctive pattern.

Fig. 5 Performance results for SDR algorithm in contest example queries



(a) SDR results for Conditional pattern. **(b)** SDR results for Negative pattern. **(c)** SDR results for Conjunctive pattern with 2 predicates.

(d) SDR results for Conjunctive pattern with 3 predicates. **(e)** SDR results for Disjunctive pattern. **(f)** SDR results for Aggregation pattern.

Fig. 6 Performance results for SDR algorithm in YouTube example queries

assigned weights > 0. This would not be the case in a real system, where many elements would be discarded, as we describe in Sect. 5.

C Appendix: Additional charts and tables displaying experiments results

To improve the readability of the manuscript, this appendix contains some of the tables and figures that show the results

Table 12 Ratio incremental gain results for Contest case study

Query Name		Models							
		$\alpha = 5$				$\alpha = 10$			
	β	1M	4M	9M	12M	1M	4M	9M	12M
RecentPart (Simple)	50	−0.1308	−0.0055	**0.1260**	**0.1457**	−0.3042	−0.0773	**0.0280**	**0.2001**
	100	−0.1054	**0.0915**	0.2004	0.2934	−0.2289	**0.0222**	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	**0.0413**	**0.1419**	**0.2094**	−0.3830	−0.0405	**0.0304**	**0.1322**
	100	−0.0419	0.1647	0.2586	0.3449	−0.2266	**0.0628**	0.1177	0.2944
	150	**0.0118**	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	**0.0781**	**0.1573**	−0.4187	−0.0631	−0.0055	**0.0563**
	100	−0.1289	**0.1242**	0.1820	0.3161	−0.2622	**0.0616**	**0.1023**	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	**0.7680**	**0.6378**	**0.6803**	**0.8520**	**0.7680**	**0.6378**	**0.6803**	**0.8520**
	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	**0.0783**	**0.0992**	−0.3074	−0.0394	−0.0162	**0.0008**
	100	−0.1580	**0.0647**	0.1569	0.2389	−0.1605	**0.0626**	**0.0611**	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	**0.3517**	−0.2278	−0.1412	−0.0071	**0.3517**	−0.2278	−0.1412	−0.0071
	10	0.6859	**0.2869**	**0.3416**	**0.4921**	0.5461	**0.1331**	**0.1371**	**0.3585**
	20	0.7596	0.4458	–	–	0.6181	0.2902	0.2865	0.5142
	50	0.8145	0.6193	–	–	–	–	–	–
	100	0.8563	0.6602	–	–	-	-	–	–

of the evaluations. Specifically, Figs. 5 and 6 show the execution time and memory consumption of the experiments with static information of *Contest* and *YouTube* case studies, respectively. Then, Tables 12 and 13 show the gain results of the experiments with dynamic information of the same case studies.

Table 13 Ratio incremental gain results for Youtube case study

Query name		Models							
		$\alpha = 5$				$\alpha = 10$			
	β	2M	4M	6M	8M	2M	4M	6M	8M
GetAnimalVideos (Conditional)	50	− 0.0626	**0.0443**	**0.0761**	**0.1770**	− 0.1712	**0.0113**	**0.0317**	**0.1305**
	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	**0.0296**	**0.0617**	− 0.2094	− 0.0930	− 0.0118	**0.1231**
	100	− 0.2305	− 0.0230	0.1106	0.1157	− 0.1359	**0.0168**	**0.0790**	0.2155
	150	− 0.2195	**0.0087**	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	**0.0048**	**0.0635**	**0.1892**	− 0.3417	− 0.0806	− 0.0405	**0.0653**
	100	− 0.1288	0.1011	0.1633	0.2757	− 0.2473	**0.0081**	**0.0563**	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	**0.0273**	**0.1051**	− 0.1661	− 0.0780	− 0.0275	**0.0162**
	100	− 0.1376	**0.0806**	0.1389	0.2266	− 0.1034	**0.0341**	**0.0728**	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	**0.0540**	**0.1135**	**0.1910**	**0.2368**	− 0.0213	− 0.0009	**0.1418**	**0.1334**
	100	0.1654	0.2239	0.3177	0.3244	**0.0533**	**0.1239**	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	**0.0419**	− 0.3103	− 0.1052	− 0.0306	**0.0649**
	100	− 0.1728	− 0.0020	**0.0411**	0.1372	− 0.1885	− 0.0016	**0.0379**	0.1518
	150	− 0.1280	**0.0230**	0.0933	0.1723	− 0.1480	**0.0325**	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

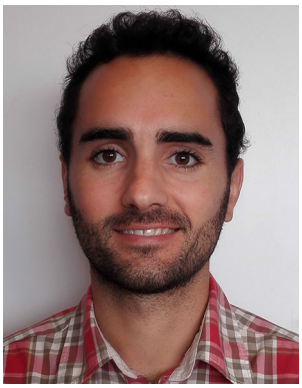
References

1. Acharya, S., Gibbons, P.B., Poosala, V.: Congressional samples for approximate answering of group-by queries. In: Proc. of SIGMOD'00, pp. 487–498. ACM (2000). https://doi.org/10.1145/342009.335450
2. 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). https://doi.org/10.14778/2367502.2367533
3. 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). https://doi.org/10.1145/3104031
4. Apache Spark: Spark streaming programming. https://spark.apache.org/docs/latest/streaming-programming-guide.html. Accessed May 2019
5. Apache Spark: GraphFrames. https://graphframes.github.io/graphframes/docs/_site/index.html. Accessed Nov 2019
6. Apache TinkerPop: The Gremlin graph traversal machine and language. https://tinkerpop.apache.org/gremlin.html. Accessed Jan 2020
7. Babcock, B., Chaudhuri, S., Das, G.: Dynamic sample selection for approximate query processing. In: Proc. of SIGMOD'03, pp. 539–550. ACM (2003). https://doi.org/10.1145/872757.872822
8. Barceló, P.: Querying graph databases. In: Proc. of PODS'13, pp. 175–188. ACM (2013). https://doi.org/10.1145/2463664.2465216
9. Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Extending complex event processing to graph-structured information. In: Proc. of MODELS'18, pp. 166–175. ACM (2018). https://doi.org/10.1145/3239372.3239402
10. Barquero, G., Troya, J., Vallecillo, A.: Trading accuracy for performance in data processing applications. J. Object Technol. **18**(2), 9:1–9:24 (2019). https://doi.org/10.5381/jot.2019.18.2.a9

11. Barquero, G., Troya, J., Vallecillo, A.: SDR algorithm git repository. https://github.com/atenearesearchgroup/SDRalgorithm. Accessed Jan 2020
12. Barquero, G., Troya, J., Vallecillo, A.: SDR algorithm website. http://atenea.lcc.uma.es/projects/SDRAlg.html. Accessed Jan 2020
13. 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 Jan 2020
14. 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 MODELS'10, pp. 76–90 (2010). https://doi.org/10.1007/978-3-642-16145-2_6
15. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA model transformation system. In: Proc. of GRAMOT'08, pp. 25–32. ACM (2008)
16. Besta, M., Fischer, M., Kalavri, V., Kapralov, M., Hoefler, T.: Practice of streaming and dynamic graphs: concepts, models, systems, and parallelism. CoRR arXiv:1912.12740 (2019)
17. Besta, M., Peter, E., Gerstenberger, R., Fischer, M., Podstawski, M., Barthels, C., Alonso, G., Hoefler, T.: Demystifying graph databases: analysis and taxonomy of data organization, system designs, and graph queries. CoRR arXiv:1910.09017 (2019)
18. Callidus Software Inc.: OrientDB. The database designed for the modern world. https://orientdb.com/. Accessed June 2020
19. Chaudhuri, S., Das, G., Datar, M., Motwani, R., Narasayya, V.R.: Overcoming limitations of sampling for aggregation queries. In: Proc. of ICDE'01, pp. 534–542. IEEE Computer Society (2001). https://doi.org/10.1109/ICDE.2001.914867
20. Chaudhuri, S., Das, G., Narasayya, V.R.: A robust, optimization-based approach for approximate answering of aggregate queries. In: Proc. of SIGMOD'01, pp. 295–306. ACM (2001). https://doi.org/10.1145/375663.375694
21. Chaudhuri, S., Ding, B., Kandula, S.: Approximate query processing: no silver bullet. In: Proc. of SIGMOD'17, pp. 511–519. ACM (2017). https://doi.org/10.1145/3035918.3056097
22. 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). https://doi.org/10.1145/2187671.2187677
23. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications, New York (2010)
24. Fan, W., Geerts, F., Cao, Y., Deng, T., Lu, P.: Querying big data by accessing small data. In: Proc. of PODS'15, pp. 173–184. ACM (2015). https://doi.org/10.1145/2745754.2745771
25. 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). https://doi.org/10.14778/1920841.1920878
26. Fan, W., Wang, X., Wu, Y.: Querying big graphs within bounded resources. In: Proc. of SIGMOD'14, pp. 301–312. ACM (2014). https://doi.org/10.1145/2588555.2610513
27. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: graph processing in a distributed dataflow framework. In: Proc. of OSDI'14, pp. 599–613 (2014)
28. Group, C.M.D.: BerkeleyDB. https://dbdb.io/db/berkeley-db. Accessed July 2020
29. Holzschuher, F., Peinl, P.D.R.: Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. In: Proc. of GraphQ@EDBT/ICDT'13, pp. 195–204 (2013). https://doi.org/10.1145/2457317.2457351
30. JanusGraph: Distributed, open source, massively scalable graph database . https://janusgraph.org/. Accessed June 2020
31. Johann, S., Egyed, A.: Instant and incremental transformation of models. In: Proc. of ASE'04, pp. 362–365. IEEE Computer Society (2004). https://doi.org/10.1109/ASE.2004.10047
32. Jouault, F., Tisi, M.: Towards incremental execution of ATL transformations. In: Proc. of ICMT'10, LNCS, vol. 6142, pp. 123–137. Springer (2010). https://doi.org/10.1007/978-3-642-13688-7_9
33. Kafka, A.: Apache Kafka. A distributed streaming platform. https://kafka.apache.org/intro. Accessed May 2019
34. Kalavri, V., Vlassov, V., Haridi, S.: High-level programming abstractions for distributed graph processing. IEEE Trans. Knowl. Data Eng. **30**(2), 305–324 (2018). https://doi.org/10.1109/TKDE.2017.2762294
35. Lee, K., Liu, L.: Scaling queries over big RDF graphs with semantic hash partitioning. PVLDB **6**(14), 1894–1905 (2013). https://doi.org/10.14778/2556549.2556571
36. Li, K., Li, G.: Approximate query processing: What is new and where to go?—a survey on approximate query processing. Data Sci. Eng. **3**(4), 379–397 (2018). https://doi.org/10.1007/s41019-018-0074-4
37. Ltd, M.: Memgraph. Difference from Neo4j's cypher implementation. https://docs.memgraph.com/memgraph/reference-overview/differences. Accessed Sept 2020
38. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Boston (2002)
39. Luckham, D.C.: Event Processing for Business: Organizing the Real-Time Enterprise. Wiley, New York (2012)
40. 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: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, June 6–10, 2010, pp. 135–146 (2010). https://doi.org/10.1145/1807167.1807184
41. Memgraph Ltd: Memgraph graph database. https://memgraph.com/. Accessed Nov 2019
42. Memgraph Ltd: Memgraph indexing. https://docs.memgraph.com/memgraph/concepts-overview/indexing. Accessed Sept 2020
43. Mhedhbi, A., Gupta, P., Khaliq, S., Salihoglu, S.: A+ indexes: lightweight and highly flexible adjacency lists for graph database management systems. CoRR arXiv:2004.00130 (2020)
44. Neo4j: Neo4j graph platform. https://neo4j.com/. Accessed Jan 2020
45. Neo4j: Cypher query language. https://neo4j.com/developer/cypher-query-language/. Accessed Nov 2019
46. Neo4j: Neo4j—indexes for search performance. https://neo4j.com/docs/cypher-manual/current/administration/indexes-for-search-performance/index.html. Accessed Sept 2020
47. OrientDB: LiveQuery. https://orientdb.com/nosql/livequery/. Accessed July 2020
48. 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)
49. Peng, P., Zou, L., Chen, L., Zhao, D.: Adaptive distributed RDF graph fragmentation and allocation based on query workload. IEEE Trans. Knowl. Data Eng. **31**(4), 670–685 (2019). https://doi.org/10.1109/TKDE.2018.2841389
50. Perliger, A., Pedahzur, A.: Social network analysis in the study of terrorism and political violence. PS Polit. Sci. Polit. **44**(1), 45–50 (2011). https://doi.org/10.1017/S1049096510001848
51. Razavi, A., Kontogiannis, K.: Partial evaluation of model transformations. In: Proc. of ICSE'12, pp. 562–572. IEEE Computer Society (2012). https://doi.org/10.1109/ICSE.2012.6227160
52. Real, E., Shlens, J., , Pan, S.M.X., Vanhoucke, V.: YouTube-BoundingBoxes dataset. https://research.google.com/youtube-bb/. Accessed Oct 2019
53. Richardson, M., Domingos, P.M.: The intelligent surfer: probabilistic combination of link and content information in PageRank. In: proc. of NIPS'01, pp. 1441–1448. MIT Press (2001)

54. 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 MODELS'14, pp. 653–669 (2014). https://doi.org/10.1007/978-3-319-11653-2_40
55. Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The Train Benchmark: cross-technology performance evaluation of continuous model queries. Softw. Syst. Model. **17**(4), 1365–1393 (2018). https://doi.org/10.1007/s10270-016-0571-8
56. Szárnyas, G., Marton, J., Maginecz, J., Varró, D.: Reducing property graph queries to relational algebra for incremental view maintenance. CoRR arXiv:1806.07344 (2018)
57. The New Yorker: Data from the New Yorker caption contest. https://github.com/nextml/caption-contest-data. Accessed Oct 2019
58. TinkerPop: Apache TinkerGraph. http://tinkerpop.apache.org/docs/current/reference/#tinkergraph-gremlin. Accessed Oct 2019
59. TinkerPop: TinkerGraph indices. https://tinkerpop.apache.org/javadocs/3.2.2/full/org/apache/tinkerpop/gremlin/tinkergraph/structure/TinkerGraph.html#vertexIndex. Accessed Sept 2020
60. Tinkerpop, A.: Interface vertex program. http://tinkerpop.apache.org/javadocs/3.1.4/core/org/apache/tinkerpop/gremlin/process/computer/VertexProgram.html. Accessed Jan 2020
61. Troya, J., Wimmer, M., Burgueño, L., Vallecillo, A.: Towards approximate model transformations. In: Proc. of AMT@MoDELS'14, pp. 44–53. CEUR-WS (2014)
62. Trushkowsky, B., Kraska, T., Franklin, M.J., Sarkar, P.: Crowdsourced enumeration queries. In: Proc. of ICDE'13, pp. 673–684 (2013). https://doi.org/10.1109/ICDE.2013.6544865
63. Trushkowsky, B., Kraska, T., Franklin, M.J., Sarkar, P.: Answering enumeration queries with the crowd. Commun. ACM **59**(1), 118–127 (2016)
64. 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). https://doi.org/10.1016/j.scico.2014.01.004
65. Uta, A., Ghit, B., Dave, A., Boncz, P.A.: [Demo] Low-latency spark queries on updatable data. In: Proc. of SIGMOD'19, pp. 2009–2012 (2019). https://doi.org/10.1145/3299869.3320227
66. W3C RDF Data Access Working Group: SPARQL query language. https://www.w3.org/TR/rdf-sparql-query/. Accessed Jan 2020
67. Wang, Y., Parthasarathy, S., Sadayappan, P.: Stratification driven placement of complex data: a framework for distributed data analytics. In: Proc. of ICDE'13, pp. 709–720. IEEE Computer Society (2013). https://doi.org/10.1109/ICDE.2013.6544868
68. Webber, J., Robinson, I., Eifrem, E.: Graph databases. O'Reilly Media (2013)
69. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in Software Engineering. Springer, Berlin (2012)
70. Wood, P.T.: Graph database. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems, 2nd edn. Springer, New York (2018). https://doi.org/10.1007/978-1-4614-8265-9_183
71. Yang, C.C., Ng, T.D.: Terrorism and crime related weblog social network: link, content analysis and information visualization. In: Proc. of ISI'07, pp. 55–58. IEEE (2007). https://doi.org/10.1109/ISI.2007.379533

Gala Barquero is Ph.D. student at the University of Málaga, Spain. She worked as Java developer at Viewnext company from the IBM group before that time (2015–2017). She graduated as Telecommunications Engineer at University of Jaén, Spain (2015). Her current research interests include Model-based Software Engineering, Real-time Analytics and Software Quality.



Javier Troya is Associate Professor of Software Engineering at the University of Seville, Spain. Before, he was a post-doctoral researcher in the TU Wien, Austria (2013–2015), and obtained his International Ph.D. with honors from the University of Málaga, Spain (2013). His current research interests include Model-based Software Engineering, Software Testing and Software Quality.

Antonio Vallecillo is Professor of Software Engineering at the University of Málaga, Spain, where he leads the Atenea Research Group. His current research interests include Model-based Software Engineering, Open Distributed Processing, and Software Quality. More information about his publications, research projects and activities can be found at http://www.lcc.uma.es/~av.