The Graph Signature:
A Scalable Query Optimization Index for RDF Graph Databases Using Bisimulation and Trace Equivalence Summarization

Mustafa Jarrar, Sina Institute, Birzeit University, Birzeit, Palestine

Anton Deik, Sina Institute, Birzeit University, Birzeit, Palestine

ABSTRACT

Querying large data graphs has brought the attention of the research community. Many solutions were proposed, such as Oracle Semantic Technologies, Virtuoso, RDF3X, and C-Store, among others. Although such approaches have shown good performance in queries with medium complexity, they perform poorly when the complexity of the queries increases. In this paper, the authors propose the Graph Signature Index, a novel and scalable approach to index and query large data graphs. The idea is that they summarize a graph and instead of executing the query on the original graph, they execute it on the summaries. The authors' experiments with Yago (16M triples) have shown that e.g., a query with 4 levels costs 62 sec using Oracle but it only costs about 0.6 sec with their index. Their index can be implemented on top of any Graph database, but they chose to implement it as an extension to Oracle on top of the SEM_MATCH table function. The paper also introduces disk-based versions of the Trace Equivalence and Bisimilarity algorithms to summarize data graphs, and discusses their complexity and usability for RDF graphs.

Keywords: *Big Data, Bisimulation, Data Index, Data Web, Graph Databases, Mashups, Query Optimization, RDF, RDF Stores, Structural Summaries, Trace Equivalence*

1. INTRODUCTION AND MOTIVATION

Big Data, Data Web, Linked and Open Data are examples of an emerging era of data industry and data science. We are witnessing a rapid growth of the amount of available structured and linked data. As of June 2015, the Linking Open Data Statistics project (LODStats) records 3308 published datasets consisting of around 89.9 billion RDF triples.[1] Examples of published datasets are: DBPedia, Yago, DBLP, CiteSeer, ACM, Freebase, Geonames, MusicBrainz, as well as open governmental datasets such as those of the UK (data.gov.uk), the US (data.gov), and Ireland (opendata.ie), and many others. The biggest software companies are encouraging the trend of publishing and linking structured data on the web. For example, Google, Yahoo, and Microsoft

DOI: 10.4018/IJSWIS.2015040102

have joined efforts in developing a shared ontology.[2] Facebook is also providing access to parts of its data via its Graph API.[3]

To exploit structured data on the web to its full potential, people need efficient querying methods. SPARQL was introduced by W3C as a standardized query language that enables querying decentralized collections of RDF data. However, SPARQL is oriented for technical people. So, in order to allow people with limited IT skills to query structured data, many solutions were proposed, among which are those which proposed an interactive approach that allows the user to formulate queries without prior knowledge of the underlying data or its structure. Examples of such approaches are: Lore (Goldman & Widom, 1997) which was developed for querying schema-free XML, and MashQL, which is a query formulation language for RDF introduced in previous work (Jarrar & Dikaiakos, 2008; 2009; 2010; 2012). In July 2013, Facebook started rolling out a "Graph Search" functionality,[4] allowing users to formulate structured queries over the Facebook data graph. Not only do such approaches motivate the importance of querying data graphs, but they also emphasize the significance of having fast responses for queries executed over large data graphs in an interactive environment.

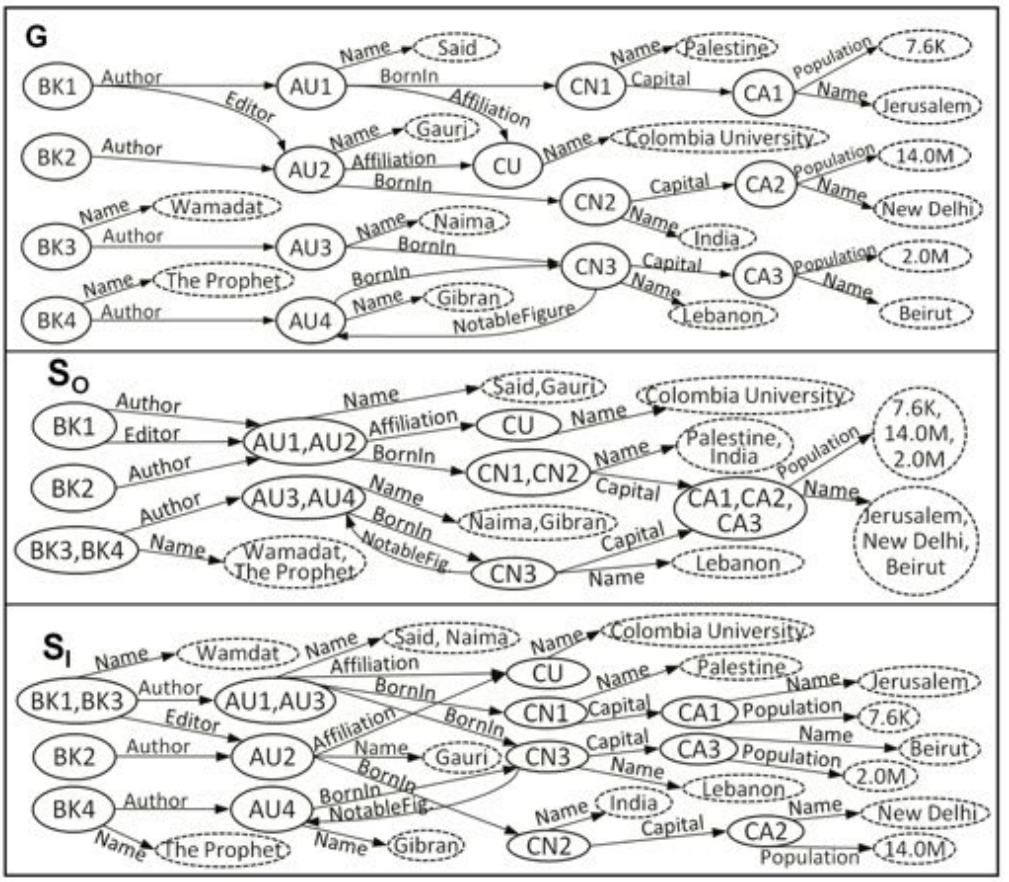
The most widely adopted data model specification for representing structured data on the web is RDF (Resource Description Framework). RDF syntax is based on XML and reflexes simple graph-based data model. RDF represents data as triples <*Subject, Predicate, Object*>.[5] For instance, the fact that the book called *Wamadat* is authored by *Naima* can be represented by the following three triples which form a directed labeled graph (see Figure 1):

```
< BK3, Name, Wamadat >
< BK3, Author, AU3 >
< AU3, Name, Naima >
```

Data representation using RDF is more elementary than relational databases and XML models, which enables easy data integration and interoperability of systems. However, querying RDF data graphs (especially large graphs) is a major challenge that faces all querying approaches, and therefore has brought the attention of the research community (e.g., Abadi et al., 2007; Chong et al., 2005; Schätzle et al., 2013; Tran et al., 2013; Yuan et al., 2013). Querying such data, which is typically stored in one relational table denoted by <*S,P,O*> is of high complexity because traversing a graph stored in relational model involves many self-joins of that table. More specifically, a query with $\left(n\right)$ edges on such a table requires $\left(n-1\right)$ self joins of that table (Abadi et al., 2007).

Several solutions were proposed to solve the problem of querying large RDF datasets. Among these solutions are Oracle Semantic Technologies (Chong et al., 2005),[6] OpenLink Virtuoso (Erling & Mikhailov, 2007; 2010), Vertical Partitioning (Abadi et al., 2007), and RDF3X (Neumann & Weikum, 2008), to name a few. Although these approaches have shown good performance in queries with medium complexity (such queries cost several seconds), they tend to perform poorly when the complexity of the queries increase. In this paper, we provide an optimization solution for a type of queries that these solutions tend to perform poorly in. Specifically, we propose an index called the Graph Signature Index (GS). The idea of our index is to summarize the RDF graph through grouping nodes based on their outgoing and their incoming paths, and instead of executing the query on the original data graph, we execute it on the index. Because the resultant summaries are typically much smaller than the original graph, executing the query on the index is faster. Our proposed index can be implemented on top of any of the aforementioned solutions. However, we choose here to demonstrate our work on top of Oracle Semantic Technologies because of its availability and support by one of the most reputable Database Management Engines in the world (Oracle Database). Our experiments have shown a promising enhancement; e.g., a

Figure 1. A data graph and it Graph Signature (S_O and S_I)



query of medium complexity that consists of only one path spanning three levels costs about 20 seconds using Oracle but it only costs 0.3 seconds with our proposed index.

Generating graph summaries is known to be a challenging problem, especially when dealing with large graphs. The XML research community (Goldman & Widom, 1997; Kaushik et al., 2002a; 2002b; Milo & Suciu, 1999; Nestorov et al., 1997) has utilized two well-known algorithms from the theoretical computing world (Paige & Tarjan, 1987; Fernandez, 1990; Henzinger, 1995) to summarize relatively small XML data graphs. These two algorithms are the *Trace Equivalence* and *Bisimilarity (or Bisimulation)* algorithms. For instance, indexing techniques for XML such as DataGuides (Goldman & Widom, 1997; Nestorov, 1997), 1-index (Milo & Suciu, 1999), A(k)-index (Kaushik et al., 2002a), and F&B (Kaushik et al., 2002b) all use Trace Equivalence and Bisimilarity algorithms for summarizing XML data. Using these two algorithms to summarize large graph-shaped data (e.g., RDF) is a challenging problem as the versions of these two algorithms that are found in literature are memory-based; that is, they not scalable to be used to summarize large RDF datasets. In addition, these versions found in literature are used to summarize XML data, which is tree-shaped, unlike RDF data which is graph-shaped. In this paper, we propose disk-based versions of Trace Equivalence and Bisimilarity to summarize RDF data.

The performance and resultant summaries of these two algorithms were experimentally analyzed for usage in the construction of the Graph Signature Index.

In short, the original contribution of this paper is two-fold:

1. The Graph Signature Indexing approach, which proposes to construct a twofold summary: one based on outgoing paths and another based on incoming paths. Our approach proposes to store both summaries separately but uses them jointly to precisely answer queries, instead of answering them using the original graph. A distinct feature of the Graph Signature Indexing approach is that it is a generic solution to be built on top of any RDF store or any other query optimization solution – rather than being proposed as a parallel alternative.
2. Disk-based versions of Trace Equivalence and Bisimilarity used to summarize medium to relatively large data graphs. This paper also comparatively studies the behavior of both algorithms in summarizing RDF data and proposes not to neglect Trace Equivalence but instead to use it in particular practical and real-life scenarios.

Preliminary versions of our disk-based algorithms for data graph summarization appeared in previous work (Hawash, Deik, & Jarrar, 2010), without analyzing their performances and resultant summaries to be used for the Graph Signature. In this paper, we position and revise our disk-based algorithms, analyzing their performances and resultant summaries thoroughly to be used for the construction of the Graph Signature (section 5). Also, primitive ideas of the Graph Signature approach appeared in previous work (Jarrar & Dikaiakos, 2010; 2012) only to optimize the background queries of MashQL. Here we extend our solution and redesign it as a generic solution; we expand the query model, introduce a new generic query execution plan to be implemented on top of any Graph Store, introduce new query evaluation theories to support our execution plan, and demonstrate our approach by implementing it as an enhancement to Oracle and experimenting it on a different RDF dataset.

The remainder of this paper is organized as follows. Related work is discussed in section 2. In section 3, we present the intuition of the Graph Signature Index. Section 4 presents an application use case of the Graph Signature. In section 5, we present our two-disk based versions of Trace Equivalence and Bisimilarity in addition to their experimental evaluation. Section 6 presents the query model and section 7 discusses query evaluation with the Graph Signature. In section 8, we discuss our implementation of the Graph Signature on top of Oracle and we experimentally evaluate it in section 9. Section 10 concludes our discussion and provides directions for future work.

2. RELATED WORK

In this section we review the work related to our Graph Signature Indexing approach and compare and position our work among those available in literature. We present related work in three categories: (i) indexing of XML data, (ii) recent work on indexing RDF data, and (iii) a review of some of the most popular RDF storage systems.

2.1. Indexing of XML Data

Several techniques have been proposed in literature to summarize XML data for XQuery optimization. The *DataGuide* (Nestorov, 1997) was the first to suggest summarizing XML by grouping nodes reachable by *any* incoming path. The problem with this way is that, because nodes that

extrinsically have some similar property labels are grouped together, many false positives are generated. The *Strong DataGuide* (Goldman & Widom, 1997) proposed to solve this issue by grouping nodes reachable by simple paths, as the DataGuide; but, it allows a node to exist in multiple groups. As pointed by the authors, this approach is efficient for tree-shaped data, but the size of the summary grows exponentially the more the data is graph-shaped (and can be larger than the original graph). Further, DataGuides are not adequate for complex queries having several regular expressions and variables (Milo & Suciu, 1999). In practice, this approach becomes very problematic when applied to cyclic graphs, as Goldman and Widom (1997) were unable to compute the strong DataGuide on a small subset of the IMDB dataset.

The *1-index* (Milo & Suciu, 1999) proposed to group nodes reachable by *all* incoming paths (which is analogous to our I-Signature), but it does not consider the outgoing paths (as our O-Signature) that yields an efficient reduction of false positives. A similar approach to the 1-index, namely, the *A(k) index* (Kaushik et al., 2002a) was suggested, based on the concept of k-bisimulation, to also group nodes reachable by all incoming paths up to k levels, thus it can only answer queries with k levels. Since this approach generates many false positives, the same authors of the A(k) suggested another approach called *F&B index* (Kaushik et al., 2002b). This approach groups nodes reachable by both all incoming *and* all outgoing paths, i.e., forward and backward at the same time. This approach produces much less false positives in query evaluation, but its size is *not* much less than the original. For example, the size of the F&B index for the Xmark dataset is only 10% less than the original (Kaushik et al., 2002b). As such, the time needed to query the F&B summary is close to querying the original data.

In general, our work differs from the work presented above in the following: (i) Our focus is on RDF rather than XML; that is, we adapt both Trace Equivalence and Bisimilarity to RDF. (ii) Unlike the F&B approach that generates one large incoming-and-outgoing index, we store the incoming and outgoing indexes separately, but they are jointly used, thus achieving small indexes and less false positives at the same time. (iii) Our Graph Signature Index relies on disk-based versions of Trace Equivalence and Bisimilarity which scale to relatively large data graphs, as opposed to the memory-based summarization algorithms used for summarizing XML. (iv) A query model and an evaluation scenario for RDF query paths is proposed, which is different from XML paths, as for instance, property labels, not only node labels, can be retrieved.

2.2. Recent Work on Indexing RDF Data

After publishing preliminary ideas of our Graph Signature approach in (Hawash, Deik, & Jarrar, 2010; Jarrar & Dikaiakos, 2010; 2012), other researchers also worked on the idea of summarizing data graphs, in particular, using Bisimulation. Tran et al. (2013), used Bisimilarity-based summaries in a query execution plan that combines both querying the summary as well as the original data, which they store in a way similar to the Vertical Partitioning approach (Abadi, 2007), but which utilizes information captured by the Bisimilarity summary. However, Tran et al. do not tackle the problem of generating the Bisimilarity summaries, but rather naively skip this problem and appear satisfied with only presenting the definition of the Bisimulation relation. They also store the summary in the main memory, which is not scalable when dealing with large RDF graphs. Furthermore, the authors exactly use the F&B approach introduced by Kaushik et al. (2002b) to summarize XML data, and apply it for RDF, resulting in summaries that are *not* much smaller than the original graph (as in the case of using F&B for XML). For example, they summarized about 12.9 million triples of DBLP data in 11.6 million (a mere 10% reduction). In order to solve this problem, the authors parameterized the Bisimilarity algorithm to produce summaries up to a certain number of levels and for certain labels in the graph, but this generates

more false positives. As for their query evaluation, instead of building upon and utilizing well-established fully-fledged RDF stores and optimization solutions, the authors restrict themselves to storing the original data in a way similar to Vertical Partitioning, and use their F&B summary as an enhancement only to their particular RDF storage solution. Moreover, they do not fully study how to answer queries precisely from the summary without going back to the original graph.

Other researchers recently introduced disk-based versions of the Bisimilarity algorithm. Hellings et al. (2012) introduced an efficient disk-based algorithm for computing Bisimilation. However, this algorithm is designed for Directed Acyclic Graphs (DAGs) and therefore cannot be applied for RDF graphs as they typically contain cycles (Schätzle et al., 2013). Schätzle et al (2013) presented two implementations of Bisimilarity for RDF summarization; one using SQL, similar to the initial version of our Bisimilarity disk-based algorithm (Hawash, Deik, & Jarrar, 2010), and the other using MapReduce Technology. The MapReduce implementation showed promising results for summarizing datasets of several-hundred-million to billion triples. Another effort to introduce a MapReduce implementation of Bisimilarity was presented by Luo et al. (2013), also with promising results for massive graphs. In spite of such promising results, an SQL version of Bisimilarity remains of great practical importance, due to its suitability for datasets of several-million triples (see section 5.6), as well as its simple implementation, as also acknowledged by Schätzle et al. (2013). Nevertheless, none of these efforts on disk-based Bisimilarity have studied deeply the theoretical or practical features of the algorithm for RDF usage, nor did any of them compare Bisimilarity with Trace Equivalence (neither theoretically nor empirically). In fact, they did not even consider introducing a disk-based Trace Equivalence algorithm, and did not investigate how to utilize such summaries for query evaluation.

Our work is different from the work above in the following: (i) We pay a close attention to the problem of generating summaries, unlike Tran et al. (2013), and solve it by introducing disk-based versions of Bisimilarity as well as Trace Equivalence. (ii) Not only that, but also, unlike the work of Hellings et al. (2012), Luo et al. (2013), and Schätzle et al. (2013), we do study and compare the resultant summaries of both algorithms (both theoretically and empirically) and conclude in suggesting the usage of both algorithms depending on the practical scenario at hand. (iii) Unlike the F&B approach (Kaushik et al., 2002b) used by Tran et al. (2013), we propose a novel approach to store our generated incoming and outgoing summaries separately, but use them jointly, which results in smaller summaries and also less false positives. (iv) Graph Signature focuses on answering complex queries precisely using the summary alone without going back to the original data. (v) Our approach is generic; it is designed to be implemented on top of any RDF store utilizing its features and capabilities, unlike Tran et al. (2013).

2.3. RDF Stores

Our Graph Signature is a generic index that can be implemented on top of any RDF Store. The summary itself is a data graph and therefore both storing and querying it is done in the same way as the original data graph. This section discusses the most prominent RDF Stores and how our index could be implemented on top of them as an enhancement.

Oracle introduced an SQL-based scheme to query RDF data in (Chong et al., 2005). Specifically, an SQL table function called "RDF_MATCH" (or "SEM_MATCH") was introduced, which takes a SPARQL-like query as an argument, and returns a table of results that can be further queried using SQL. Oracle stores RDF triples in one table G(S,P,O); thus a query with *n*-levels implies joining the table *n*-1 times. In addition, Oracle builds several B-tree indexes on G, as well as subject-property materialized views, such as V_1 (s, p_1, p_2, …, p_n). A tuple in V_1 is a subject identifier x, and the value of the column p_i (i=1,2,…,n) is an object y. In this way, data

is transformed from a graph into a relational form; thus, less joins are needed when executing a query. These subject-property views are seen as auxiliary, rather than core, indexes. This is because there is no general criteria to know which subjects and which properties to group. Oracle uses statistics to find possibly good groupings, otherwise, queries are executed on the original graph; hence queries with many joins remain a challenge.

OpenLink has developed native RDF support within its *Virtuoso* RDBMS (Erling & Mikhailov, 2007; 2010). Its initial storage solution for RDF is fairly conventional and similar to Oracle. Virtuoso stores RDF as a single table of four columns (G,S,P,O), with two covering indices: <G,S,P,O> and <O,G,P,S>. Also, Virtuoso assigns the O's of string type that are longer than 12 characters a unique ID, which enhances query performance. However, deep linear queries of many variables remain a challenge as answering such queries involve several expensive self-joins of the table.

RDF3X (Neumann & Weikum, 2008) proposes to store the RDF triples as a single (S,P,O) table and to build B^+-tree indexes over 6 permutations of the three dimensions (i.e. SPO, SOP, PSO, POS, OSP, OPS). To exploit these indexes to their full potential, the query optimizer chooses the best order and the types of joins to build an execution plan. Experiments conducted by Neumann and Weikum (2008) show that execution time is still relatively large; a query of three levels on the Yago RDF dataset took about 22 seconds.

Vertical Partitioning (Abadi et al., 2007) proposes to store RDF triples as n two-column tables, where n is the unique number of predicates P(S,O). Although this method attains significant optimization, queries spanning several levels with many variables are expensive; a query of only two levels took 15.88 seconds (Abadi et al., 2007).

Our Graph Signature can be built on top of any of the above mentioned RDF stores. The idea of our solution is to reduce the size of the data by means of summarization. Implementing our Graph Signature on top of an RDF Store means that we store and query our summaries in the same way as the original data. Specifically, in Oracle, we store our summary as an SPO table and query it using RDF_MATCH, whereas in OpenLink Virtuoso, one may store it as the two suggested covering indices and query it using the Virtuoso API. In the case of RDF3X, our summaries can be stored in SPO tables applying all kinds of suggested indices over them and using the same query optimizer of RDF3X. In the case of using Vertical Partitioning, the Graph Signature can be vertically partitioned (in n two-column tables) and queried accordingly. It is worth mentioning that several other promising RDF stores are available in literature and industry such as Neo4j,[7] TripleBit (Yuan et al., 2013), and Trinity.RDF (Zeng et al., 2013), on top of which we can also implement our Graph Signature. In this paper, we choose to demonstrate our approach by implementing it on top of Oracle as it is one of the most reputable Database Management Engines with wide-spread industrial support and commercial availability. The details of our implementation and experimental evaluation are in sections 8 and 9, respectively.

3. INTUITION OF THE GRAPH SIGNATURE INDEX

The idea of the Graph Signature Index is to summarize a data graph such that queries are evaluated using the summary instead of the original graph. Because the size of the summary is smaller than that of the original graph, query evaluation using the Graph Signature can be faster. For instance, we summarized 16 million triples of Yago in 0.5 million such that a query that costs 62.7 seconds on the 16 million triples, costs less than 0.6 seconds on the 0.5 million triples, while producing exactly the same set of results.

Given a data graph G, its Graph Signature is a two-fold summary: the O-Signature (S_O) and the I-Signature (S_I). The O-Signature is a summary of the original data graph constructed by grouping nodes that have the same set of their *outgoing* paths. The I-Signature is another summary of the original graph, but it is constructed by grouping nodes that have the same set of their *incoming* paths. Figure 1 shows a data graph (G) with its Graph Signature (S_O and S_I). One can notice that nodes in the O-Signature are combined based on their outgoing paths. For example, nodes CN1 and CN2 are grouped together because the sets of their outgoing paths are the same: {(Name), (Capital, Population), (Capital, Name)}. However, node CN3 is not put into the same group with CN1 and CN2 because there are two extra outgoing paths from CN3: {(NotableFigure, BornIn), (NotableFigure, Name)}. The same can be noticed in the I-Signature. For example, AU2 is not combined with AU1 and AU3 because it has an extra incoming path: {(Editor)}. Each of the two summaries (S_I and S_O) is computed and stored separately the same way the original graph G is stored, but they are jointly used when evaluating a query to produce *precise answers* (answers that are equal to those resulting from querying the original graph). To illustrate this, in Table 1 we execute 6 queries on both summaries of the graph G depicted in Figure 1, and compare the results with the *target answer* (the answer obtained from G).

For any query, each part of the Graph Signature produces the correct answer and some more results, called *false positives*. That is, the target answer is equal to or is a subset of the answer of each part of the signature. Hence, the intersection of the S_O and S_I answers equals to or is a smaller superset of the target answer. One can notice from Table 1 that some queries do produce precise results when queried over either S_O or S_I (e.g., Q1), while in Q2, S_O is enough to answer this query precisely and in Q3, S_I is enough. Q4 represents the case of queries producing empty results. In this case, the empty result is the precise answer. In Q5, three queries are presented with different projections pertaining to the same query body: when the first node is projected (Q5.a) S_O is enough to produce precise results, while in (Q5.b), the intersection of the answers of S_I and S_O suffice to precisely answer the query, and in (Q5.c) S_I is enough. Q6 is an example of a query that does not conform to our query model (section 6) and is not guaranteed to produce precise results using S_I, S_O, nor the intersection of their results. The answer of such queries includes the target answer in addition to false positives. In our execution plan (section 7), queries that conform to our query model (such as Q1-Q5) are executed on the summaries while others are executed via the native RDF store on the original data.

4. APPLICATION USE CASE OF THE GRAPH SIGNATURE INDEX

This section presents a use case where challenging queries over large RDF datasets must be answered within very short response-time. In particular, we illustrate MashQL, a graphical query formulation language introduced in previous work (Jarrar & Dikaiakos, 2008; 2009; 2010; 2012), which allows non-technical people to easily query and fuse RDF data on the web. Formulating a query in MashQL is an interactive process done using a "MashQL editor", during which the user performs selections from drop-down lists. While the user interacts with the editor, the editor performs queries in the background to generate these lists. These queries are called *Background Queries*.

Figure 2.a depicts six snapshots (Windows I - VI) of the MashQL editor taken while a user is formulating a query over the data graph in Figure 1. The query means "Give me all authors who are born in a place whose capital is called Beirut". While formulating this query, the MashQL editor executes several queries in the background (Q1-Q5). In order to maintain an acceptable

Table 1. Graph Signature answers compared with the target answers

	Query	G answer	S_O Answer	S_I Answer	$S_O \cap S_I$
Q1	$?p_1$ $?p_4$: ($?s_1$ $?p_1$ $?o_1$ $?p_2$ $?o_2$ $?p_3$ $?o_3$ $?p_4$ $?o_4$)	Author Name, Author Population, Author BornIn, Editor Name, Editor Population, BornIn Capital, BornIn Name, BornIn NotableFigure, NotableFigure BornIn, NotableFigure Name, NotableFigure Population	Author Name, Author Population, Author BornIn, Editor Name, Editor Population, BornIn Capital, BornIn Name, BornIn NotableFigure, NotableFigure BornIn, NotableFigure Name, NotableFigure Population	Author Name, Author Population, Author BornIn, Editor Name, Editor Population, BornIn Capital, BornIn Name, BornIn NotableFigure, NotableFigure BornIn, NotableFigure Name, NotableFigure Population	Author Name, Author Population, Author BornIn, Editor Name, Editor Population, BornIn Capital, BornIn Name, BornIn NotableFigure, NotableFigure BornIn, NotableFigure Name, NotableFigure Population
Q2	?p: (<BK3><Author> $?o_1$?p $?o_2$)	Name, BornIn	Name, BornIn	Name, BornIn, Affiliation	Name, BornIn
Q3	?p: ?s ?p <AU1>	Author	Author, Editor	Author	Author
Q4	$?o_2$: (<BK3><Author> $?o_1$ <Affiliation> $?o_2$)	{}	{}	CU	{}
Q5	*(a)* ?s | *(b)* $?o_1$ | *(c)* $?o_3$: (?s <Author> $?o_1$ <BornIn> $?o_2$ <NotableFigure> $?o_3$) **Three queries with three different projections: ?s,?o$_1$,?o$_3$*	(a) BK3, BK4 (b) AU3, AU4 (c) AU4	(a) BK3, BK4 (b) AU3, AU4 (c) AU3, AU4	(a) BK3, BK4, BK1 (b) AU3, AU4, AU1 (c) AU4	(a) BK3, BK4 (b) AU3, AU4 (c) AU4
Q6	?s: (?s <BornIn> $?o_1$ <Capital> $?o_2$ <Name> <Jerusalem>)	AU1	AU1, AU2, AU3, AU4	AU1, AU3	AU1, AU3

interactive environment in such applications, the response time of background queries is expected to be very small; within a few hundred milliseconds (Miller, 1968).

This use case demonstrates the importance of handling deep and complex query models. The complexity of queries does not necessarily come from the many constraints they include. For example, a star-shaped query (which is more generic than linear-shaped) might include many constraints, but with short paths. In general, one of the most challenging types of queries are those that span several levels (i.e. deep queries), as they require many expensive joins. It will be shown later how such queries (deep, linear and demanding fast response) can be answered precisely (and quickly) using our Graph Signature Index without the need to query the original data.

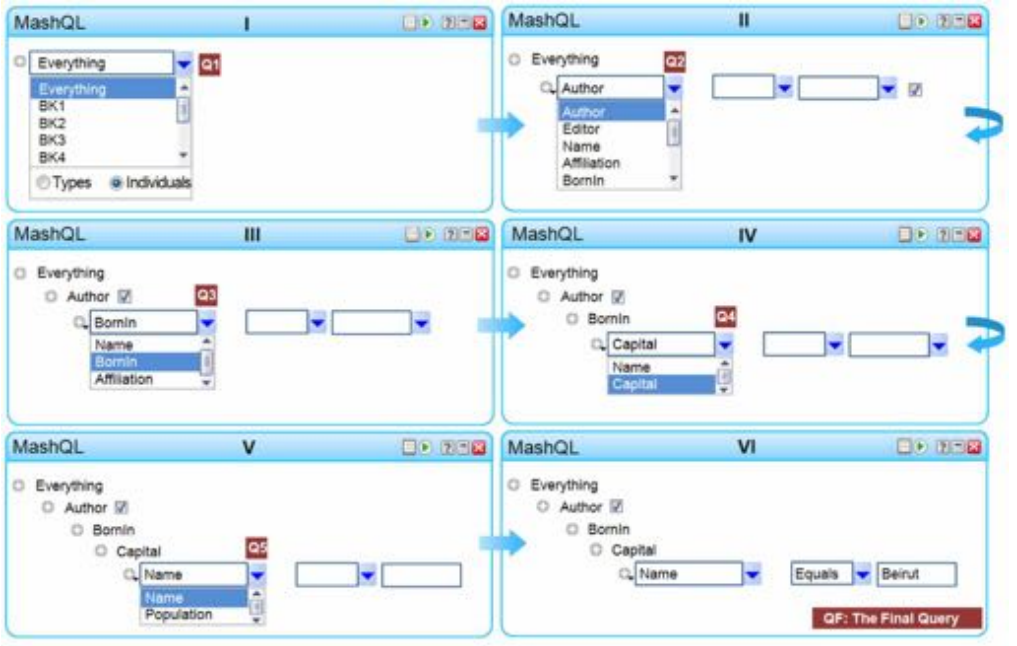
5. CONSTRUCTION OF THE GRAPH SIGNATURE

Before presenting our query model and execution plan, we present the Graph Signature Index and the algorithms to construct it. The reader can also glimpse at the query model and evaluation in sections 6 and 7 first.

5.1. Basic Definitions

A directed labeled graph (also referred to here by data graph) is composed of vertices (nodes) and edges connecting those vertices. The formal definition below is based on the definition of the labeled graph of Champin and Solnon (2003):

Figure 2. A MashQL query formulation session and its accompanying queries



(a) A MashQL session

Q1	?S: (?S ?P ?O)
Q2	?P: (?S ?P ?O)
Q3	?P: (?S <Author> $?O_1$?P $?O_2$)
Q4	?P: (?S <Author> $?O_1$ <BornIn> $?O_2$?P $?O_3$)
Q5	?P: (?S <Author> $?O_1$ <BornIn> $?O_2$ <Capital> $?O_3$?P $?O_4$)
QF	$?O_1$: (?S <Author> $?O_1$ <BornIn> $?O_2$ <Capital> $?O_3$ <Name> <Beirut>)

(b) Queries generated from the MashQL session in Figure 2.a

Definition 1 (Directed Labeled Graph): *Given a finite set of vertex labels* L_V *and a finite set of edge labels* L_E*, a directed labeled graph is defined by a triple* $G = \langle V, r_V, r_E \rangle$*, such that:*

- V *is a finite set of vertices.*
- $r_V \subseteq V \times L_V$ *is the relation that associates vertices with labels, such that each vertex in V is associated with exactly one vertex label in* L_V *and each vertex label in* L_V *is associated with exactly one vertex in V, i.e.,* r_V *is the set of couples* (v, l) *such that each vertex* v *has exactly one label* l *and l is not associated with any other vertex.*

- $r_E \subseteq V \times V \times L_E$ *is the relation that associates edges with labels, i.e.,* r_E *is the set of triples* (v,u,l) *such that edge* (v,u) *has label* l. *We can define the set of edges* E *as:* $E = \{(v,u,l) \epsilon r_E\}$.

It is worth mentioning here that an RDF graph is in fact a directed labeled graph. Thus, in this paper we use the terms *Data Graph*, *RDF Graph*, and *Directed Labeled Graph* interchangeably. Definition (2) below defines the notion of a *Path* based on Definition (1). A Node Path can be defined as: $\langle (v_1,v_2,l_1),(v_2,v_3,l_2),\ldots,(v_{n-2},v_{n-1},l_{n-2}),(v_{n-1},v_n,l_{n-1})\rangle$, such that, $(v_i,v_{i+1},l_i) \in r_E$ for *i=1,2,...,n-1*. Because in this paper we are interested in the structure of the path regardless of the nodes $(v_1,v_2,\ldots,v_n)$, we define the *Path* as an ordered set of consecutive edge labels.

Definition 2 (Path): *A Path is an ordered set of consecutive edge labels* $l_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_m$, *such that for* l_i *and* l_{i+1} *there exist some nodes* $v,u,w \in G$, *such that* $(v,u,l_i) \in r_E$, *and* $(u,w,l_{i+1}) \in r_E$ *(i=1,2,...,m-1).*

5.2. The Notions of Trace Equivalence and Bisimilarity

Trace Equivalence is defined (based on outgoing paths) as an equivalence relation on the set of vertices V, such that two vertices (v,u) are Trace Equivalent if and only if the set of all paths rooted in vertex v is equal to the set of all paths rooted in vertex u. We call the Trace Equivalence relation when defined based on outgoing paths, *O-Trace-Equivalence*, and when it is based on incoming paths, *I-Trace-Equivalence*. The formal definition of *O-Trace-Equivalence* based on (Henzinger et al., 1995) is as follows.

Definition 3.a (O-Trace-Equivalence $\equiv_O$): *The vertex* v *trace-dominates the vertex* u *if for every finite path rooted in* u *(*$\overline{u}$*),* there is a path rooted in v *(*$\overline{v}$*)* such that $\overline{u} = \overline{v}$. *The vertices* u *and* v *are O-Trace-Equivalent, written* $(u \equiv_O v)$ *if* u *trace-dominates* v *and* v *trace-dominates* u.

This definition can be trivially adapted to be based on incoming paths, in order to define *I-Trace-Equivalence* as follows:

Definition 3.b (I-Trace-Equivalence $\equiv_I$): *The vertex* v *trace-dominates the vertex* u *if for every finite path terminated in* $\overline{\overline{u}}$, *there is a path terminated in* v *(*$\overline{\overline{v}}$*)* such that $\overline{\overline{u}} = \overline{\overline{v}}$.

The vertices u *and* v *are I-Trace-Equivalent, written* $(u \equiv_I v)$ *if* u *trace-dominates* v *and* v *trace-dominates* u.

A memory-based algorithm to generate summaries based on the Trace Equivalence relation would simply suggest that we fully traverse the graph G and take each node in the graph and find all outgoing paths from it and all incoming paths to it. After that, nodes having the same set of outgoing paths are grouped together into one Equivalence Class, generating the O-Signature. Similarly, nodes having the same set of incoming paths are grouped together, generating the I-

Signature. Each group in either summary is seen as an Equivalence Class of its members. Although the idea of the Trace Equivalence algorithm appears to be straight-forward, unfortunately this algorithm is computationally expensive; it is known to be PSPACE-complete (Henzinger et al., 1995). However, as will be discussed later, in some practical cases one may use Trace Equivalence to compute the data summary (e.g., when the data graph is simple or the summarization time in not critical). Because of its usefulness in such cases, we decided to provide a disk-based version of Trace Equivalence (section 5.4).

To avoid the computational complexity of Trace Equivalence (when it matters), we introduce an alternative notion called Bisimilarity - an extensively discussed notion in the literature of process algebra (Henzinger et al., 1995; Paige & Tarjan, 1987). Bisimilarity has the complexity of $O\left(m \log n\right)$ for a graph with n vertices and m edges (Henzinger et al., 1995; Milo & Suciu, 1999). Thus, it is easier to compute than Trace Equivalence. Furthermore, we introduce the notion of Bisimilarity as an *approximation* of Trace Equivalence, as suggested by Milo and Suciu (1999). An equivalence relation ($\approx$) on a data graph G is called an *approximation* of another relation ($\equiv$), if it satisfies the following condition for nodes v, u in the data graph G: $u \approx v \Rightarrow u \equiv v$ (Milo & Suciu, 1999). Because Bisimilarity satisfies this condition w.r.t Trace Equivalence (i.e., Bisimilarity implies Trace Equivalence), Bisimilarity is an *approximation* of Trace Equivalence. That is, if two nodes are Bisimilar, this necessarily implies that they are Trace Equivalent (Fernandez, 1990; Henzinger et al., 1995; Milo & Suciu, 1999).

Bisimilarity in a directed labeled graph is an Equivalence Relation defined on a set of vertices V, such that two vertices $\left(u, v\right)$ are Bisimilar if and only if the set of edges coming immediately out of u (predicates in RDF) is equal to the set of edges coming immediately out of v (definition based on outgoing edges). Also, all successor nodes of u and v must be Bisimilar. We call the Bisimilarity relation when defined based on outgoing paths, *O-Bisimilarity*, and when it is based on incoming paths, *I-Bisimilarity*. The formal definitions are given in Definitions (4.a) and (4.b) below.

Definition 4.a (O-Bisimilarity $\approx_O$): *Given a directed labeled graph G=<V, r_V, r_E> as in Definition (1), two vertices* $\left(u, v\right) \in V$ are O-Bisimilar, written $\left(u \approx_O v\right)$, *if and only if:*

1. For the set of all immediate edges out of v: $\{\left(v, v_1', l_{11}\right), \left(v, v_2', l_{12}\right), \ldots, \left(v, v_n', l_{1n}\right)\} \subseteq E$, there exists a set of immediate edges out of u: $\left\{\left(u, u_1', l_{21}\right), \left(u, u_2', l_{22}\right), \ldots, \left(u, u_n', l_{2n}\right)\right\} \subseteq E$, such that $l_{1i} = l_{2i}$ for $\left(i = 1, 2, \ldots, n\right)$.
2. Conversely, for the set of all immediate edges out of u: $\left\{\left(u, u_1', l_{11}\right), \left(u, u_2', l_{12}\right), \ldots, \left(u, u_n', l_{1n}\right)\right\} \subseteq E$, there exists a set of immediate edges out of v: $\left\{\left(v, v_1', l_{21}\right), \left(v, v_2', l_{22}\right), \ldots, \left(v, v_n', l_{2n}\right)\right\} \subseteq E$ such that $l_{1i} = l_{2i}$ for $\left(i = 1, 2, \ldots, n\right)$.
3. The set of vertices $\left(v_i', u_i'\right)$ for $\left(i = 1, 2, \ldots, n\right)$ are also O-Bisimilar.

The definition of I-Bisimilarity is trivially derived from Definition 4.a, as it is the inverse of O-Bisimilarity:

Definition 4.b (I-Bisimilarity $\approx_I$): *Given a directed labeled graph G=<V, r_I, r_E> as in Definition (1), two vertices* $(u,v) \in V$ are I-Bisimilar, written $(u \approx_I v)$, *if and only if:*

1. For the set of all immediate edges into v : $\{(v_1'', v, l_{11}), (v_2'', v, l_{12}), \ldots, (v_n'', v, l_{1n})\} \subseteq E$, there exists a set of immediate edges into u : $\{(u_1'', u, l_{21}), (u_2'', u, l_{22}), \ldots, (u_n'', u, l_{2n})\} \subseteq E$, such that $l_{1i} = l_{2i}$ for $(i = 1,2,\ldots,n)$.
2. Conversely, for the set of all immediate edges into u : $\{(u_1'', u, l_{11}), (u_2'', u, l_{12}), \ldots, (u_n'', u, l_{1n})\} \subseteq E$, there exists a set of immediate edges into v : $\{(v_1'', v, l_{21}), (v_2'', v, l_{22}), \ldots, (v_n'', v, l_{2n})\} \subseteq E$ such that $l_{1i} = l_{2i}$ for $(i = 1,2,\ldots,n)$.
3. The set of vertices (v_i'', u_i'') for $(i = 1,2,\ldots,n)$ are also I-Bisimilar.

The memory-based version of the Bisimilarity algorithm can be found in (Paige & Tarjan, 1987). The idea of the algorithm is to create summaries in an iterative process. In each iteration, the algorithm groups nodes up to a certain number of levels. Applying the Bisimilarity algorithm to a data graph G to produces the summary based on outgoing paths (S_O) involves two steps. First, the algorithm *groups* the nodes based on the immediate edge labels (or predicates in RDF). Second, these groups go through a number of iterations to *split* the nodes that conflict with the Bisimilarity relation, until no more splitting can be done. The main issue with the memory-based versions of both Bisimilarity and Trace Equivalence is that they are not suitable for large data graphs, since a data graph needs to be loaded into the computer's memory and the algorithms are executed there. To solve this problem, disk-based versions of the algorithms need to be introduced (see also (Hellings et al., 2012; Luo et al., 2013; Schätzle et al., 2013) on the importance of introducing disk-based algorithms for large data graphs). In section 5.4, we introduce disk-based versions of Trace Equivalence and Bisimilarity and experiment them in section 5.6 on relatively large RDF datasets. Before that, we formally define our Graph Signature Index.

5.3. The Graph Signature Index

The Graph Signature Index consists of two summaries; one based on outgoing paths, called, the O-Signature (S_O) and another based on incoming paths, called, the I-Signature (S_I). Each of these two signatures can be built in two ways: either using Trace Equivalence, or Bisimilarity. We call the O-Signature produced by Trace Equivalence S_{OT}, and S_{OB} when produced by Bisimilarity. Similarly, when the I-Signature is produced by Trace Equivalence, we call it S_{IT}, and S_{IB} when it is produced by Bisimilarity. Every node in any summary is an *Equivalence Class* of some node in the original graph *G*. We define the Equivalence Class in S_{OT} by the O-Trace-Equivalence relation ($\equiv_O$) defined above, whereas, in S_{OB}, the Equivalence Class is defined by the O-Bisimilarity relation ($\approx_O$). The Equivalence Class in S_{IT} is defined by I-Trace-Equivalence ($\equiv_I$) and, in S_{IB}, it is defined by I-Bisimilarity ($\approx_I$).

As will be discussed later, Trace Equivalence produces smaller summaries thus query evaluation using Trace Equivalence summaries is faster than using Bisimilarity summaries. However, computing Trace Equivalence is of high complexity (PSPACE complete). Nevertheless, it might be still used in case the summarization time is not very critical or in case the data graph is simple. On the other hand, Bisimilarity produces larger summaries – thus query evaluation using them can be slower – but the algorithm itself is easier to compute: $O(m \log n)$

complexity. As a result, it is recommended to have Bisimilarity as a default summarization algorithm while keeping the option for the user to choose Trace Equivalence if needed (see section 5.6). Here we present two versions of the definition of our Graph Signature, one based on Trace Equivalence (Definition 5) and another based on Bisimilarity (Definition 5').

Definition 5 (Graph Signature based on Trace Equivalence S_T): *Given a data graph G, its Graph Signature S_T is the twofold summary: the O-Signature S_{OT} and the I-Signature S_{IT}. In general, $S_T = <S_{OT}, S_{IT}>$.*

Definition 5.a (O-Signature based on Trace Equivalence S_{OT}): *Given a data graph G, its S_{OT} summary is a directed labeled graph where each node in S_{OT} is an Equivalence Class on O-Trace-Equivalence ($\equiv_O$) of some node in G, such that every node in G has exactly one Equivalence Class in S_{OT}, and there exists an edge p in S_{OT} from u to v ($u \xrightarrow{P} v$) iff G contains an edge p from a to b ($a \xrightarrow{P} b$) and $a \in u$, $b \in v$.*

Definition 5.b (I-Signature based on Trace Equivalence S_{IT}): *Given a data graph G, its S_{IT} summary is a directed labeled graph where each node in S_{IT} is an Equivalence Class on I-Trace-Equivalence ($\equiv_I$) of some node in G, such that every node in G has exactly one Equivalence Class in S_{IT}, and there exists an edge p in S_{IT} from u to v ($u \xrightarrow{P} v$) iff G contains an edge p from a to b ($a \xrightarrow{P} b$) and $a \in u$, $b \in v$.*

In the following we present the same definition of the Graph Signature above, but this time based on Bisimilarity:

Definition 5' (Graph Signature based on Bisimilarity S_B): *Given a data graph G, its Graph Signature S_B is the twofold summary: the O-Signature S_{OB} and the I-Signature S_{IB}. In general, $S_B = <S_{OB}, S_{IB}>$.*

Definition 5'.a (O-Signature based on Bisimilarity S_{OB}): *Given a data graph G, its S_{OB} summary is a directed labeled graph where each node in S_{OB} is an Equivalence Class on O-Bisimilarity ($\approx_O$) of some node in G, such that every node in G has exactly one Equivalence Class in S_{OB}, and there exists an edge p in S_{OB} from u to v ($u \xrightarrow{P} v$) iff G contains an edge p from a to b ($a \xrightarrow{P} b$) and $a \in u$, $b \in v$.*

Definition 5'.b (I-Signature based on Bisimilarity S_{IB}): *Given a data graph G, its S_{IB} summary is a directed labeled graph where each node in S_{IB} is an Equivalence Class on I-Bisimilarity ($\approx_I$) of some node in G, such that every node in G has exactly one Equivalence Class in S_{IB}, and there exists an edge p in S_{IB} from u to v ($u \xrightarrow{P} v$) iff G contains an edge p from a to b ($a \xrightarrow{P} b$) and $a \in u$, $b \in v$.*

5.4. Disk-based Versions of Trace Equivalence and Bisimilarity Summarization Algorithms

In order to summarize large graph-shaped data, we introduce disk-based versions of the algorithms. Our disk-based version of Trace Equivalence is presented formally in Figure 3 using Relational Algebra notation (as it is SQL-based). Its idea is based on Definitions 3.a and 3.b of Trace Equivalence. That is, in order to generate the summary of a graph (based on outgoing paths), we first find all outgoing paths of a node and then group the nodes that have the same set of paths together. Since a data graph is represented as an <S,P,O> table, finding the set of all paths from a node is done by continuously performing self-left joins to this table until all paths are determined. The number of self-left joins required depends on the levels the longest

Figure 3. The disk-based Trace Equivalence algorithm

Algorithm 1: Trace Equivalence (G)

Input: A graph G (three-tuple table,$< S, P, O >$).
Output: Summarized Graph S_{OT}
Begin
1. $R_0 = (copy\ of) G$
2. While (no more left joins possible) do
3. For each row in R_0
4. if $s = o$ //checks for possible loops to isolate them
5. Copy row to table R
6. Delete row from R_0
7. $\rho\left(R_0, \pi_{s,p_1,p_2,\ldots,p_n,o}\left(R_0 \bowtie_{R_0.O=R_0.S} R_0\right)\right)$ // self left join
8. End While
9. Insert into R_0 all rows from R.
10. $\rho\left(spath, \pi_{s,concat(p_1,\ldots,p_n)as\ PATH}(R_0)\right)$
11. $pathID$ = *All distinct (PATH) in spath*
12. For each (PATH) in pathID assign a unique number
13. $\rho\left(sid, \pi_{spath.s,pathID.id}\left(spath \bowtie_{spath.PATH=pathID.PATH} pathID\right)\right)$
14. $\rho(subCat, \pi_{s,hashFunction(id),group\ by\ s}(sid))$
15. Generate graph summarization from $subCat$.
End

path spans, e.g., if the longest path is only two levels only one join is needed, if it is four levels deep then two joins are needed, and so on. The stopping condition of this process is: when no more left joins possible; i.e., the longest path has been retrieved and further joins do not retrieve additional paths.

Because a self-left join becomes more expensive as the table becomes larger, the algorithm eliminates the columns that are not needed before performing each join in order to enhance the performance. Specifically, the result of the first self-left join of the table $\langle s, p, o \rangle$ is a table named $\langle s_1, p_1, o_1, s_2, p_2, o_2 \rangle$ (self-left join is done on $o_1 = s_2$). Before performing the second self-left join we eliminate columns $\langle o_1, s_2 \rangle$, resulting in redundant rows, which are then eliminated when creating the reduced table, $\langle s, p_1, p_2, o \rangle$ on which the second self-left join is performed, and so on. This column elimination is done in order to remove redundant rows that appear after removing the columns, thus reducing the size of the table, which results in significant enhancement in performance as well as reducing space requirements. An additional step to detect and resolve loops before performing each self-left join is done by checking the condition "$s = o$" for each row (line 4 in Figure 3), i.e., to check whether the starting vertex of the path is equal to the terminating one. Such rows are isolated and are not considered in the further joins.

Although we have implemented some technical enhancements in our disk-based version of Trace Equivalence as described above, the computational complexity of our version of the algorithm is still the same as the memory-based version (PSPACE Complete). Improving the complexity of the algorithm is beyond the scope of our paper.

In what follows we introduce a disk-based version of the Bisimilarity algorithm, which is computationally less expensive than Trace Equivalence. The complexity of our disk-based version of Bisimilarity is again similar to that of the memory-based version; $\mathrm{O}\left(\mathrm{m}\log\mathrm{n}\right)$ for a graph

with n vertices and m edges. To generate the O-Signature using our disk-based version of Bisimilarity (Figure 4), the initial grouping of nodes with similar outgoing predicates is done by assigning to each distinct predicate (P) a unique hash value (lines 2-4 in Figure 4). Then, for each node, the algorithm calculates the sum of the hash values of its predicates (line 5). Nodes that end up having the same summation belong to the same group. That is, this summation reflects the category of the node.

In the second step (i.e., the iterative step), for each node in the graph, we find its successors, and then we sum the node's category with a hash value of the sum of all its successors' categories. This new sum is then hashed, and its new value is used as the new category number of the node. In this way, nodes in each category are split according to their successors' categories. In general, this step is repeated k times, until the table stabilizes (no more categories are updated), with k corresponding to the longest acyclic path in the graph. The problem of cycles in the graph are solved by the condition of the iterative step (the while-loop condition, line 6 in Figure 4), which breaks the while-loop when the number of the categories are not changing anymore. This loop condition guarantees that the algorithm stops regardless of the existence of cyclic paths in the graph. It is worth mentioning that in our implementation using Oracle DBMS, we used Oracle's hash function (ORA_HASH[8]) that produces a number between 1 and 2^{32}-1.

The algorithms in Figures 3 and 4 produce summaries based on outgoing paths (S_{OT}, S_{OB}). Adapting these algorithms to generate summaries based on incoming paths (S_{IT}, S_{IB}) is straight forward. We simply use the same algorithms presented in Figures 3 and 4, but in line (1) of both algorithms, instead of copying G into table R_0, we copy the inverse of G. That is, instead of copying <S,P,O> from G into R_0, we copy <O,P,S> from G into R_0. After that the algorithms continue normally as described in Figures 3 and 4. Using relational algebra, line (1) in Figures 3 and 4 is replaced by: $\rho\left(R_0, \pi_{O,P,S} G\right)$. In SQL, this can be written as:

```
create table R0 (S,P,O) as select (O,P,S) from G.
```

Figure 4. The disk-based Bisimilarity algorithm

Algorithm 2: Bisimilarity (G)

Input: A graph G (three-tuple table,$< S, P, O >$).
Output: Summarized Graph S_{OB}
Begin
1. R_0 = (Copy of) G
2. R_1 = all distinct (P) in R_0
3. For each (P) in R_1 assign a unique hash value
4. Update (P) in R_0 with the corresponding unique hash value in R_1
5. $\rho\left(subCat, \pi_{S,sum\ (P)\ as\ Cat, Group\ by\ S}\ (R_0)\right)$
6. WHILE (number of unique categories in $subCat$ is changing)
7. $\rho(T_0, \pi_{R_0.S,\ subCat.Cat}(R_0 \bowtie_{R_0.S = subCat.S} subCat))$
8. $\rho(T_1, \pi_{T_0.S,\ T_0.Cat\ as\ sCat,\ subCat.Cat\ as\ oCat}(T_0 \bowtie_{T_0.O = subCat.S} subCat))$
9. $\rho(T_2, \pi_{S,\ hash(sCat+hash\ (sum(oCat)))\ as\ Cat,\ Group\ by\ (S,sCat)}\ (T_1))$
10. $\rho(T_3, T_2 \cup \pi_{S,Cat}\ (\sigma_{S\ NOT\ IN\ \pi_S(T_3)}(subCat)))$
11. $subCat$ = (copy of) T_3
12. END WHILE
13. Generate graph summarization from $subCat$
End

As mentioned previously, the memory-based version of the Bisimilarity algorithm was used in literature to summarize XML data using several different techniques, such as 1-index (Milo & Suciu, 1999), A(k)-index (Kaushik et al., 2002a), and F&B (Kaushik et al., 2002b). However, RDF is more complex than XML in several ways. RDF in its nature forms a graph, thus there is no single root, whereas XML is tree structured. This tree structure implies that every node has only one parent, and that for every node there is only one unique path that this node can be reached from (Milo & Suciu, 1999). Conversely, in RDF, the same node may be accessed via several paths, and through different nodes. Also, RDF may contain cycles that should be taken into consideration.

These differences between tree-shaped data (XML) and graph-shaped data (RDF) are reflected on the summaries of Trace Equivalence and Bisimilarity. For tree-shaped data the two equivalence relations coincide, thus their algorithms generate identical summaries (Milo & Suciu, 1999). However, in the case of graph-shaped data, Bisimilarity summaries are *approximations* of Trace Equivalence summaries. This is because Bisimilarity does not catch/summarize all cases that are Trace Equivalent. In other words, some nodes that are grouped together because they are Trace Equivalent might not be Bisimilar and thus might not be grouped together. In the following, we explain this issue by pointing out two typical cases.

The first case (see Figure 5.a) emerges from the fact that, in graph-shaped data, edge labels (predicates) coming out of a node (or going into a node) might not be unique. Using Trace Equivalence, nodes A1, A2, and A3 in Figure 5.a have the same set of outgoing paths, so they are grouped together in the O-Signature (S_{OT}). However, this is not the case in the Bisimilarity algorithm, because the successors of these nodes do not fall in the same category (i.e., they are not bisimilar). Thus, in Bisimilarity, the three nodes A1, A2, and A3 are put into three different categories in S_{OB}. The second case that appears in graph-shaped data, which is discussed by Milo and Suciu (1999), is depicted in Figure 5.b. One can notice that A1 and A2 are grouped together using the Trace Equivalence algorithm to generate the O-Signature as they have the same set of paths. Meanwhile, these two nodes are not Bisimilar; because their successors (M1 and M3) are not Bisimilar. *Such cases result in Bisimilarity summaries larger than Trace Equivalence summaries. Thus, query evaluation on Trace Equivalent summaries can be faster, although the Trace Equivalence algorithm is computationally more expensive.*

5.5. Storage and Size of the Graph Signature

The Graph Signature Index is an RDF Graph that can be stored the same way the original RDF graph is stored. Thus, storing it depends on the particular RDF store used. In this paper, we choose to demonstrate our work on top of Oracle as it is a well known database solution that supports RDF. Since Oracle basically stores the RDF graph in one table of <S, P, O>, each part of the Graph Signature (i.e., the O/I-Signature) is stored as an <S,P,O> table. The subjects and objects of each signature table are category numbers corresponding to summarization groups. In addition, we build two *Graph Signature Extents*, each of which is a lookup table storing each node in the original graph and its corresponding category in the O/I-Signature. The schemas of the extent tables are:

```
extent_o(SoID,Node), extent_i(SiID,Node).
```

If our Graph Signature is to be implemented on top of another solution, its storage will be different. For example, if we are to implement our index using C-Store (Stonebraker et al., 2005) with Vertical Partitioning (Abadi et al., 2007), then it needs to be vertically partitioned into *n* two-column tables, where *n* is the unique number of predicates.

The space cost for storing each part of the Graph Signature consists of the space of the signature and the space of the extent. The size of each part of the Graph Signature is at most as

large as the data graph; but in practice, it is much less, as our experimental evaluation shows (section 5.6). The size of the extent is exactly the number of unique nodes in the data graph.

5.6. Experimental Evaluation of the Summarization Algorithms

5.6.1. Experiment Setup

Our disk-based versions of Trace Equivalence and Bisimilarity were implemented using PL/SQL in Oracle 11g. The experiments were conducted on a PC with a 2.50 GHz Intel Core 2 Quad CPU and a 250 GiB SATA Hard Disk, on a Windows XP SP2 Operating System. Each experiment was conducted twice; once with 2GiB of memory and another time with only 1GiB, in order to demonstrate that our algorithms are disk based. For the purpose of the experiments presented here we used both algorithms to generate summaries based on outgoing paths ($S_{O,T}$ and $S_{O,B}$). The behavior of both algorithms to generate I-Signatures is similar to the O-Signatures case presented here.

In the experiments, we used two real-world RDF datasets: DBLP and Yago. Yago is a large semantic knowledge base derived from Wikipedia and WordNet containing more than 2 million entities (e.g., persons, organizations, cities), whereas DBLP is about scientific publications from the DBLP Bibliography. Our Yago dataset contains 15 million triples (1.10GiB) whereas the DBLP dataset contains 8 million RDF triples (1.07GiB). We partitioned Yago into 5 tables: Y3 with 3 million triples, Y6 with 6 million, Y9 9 million, Y12 12 million, and Y15 with 15 million triples. DBLP was partitioned into 4 tables: D2, D4, D6, and D8 with 2, 4, 6, and 8 million triples, respectively. Note that no sorting was applied on the data before partitioning (e.g., D4 was created by:

```
create table D4 as select * from D8 where rownum<=4000000).
```

The two datasets differ in the nature of the data they contain. DBLP data tends to be more homogenous as most of its nodes have similar paths. Also, DBLP paths tend to be shorter than those in the Yago dataset (the longest path in our DBLP dataset is 4 levels long). Yago data, on the other hand, is more heterogeneous as it contains data from a very diverse spectrum of domains (the domains of Wikipedia and Wordnet) and node paths tend to be longer than those in DBLP. Note that these observations impact the results of the experiments (Tables 2 and 3), as will be discussed shortly.

5.6.2. Analysis of the Experimental Results

The fact that our algorithms are written in SQL and executed using Oracle DBMS shows that they are disk-based. This is further confirmed by the results of our experiments in Tables 2 and 3; reducing the memory installed on the machine from 2 GiB to 1 GiB had no impact on the time cost of summarization for both algorithms. Furthermore, one can notice the scalability of the behavior of both algorithms with respect to the number of triples. For instance, in the case of Trace Equivalence (and similar for Bisimilarity), the whole Yago dataset (Y15) is summarized by 462K triples. This number tends to increase when the data is smaller (e.g. 658K for Y12). The same is true with DBLP; D8 is summarized in 30K triples using Trace Equivalence while D6 is summarized in 118K triples. This is because more similarities were found when the whole data is put together (notice that we did not apply any sorting before partitioning the data). In other words, some nodes in D6 are grouped in several equivalence classes (instead of one) as they have different paths, while when all data is put together in D8, it is found that these nodes have the same paths. This implies that the size of the summary does not necessarily increase when more triples are added to the data graph. Moreover, it is noticed that the nature of the data being summarized impacts the size of the summaries and the summarization time. For example,

Table 2. Summary of experimental results using our disk-based versions of Trace Equivalence and Bisimilarity on YAGO dataset

	Number of		Y3	Y6	Y9	Y12	Y15
Original Graph	Unique Triples		3M	6M	9M	12M	15M
	Unique Subjects		1.78M	2.67M	3.30M	3.84M	4.34M
	Unique Predicates		81	83	83	84	84
	Unique Objects		1.50M	3.04M	4.54M	6.09M	7.64M
	Data Size		226MiB	459MiB	678MiB	904MiB	1.10GiB
Summary (Trace Equivalence)	Unique Categories		32K	82K	121K	148K	100K
	Triples in Summarized Graph		102K	312K	501K	658K	462K
	2GiB	Summ. Time (sec)	95	311	1,025	3,507	9,869
	1GiB	Summ. Time (sec)	94	314	1,022	3,511	9,873
Summary (Bisimilarity)	Unique Categories		33K	86K	128K	159K	109K
	Triples in Summ. Graph		106K	325K	530K	703K	499K
	2GiB	Summ. Time (sec)	96	241	504	1,481	1,818
	1GiB	Summ. Time (sec)	101	243	498	1,483	1,816

the summary of D6 (118K, using Trace Equivalence) is smaller than the summary of Y6 (312K, using Trace Equivalence), as DBLP is more homogenous. Also notice that Y6 is summarized in 241 seconds while D6 is summarized in 68 seconds (using Bisimilarity) though both contain 6 million triples. This is because DBLP is more homogenous and its paths are shorter.

In comparing between the behavior of Bisimilarity and Trace Equivalence algorithms, one notices some differences. First, the time cost of the Bisimilarity algorithm is almost always less than that of Trace Equivalence. In particular, as the number of triples increases in the graph, the difference between the time-costs of both algorithms increases. For example, Y3 was summarized in 95 seconds using Trace Equivalence and in 96 seconds using Bisimilarity using 2 GiB of memory, i.e., no significant difference between the performances of both algorithms. However, the summarization of Y6 using Trace Equivalence costs 311 seconds whereas, using Bisimilarity, the cost is 241 seconds; a difference of 70 seconds (22.5% less). For the Y15 dataset, the Trace Equivalence algorithm finishes after 9,869 seconds, whereas the Bisimilarity algorithm finishes after 1,818 seconds (about 81.6% less than the time-cost of Trace Equivalence).

A second difference between Bisimilarity and Trace Equivalence is in the performance behavior of both algorithms with respect to the data size. An initial look at the results of the experiments conducted on the DBLP dataset (Table 3) reveals no major difference in the performance behavior of both algorithms; both appear to be almost linear (Figure 6.a). Due to the simple homogenous nature of the DBLP data, the behavior of Trace Equivalence becomes closer to that of Bisimilarity. This is also because the longest path in DBLP spans four levels; that is, Trace Equivalence takes a maximum of two joins to traverse the D8 graph. However, the experiments on the more heterogeneous Yago dataset reveal the more accurate performance behavior of both algorithms. This behavior is depicted in Figure 6.b, which shows that the performance behavior of the Bisimilarity algorithm is linear with respect to the number of triples in the graph, whereas the performance of the Trace Equivalence algorithm is exponential. In fact, one can expect such performance behavior from both algorithms; theoretically, Trace Equivalence is known to be PSPACE complete, whereas Bisimilarity is $\mathrm{O}\left(\mathrm{m}\log\mathrm{n}\right)$ for a graph with n vertices and m edges.

Table 3. Summary of experimental results using our disk-based versions of Trace Equivalence and Bisimilarity on DBLP dataset

	Number of		D2	D4	D6	D8
Original Graph	Unique Triples		2M	4M	6M	8M
	Unique Subjects		791K	942K	1.04M	1.15M
	Unique Predicates		24	25	27	27
	Unique Objects		659K	1.23M	1.76M	2.30M
	Data Size		275MiB	541MiB	824MiB	1.07GiB
Summary (Trace Equivalence)	Unique Categories		5K	13K	16K	4K
	Triples in Summarized Graph		28K	87K	118K	30K
	2GiB	Summ. Time (sec)	44	70	116	174
	1GiB	Summ. Time (sec)	43	72	119	175
Summary (Bisimilarity)	Unique Categories		6K	16K	19K	5K
	Triples in Summarized Graph		32K	102K	139K	35K
	2GiB	Summ. Time (sec)	23	41	68	107
	1GiB	Summ. Time (sec)	25	40	69	110

A third difference between Trace Equivalence and Bisimilarity is in the summary itself. One can notice from Tables 2 and 3 that in all the experiments, Bisimilarity summaries are always larger than Trace Equivalence summaries. For example, the summary of D8 using Trace Equivalence contains 30K triples, whereas using Bisimilarity, it contains 35K triples. The same can be noticed in the Y15 dataset: it is summarized into 462K using Trace Equivalence and into 499K using Bisimilarity. As explained previously, the reason behind this is that Bisimilarity is an *approximation* of Trace Equivalence. One might also notice that the differences in the sizes of the summaries between Trace Equivalence and Bisimilarity tend to be higher in DBLP data than Yago; Bisimilarity summaries are 5.7% larger (on average) in the case of Yago than their Trace Equivalence counterparts, while DBLP Bisimilarity summaries are 16.5% larger, on average. The reason for this is the nature of the data itself; DBLP data is about scientific publications, where a publication tends to have several authors, resulting in many nodes in the data graph having several non-unique predicates, which might cause cases such as those depicted in Figure 5, which Bisimilarity does not summarize.

Figure 5. Special cases that appear only in graph-shaped data

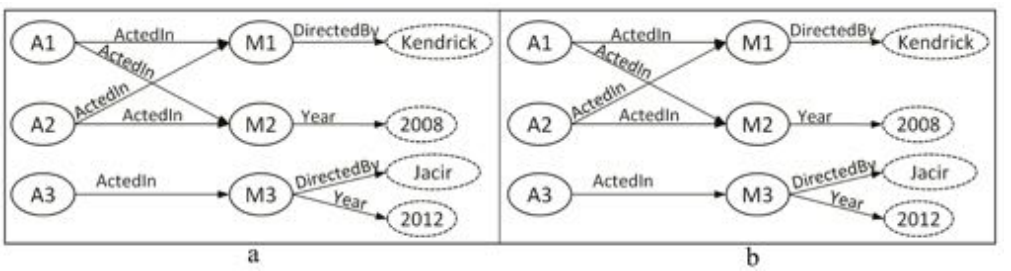
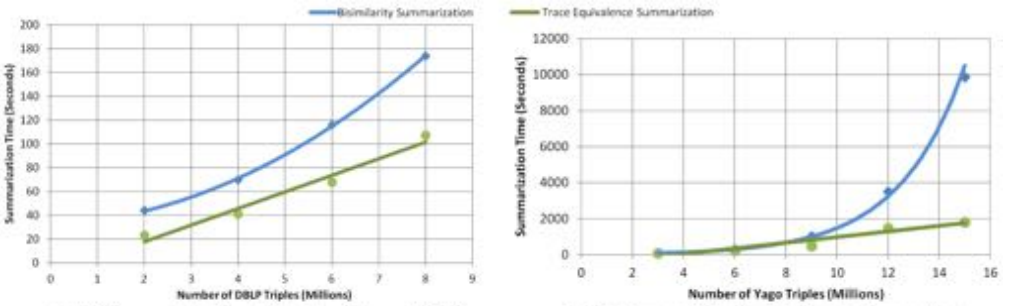


Figure 6. Performance behavior of Trace Equivalence and Bisimilarity



(a) Performance behavior of both algorithms on DBLP

(b) Performance behavior of both algorithms on Yago

From our discussion above, we noticed that the summarization time cost of Bisimilarity is less than that of Trace Equivalence. On the other hand, Trace Equivalence produces smaller summaries than Bisimilarity, which means evaluating queries over Trace Equivalence summaries is typically faster. We also noticed that when the data itself tends to be more homogenous and its paths shorter (as in DBLP), the behavior of Trace Equivalence becomes close to that of Bisimilarity. Based on this, we propose using Bisimilarity in some cases and Trace Equivalence in others. In general, because of its better performance, we recommend Bisimilarity as the default summarization algorithm, while using Trace Equivalence in particular scenarios. For example, when the summarization time-cost is not critical for the user and/or the data is homogenous with short paths, one might use Trace Equivalence. For instance, for our particular setup here, we would recommend using Bisimilarity to summarize Yago and Trace Equivalence to summarize DBLP.

6. THE QUERY MODEL

As mentioned earlier, our query model tackles one of the most challenging query types; deep linear-shaped queries of many variables, spanning several levels. Such queries are among the most expensive, especially when evaluated on a large RDF graph stored in one <S,P,O> table, as they require multiple expensive self joins. The complexity of such queries does not necessarily come from the many conditions they include, but rather from the depth of the query (the levels it spans) and the number of variables. The more levels a query spans and the more variables it contains, the more joins are required and the more expensive these joins are. Optimizing such queries, however, leads to optimizing more generic query models such as tree and star-shaped queries, as the linear query model is the building block of these models.

Definition 6 (Query Path): We define a query path as an expression of the form: $\{O_1P_1O_2P_2 \ldots P_{n-1}O_n\}$, where O_i is a query node, and P_j is a query edge (*i=1,2,...,n; j=1,2,...,n-1*). A variable node (unbound node) is denoted by $?O_i$ and a variable edge (unbound edge) by $?P_j$. A non-variable node (bound node) is denoted by $<O_i>$ and a non-variable edge (bound edge) by $<P_j>$. To refer to nodes and edges that can be either bound or unbound we use the notation O_i for nodes and P_j for edges.

Definition 7 (Graph Signature Query Model $\mathbb{Q}$): Our Graph Signature Query Model ($\mathbb{Q}$) is defined as: $?P_1,\ldots,?P_{n-1} \mid ?O_i : \{O_1P_1O_2P_2\ldots P_{n-1}O_n\}$, $i = 1,2,3,\ldots, n$, where the query condition is a linear path $\{O_1P_1O_2P_2\ldots P_{n-1}O_n\}$ and the projections are only predicates ($?P_1,\ldots,?P_{n-1}$) or alternatively one node ($?O_i$), such that when a query node is projected ($?O_i$) all other query nodes must be unbound.

7. EVALUATING QUERIES WITH THE GRAPH SIGNATURE

Recall from our earlier discussion, that the answer obtained from executing a query on the O-Signature -and similarly on the I-Signature- is always a superset or equals the target answer (the answer obtained from the original data graph). In case the answer of the I/O-Signature equals the target answer, we call it a *precise* answer. Otherwise, it is called a *safe* answer, since it equals the target answer and some more *false positives*. In this section, we illustrate how queries that conform to our query model ($\mathbb{Q}$) can be answered *precisely* using only the Graph Signature. This is done through introducing a set of nine query evaluation theorems and an execution plan based on them. Our theorems and execution plan produce *precise* results whether they are evaluated on Trace-Equivalence-based Graph Signature (S_T) or on Bisimilarity-based Graph Signature (S_B), following the proposition below.[9]

Proposition. *Given a data graph G, its Trace-Equivalence summary S_T, and its Bisimilarity summary S_B, if a query (that conforms to $\mathbb{Q}$) produces precise results when evaluated on S_T, then it produces precise results if evaluated on S_B.*

Theorems 1 and 2 below are two intuitive theorems that apply to any arbitrary query:

Theorem 1. *Given an arbitrary query, the answer of the O-Signature is always safe and similarly the answer of the I-Signature.*

Theorem 2. *Given an arbitrary query, if the answer of the I/O-Signature is empty or the intersection of the results of both is empty, then this answer is always precise.*

The following theorems apply only to queries that conform to our query model such that all projections are query edges:

Theorem 3. *Given a query that conforms to our query model $\mathbb{Q}$ such that all projections are query edges, if all query nodes are unbound:$?P_1,\ldots,?P_{n-1}$:$\{?O_1 P_1 \ldots P_{n-1} ?O_n\}$, the answer of either the O-Signature or the I-Signature is always precise.*

Theorem 4. *Given a query that conforms to our query model $\mathbb{Q}$ such that all projections are query edges, if the first query node in the query path is bound and all remaining query nodes are unbound:$?P_1,\ldots,?P_{n-1}$: $\{<O_1>P_1 ?O_2 P_2 \ldots P_{n-1} ?O_n\}$, then the answer of the O-Signature is always precise.*

Theorem 5. *Given a query that conforms to our query model $\mathbb{Q}$ such that all projections are query edges, if the last query node in the query path is bound and all remaining query nodes are unbound: $?P_1,\ldots,?P_{n-1}$:$\{?O_1 P_1 ?O_2 P_2 \ldots P_{n-1} <O_n>\}$, then the answer of the I-Signature is always precise.*

Theorem 6. *Given a query that conforms to our query model $\mathbb{Q}$ such that all projections are query edges, if one node located anywhere in the query path is bound and all remaining query nodes are unbound: $?P_1,\ldots,?P_{n-1}$: $\{?O_1\ P_1 \ldots <O_i> \ldots P_{n-1}\ ?O_n\}$ (where; $1 \leq i \leq n$), then the intersection of the answers of both the I-Signature and the O-Signature is always precise.*

The following theorems also apply only to queries that conform to our query model such that the projection is exactly one query node and all query nodes in the query path are unbound.

Theorem 7. *Given a query that conforms to our query model $\mathbb{Q}$ such that the projection is the first query node in the query path and all query nodes are unbound: $?O_1$: $\{?O_1\ P_1\ ?O_2\ P_2 \ldots P_{n-1}\ ?O_n\}$, then the answer of the O-Signature is always precise.*

Theorem 8. *Given a query that conforms to our query model $\mathbb{Q}$ such that the projection is the last query node in the query path and all query nodes are unbound: $?O_n$: $\{?O_1\ P_1\ ?O_2\ P_2 \ldots P_{n-1}\ ?O_n\}$, then the answer of the I-Signature is always precise.*

Theorem 9. *Given a query that conforms to our query model $\mathbb{Q}$ such that the projection is exactly one query node located anywhere in the query path and all query nodes are unbound: $?O_i$: $\{?O_1\ P_1 \ldots ?O_i \ldots P_{n-1}\ ?O_n\}$ (where; $1 \leq i \leq n$), then the intersection of the answers of both the I-Signature and the O-Signature is always precise.*

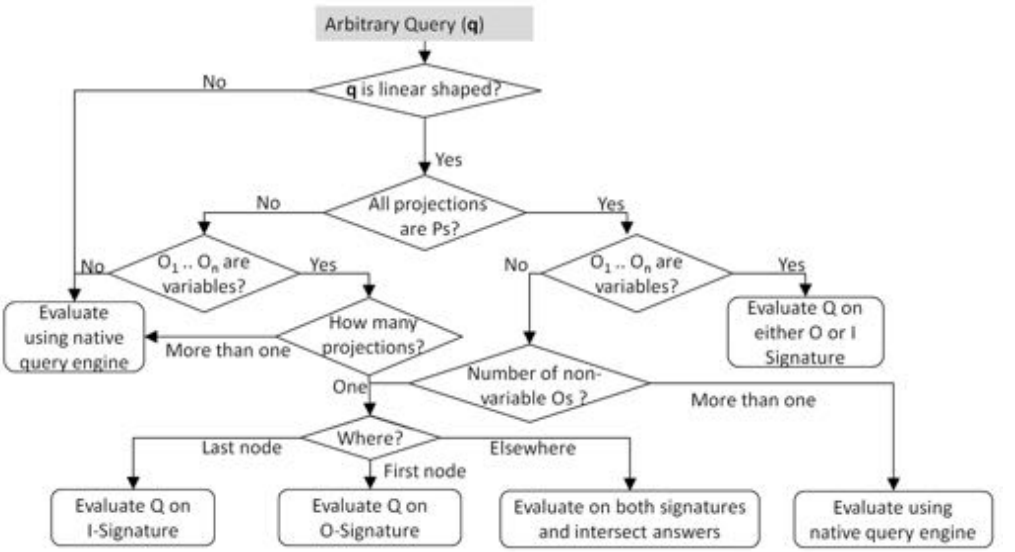
The flowchart in Figure 7 depicts the evaluation scenario based on the above theories. As shown, the Graph Signature is used to answer queries that conforms to our query model $\mathbb{Q}$ while other queries are executed using the native query engine that the Graph Signature is built on.

As the flowchart in Figure 7 depicts, an arbitrary query is first checked against our query model $\mathbb{Q}$. If the query does not conform to the query model, it is directed to the native query engine; otherwise it is checked for further conditions. If the query meets the exact conditions imposed by the theories sketched above, it is executed according to the theories (either on the O-Signature, the I-Signature, or both). A query that conforms to $\mathbb{Q}$ is first checked for projections; if all projections are predicates, then the query is checked for bound nodes. If all nodes are unbound, the query can be executed on either summary. If one node is bound, then its position is checked and the query is executed accordingly; on the O-Signature if the bound node is the first node in the query path, on the I-Signature if the bound node is the last node, otherwise, the query is executed on both signatures and the intersection of the answers is the precise result set (this implements theories 3-6). If more than one bound node is found, the query is directed to the native query engine. In the case that the projection is one query node, the query is checked for bound nodes. If there is any bound node, the query is directed to the native query engine, else, it is processed according to theories (7-9), where the position of the projection node determines which signature it is executed on. However, no matter where the projection node is located, the intersection of the answers of the O-Signature and the I-Signature always produces precise results. Nevertheless, if the projection node is the first one in the query path, then the O-Signature suffices to produce precise results, if it is the last, then the I-Signature is sufficient.

8. IMPLEMENTATION OF THE QUERY EXECUTION PLAN

The query evaluation scenario depicted in Figure 7 is implemented as a java stored procedure that takes a SPARQL query as an input and returns precise results that can be further processed using SQL . Java stored procedures are Java methods published to SQL and stored in the Oracle

Figure 7. Flow chart depicting the execution plan



DBMS for general use. They are compiled once and stored in an executable form, which results in quick and efficient procedure calls with minimal overhead. Querying a data graph is done using Oracle's SEM_MATCH function. In specific, we have implemented our java stored-procedure on top of SEM_MATCH, such that the stored procedure evaluates queries that conform to our query model using the SEM_MATCH function over the Graph Signature (following the execution plan in Figure 7). The execution process follows these steps:

1. The function parses the SPARQL query to determine the variable and non-variable nodes in addition to the number and types of projections in the query.
2. Using the information determined in step 1, the target summary to be used in query evaluation is determined based on the conditions imposed by the theorems presented in section 7, as depicted in Figure 7.
3. Before executing the query over the target summary, the non-variable node labels in the query are replaced with their corresponding category numbers found in the Graph Signature Extent.
4. The query is then evaluated on the target summary using SEM_MATCH, according to the chosen evaluation path. In the case where a node is projected, the results obtained contain only group IDs. Therefore, these results are joined with the Graph Signature Extent to obtain (i.e., lookup) the target answer for the query.

We have also implemented a *Mapping Dictionary* to gain the best performance results. The Mapping Dictionary is a lookup table that maps each node, and edge label in the RDF dataset into a unique identifier (*dID*). All literals and URIs in the original data graph are replaced with their corresponding *dIDs*, such that the data graph would consist of numerals only instead of long URIs and literals. This is done before computing the Graph Signature and its extent. This has two advantages. (i) The size of the O-Signature and the I-Signature and the extent are compacted in

terms of disk space usage. (ii) Executing queries on tables containing only numerals allows for faster joins, comparison, and matching operations.

Our query execution plan does not currently optimize all types of queries, but rather optimizes linear queries with many variables; a query type which Oracle performs poorly in. Oracle's query optimizer, however, can utilize our optimization of linear queries for evaluating more generic query models such as tree-shaped and star-shaped queries. This could be done, for instance, by submitting certain query paths to our query execution plan. In our implementation, we built our execution plan on top of Oracle instead of tuning its query optimizer, as Oracle's query optimizer is not open.

9. EXPERIMENTAL EVALUATION OF THE GRAPH SIGNATURE INDEX

This section presents the experimental evaluation of our proposed Graph Signature Index, which we implemented on top of Oracle. Our experiment was conducted on the same PC we have used in evaluating the summarization algorithms (section 5.6.1). For this experiment, we define two sets of benchmark queries and execute them twice on a relatively large RDF graph in Oracle: once without the Graph Signature and another time with the Graph Signature. For the RDF dataset, we use the largest and most heterogeneous datasets of those presented earlier: Yago with 15 million RDF triples (Y15), which is 1.10 GiB. Our queries are evaluated on the Bisimilarity-based Graph Signature, which is larger than the Graph Signature produced by Trace Equivalence. Thus, we demonstrate here a more challenging case of our approach, since if our queries were evaluated on the summaries produced by Trace Equivalence, our query execution would be faster as the summaries are smaller. Furthermore, our choice of the Y15 dataset reflects a realistic use case; a real-world RDF dataset of medium complexity presenting data from Wikipedia and Wordnet.[10]

From the Yago dataset we derived 13 benchmark queries, divided into two groups (Table 4). The first group contains several queries that are encountered in some Web 3.0 applications such as MashQL. The second group of queries can be considered an extreme case of linear shaped queries where all the query nodes and edges are variables - thus requiring expensive joins. The aim of this second group of queries is to demonstrate the power of the enhancement that the Graph Signature introduces to Oracle. Even though these queries are not necessarily faced in practice, they form a good showcase for exposing the limits of Oracle Semantic Technologies with and without the Graph Signature Index.

The results of our experiments are shown in Table 4 and Figure 8. One can notice that the performance of Oracle is much improved with the Graph Signature Index. This improvement is specially noticed in queries that span a long path (e.g., A6, A7). Queries in group B present an extreme case for Oracle with and without the Graph Signature Index. As evident from the results, the response of Oracle's SEM_MATCH table function after the 4th level (B5), was larger than 20 minutes. On the other hand, although the execution time using our index increases at each level, this increase remains acceptable for such type of extreme queries. Again, the performance boost that the Graph Signature Index provides is due to its size, which is much smaller than that of the original graph.

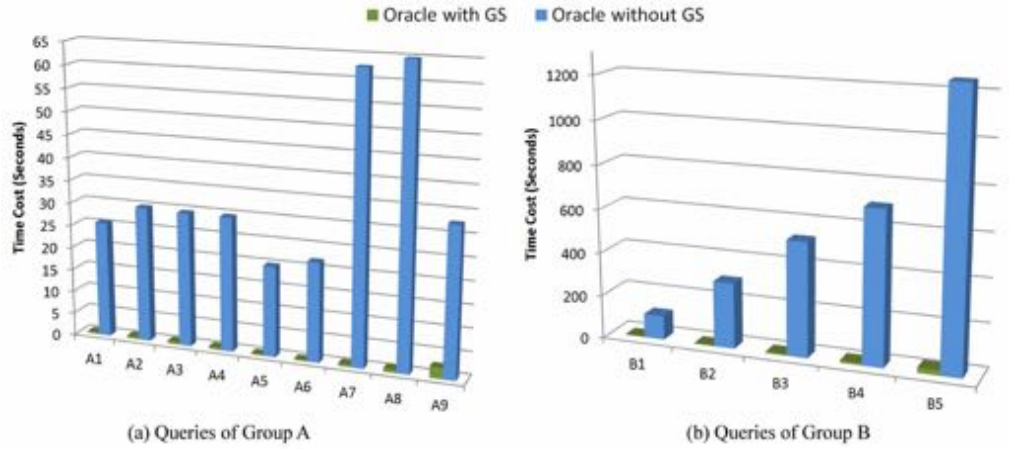
10. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a novel approach for query optimization for RDF stores, namely, the Graph Signature Index, which is a generic index that can be implemented on top of any RDF

Table 4. The benchmark queries and their evaluation time-cost

	Benchmark Queries	Time costs in seconds	
		Oracle with GS	Oracle without GS
Group A			
A1	?p_1:{?s <y:wrote> ?o ?p_1 ?o_1 ?p_2 ?o_2}	0.441	25.640
A2	?p,?p_1,?p_2:{<Palestine> ?p ?o ?p_1 ?o_1 ?p_2 ?o_2}	0.578	29.875
A3	?p,?p_1,?p_2:{?s ?p ?o ?p_1 <Palestine> ?p_2 ?o_2}	0.587	29.593
A4	?p,?p_1,?p_2:{?s ?p ?o ?p_1 ?o_1 ?p_2 <Palestine>}	0.556	29.641
A5	?o_2:{?s <y:isMarriedTo> ?o ?p ?o_1 <y:locatedIn> ?o_2}	0.291	21.922
A6	?s:{?s <y:happenedIn> ?o ?p_1 ?o_1 ?p_2 ?o_2 ?p_3 ?o_3}	0.587	62.734
A7	?o_3:{?s ?p ?o ?p_1 ?o_1 ?p_3 ?o_2 <y:happenedIn> ?o_3}	0.785	64.813
A8	?o_2:{?s ?p ?o ?p_1 ?o_1 <y:hasCurrency> ?o_2 ?p_3 ?o_3}	2.469	32.703
Group B			
B1	?p:{?s ?p ?o}	0.344	110.390
B2	?p:{?s ?p ?o ?p_1 ?o_1}	1.953	302.672
B3	?p:{?s ?p ?o ?p_1 ?o_1 ?p_2 ?o_2}	5.250	525.844
B4	?p:{?s ?p ?o ?p_1 ?o_1 ?p_2 ?o_2 ?p_3 ?o_3}	10.234	702.969
B5	?p:{?s ?p ?o ?p_1 ?o_1 ?p_2 ?o_2 ?p_3 ?o_3 ?p_4 ?o_4}	24.672	>1200

Figure 8. Graphs showing the evaluation time costs in seconds for the benchmark queries



store as an enhancement. The idea is to summarize a data graph and instead of querying the original data, we query the summary. Because the size of the summary is typically smaller than the original data, querying it can be faster than querying the original data. Our summary (the Graph Signature) is two-fold; the O-Signature, which groups nodes based on their outgoing paths, and the I-Signature which is based on incoming paths. Both summaries are stored separately but

are used jointly to produce precise results, using an original execution plan and its underpinning query evaluation theorems. We addressed the challenge of generating the Graph Signature for relatively large RDF graphs by introducing two disk-based versions of the Bisimilarity and Trace Equivalence algorithms. As the Graph Signature is a data graph, storing and querying it is done in the same way as the original data graph, using the techniques of the host RDF store. We have demonstrated our approach by implementing it on top of Oracle and experimenting it on a relatively large RDF dataset, showing that it indeed enhances Oracle.

We plan to extend our query model to cover tree and star-shaped queries. This might be a difficult challenge, but it is achievable since the linear query model is the building block for tree and star-shaped queries. For instance, Theorem 7 can be extended to tree-shaped queries of unbound nodes with the projection being the root of the query. Such query can be executed precisely using the O-Signature. Similar extensions will be made to the theorems and consequently to the execution plan. In addition, we plan to study and utilize the false-positive answers that are generated from queries not conforming to our query model. Such answers may be good-enough for certain types of applications such as information retrieval and search engines, where precise results are not necessarily needed. Also, we plan to develop a maintenance strategy to support querying dynamic datasets.

ACKNOWLEDGMENT

This research is supported by Sina Institute at Birzeit University and is an extension of a research that was originally supported by the SEARCHiN project (FP6-042467, Marie Curie Actions), coordinated by Prof. Marios Dikaiakos. We would like also to thank Ala' Hawash and Bilal Farraj for their valuable contributions in developing the initial work of this research.

REFERENCES

Abadi, D. J., Marcus, A., Madden, S. R., & Hollenbach, K. (2007). Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*, Vienna, Austria (pp. 411-422).

Champin, P., & Solnon, C. (2003). Measuring the similarity of labeled graphs. In *Proceedings of the Fifth International Conference on Case-Based Reasoning*, Berlin, Germany (pp. 80–95).

Chong, E. I., Das, S., Eadon, G., & Srinivasan, J. (2005). An Efficient SQL-Based RDF Querying Scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*, Trondheim, Norway (pp. 1216-1227).

Erling, O., & Mikhailov, I. (2007). RDF Support in the Virtuoso DBMS. In *Proceedings of the 1st Conference on Social Semantic Web (CSSW)*, Leipzig, Germany (pp. 59-68).

Erling, O., & Mikhailov, I. (2010). Virtuoso: RDF Support in a Native RDBMS. In R. de Virgilio, F. Giunchiglia, & L. Tanca (Eds.), *Semantic Web Information Management - A Model-Based Perspective* (pp. 501–519). Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-04329-1_21

Fernandez, J. C. (1990). An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, *13*(2), 219–236. doi:10.1016/0167-6423(90)90071-K

Goldman, R., & Widom, J. (1997). Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, Athens, Greece (pp. 436–445).

Hawash, A., Deik, A., & Jarrar, M. (2010). Towards Query Optimization for the Data Web - Disk Based Algorithms: Trace Equivalence and Bisimilarity. In: *Proceedings of the International Conference on Intelligent Semantic Web - Services and Applications (ISWSA'10)*, Amman, Jordan (pp. 131-137). doi:10.1145/1874590.1874607

Hellings, J., Fletcher, G. H., & Haverkort, H. (2012). Efficient External-Memory Bisimulation on Dags. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA (pp. 553-564). doi:10.1145/2213836.2213899

Henzinger, M. R., Henzinger, T., & Kopke, P. W. (1995). Computing Simulations on Finite and Infinite Graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, Milwaukee, Wisconsin, USA (pp. 453-462). doi:10.1109/SFCS.1995.492576

Jarrar, M., & Dikaiakos, M. D. (2008). MashQL: A Query-by-Diagram Language - Towards Semantic Data Mashups. *Proceedings of the 2nd International Workshop on Ontologies and Information Systems for the Semantic Web (ONISW'08), Part of the ACM CIKM Conference*. Napa Valley, California, USA (pp. 89-96). doi:10.1145/1458484.1458499

Jarrar, M., & Dikaiakos, M. D. (2009). A Data Mashup Language for the Data Web. In *Proceedings of the WWW2009 Workshop on Linked Data on the Web (LDOW'09)*, Madrid, Spain, online ceur-ws.org/Vol-538/ldow2009_paper14.pdf

Jarrar, M., & Dikaiakos, M. D. (2010). Querying The Data Web: The MashQL Approach. *IEEE Internet Computing*, *14*(3), 58–67. doi:10.1109/MIC.2010.75

Jarrar, M., & Dikaiakos, M. D. (2012). A Query Formulation Language for the Data Web. *IEEE Transactions on Knowledge and Data Engineering*, *24*(5), 783–798. doi:10.1109/TKDE.2011.41

Kaushik, R., Bohannon, P., Naughton, J. F., & Korth, H. F. (2002b). Covering Indexes for Branching Path Queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA (pp. 133-144). doi:10.1145/564691.564707

Kaushik, R., Shenoy, P., Bohannon, P., & Gudes, E. (2002a). Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, San Jose, California (pp. 129-140). doi:10.1109/ICDE.2002.994703

Luo, Y., de Lange, Y., Fletcher, G. H., De Bra, P., Hidders, J., & Wu, Y. (2013). Bisimulation Reduction Of Big Graphs On MapReduce. In *Proceedings of Big Data, the 29th British National Conference on Databases*, Oxford, UK (pp. 189-203). doi:10.1007/978-3-642-39467-6_18

Miller, R. B. (1968). Response Time in Man-Computer Conversational Transactions. In *Proceedings of the December 9-11, 1968*, *Fall Joint Computer Conference, Part I (AFIPS'68)*, San Francisco, California, USA (pp. 267-277).

Milo, T., & Suciu, D. (1999). Index Structures for Path Expressions. In *Proceedings of 7th International Conference on Database Theory (ICDT'99)*, Jerusalem (pp. 277-295).

Nestorov, S., Ullman, J., Wiener, J., & Chawathe, S. (1997). Representative Objects: Concise Representations of Semistructured, Hierarchical Data. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, Birmingham, UK (pp. 79-90). doi:10.1109/ICDE.1997.581741

Neumann, T., & Weikum, G. (2008). RDF-3X: A RISC-Style Engine for RDF. *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*, Auckland, New Zealand (pp. 647-659).

Paige, R., & Tarjan, R. E. (1987). Three Partition Refinement Algorithms. *SIAM Journal on Computing*, *16*(6), 973–989. doi:10.1137/0216062

Schätzle, A., Neu, A., Lausen, G., & Przyjaciel-Zablocki, M. (2013). Large-Scale Bisimulation of RDF Graphs. In *Proceedings of the Fifth Workshop on Semantic Web Information Management (SWIM '13)*, New York, NY, USA. doi:10.1145/2484712.2484713

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., & Zdonik, S. B. et al. (2005). C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*, Trondheim, Norway (pp. 553-564).

Tran, T., Ladwig, G., & Rudolph, S. (2013). Managing Structured And Semistructured RDF Data Using Structure Indexes. *IEEE Transactions on Knowledge and Data Engineering*, *25*(9), 2076–2089. doi:10.1109/TKDE.2012.134

Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., & Liu, L. (2013). Triplebit: A Fast and Compact System for Large Scale RDF Data. *Proceedings of the 39th International Conference on Very Large Databases (VLDB'13)*, Trento, Italy (517-528). doi:10.14778/2536349.2536352

Zeng, K., Yang, J., Wang, H., Shao, B., & Wang, Z. (2013). A Distributed Graph Engine For Web Scale RDF Data. *Proceedings of the 39th International Conference on Very Large Databases (VLDB'13)*, Trento, Italy (pp. 265-276). doi:10.14778/2535570.2488333

ENDNOTES

1 http://stats.lod2.eu. Accessed: June 2015.

2 In early June 2011, Bing, Google and Yahoo! introduced schema.org (accessed: June 2015); an ontology to be used to markup web pages.

3 http://developers.facebook.com/docs/reference/api. Accessed: June 2015.

4 http://search.fb.com. Accessed: June 2015.

5 For more about RDF, refer to the W3C documentation: http://www.w3.org/TR/rdf11-concepts/. Accessed: July 2015.

6 In Oracle 12c, the commercial name "Oracle Semantic Technologies" was renamed to "Oracle Spatial and Graph".

7 See www.neo4j.com. Accessed: July 2015.

8 This hashing function randomly generates a number between 1 and 4,294,967,295, which is sufficient for our purposes especially that the number of unique edge labels is typically not large and therefore the chance of collisions is negligible. Nevertheless, more complex hashing functions might be used such as MD5 or SHA, which produce 128-bit and 160-bit hash values, respectively – however on the cost of performance overhead.

9 For space limitations, proofs of the proposition and all the theorems can be accessed through www.jarrar.info/publications/GSProofs.pdf.

10 The reader might be interested to have a look at other experiments (with DBLP and DBPedia) that we conducted on a preliminary previous work (Jarrar & Dikaiakos, 2012).

Mustafa Jarrar is an associate professor of Computer Science at Birzeit University in Palestine. Before joining Birzeit in 2009, he was a Marie Curie Fellow at the University of Cyprus (2007-2009), and a Senior Research Scientist at Vrije Universiteit Brussel (1999-2007), where he completed his Masters (2000) and PhD (early 2005). He is the founder of both Sina Institute for Knowledge Engineering and Arabic Technologies, and the Palestinian e-Government Academy, at Birzeit University. Jarrar published +75 articles and refereed reports in the areas of Ontology Engineering, Lexical Semantics, Semantic Web, and Databases.

Anton Deik holds an MA in Hermeneutics from London School of Theology and a BEng in Computer Systems Engineering from Birzeit University. His areas of interest in Computer Science include Ontology Engineering, Semantic and Data Web, Graph Databases, Query Optimization, in addition to e-Learning. He also takes interest in Biblical Studies and Biblical Hermeneutics in the Palestinian context. He is a research collaborator at Sina Institute for Knowledge Engineering and Arabic Technologies at Birzeit University and a faculty member at Bethlehem Bible College, where he also serves as director for e-learning.