Bisimulation-based Structural Summaries of Large Graphs

by

Shahan Khatchadourian

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2016 by Shahan Khatchadourian
Abstract

Bisimulation-based Structural Summaries of Large Graphs

Shahan Khatchadourian
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2016

With an increasing number of heterogeneous entity descriptions available as large graphs that grow to millions of nodes and billions of edges, it is a challenge to understand, explore, and query these large graphs. Bisimulation-based structural summaries have often been used as a compact representation of the dataset that can improve query performance. However, current bisimulation summary construction techniques for large graphs do not scale and do not facilitate the use of summaries within existing systems. We address these challenges with three contributions. First, we describe bisimulation summary construction techniques for large graphs that leverage a novel singleton optimization which drastically reduces construction time. Second, we show how structural summaries can be used to improve query performance within existing RDF systems. Third, we give an ontology for describing structural summaries as RDF that enables their use and verification with existing RDF tools. Our work also demonstrates that the S+EPPs system, built on top of existing RDF processors, is an efficient, scalable, and flexible approach to exploring and querying large graphs using bisimulation-based structural summaries.
To my family
Acknowledgements

I would like to give special thanks to my supervisor Prof. Mariano P. Consens for his unwavering and extensive energy, advice, teaching, guidance, and support. I would also like to thank my co-supervisor Prof. Michael Grüninger for his advice, comments, and encouragement. There is no doubt in my mind that my unique and memorable experience under their supervision forged my success.

For their valuable advice and feedback to improve my work, I thank my committee members Prof. Eyal de Lara, Prof. Cristiana Amza, Prof. Gerald Penn, and my external examiner Prof. Josep-Lluis Larriba-Pey. I would like to thank my collaborators, Dr. Valeria Fionda and Dr. Giuseppe Pirrò, and my mentor at IBM Jérôme Siméon, for their research insights. I would like to express my deep gratitude to Dr. Mir Sadek Ali, Dr. Daniele Barone, Ph.D. candidate Agop Koulakezian, Prof. Kelly Lyons, Prof. Sheila A. McIlraith, Prof. Renée J. Miller, Prof. Reza Samavi, and Dr. Thodoros Topaloglou for their guidance and mentorship. My ideas and goals would not have been the same without them all.

I would like to thank two very important families for their boundless support and incomparable love. I thank my parents Dr. Arthur and Nora, my brother Yervant, and my brother Varant and his wife Lucia. I also thank Katie and her family, Marsha, Dr. Mark, Dan, Adam, and Monica.
## Contents

1 Introduction .......................... 2
   1.1 Querying RDF using bisimulation summaries ......................... 3
   1.2 Thesis outline and contributions .................................. 7

2 Related Work .......................... 11
   2.1 Summary construction ........................................... 12
   2.2 Query optimization using structural summaries ...................... 15
   2.3 Summary representations ........................................ 17

3 Summary Construction ................. 19
   3.1 Hadoop Implementations ........................................ 20
      3.1.1 Dataset graph translation .................................. 21
      3.1.2 p/-p variation ............................................. 22
      3.1.3 p&-p variation ............................................. 24
      3.1.4 p&-p/reset variation ....................................... 26
      3.1.5 Discussion ................................................ 27
   3.2 GraphChi summary construction ................................... 27
      3.2.1 Dataset graph translation .................................. 27
      3.2.2 Summary construction algorithm ................................ 28
      3.2.3 Singleton optimization ..................................... 29
   3.3 Evaluation ..................................................... 33
      3.3.1 Datasets .................................................. 34
      3.3.2 Constructing FWBW summaries ................................ 34
      3.3.3 Comparing GraphChi with Hadoop ............................. 39
   3.4 Summary ....................................................... 41

4 Summary-based Query Optimization .... 42
   4.1 EPPs query expressions .......................................... 44
      4.1.1 EPPsToEPPsS translation ................................... 44
      4.1.2 EPPsToSPARQL translation .................................. 45
      4.1.3 Properties of selective summaries ............................ 46
   4.2 Experimental results ............................................ 46
      4.2.1 Datasets .................................................. 47
      4.2.2 Selecting queries .......................................... 47
      4.2.3 Comparing SPARQL engines .................................. 49
      4.2.4 Comparing query plan evaluation ............................. 52
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Dataset graph statistics</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>FWBW summary statistics, with M1 times (in mins) to load input, construct summary (using B or S variation), and write summary</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>FW summary statistics, with M2 construction times (in mins)</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Syntax of EPP expressions</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Instance and FWBW summary statistics for DBpedia full and selective summaries</td>
<td>47</td>
</tr>
<tr>
<td>4.3</td>
<td>Structure, number of predicates, and number of results for LMDB queries</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Structure, number of predicates, and number of results for DBpedia queries</td>
<td>49</td>
</tr>
<tr>
<td>4.5</td>
<td>Query time (in secs) with lmdb on Virtuoso (v-), Fuseki (f-), and Sesame (s-) with dataset (-inst) and summary (-sum) graphs</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>Disk space usage (with load time in secs in parentheses) of lmdb for Fuseki, Sesame, and Virtuoso with dataset (I) and summary graphs (I+S)</td>
<td>51</td>
</tr>
<tr>
<td>4.7</td>
<td>For each predicate of LinkedMDB query LQ01 of Table 4.5, we show the number of statements in the graph (suffix -stmts), the number of intermediate rows in Virtuoso’s query evaluation (-rows), and the order in which Virtuoso evaluates predicates (-order) in the EPPsI (prefix i-) and EPPsS (s-) summaries</td>
<td>53</td>
</tr>
<tr>
<td>4.8</td>
<td>Query time (in secs) of DBpedia instance, dbp summary, and d30 and d13 selective summaries</td>
<td>56</td>
</tr>
<tr>
<td>4.9</td>
<td>Virtuoso disk space usage (in GB) with load time (in mins in parentheses) of dbp dataset graph (I), plus full and selective summaries (I+S)</td>
<td>57</td>
</tr>
<tr>
<td>4.10</td>
<td>Time (in mins) for RDF dataset load, full and selective summary construction with B and S variations, and RDF summary write</td>
<td>58</td>
</tr>
<tr>
<td>A.1</td>
<td>DBpedia queries</td>
<td>86</td>
</tr>
<tr>
<td>A.2</td>
<td>Statistics of predicates in instance graph, FWBW full, d30, and d13 summary graphs, and count of distinct queries in DBSB logs</td>
<td>87</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Dataset and summary graphs of motivating example ........................................... 4  
1.2 Example instance query graph .............................................................................. 5  
1.3 Example instance query answer navigation ............................................................ 6  
1.4 Example summary query graph ............................................................................ 6  
1.5 Example summary query answer navigation across block edges (red) and extent edges (orange) ........................................................................................................... 7  
1.6 Summary graph highlighting singleton blocks (marked with a dark red border) .......... 7  
1.7 Architecture for S+EPPs system ............................................................................ 9  

3.1 Pig Latin dataset graph construction .................................................................... 21  
3.2 Bisimulation using p/-p, initialization map phase .................................................... 22  
3.3 Bisimulation using p/-p, update jobs .................................................................... 23  
3.4 Bisimulation using p&-p, initialization map-only job ............................................. 24  
3.5 Bisimulation using p&-p, update jobs .................................................................... 25  
3.6 Bisimulation using p&-p/reset, initialization map-only job .................................... 26  
3.7 Bisimulation using p&-p/reset, update job ............................................................. 26  
3.8 B variation algorithm ........................................................................................... 29  
3.9 S variation algorithm ........................................................................................... 30  
3.10 Unstable singleton example ................................................................................ 32  
3.11 Unstable/stable/unstable example ....................................................................... 33  
3.12 LMDB FWBW summary construction ................................................................. 36  
3.13 spb256 FWBW summary construction ................................................................ 37  
3.14 DBpedia FWBW summary construction .............................................................. 37  
3.15 Twitter FWBW summary construction ................................................................ 38  
3.16 LMDB FW summary construction ...................................................................... 40  
3.17 DBpedia FW summary construction ................................................................... 40  
4.1 Iterations of FW full and selective summaries on an example instance .................. 46  
4.2 Query performance ratio of summary versus instance on Virtuoso (v-), Fuseki (f-), and Sesame (s-) systems ................................................................. 51  
4.3 For each of LQ01’s predicates (in alphabetical order), ratio of number of statements in summary graph over number of statements in instance graph (ratio-stmts), and ratio of intermediate rows in EPPsS over intermediate rows in EPPsI (ratio-rows). .......... 54  
4.4 Query performance ratio of summary versus instance for DBpedia dbp summary, and d30 and d13 selective summaries ..................................................... 57  
4.5 d30 FWBW selective summary construction ......................................................... 58
Chapter 1

Introduction

In this chapter, we motivate and introduce an approach to answer SPARQL \[54\] queries on Resource Description Framework (RDF) \[35\] datasets using RDF summaries. We show a scalable approach to construct summaries of large RDF datasets using graph analytics platforms with an effective optimization that dramatically reduces summary construction time for both real-world and synthetic datasets. We further show that the use of RDF summaries in unmodified SPARQL processors substantially improves query performance. We use a formal representation that also allows the use of unmodified semantic web tools such as those used for verification using automated reasoning. This chapter ends with the thesis outline and contributions.

We can model real-world relationships, such as online social networks \[76, 84\] and links across the broader World Wide Web \[20\], as graphs using multi-relational, predicate-labeled edges. In Figure 1.1 (i) for example, which shows a fragment of the LinkedMDB \[56\] RDF dataset, edges labeled :country, :actor, :type, and :movie contribute to descriptions of relationships between people, places, and movies. The volume and variety of data that organizations and individuals are generating is increasing at a notable pace and a recent business survey \[1\] captures the need for businesses to incorporate strategies involving analysis of Big Data in order to remain competitive. For example, the quintessential RDF dataset of the Linked Open Data (LOD) initiative, DBpedia \[13, 17\], contains tens of millions of nodes and hundreds of millions of edges including descriptions of people and places. Furthermore, a Twitter dataset that we use in our experiments has almost 2 billion follower edges between users.

To process large datasets, users often turn to distributed parallel processing paradigms including Message Passing Interface (MPI), a distributed shared-memory architecture, as well as MapReduce \[89\] and Pregel \[83\], which are distributed shared-nothing architectures. These paradigms aim to process large datasets by assigning portions of work amongst parallel tasks on different machines, providing scalability by adding more machines. Recently, parallel processing frameworks such as GraphChi \[77\] show the effectiveness of processing large datasets using (non-distributed) multi-core architectures and demonstrate the need for much less parallelism and hardware than distributed architectures.

An additional motivation for this work, which we do not pursue in this thesis, is the visualization and exploration of RDF datasets \[67, 68, 69\]. In this work, we focus on an approach to query and explore large graphs that leverages processing of large datasets using multi-core architecture. One approach to query and explore large graphs includes writing navigational queries, queries that return only the endpoints, in SPARQL 1.1 \[54\] and the EPPs \[41\] language. Navigational queries try to address the
complexity of formulating queries and can also express nested navigations succinctly.

Given the large graphs that users wish to explore and query, the challenge is to understand and query the large number of nodes and their relationships. One way to improve query performance is to construct and use a *structural summary* [43], a graph-based index that groups nodes using some notion of equivalence such as bisimilarity [58], which has broad applicability [107] (in areas such as for model checking [65] and information retrieval [5]). Structural summaries have seen extensive use with different data formats including XML [32, 25, 66], RDF [67, 51, 106], and more recently, property graphs [7, 9, 55], a graph model with key-value pairs on nodes and edges. *Bisimulation summaries* provide valuable insight into describing the semi-structure of graph data [67], and existing literature shows that summaries can improve query performance [30, 105], in particular, SPARQL query performance [98, 118].

The outline of this chapter is as follows. In Section 1.1, we give a motivating example that shows how an RDF summary can support query optimization. In Section 1.2, we give the outline of this Thesis, then we give the Thesis contributions. We give an architectural overview of S+EPPs [30], a system that combines fast summary construction, plus optimization of navigational queries on large graphs using summaries on existing SPARQL systems.

### 1.1 Querying RDF using bisimulation summaries

In this section, we present a motivating example that shows how to answer a query on a fragment of the LinkedMDB RDF dataset, then to answer its corresponding optimized query using a summary instead. We first describe a translation of RDF data into a dataset graph.

Let $S$ represent a set of *URIs*, labels that identify web addresses, and let $T$ represent a set of XSD-typed [127] literal values. An *RDF dataset* [35] is a set of *triples* $D \subseteq S \times S \times (S \cup T)$ where each triple has a subject, predicate, and object URI, and the object can optionally be an XSD literal value. We call a *quad* a triple that is extended with a *graph* URI to identify its origin. For example, the quad

(http://linkedmdb.org, dir:8487, rdfs:label, “Stanley Kubrick”) from LinkedMDB [56] says that dir:8487 has text label “Stanley Kubrick”.

A *dataset graph* $G = (V, E, l, L, m, M)$ satisfies the following properties: (1) $V$ is a finite set of nodes; (2) $l$ is a bijective label function that maps each $v \in V$ to a distinct label $l(v) \in L$, the set of node labels; (3) $E \subseteq V \times V$ is a finite multi-set of labeled, directed edges, where an edge $(v, v') \in E$ between two nodes $v, v' \in V$ represents an edge from a node $v$ to a node $v'$; and (4) $m$ is a label function that maps each edge $e \in E$ to a label $m(e) \in M$, the set of edge labels.

Figure 1.1 (i) shows a sample of triples from the LinkedMDB [56] RDF dataset as a dataset graph. We represent RDF quads as a *dataset graph* as follows. First, for each distinct URI in a subject or object position, we create a node with the same label. The dataset graph describes 11 URIs (each one is a node) with 14 triples (edges). We then create a node for each literal value, adding 7 literal nodes. Finally, for each quad, we create an edge from the subject node to the object node, and the edge’s label is a pair that consists of the quad’s predicate and graph URIs; for succinctness, we do not display the LinkedMDB graph URI in this figure. An example quad that says that “Stanley Kubrick” is a director is shown as the node dir:8487 with a rdfs:type predicate-labeled edge to node :Director. The reverse of this translation can convert a dataset graph into a set of quads.

We construct bisimulation summaries that group nodes using bisimilarity as the notion of equivalence.
Chapter 1. Introduction

Figure 1.1: Dataset and summary graphs of motivating example

Given a dataset graph $G = (V, E, l, L, m, M)$, we define two nodes $v, v' \in V$ as forward bisimilar (which we write as $v \approx_{FW} v'$), which determines node equivalence along outgoing edges, iff either of the following conditions hold: (1) if $v$ and $v'$ have no outgoing edges; or (2) if $v$ has an edge $(v, w) \in E$ with label $p \in M$, there exists an edge $(v', w') \in E$ with label $p$, and $w \approx_{FW} w'$, and if $v'$ has an edge $(v', w') \in E$ with label $p \in M$, there exists an edge $(v, w) \in E$ with label $p$, and $w \approx_{FW} w'$. A closely related counterpart to forward bisimilarity, backward bisimilarity (which we write as $v \approx_{BW} v'$), which determines node equivalence along incoming edges, iff either of the following conditions hold: (3) if $v$ and $v'$ have no incoming edges; or (4) if $v$ has an edge $(w, v) \in E$ with label $p \in M$, there exists an edge $(w', v') \in E$ with label $p$, and $w \approx_{BW} w'$, and if $v'$ has an edge $(w', v') \in E$ with label $p \in M$, there exists an edge $(w, v) \in E$ with label $p$, and $w \approx_{BW} w'$. We say that two nodes $v, v'$ are FWBW bisimilar (in short, bisimilar), which we write as $v \approx v'$, iff $v \approx_{FW} v'$ and $v \approx_{BW} v'$. We can materialize a summary as RDF by using the reverse translation of RDF into a dataset graph such that each directed, predicate-labeled block edge in the summary constructs an RDF triple whose subject and object are the source and target blocks, respectively, and whose predicate is that of the block edge’s predicate. Optionally, RDF summary statements can be expressed as quads, such as to store a summary’s block edges with a different graph URI that extent edges. By representing a summary as RDF, it can be loaded and used in a SPARQL store alongside its original RDF dataset. We now describe an example RDF summary then describe how to use the summary for query optimization.
Figure 1.1 (ii) shows a FWBW bisimulation summary. Each node in the summary is a block, shown as a rectangle identified with a hash URI, such as the block with URI bc:5653juk. Each block has bc:extentContains predicate-labeled edges to all the nodes in its extent, such as block bc:5653juk which contains the country URI :i in its extent, i.e., block(bc:5653juk). Additionally, each block connects to those nodes in the fragment that are in its extent using bc:extentContains predicate-labeled edges. Blocks are also connected by block edges that summarize the corresponding predicate-labeled edges between data nodes across blocks (e.g., :actor, :director, :country, etc.).

In this work, we focus on navigational queries (that return specific nodes) are part of the W3C-standard RDF query language, SPARQL [54] and its proposed EPPs [41] sub-language. We now show how to optimize navigational queries using summaries.

**Example 1** Consider the graph in Figure 1.1. Find actors who appear in a movie without country information; additionally, the director of those movies should also be the producer of a movie with country information.

We can return the answer of our example navigational query by matching node and edge labels. The above query is evaluated on the instance graph in Figure 1.1 (i) as follows. A graph navigation that matches the answer to the example query starts from the :Director node and navigates across incoming :type edges to the 3 instances of director, nodes dir:8487, dir:15437, and dir:10810, from where it is necessary to check the existence of a traversal across :producer edges followed by :country edges. Then, there is also a traversal from node dir:8487 across :director edges to movies; then, movies that do not support a :country edge traversal (i.e., movie:38125) have to be kept. Finally, a traversal across :actor edges from movie:38125 returns actor nodes, i.e., actor:45772. The answer to a navigational query is the set of distinct endpoints in the subgraphs that match a query, and so the example’s answer is the actor actor:45772.

We can visualize the example navigational query as a graph pattern, as depicted in Figure 1.2, which uses variables (prefixed with ‘?’) to match node and edge labels; the query represents a navigation in the instance graph from the :Director node to reachable actor nodes. The query checks that movies have a country by checking the existence of a :country edge, while the query uses a filter to ensure a :country edge is missing from movies that should not have country information.

The left side of Figure 1.3 shows the nodes and edges in the instance graph that the example query matches (edge traversals are highlighted in red); we rely on navigational queries based on the EPPs language [41] that return only endpoints. Figure 1.3 shows the navigation of the example query in the instance graph. Notice that the navigation proceeds from the :Director node to the node actor:45772, and the check for a missing :country edge appears as an edge that is crossed on movie:38125.
Chapter 1. Introduction

Figure 1.3: Example instance query answer navigation

![Figure 1.3: Example instance query answer navigation](image)

Figure 1.4: Example summary query graph

![Figure 1.4: Example summary query graph](image)

We can translate the example navigational query on the instance into a corresponding navigational query that is answered using the summary instead. Figure 1.4 shows the graph pattern of the example navigational query that will answer the query by traversing the summary graph instead of the dataset graph. The query graph can traverse `bc:extentContains` edges back and forth between summary blocks and instance nodes, such as for retrieving result instances (as is the case with `?Actor` instances) or other instance filters (as is the case with `:Director`).

Figure 1.5 shows the navigation of the example query in the summary graph. Notice that the query navigates block edges labeled the same as in the instance (shown in red) as well as extent edges (shown in orange) including accessing the `:Director` instance and the reachable actor node `actor:45772`. Also notice that the check for a missing `:country` edge appears as an edge that is crossed on the movie block containing `movie:38125`. Observe that a summary can help with query optimization by potentially reducing the number of navigations that do not contribute to the answer. For example, there are only two `rdf:type` and `:director` block edges involving the non-singleton blocks `bc:9ki4lq2m` and `bc:aeh0rt4j`, and is less than the number of four corresponding edges in the instance graph involving the nodes in the extents of these two blocks.

A singleton is a block whose extent contains exactly one node, i.e., a node that is not bisimilar to any other node. Notice that the majority of blocks in the summary of Figure 1.6 are singletons (highlighted...
Chapter 1. Introduction

1.2 Thesis outline and contributions

In this section, we propose our \textsc{S+EPPs} system \cite{30} then give our Thesis outline followed by our Thesis contributions.

In Chapter \cite{2}, we review existing literature with a focus on summary construction of large graphs in...
Section 2.1 using summaries for query optimization in Section 2.2 and semantic web representations of summaries in Section 2.3.

In Chapter 3, we describe several implementations of summary construction algorithms for Hadoop and GraphChi and give experimental results that evaluate and compare their performance. In Section 3.1, we give Hadoop implementations to construct FWBW summaries. Our Hadoop-based implementations to construct FWBW summaries extend the work of [108], a Hadoop-based FW summary construction (they do not provide a FWBW variation). To the best of our knowledge, our Hadoop-based FWBW summary constructions are the first to appear in literature. In Section 3.2, we give GraphChi implementations to construct FW and FWBW summaries, and propose a novel and effective singleton optimization that can drastically reduce per-iteration times within the first few iterations. We choose GraphChi as it is an efficient multi-core graph processing framework. In Section 3.3, we give an experimental validation that our GraphChi construction, with our optimization, achieves our goal of constructing summaries of large graphs in an amount of time similar to the time required to load the dataset plus write the summary. In our experiments, we use graphs ranging from millions to billions of edges. Additionally, we compare construction of FW summaries using the Hadoop-based summary construction of [108], which only constructs FW summaries, and our GraphChi implementations and give evidence that our summary construction using GraphChi is effective; we focus our comparison of approaches on FW summary construction since ours is believed to be the first Hadoop-based FWBW summary construction. In Section 3.4, we give a summary of the chapter and its contributions.

In Chapter 4, using navigational queries written using the EPPs [41] language, we show that bisimulation summaries are effective at improving query performance across a range of navigational queries, real-world datasets, and scenarios. In particular, we show that selective summaries, which summarize a subset of a dataset, such as a subset of predicates present in DBpedia’s query logs [86], can improve query performance more than a full summary that considers all the predicates in DBpedia. The predicates we choose for selective summarization range in popularity in the query logs, in the number of statements in the instance dataset, and the number of block edges in the summary, showing that our summaries are effective for improving query performance in real-world scenarios. In Section 4.1, we describe two translations for EPPs navigational expressions that allows using summaries alongside instance graphs: \texttt{EPPsToEPPsS}, that translates a navigational query on a dataset graph into a query on the dataset’s corresponding summary graph; and \texttt{EPPstoSPARQL}, that translates an EPPs expression to SPARQL. In Section 4.2, we give experimental results that show summaries can improve query performance across systems and a variety of real-world graphs including LinkedMDB and DBpedia. We first give experimental evidence that our summaries improve query performance on a variety of unmodified SPARQL processors. We then give experimental evidence that our summaries improve query performance with a variety of real-world datasets and full summaries. We then give experimental evidence that our selective summaries improve query performance more than a full summary, and that selective summaries are faster to construct while benefiting from our singleton optimization. In Section 4.3, we give a summary of the chapter and its contributions.

We propose the \texttt{S+EPPs} system [30] that addresses two main scenarios involving large graphs: efficient summary construction and query optimization. The architecture of \texttt{S+EPPs}, shown in Figure 1.7 consists of two sub-systems: a multi-core GraphChi system that constructs RDF summaries from RDF instance graphs, and a SPARQL system to query RDF instance graphs and RDF summary graphs. The SPARQL system initially has an RDF instance graph that is loaded into the multi-core GraphChi
system, which represents graphs using its internal graph format. Our bisimulation summary construction algorithm uses GraphChi for scalable construction of RDF summaries that are loaded back into the SPARQL system. The S+EPPs system accepts EPPs navigational queries on RDF graphs, translating EPPs queries on an instance graph into SPARQL queries using an EPPstoSPARQL translation, and also translating EPPs queries on an instance graph into a corresponding summary-optimized queries using an EPPstoEPPsS translation. To the best of our knowledge, S+EPPs is the first system that pairs fast building of bisimulation summaries with navigational query translations to leverage summary-based optimizations within existing SPARQL query processors (without modifying the SPARQL engine internals). In addition, by storing summaries alongside their original graphs, users can explore a dataset’s semi-structure to understand the dataset and compose relevant queries.

In Chapter 5, we describe an ontology for representing bisimulation summaries that can be used by existing semantic web tools, such as for verification using automated reasoning. The summary ontology is defined using the competency questions given in Section 5.1, which validates the need to describe a summary and its relationship with the original instance and other summaries (such as during a summary construction algorithm). To develop an ontology that satisfies the competency questions, we develop a summary ontology that has the classes and properties given in Section 5.2. Specifically, the ontology we give serves the need to describe a summary’s blocks and block edges, as well as relationships between summaries and the original instance, as well as between summaries such as during an iterative construction. Furthermore, to satisfy relevant constraints, such as that each instance may appear in exactly one block, in Section 5.3, we author the ontology using OWL 2 notation and verify logical constraints through an example summary that partitions a dataset’s nodes. In Section 5.4, we give the context for the ontology with a discussion of its semantic dimensions, such as the ontology’s use with automated reasoners, and pragmatic dimensions, such as the ontology’s degree of granularity. In Section 5.5, we give a summary of the chapter and its contributions.

Summary of contributions With organizations generating large volumes of graph semi-structured data, it becomes increasingly challenging to explore and understand this data. Examples of large graphs include DBpedia’s RDF dataset that contains tens of millions of nodes and hundreds of millions of edges to describe entities such as people and location, and Twitter’s follower graph that contains close to 2 billion edges. Users can benefit from summaries to understand and query such large datasets, summaries have seen use for multiple formats including XML and RDF. Despite existing iterative,
parallel summary construction algorithms, starting from a hash-based approach that was parallelized into an MPI implementation and then for MapReduce, there is a need to support multi-core architecture. Furthermore, to improve query performance, a summary can be used as a graph-based index. This thesis describes our multi-core approach to constructing and using RDF summaries. We describe a novel optimization that can drastically reduce the per-iteration time during summary construction of large RDF datasets. We also propose the use of an RDF summary, described using a formal ontology, within existing unmodified SPARQL processors for query optimization. We show the effectiveness of our summary construction and query optimization approaches, and validate the ontology for describing summaries.

Some of the systems experience needed to develop scalable implementations was gained while a research intern with the Scalable XML Infrastructure group at IBM Watson. Our work appears in [72, 73], with an application shown in a collaboration with Paradies [95]. Specifically, collaborations at IBM provided the author with the background and experience needed to integrate the MapReduce paradigm within IBM’s implementation of the XQuery XML query language. The Hadoop-based software was written by the author, with several modifications contributed by the author to IBM’s closed-source XQuery compiler.

The work that we describe in the introduction of Chapter 3, together with Sections 3.2 and 3.3, appears in [70] and in [30]. The work in Chapter 3 is being prepared as part of a joint submission with Consens, Fionda, Khatchadourian and Pirrò [31], part of which is also demonstrated in [30]. The authors Consens, Fionda, Khatchadourian, and Pirrò have presented initial work in [29].

We have implemented the p/-p, p&-p, and p&-p/reset FWBW summary construction variations that we give in Section 3.1 in Hadoop. We have also implemented the Hadoop-based FW summary construction of [108] used in the experimental comparison in Section 3.3.3. Furthermore, we have implemented our B and S GraphChi summary construction variations using unmodified GraphChi 0.2, which is available as open-source. Our singleton optimization is novel and we integrate it into our GraphChi summary construction. Our Hadoop-based and GraphChi-based summary construction code, which is written by Shahan Khatchadourian, is available as open source under the Apache License.

The EPPs language that we use in Chapter 4 is developed by Fionda, Pirrò, and Consens in [41], and they have kindly made the software available as open-source; their software components are identified in our architecture shown in Figure 1.7 as the boxes labeled EPPsItoEPPsS and EPPstoSPARQL. In particular, the authors have made the Java source code available with its EPPsItoEPPsS and EPPstoSPARQL translations.

The summary ontology that we describe in Chapter 5 was written solely by the authors; we also developed the ontology using OWL 2 and verified it using the open-source Java Pellet reasoner.

1https://github.com/shahankhatch/ScalableGraphSummaries
Chapter 2

Related Work

In this chapter, we describe work related to answering queries on RDF data using structural summaries. In this section, we describe scalable ways to query RDF data using Hadoop. In Section 2.1, we describe work related to scalable constructions of structural summaries. In Section 2.2, we describe work related to scalable querying of RDF using summaries. In Section 2.3, we describe work related to representation of structural summaries in RDF.

We do not consider value-based summaries [27, 40, 113, 102], which represent value aggregations over semi-structured graphs, because they cannot answer navigational queries. Furthermore, our focus on multi-core and distributed-parallel frameworks does not include NUMA architecture, a paradigm for sharing memory of multiple machines, or graphics card approaches to computing bisimulations [124] as they are out of scope.

Existing literature shows that Hadoop can process and query large RDF datasets such as by using a system [60] that stores RDF data on Hadoop’s distributed storage after partitioning the input, where files are partitioned by each statement’s predicate and object as a way to increase parallelism. The system answers queries using one or more Hadoop jobs, where each job executes a join using a variable appearing in more than one triple. As a way to improve query performance, the system executes joins in order of decreasing number of triples a variable joins; this approach is dependent on the selectivity of predicates in the dataset. System scalability follows a sub-linear increase in query times with respect to increasing the scale of synthetically generated data containing up to billions of triples.

The SHARD triple-store [103, 104] uses Hadoop to process queries on large RDF datasets using an iterative and incremental evaluation over variable bindings. SHARD answers queries by using jobs to iteratively bind data to variables, and sometimes faces the challenge of poor performance due to a chosen join order. Another system [89] also processes RDF data in an iterative manner, using multi-way joins over variable bindings as a way to reduce the number of iterations.

PigSPARQL [109] is an approach to processing SPARQL queries that leverages Pig Latin [93], a high-level declarative language for Hadoop that supports a nested-record data model. PigSPARQL processes SPARQL queries by transforming an input RDF dataset into an instance represented in Pig Latin’s data model, then translating a SPARQL expression into a Pig Latin expression that executes Hadoop jobs over the instance. PigSPARQL’s advantages include that it does not modify Hadoop and benefits from Pig Latin’s high level language and job optimizations to reduce job duration. PigSPARQL executes a query’s joins in order of increasing number of times a variable appears as a join variable as a way to
increase join selectivity, and is in contrast to other systems [89].

In this section, we introduced some Hadoop-based systems to answer queries on RDF data. In the next section, we describe related work of scalable construction of structural summaries.

## 2.1 Summary construction

In this section, we describe existing literature on scalable construction of structural summaries of RDF data. We first describe Hadoop-based systems to process large graphs, and then describe existing literature on computing structural summaries of large graphs.

Pregel [83], an approach to processing graphs using Hadoop’s parallelism, aims to simplify describing graph algorithms with an iterative computation model where a graph node can send messages to and receive messages from other nodes; messages sent in an iteration are visible to their recipient in the next iteration. Two maturing, open source Pregel implementations based on Hadoop are Apache Hama [11] and Apache Giraph [10]; their current limitations are due to the main-memory storage of messages, which can be negatively affected by potentially sending an exponential number of messages in an iteration.

Green-Marl [59] is a high-level domain-specific language for describing graph algorithms that can produce optimized code for different parallel graph paradigms, including Apache Giraph’s implementation of Pregel. Green-Marl’s simplified model on top of Pregel’s computation model allows a developer to describe their algorithm using graph notions such as nodes and outgoing edges, without specifying parallelism details such as sending of messages, but while retaining the ability to control parallelism using simple synchronization operators. A limitation that approaches such as Green-Marl face is that they cannot produce Pregel-like programs that effectively parallelize depth-first traversal due to its inherently sequential nature. Green-Marl is an interesting approach for translating high-level graph algorithms into low-level parallel implementations.

There is existing work comparing simulation and bisimulation in terms of partitioning problems [42]. In terms of query features that bisimulation summaries and simulation summaries support, their main difference is that bisimulation summaries can answer queries that use negation while simulation summaries cannot [100]. For this reason, we focus on the ‘stronger’ equivalence notion of bisimulation summaries.

Because bisimulation is a P-complete problem [14, 26, 99], it is difficult to parallelize. Attempts at scaling summary construction include improving performance by taking advantage of linear complexity of computing summaries of directed-acyclic graphs [57], using external memory to support processing of graphs that do not fit in main-memory [80], or using Hadoop for summary construction [81, 108].

One approach to constructing summaries targets acyclic graphs and uses a single CPU with external-memory data structures [57]. The linear complexity of this approach is reflective of its constraint to acyclic graphs [37]. Effective use of a single machine involves sorting the input graph nodes in reverse-topological order, that is, an edge’s target node must appear before its source. This mechanism has a worst-case IO-complexity of $O(\text{SORT}(|V| + |E|))$, where the SORT function returns the number of accesses to external memory needed to sort the $V$ nodes and $E$ edges of a dataset graph. The sort order enables time-forward processing, a message-passing approach that sends messages to its neighbourhoods, not unlike Pregel’s message-passing approach. The mechanism uses time-forward processing to compute blocks of the partition by iteratively combining smaller blocks into bigger blocks, and then iterating over a sorted set of smaller blocks. Empirical results shows that this mechanism is effective with both
real-life and synthetic datasets using a single commodity machine, with summary construction of graphs containing billions of nodes and edges taking only seconds.

ExpLOD’s usage summaries are a flexible structural summary for RDF data. A usage summary is a forward bisimulation that groups nodes based on classes and predicates. For example, one usage summary groups entities by the set of types that they are an instance of, as well as attributes that describe them. We have three implementations of ExpLOD. The first implementation constructs usage summaries of datasets that fit in main memory. This approach computes the relational coarsest partition of a graph, essentially using partition refinement and takes $O(|E| \log |V|)$ time. The second implementation issues SPARQL queries against an RDF triple store which contains RDF datasets. Although the second approach enables some form of scalability, it is slow due to the number of queries that the store needs to answer, and a query may take time only to then return an empty result. The third implementation uses Hadoop to generate usage summaries of large RDF datasets in a parallel, scalable way, and is the fastest of the three implementations. Our Hadoop-based implementation can construct structural summaries of substantial crawls of the LOD Cloud from the years 2010 and 2011, which contain billions of triples. Our results show that usage summaries are an accurate and effective tool to identify unique descriptions in large RDF datasets.

Using Pig Latin’s nested-record data model, we present in Figure A.1 of the Appendix one of our initial attempts at using Hadoop to construct two FW class usage summaries. The first usage summary groups entities based on the set of classes that it is a type of, and the second usage summary groups entities on the sets of classes that it is a types of, as well as the provider that made that assertion; notice that the second summary is a refinement of the first. Using Pig Latin allows a developer several benefits including operators on relations like selection, joins, projection, and user-defined functions. We give details of Pig Latin’s features and drawbacks using our example script, a script that computes one iteration of a FW summary computation, more specifically, a dataset’s class usage.

The input to the script in Figure A.1 are files on Hadoop’s distributed file system (HDFS) containing quads that are loaded into a relation having a column for subject, predicate, object, and graph URIs (line H1). Line H3 examines each quad to determine the class and predicate usage. Notice that the class and predicate usage neighbourhood can be decided at a statement level and so does not require a self-join to examine longer paths. In this example, since we are interested only in class usage, other usages are filtered on line H8.

Furthermore, in Figure A.1, each instance’s usages are grouped in order to assign a bisimulation identifier. Line H10 groups each instance’s class usages by placing all tuples having the same suri into a bag with the field name usagesstmtsclass. The foreach operation from lines H12 to H24 iterates over each instance’s record and computes two bisimulation identifiers. The first identifier is computed from the set of distinct classes (line H13-H16), and the second identifier is computed from the set of distinct (class usage, guri) pairs (lines H18-H21) which captures the sets of classes usages and the providers that expressed them. Each identifier is computed as an MD5 hash of the distinct and ordered set of usages; barring collisions, sets of usages containing the same members will have the same hash value. The foreach outputs records containing a distinct instance URI, its bisimulation identifiers and the (distinct) usages that contributed to each bisimulation identifier. In essence, this creates an initial partition where each block contains a single instance, and which is more refined than the coarsest, stable partition of both summaries. So to combine instances having the same bisimulation identifier into the same block, instances are grouped by bisimulation identifier (lines H35 and H36).
The output of the Pig Latin script of Figure A.1 is a set of files containing nested-records that are stored on HDFS. Distinct bisimulation identifiers and their usages are stored to HDFS on lines 26 to 32 for later reference. The two usage summaries are stored (lines 34-38) as relations containing bisimulation identifier and a bag of instance URIs that have the same bisimulation identifier. Notice that identical bisimulation identifiers computed from class usage and provider guri will also have an identical bisimulation identifier computed from just class usage, since the first is a refinement of the latter.

The script of Figure A.1 was a first-step to computing usage summaries in Pig Latin, but it differs from our intended approach in four main ways. First, ExpLOD’s mechanism for creating usage summaries accesses properties of a node’s neighbours, such as their block identifier, a feature that is not automatically available in shared-nothing frameworks such as MapReduce; though time-forward processing techniques as leveraged in [57] can be of use. Second, the Pig Latin script in Figure A.1 requires that a set of usages from which a bisimulation identifier is to be computed fits in memory. One option is to use set-similarity joins [122] by transforming each input record into multiple records based on a partition of its fields. The third problem of the above script is that Pig Latin’s lack of support for loops makes it difficult to support flexible usage summaries based on arbitrary paths (a feature of ExpLOD) and reasoning-like tasks, such as finding all equivalent instances based on owl:sameAs interlinks. An existing system [120] overcomes this challenge by precomputing a synonyms table based on equivalent instances that are replicated to each cluster node as part of a broadcast join; however, the synonym table’s size may still exceed main-memory. Fourth, despite Hadoop’s support for high-level languages, a challenge that arises is to ensure that such declarative tools generate efficient execution plans. These four challenges highlight a need for a combination of features in order to compute flexible, efficient bisimulations over large graphs. In our experience, our Pig Latin script generated an execution plan that involved several jobs per iteration, thereby decreasing performance due to an increase in the amount of intermediate data serialized to disk. It is for this reason that we sought an alternative multi-core approach.

One of the earliest, parallel versions of the bisimulation algorithm relies on shared memory within a failure-free cluster [19]. This approach leverages an optimization that skips processing of stable blocks; a block is stable if it is not connected a block that was split in the previous iteration. In a parallel framework such as Hadoop, where there is no interprocess communication, a job to construct a bisimulation would need to collect such information at the end of a phase, before supporting such optimization at a task level. Even then, such an approach results in increased cost due to Hadoop’s serialization of intermediate data between phases and jobs. In such scenarios, there is clear benefit to having some form of in-memory or inter-process communication between parallel tasks.

An evaluation of a multi-core MapReduce implementation on synthetic data shows that it is possible to achieve speedups [101]; however, there is no comparison with Hadoop as it does not support multiple cores effectively. Hadoop supports three modes: standalone, pseudo-distributed, and distributed. The standalone mode executes MapReduce jobs in a single Java virtual machine (JVM), possibly with multiple parallel threads that execute Map tasks; parallel Reduce tasks are not supported. The pseudo-distributed mode executes tasks in parallel JVMs, as a way simulate parallel JVMs executing on different machines (distributed mode, which is outside our scope). Anecdotally speaking, we report from our experiences with Hadoop in a multi-core environment that configuration sensitivity for pseudo-distributed mode, serialization cost between jobs and phases, and lack of effective multi-core support (no parallel reduce tasks when using a single JVM) results in a poor environment to compute structural summaries
of large graphs. Often, this is due to the fact that Hadoop expects to launch multiple JVMs, often one for each core on a commodity machine.

One of the advantages of multi-core parallel implementations is that their performance benefits greatly with asynchronous updates while processing large graphs, that is, where an update to a node or edge is immediately visible to other nodes. In contrast, the Bulk Synchronous Parallel (BSP) framework processes large graphs in parallel by sending explicit messages (not unlike the intermediate data serialization). Performance improvement of asynchronous implementations of popular algorithms, including breadth-first-search and connected components and approximate page rank, have been shown [96, 78, 53, 44]. GRACE [126, 123] is a parallel graph processing framework that gives a synchronous environment in which to describe an update method, then allows relaxing synchronization rules to enable asynchronous updates. Ligra [112] is a multi-core graph processing library that assumes that datasets fit in memory.

In this section, we described scalable constructions of structural summaries using Hadoop. In the next section, we describe query optimizations from using structural summaries.

### 2.2 Query optimization using structural summaries

In this section, we describe work that is related to query optimization on graph data using structural summaries. We show that there is a gap in the use of summaries for query optimization in existing unmodified RDF systems.

Graph matching is often understood as an equivalence relation over subgraphs based on isomorphism. An equivalence notion as strict as isomorphism can result in an exponential number of matches. To reduce the number of results returned a ‘weaker’ notion of equivalence can be used, such as simulation or bisimulation. The work of [82] adapts simulation to a given depth and duality. Their work compares simulation with isomorphism and approximate answers. Their work also extends simulation with duality (equivalence along incoming and outgoing edges) and locality (balls of depth k) in order to capture topological structures in the dataset graph that can be lost through simulation, since it is a notion of equivalence that is less strict than bisimulation. After describing a centralized implementation for answering queries based on simulation applied with duality and locality features, they further propose a decentralized algorithm using message-passing.

Structural summaries based on varied notions of equivalence can improve query performance of queries over RDF data. Several systems such as GRIN [118], SAINT-DB [98], RDF-3X [91], and gStore [129] use structural summaries to improve query performance with RDF data. The key similarity of many of the implementations is that systems integrate summaries at a low-level, while our RDF summaries can support existing RDF processors without modification.

One of the earliest works that involves structural summaries of RDF data is GRIN [118], which takes as input an instance graph and constructs a union of forward bisimulations of neighbourhoods reachable within a given distance from ’center’ nodes. Their key contribution is the fast construction of structural indexes for RDF datasets that provide better query performance and require less storage overhead than the default indexes created by popular libraries such as Jena and Sesame.

The SAINT-DB [98] project is interesting because its structural summaries group RDF triples using a join-based scheme. Specifically, SAINT-DB groups triples using label positions as a way to summarize the equality that exists between triples, and a query can then use the structural summary to prune triples
Chapter 2. Related Work

from the result that do not match structural constraints. As well, their work considers depth-k summaries that can summarize structural features of up to length k. The SAINT-DB implementation extends RDF-3X. Their experimental results on small datasets show that navigational queries on synthetic data can improve by over 10x times, while depth-2 summaries can be effective at pruning results.

The gStore [129] system takes a different approach than SAINT-DB to answer SPARQL queries on RDF data. Specifically, gStore stores both the RDF data and its structural summary graphs using adjacency lists, instead of within an existing RDF store. Experimental comparisons with several systems, including RDF-3X and GRIN, report that gStore can provide improved query performance for RDF data containing tens of millions of triples. A unique feature of gStore is their ability to also answer wildcard value-based queries with their structural summaries.

A parameterized structural summary [117] maintains predicates to compute forward bisimulation separate from predicates to compute backward bisimulation. To answer dataset queries, a parametrized summary first returns those triples in the dataset graph which a query will consider, then prunes non-structural parts of the query, then applies the pruned query to the remaining results. Furthermore, query performance of summaries of depth-1 is better than that of summaries of depth-3, improving performance by up to 10x times for structural queries against a small portion of the DBpedia dataset. The Apache Lucene-based implementation shows merit in comparison to querying a dataset using the Sesame and RDF-3X RDF processors.

Summaries group nodes using bisimulation as the notion of equivalence, thereby allowing a user to examine (possibly) much fewer number of groups; however, there remains a challenge of exploring and understanding large datasets if there are tens of millions of instance nodes and few instances are bisimilar. Summaries that allow some error can potentially increase the number of instance nodes that are considered equivalent, thereby reducing the number of blocks in a structural summary (such as keeping two non-equivalent nodes in the same block). Summaries with error have also been described [23] as having a block edge in a summary that represents an edge that does not exist in its instance graph, or alternatively a missing block edge in a summary that does not accurately capture an edge that does exist in its instance graph. Precision and recall can be used to compare the error rates of different summaries for different queries, such as the accuracy of a summary with error for determining the set of classes an instance belongs to.

A summary constructed using SNAP [116] constructs a relational coarsest partition without error. The operation takes an initial partition and iteratively refines it by examining each block, and using a radix sort amongst the nodes in order to split them into smaller blocks. A k-SNAP operation adds error to the SNAP operation by using k as the number of blocks to have in the final partition, mimicking OLAP’s ‘roll-up’ and ‘drill-down’ abilities. To address the NP-completeness of k-SNAP, two heuristic methods are proposed to approximate k-SNAP results, a splitting approach and a combining approach. The k-SNAP splitting approach takes an initial partition and iteratively refines it until there are k blocks in the partition. In contrast, the k-SNAP combining approach first creates the most refined partition possible (such as placing each node in its own block) and iteratively combines blocks until the there are k blocks. Since the combining approach has a higher complexity at low k in a large graph due to the potential need to combine many blocks, the splitting approach is often more amenable for large graphs.

Another way of incorporating error into structural summaries [91] constructs a summary based on an information-theoretic measure, the minimum description length (MDL). The MDL approach identifies a good summary as a minimal one that encodes data maximally. The size of the summary is based on
the number of edges between blocks of the partition, where a block edge represents a complete bipartite graph between the nodes of the connected blocks. Since the nodes in blocks may not represent the actual instance data, adjustments to block edges are captured by annotating a block edge with correction edges. A correction edge is either an edge between nodes that exists in the instance data but not as a block edge (as in the case when a block edge does not exist between two blocks), or an edge between nodes that does not exist in the instance data but where there is an edge between the blocks of the nodes (thereby correcting the node edges implied by the block edge). The partition’s blocks and block edges, together with the correction edges, then recreate the instance data exactly, and a summary’s MDL cost is the sum of blocks, block edges, and correction edges. Their mechanism iteratively combines blocks in a way that tries to reduce a summary’s MDL cost. Their work has two implementations for creating a summary. Their GREEDY algorithm is based on a max-heap and their RANDOMIZED algorithm avoids using the max-heap and instead examines the blocks containing the neighbours of a randomly selected node, and chooses the best block with which to combine with the node’s block.

Errors are introduced into the MDL approach by reducing the number of block edges and edge corrections. Specifically, an error bound is used to specify the fraction of a node’s neighbours in a summary that can differ from its neighbours in its instance graph. Two algorithms for computing summaries with error are APXMDL and APXGREEDY. APXMDL takes a summary without errors and first removes edge corrections by applying a matching problem that selects edge corrections to remove, and then applies a greedy approach to find block edges to remove. The second algorithm, APXGREEDY, computes a summary with error directly from the instance data instead of starting from a summary without error. In the first of its two steps, APXGREEDY proceeds as in GREEDY to produce a summary without error, but records the costs associated with the specified error. These costs are then be used by the second step which feeds the summary to the APXMDL algorithm to choose the blocks to combine and generate the final summary. The intuition behind this approach is that, when considering the error bound during GREEDY, block edges cannot be deleted without affecting the cost of their neighbours so these block edges are instead considered for deletion during APXMDL.

In this section, we described related literature for querying RDF data using structural summaries. In the next section, we describe representations for structural summaries.

2.3 Summary representations

In this section, we describe literature related to ontologies of structural summaries.

The 'Vocabulary of Interlinked Datasets' (VoID) RDFS schema is an ontology for a publisher to give light-weight metadata about an RDF dataset, such as ontologies, classes, and predicates in a dataset’s descriptions of real-world entities. SchemeX is a system that uses a VoID-based summary to enable a user to understand and query the semi-structure of streaming data. SchemeX summarizes batches of streaming RDF data to construct FW summaries based on the classes and predicates that nodes are directly connected to. To understand portions of a streaming RDF dataset, a user can then query a batch’s SchemeX summary using an unmodified SPARQL processor.

Ontology-based data access (OBDA) is an approach to explore and query data by automatically rewriting user queries so that they support ontology based reasoning. OBDA systems often rely on description logics, such as the popular OWL ontology, to add extensional and reasoning applications on databases that do not support it by default. The key idea is to a rewrite a query to a more complex
set of queries that includes inferred answers that arise from following logical rules, such as transitive relationships. Queries are rewritten using an ontology that describes the meaning of the data as well as how to formulate queries against the data. In addition to query rewriting, OBDA often involves result translation \cite{45}, or alternatively, mappings between ontologies that integrate datasets \cite{33}.

A query that undergoes OBDA rewriting is often a union of conjunctive queries such that they return the answer to the original query along with answers that result from reasoning and inferencing. Query rewriting faces a challenge with the potentially many rewritings, for example, a query rewrite may include a query that returns an empty answer, or a query that encompasses another. Optimization of query rewriting \cite{45,46} often targets performance by examining query translation time as well as the number of conjunctive queries that a database system executes. Some OBDA systems have found it a challenge to ensure that rewrite rules produce correct ontological inferences; a recent verification of OBDA systems shows that their implementations require improvement \cite{61}. A benchmark of ontology-based query rewriting systems that measures their performance \cite{85} also considers expressiveness of ontologies, types and limitations of logics that an ontology supports, complexity of rewritten queries, and support for complex structural queries on the instance data.

Our use of RDF summaries to answer queries is a form of OBDA access since we rewrite dataset queries into summary-optimized queries as a way to improve query performance on large graphs \cite{30}. Our summary ontology is flexible since it allows describing a structural summary using the same predicates that an RDF dataset actually uses. Our work differentiates itself clearly from existing work in that we use our summary ontology to answer queries on the instance while improving query performance. Our performance results show that rewriting queries to use structural summaries is a light-weight and feasible way to answer queries faster than querying instance data.

In this section, we described related work of summary ontologies and ontology-based query rewriting systems. In this chapter, we described work related to constructing summaries using parallelism, querying RDF data using Hadoop and structural summaries, and representations of structural summaries. In the next chapter, we describe how to construct summaries of large RDF datasets.
Chapter 3

Summary Construction

In our work to improve query performance on large graphs, we propose using bisimulation summaries as a graph-based index. However, the challenge is that there needs to be an effective way to construct summaries in a scalable manner. We address this challenge by targeting a construction time that is within a factor of time to load input plus write output. This also means that effective optimization techniques should be employed in order to achieve our goal.

For scalable construction of summaries, there exists an iterative, hash-based algorithm \[19\] that extends a simple algorithm \[65\] to iteratively update each node’s block identifier using the block identifier of neighbours in parallel. Existing scalable implementations \[19, 81, 108\] rely on distributed frameworks like MapReduce \[36\] and message-passing to compute summaries of large graphs. There is literature that points to multi-core systems with large amounts of main-memory \[12, 75\] as a viable parallel processing architecture; however, there are no summary construction implementations for multi-core systems in the literature. Furthermore, existing implementations use optimizations that tend to have equal per-iteration time. Per-iteration optimizations focus on different aspects of the construction and include reducing the number of blocks processed but that still requires processing all edges \[19\], reducing the number of Hadoop jobs \[108\] per iteration, or balancing skew of edges processed amongst parallel tasks \[79, 81\] in an iteration.

In this work, we give a vertex-centric summary construction algorithm, implemented in the GraphChi \[77\] multi-core graph processing framework, to construct FWBW bisimulation summaries \[37, 94\] of large graphs with the goal of computing a summary in an amount of time similar to the time required to load the input dataset plus write the summary. In order to achieve our goal, we propose a novel and very effective singleton optimization that drastically reduces per-iteration times after only a few iterations by processing fewer edges in each iteration.

The outline of this chapter is as follows. In Section 3.1 we give Hadoop-based implementations to construct FWBW summaries that extend the work of \[108\], a Hadoop-based FW summary construction. To our knowledge, our Hadoop-based FWBW summary constructions are the first to appear in literature. We leverage the declarative Pig Latin language and give a Pig Latin script that translates an RDF dataset into a dataset graph. We then give three Hadoop summary constructions, each of which varies in their per-iteration properties, such as the amount of intermediate data or number of jobs. Specifically, we describe a p/-p variation that has lower amounts of intermediate data, but requires more jobs, than a p&-p variation, and a p&-p/reset variation.
variation that processes less intermediate data with fewer jobs than the p&-p variation.

In Section 3.2, we give GraphChi implementations to construct summaries, including a novel singleton optimization that can reduce the number of edges processed in each iteration. We start by describing our translation from an RDF dataset into GraphChi’s internal format. We then give a GraphChi B variation that computes structural summaries without the singleton optimization. Then we describe our GraphChi S variation that uses includes the singleton optimization. We then describe how our singleton optimization is novel and unique in comparison to the existing notion of stable optimization.

In Section 3.3, we evaluate the performance of our GraphChi variations and compare them with Hadoop-based implementations described in existing literature. We first give an experimental validation that our GraphChi S variation is faster than the B variation. We also validate that our S variation achieves our goal of constructing summaries of large graphs in an amount of time similar to the time required to load the dataset plus write the summary. In our experiments, we use graphs ranging from millions to billions of edges. Our comparison to existing literature involves comparing our FW summary construction using our S variation to an implementation of FW summary construction \[108\]. We give evidence that summary construction on multi-core architecture is effective with our GraphChi S variation. We do this by constructing FW summaries (which are supported by the literature) using different Hadoop execution modes that are appropriate for multi-core execution. We summarize our chapter contributions in Section 3.4.

### 3.1 Hadoop Implementations

In this section, we describe FWBW summary construction algorithms based on the Hadoop \[52\] implementation of MapReduce \[36\] that extends the work of \[108\], a Hadoop-based FW summary construction. We first describe the Hadoop processing model. Then in Section 3.1.1, we give a translation of RDF data into a dataset graph for Hadoop to process.

Hadoop processes a dataset using one or more jobs. Each job comprises a sequence of three phases: a map phase, a shuffle phase, and a reduce phase. Each phase executes one or more tasks in parallel, where each task operates on key-value pairs, also called records, that can contain any type of data; records can have an annotation in order to distinguish their type. The map phase is a function

\[
\text{map}(k_1, v_1) \rightarrow (k_2, v_2)^* 
\]

that transforms each input record into zero or more output records. At the end of the map phase, the shuffle phase groups its output into lists of records using key equality. Alternatively, a Hadoop job can specify the shuffle phase as a grouping comparator that groups records by a portion of their key, and a secondary sort that orders each list of grouped records based on a custom key sort order. The reduce phase is a function

\[
\text{reduce}(k_3, v_3^*) \rightarrow (k_3, k_4)^* 
\]

that executes a reduce task for each key to process its list of grouped records, and outputs zero or more records which a subsequent job can then process. Hadoop’s ability to execute parallel tasks in each phase makes it well-suited to process large datasets in a structured, batch-oriented way.

In this section, we described how Hadoop’s processing model, consisting of a sequence of phases that execute parallel tasks over key-value pairs, is able to process large datasets. In the following sections, we describe how to construct a dataset graph from an input RDF dataset, how to construct a structural summary of a dataset graph, and how to output a summary as RDF. In the next section, we describe
how Hadoop can construct a dataset graph from input RDF data.

### 3.1.1 Dataset graph translation

In this section, we describe how to translate an input RDF dataset into a dataset graph using Hadoop.

**Procedure:** Dataset graph construction

**Input:** $input(s, p, o)$, RDF input triples

1. - read input
2. $stmts = FOREACH input GENERATE slabel, plabel, olabel;$
3. 
4. - handle predicate label id mapping
5. $puris = FOREACH stmts GENERATE plabel;$
6. $puris = DISTINCT puris;$
7. $puris = RANK puris;$
8. 
9. - handle subject and object label id mapping
10. $suris = FOREACH stmts GENERATE slabel AS nlabel;$
11. $ouris = FOREACH stmts GENERATE olabel AS nlabel;$
12. $nodes = UNION suris, ouris;$
13. $nodes = DISTINCT nodes;$
14. $nodes = RANK nodes;$
15. 
16. - handle edges
17. $edgesTemp1 = JOIN stmts BY slabel, nodes BY nlabel;$
18. $edgesTemp1 = FOREACH edgesTemp1 GENERATE id AS sid, plabel, olabel;$
19. $edgesTemp2 = JOIN edgesTemp1 BY olabel, nodes BY nlabel;$
20. $edgesTemp2 = FOREACH edgesTemp2 GENERATE sid, plabel, id AS oid;$
21. $edgesTemp3 = JOIN edgesTemp2 BY plabel, puris BY plabel;$
22. $edges = FOREACH edgesTemp3 GENERATE sid, id AS pid, oid;$

**Figure 3.1: Pig Latin dataset graph construction**

In Figure 3.1 we give a Pig Latin script that generates an integer-mapped dataset graph; we use integers as they are more compact to store and process than string labels. The script starts by representing each triple of an input RDF dataset as a record with three fields, $slabel$, $plabel$, and $olabel$, as part of the $stmts$ relation. From the $stmts$ relation, the script generates two relations, $puris$ containing distinct predicate labels, each with a distinct id, and $nodes$ containing distinct subject and object labels, each with a distinct id; both relations use the Pig built-in $RANK$ function that adds a field $id$ with a distinct identifier value to each record in a relation. To replace each label of the input RDF dataset by the label’s corresponding integer id, also referred to as $normalization$ in literature, the script joins each part of the input triple relation with the corresponding $puris$ or $nodes$ relation, and generates a new output relation $edges$ containing the dataset graph’s edges. The output of the Pig Latin script is a set of triples with integer identifiers for each of the subject, predicate, and object labels.

GraphBuilder [63] is a toolkit that provides Hadoop jobs written in the Java programming language to extract, transform and load datasets into a given representation, and as such can translate the labels of an input graph using an integer mapping. In our work, we use Pig Latin since it executes as a sequence of Hadoop jobs and is simpler to express, more extensible, and can undergo optimization.
In this section, we described a Pig Latin script that translates an input RDF dataset into an integer-mapped dataset graph. In the next section, we describe a Hadoop implementation that constructs a structural summary of a dataset graph.

### 3.1.2 p/-p variation

In this section, we describe a p/-p variation that computes the structural summary of a dataset graph using Hadoop.

**Procedure:** Bisimulation using p/-p, initialization map-only job

**Initialization input:**
- **key:** byte offset in input file, can ignore
- **value:** an edge $(v, p, v')$ with label $p = m(v, v')$

1. $(v, p, v') \leftarrow \text{parseEdge}(value)$
2. emit ('signature', $(v, p), (\text{block}_0(v'), v')$)
3. emit ('template', $(v', -p), (v)$)

![Figure 3.2: Bisimulation using p/-p, initialization map phase](image)

We give a Hadoop p/-p summary construction algorithm that iteratively executes three MapReduce jobs in sequence, two update jobs and a count job. In each iteration $k$, the first update job computes the forward $(k+1)$-block identifier for each node, the second update job computes the $(k+1)$-FWBW block identifier for each node, and the third job counts the number of distinct block identifiers. The algorithm terminates when the number of distinct blocks remains the same in two consecutive iterations. Next, we describe the Hadoop jobs that execute in each iteration.

Aside from an initialization map, we express the map task of both update jobs as an identity function that emits input records without modification. The algorithm initializes records to process in an initialization map phase, the first map phase of the first update job when the algorithm begins, and is the identify function for all the following map phases. Figure 3.2 shows the algorithm for the initialization map phase. An initialization map task reads a dataset graph edge $(v, p, v')$ as an edge with label $p$ from a node $v$ to a node $v'$, and emits two records, a signature record and a template record. A signature record represents the outgoing edge of a node $v$ as follows: a key that is a pair of the node identifier of $v$ and its outgoing edge label $p$, and a value that is a pair of the target node $v'$ and its initial block identifier. Initially, the summary is a partition with a single block, so every call to $\text{block}_0(v')$ returns that single block’s identifier. A template record that an initialization task emits represents the incoming edge of $v'$ as follows: a key that is a pair of the node identifier of $v$ and its outgoing edge label $p$, and a value that is a pair of the target node $v'$ and its initial block identifier. Initially, the summary is a partition with a single block, so every call to $\text{block}_0(v')$ returns that single block’s identifier. A template record that an initialization task emits represents the incoming edge of $v'$ as follows: a key composed of $v'$ and its incoming edge label marked as $-p$, and a value composed of only $v$ and its initial block identifier; in the implementation, predicate identifiers start from 1 to allow negation. The p/-p variation processes double the number of edges since it represents both incoming and outgoing edges.

We now describe the shuffle phase of the update jobs. The shuffle phase has a grouping comparator that groups signature records and template records by the node identifier of their key, thus grouping each node’s incoming and outgoing edges. The shuffle phase has a secondary sort that orders each list of grouped records such that signature records appear before template records. As well, if there exists a key-value pair from the node stream, that consists of the node’s block identifier (or assume a default block identifier if such a record does not exist), it appears first in the list; the map initialization phase...
does not emit a node record since it cannot generate a single node record for each node. Such a grouping comparator and secondary sort gives an order, for each node, consisting of its node record, its signature records, and lastly, its template records.

Next, we describe the reduce phases of the two update jobs.

Algorithm 3.3 shows the reduce phases of the two update jobs that execute in each iteration. At a high level, the first update job computes each node’s (k+1)-forward block identifier, and the second update job computes each node’s (k+1)-FWBW block identifier. In the first update job, a reduce task processes the list of records of a node \( v \), starting with a node record that captures its current (k)-FWBW block identifier, assuming a default block identifier if the record is not present (e.g. right after initialization).

Next, a reduce task computes the (k+1)-forward signature of \( v \) by iterating over its signature, each of which represents an outgoing edge to a node \( v' \) and its (k)-FWBW block identifier. For each signature record, a reduce task emits a template record that represents an incoming edge from \( v' \), with a key \( v' \) and the negated predicate id, and \( v \) as the record’s value. A reduce task then computes the (k+1)-forward block identifier of \( v \) as a hash value that incorporates its (k)-FWBW block identifier and its (k+1)-forward signature; sorting gives a canonical value as input to the hash function regardless of the
order in which signature records appear in the input list. A reduce task then emits the (k+1)-forward block identifier as a node record with \( v \) as the key and its block identifier as the value. A reduce task then generates a signature record for each template record, each of which represents an incoming edge of \( v \) (with negative edge label) from \( v'' \); the signature record expresses an outgoing edge from \( v'' \) to \( v \) as a record with a key that is a pair \( v'' \) and a positive edge label (the negation of a negative label), and a pair \( v \) with its (k+1)-forward block identifier as the record’s value.

The second job of the reduce phase of Figure 3.3 proceeds similarly to the first update job except that the job uses each node’s (k+1)-forward block identifier to compute a node’s (k+1)-FWBW block identifier. The second job emits signature records as outgoing edges that embed the (k+1)-FWBW block identifier of target nodes, and template records as incoming edges. The count job counts the number of distinct block identifiers in the node stream (its code is not shown).

To generate an RDF summary from an iteration’s output node stream, we normalize the edges of the input dataset graph using the block identifiers that the summary construction algorithm emits in the node stream. We use a normalization approach similar to the one shown in Figure 3.1. We re-use this approach to output RDF summaries in all our Hadoop variations.

In this section, we described the p/-p variation to construct structural summaries of a dataset graph using Hadoop. The p/-p variation uses two update jobs in each iteration to compute each node’s block identifier. In the next section, we describe a p&-p variation that computes a structural summary using a single update job in each iteration.

### 3.1.3 p&-p variation

In this section, we describe the p&-p variation of a Hadoop summary construction algorithm.

#### Procedure: Bisimulation using p&-p, initialization map-only job

**Initialization input:**
- **key:** byte offset in input file (can ignore),
- **value:** an edge \((v, p, v')\) with label \(p = m(v, v')\)

1. \((v, p, v') \leftarrow \text{parseEdge(value)}\)
2. emit('signature', \((v, p), (\text{block}_0(v'), v')\))
3. emit('signature', \((v', -p), (\text{block}_0(v), v)\))
4. emit('template', \((v, p), (v')\))
5. emit('template', \((v', -p), (v)\))

Figure 3.4: Bisimulation using p&-p, initialization map-only job

The p&-p summary construction algorithm iteratively executes one update job followed by a count job. The main difference of this variation is that, instead of computing forward block identifiers in the first update job followed by FWBW block identifiers in a second update job in the p/-p variation, the p&-p variation computes a FWBW block identifier for each node using one update job. To do so, a p&-p update job processes signature records that represent a node’s incoming and outgoing edges to compute its FWBW block identifier, and emits a node’s incoming and outgoing edges as template records.

In the p&-p variation, the map phase of the update job is an identity function that receives and emits records without modification. Figure 3.4 shows the initialization map task that takes place of the first map phase of the iterative sequence; the first map phase initializes records that represent each node’s
Procedure: Bisimulation using p&-p, reduce phase of update job

Input: list of signature and template records of a node $v$

1. $\text{sig}_{k+1}(v) \leftarrow \{\}$
2. while $value \in \text{signature}$ do
3.   $(\text{block}_{k}(v'), v') \leftarrow \text{parseValue}(value)$
4.   $\text{sig}_{k+1}(v) \leftarrow \text{sig}_{k+1}(v) \cup \{(p, \text{block}_{k}(v'))\}$
5.   emit(‘template’, $(v, p)$, $(v')$)
6. end
7. $\text{block}_{k+1}(v) \leftarrow \text{hash} \left(\text{sort}(\text{sig}_{k+1}(v))\right)$
8. emit(‘node’, $v$, $\text{block}_{k+1}(v)$)
9. while $value \in \text{template}$ do
10.  $(v'') \leftarrow \text{parseValue}(value)$
11.  emit(‘signature’, $(v'', -p)$, $(\text{block}_{k+1}(v), v)$)
12. end

Figure 3.5: Bisimulation using p&-p, update jobs

incoming and outgoing edges. The initialization translates each edge $(v, p, v')$ in the dataset graph into 4 records; there is a pair of signature and template records for the outgoing edge of $v$ and the incoming edge of $v'$. One signature record encodes a node’s outgoing edge with the initial block identifier of the target node. The outgoing signature’s key is the source node identifier $v$ with a positive edge label $p$, and its value is the target node identifier $v'$ and its initial block identifier $\text{block}_0(v')$. The other signature record encodes a node’s incoming edge with the initial block identifier of the source node. The incoming signature’s key is the target node identifier $v'$ with a positive edge label $p$, and the value is the target node identifier $v$ and the negation of its initial block identifier $-\text{block}_0(v)$; the negation helps to distinguish outgoing edge signature records from incoming edge signature record. The two template key-value pairs also represent incoming and outgoing edges into which an update job will embed block identifiers that the next iteration’s update job will process.

The shuffle phase of the p&-p variation is a grouping comparator that groups key-value pairs by the node identifier of each record’s key, and a secondary sort that orders signature records ahead of template records. Figure 3.5 shows the reduce phase of the update job for the p&-p variation. A reduce task computes the $(k+1)$-FWBW block identifier of a node as a hash of its sorted signature records which represent the $(k)$-FWBW block of connected nodes. A reduce task emits a template record for each signature record, which has the same key and only the connected node as the value. The hash computation of a node’s $(k+1)$-FWBW block identifier does not incorporate a node’s $(k)$-FWBW block identifier since both incoming and outgoing edges are present as signature records. After computing a node’s FWBW block identifier, a reduce task emits a node’s template records as signature records by embedding the node’s $(k+1)$-FWBW block identifier. After the update job completes, a count job counts the number of distinct block identifiers, and the algorithm terminates if two consecutive iterations have equal number of distinct block identifiers.

In this section, we described the p&-p variation of the Hadoop summary construction algorithm, which uses a single update job. In the next section, we describe the p&-p/reset variation of the summary construction algorithm.
3.1.4 p&-p/reset variation

In this section, we describe the p&-p/reset Hadoop implementation that computes a structural summary without using template records.

**Procedure:** Bisimulation using p&-p/reset, initialization map-only job

**Initialization input:**
- **key:** byte offset in input file, can ignore
- **value:** an edge \((v, p, v')\) with label \(p = m(v, v')\)

1. \((v, p, v') \leftarrow \text{parseEdge}(\text{value})\)
2. \((v, p) \leftarrow \text{emit('signature',}(v, p), (\text{block0}(v'), v'))\)
3. \((v', -p) \leftarrow \text{emit('signature',}(v', -p), (\text{block0}(v), v))\)

**Procedure:** Bisimulation using p&-p/reset, reduce phase of update job

**Input:** list of a node’s edges

1. \text{markStream}()
2. \(\text{sig}_{k+1}(v) \leftarrow \{\}\)
3. while \(\text{value} \in \text{signature}\) do
4. \((\text{block}_k(v'), v') \leftarrow \text{parseValue}(\text{value})\)
5. \(\text{sig}_{k+1}(v) \leftarrow \text{sig}_{k+1}(v) \cup \{(p, \text{block}_k(v'))\}\)
6. end
7. \(\text{block}_{k+1}(v) \leftarrow \text{hash}(\text{sort}(\text{sig}_{k+1}(v)))\)
8. \((v, \text{block}_{k+1}(v)) \leftarrow \text{emit('node',} v, \text{block}_{k+1}(v))\)
9. \text{resetStream}()
10. while \(\text{value} \in \text{signature}\) do
11. \((\text{block}_k(v'), v') \leftarrow \text{parseValue}(\text{value})\)
12. \((v', -p) \leftarrow \text{emit('signature',}(v', -p), (\text{block}_k(v), v))\)
13. end

Like the p&-p variation, the p&-p/reset variation of the summary construction algorithm consists of the iterative execution of an update job and a count job. We give details of the jobs now.

We show the initialization map phase in Figure 3.6. Like in the p&-p variation, an initialization map phase emits signature records for each node that represents its incoming and outgoing edges with an initial FWBW block identifier. This variation does not emit any template records; we show later how the signature records themselves serve as template records. Aside from the initialization map phase, the map phase of the update job is an identity function that receives records from any stream and emits them to the same stream without modification. The shuffle phase has a grouping comparator that groups records by the node identifier of their key, and does not use a secondary sort order.

Figure 3.7 shows the reduce phase of the p&-p/reset variation’s update job. Unique to this variation, before a reduce task processes a node’s signature records, a reduce task first marks the beginning of the stream which will allow the reduce task to re-iterate over the stream by resetting the stream iterator to the marked location. A reduce task computes a node’s \((k+1)\)-FWBW block identifier as the hash of its ordered signature, then resets the stream back to the start marker. The reduce task then iterates over
the node’s signature records, using them as template records to write new signature records that contain
an updated block. After the update job completes, a job counts the number of distinct block identifiers in
the node stream and the algorithm terminates when the count is the same in two consecutive iterations.

In this section, we described the p&-p variation of the summary construction algorithm. Next, we
discuss the Hadoop variations that we describe in this section.

3.1.5 Discussion

Notice that the p/-p variation has 2 copies of each dataset graph edge and a node record for each node,
requiring \( O(2 |E| + |V|) \) space but 2 update jobs in sequence. In contrast, the p&-p variation requires
\( O(4 |E| + |V|) \) space to compute a node’s FWBW bisimulation block identifier in a single update job,
which has the advantage of increasing computation time within tasks by using fewer update jobs, and
thus reducing costs such as job setup and serialization of intermediate data between the map and reduce
phases. An alternative, the p&-p/reset variation, leverages executing only 2 jobs per iteration and also
using a resettable-iterator in order to iterate over a single copy of incoming and outgoing edges, which
uses \( O(2 |E| + |V|) \) space.

In this section, we described Hadoop’s processing model, a Pig Latin script to translate RDF input
data into a dataset graph, followed by our p/-p, p&-p, and p&-p/reset Hadoop-based FWBW summary
construction algorithms. To our knowledge, these are the first Hadoop-based FWBW summary
constructions. In the next section, we describe our GraphChi summary construction approach.

3.2 GraphChi summary construction

In this section, we describe our GraphChi summary construction that achieves our goal of con-
structing summaries in around the same amount of time as it takes to load the instance dataset and
write the summary. In Section 3.2.1, we describe how we translate an RDF dataset into a dataset graph
represented in GraphChi’s internal format. In Section 3.2.2, we describe how to construct summaries
using GraphChi and present a novel and very effective singleton optimization that allows us to achieve
the goal of drastically reducing per-iteration times after only a few iterations.

3.2.1 Dataset graph translation

In this section, we describe how to translate an input RDF dataset into a dataset graph represented in
GraphChi’s internal format.

A translation of an input RDF dataset into an integer-mapped dataset graph proceeds as follows. For
each triple in an RDF dataset, we integer-map distinct subject and object labels to a node identifier using
a hash map in main-memory (with optional backing on secondary-storage), and integer-map distinct
predicate labels to a second hash map in main-memory; since a dataset often contains a small number of
distinct predicates in comparison to subjects and objects, we keep the predicate map in main-memory
without any secondary-storage backing. While reading an input RDF dataset, we add nodes, that
contain their node identifier, and edges, that contain their predicate identifier, to GraphChi’s internal
graph. As well, for each integer-mapping, we construct an inverse mapping from integers to labels so
that, after construction, we can write structural summaries as RDF.
In this section, we described a way to translate input RDF into GraphChi’s internal format with support for writing summaries. In the next section, we describe our GraphChi summary construction.

3.2.2 Summary construction algorithm

In this section, we describe the GraphChi processing model, and a GraphChi implementation of our summary construction.

GraphChi is a multi-core processing framework that supports the Bulk Synchronous Parallel (BSP) \cite{121} processing model, an iterative, node-centric processing model by which nodes in the current iteration execute an \textit{Update} function in parallel that depends on values from the previous iteration. Targeting scalability on ‘just a laptop’, GraphChi partitions graph that do not fit in main memory in a way that makes it efficient to load and process subgraphs that fit in main memory. GraphChi uses \textit{parallel sliding windows} (PSW), a subgraph partitioning and loading mechanism that involves both main-memory and secondary storage. GraphChi partitions a graph’s nodes into equal \textit{intervals}, storing each interval’s incoming edges in a \textit{shard} file in sorted order by the source node. In each iteration, the PSW mechanism processes each interval in turn. For each interval, PSW first loads incoming edges from disk into main-memory. Then, relying on the fact that all shards sort the edges they contain by each edge’s source, the PSW streams the current interval’s outgoing edges from disk. GraphChi processes an interval by executing the \textit{Update} method of each node in the interval using parallelism.

We base our summary construction algorithm on the parallel, hash-based approach of \cite{19}. The hash-based approach is an iterative computation that updates each node’s block identifier. The block identifier is a hash value that results from the node’s signature, the set of block identifiers from the previous iteration to which the node’s neighbours belong. The main idea is that bisimilar nodes will have the exact same signature (and the same hash value) and thus have the same block identifier. As well, two nodes that are not bisimilar will not have hash values that collide and will have different block identifiers. A parallel variant of this notion enables executing multiple node update methods at the same time since new block identifiers are computed only using values from the previous iteration.

The hash-based summary construction can be implemented by the iterative Bulk Synchronous Parallel (BSP) processing model \cite{121}, an iterative, node-centric processing model by which nodes in the current iteration execute an \textit{Update(·)} function in parallel that depends on values from the previous iteration. The BSP model communicates a node’s new value to its neighbours via message-passing but GraphChi is more general in that it does not require a user to implement the BSP model, and so does not provide a message-passing mechanism by default. GraphChi suggests a BSP implementation that stores each node’s previous and current block identifier within its edges. This is because the PSW mechanism only loads adjacent edges during execution of a node’s \textit{Update(·)} method, and means that a node can read and write its own properties, the properties of connected edges, and shared-memory data structures, but cannot read or write the properties of neighbours. Thus, to store and transmit each node’s block identifier to its neighbours requires keeping each node’s block identifier within its connected edges. So, GraphChi’s internal graph representation is such that each node stores its label integer identifier, and each edge stores its predicate label integer identifier and 4 byte arrays. One pair of byte arrays represents the block identifiers of the edge’s source and target for iteration \( k \), and the second pair of byte arrays represents the block identifiers of the edge’s source and target for iteration \( k + 1 \). Initially, when an algorithm begins, all 4 values have the same block identifier. In each iteration, a node obtains a signature by reading block identifiers from one pair of source and target block identifiers, which represents values
Chapter 3. Summary Construction

Table 3.8: B variation algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1.</td>
<td>foreach $v \in V_G$ do $block_0(v) = '0'$ end</td>
</tr>
<tr>
<td>L2.</td>
<td>$count_{old} \leftarrow 1, k \leftarrow 0, count_{new} \leftarrow 0$</td>
</tr>
<tr>
<td>L3.</td>
<td>while $count_{old} \neq count_{new}$ do</td>
</tr>
<tr>
<td>L4.</td>
<td>parallel foreach $v \in V_G$ do $V$-Update($v$) end</td>
</tr>
<tr>
<td>L5.</td>
<td>$count_{old} \leftarrow count_{new}, k \leftarrow k + 1$</td>
</tr>
<tr>
<td>L6.</td>
<td>$count_{new} \leftarrow {</td>
</tr>
<tr>
<td>L7.</td>
<td>end</td>
</tr>
<tr>
<td>L8.</td>
<td>$V_S \leftarrow {block_k(v)</td>
</tr>
<tr>
<td>Method: $V$-Update($v$)</td>
<td></td>
</tr>
<tr>
<td>V1.</td>
<td>$fwsig(v) = {(+m(v, v'), block_k(v'))</td>
</tr>
<tr>
<td>V2.</td>
<td>$bwsig(v) = {(-m(v', v), block_k(v'))</td>
</tr>
<tr>
<td>V3.</td>
<td>$block_{k+1}(v) = \text{hash(sort(block_k(v) \cup fwsig(v) \cup bwsig(v)))}$</td>
</tr>
</tbody>
</table>

Figure 3.8: B variation algorithm

from the previous iteration, and then writing updated block identifiers to the counterpart pair, which represents the values of the current iteration. Since block identifiers are duplicated between edges, it increases serialization cost. Anecdotally, although the performance of this BSP implementation is quite good, we choose to implement the model slightly differently.

We implement two variations: variation B avoids storing block identifiers as edge properties, and variation S which provides an optimization to improve performance. We now describe the two variations.

The algorithm in Figure 3.8 gives our B variation program and its corresponding node update method, $V$-Update. The GraphChi program takes a dataset graph as input and initializes its summary as a single block containing all nodes (L1). In each iteration $k$, GraphChi executes $V$-Update on each dataset graph node in parallel (L4); we use the keyword parallel foreach to represent how GraphChi processes all of a graph’s nodes using parallelism. Our algorithm keeps each node’s current and previous iteration block identifiers in two hash maps A node’s update function takes its forward signature from its outgoing edges (V1) and its backward signature from its incoming edges (V2) to compute the (k+1)-FWBW block identifier (V3). If we exclude line V2 then we compute a FW summary. The algorithm terminates when two consecutive iterations have the same number of distinct block identifiers (L3,L5,L6). The algorithm writes the summary to disk in parallel (L8), skipping duplicate block edges.

A node’s (k+1)-FWBW block identifier is the result of a hash function that takes as input the forward and backward signatures A sort function lexicographically sorts the set of signature entries, and is used as a way to provide a canonical input to the hash function. In addition to using canonical input values, we assume that a hash value generated from a signature does not collide with a different signature, thus block identifiers generated from a hash value can be used as a canonical block identifier. Although it is not essential, we choose to reuse a node’s block identifier as part of its hash computation as a way to maintain a node’s ’history’. In this section, we described our vertex-centric approach to constructing a FWBW summary using multi-core parallelism. In the next section, we describe a novel and effective singleton optimization.

3.2.3 Singleton optimization

In this section, we describe variation S, our optimized summary construction.
Since it is common in graph algorithms to skip computations, GraphChi provides a scheduler that maps each node to a boolean value which decides whether to disable processing of specific nodes in each iteration. Using the scheduler has a benefit that GraphChi skips loading intervals of disabled nodes and their edges from disk, thereby improving performance by reducing disk access.

As stated earlier, a singleton is a block whose extent contains exactly one node, i.e., a node that is not bisimilar to any other node. Notice that summary partitions have the property such that a node that is in a singleton block is never bisimilar with any other node. This means that a singleton node’s block identifier will always remain distinct from all other nodes, and that the summary construction algorithm no longer needs to update the node’s block identifier. Our S variation integrates an algorithmic optimization into variation B that reduces the number of nodes the algorithm processes in each iteration by skipping nodes that are in the extent of a singleton block. In Figure 3.9, we give the S variation algorithm, which uses the program of variation B (Figure 3.9) but having instead the modified line L4′ that executes the custom node update function V-Update-Singleton. The algorithm keeps a hash map with the count of nodes in each block’s extent in the previous iteration. Each node executes the V-Update-Singleton method and uses the hash map to check whether the node is in a singleton’s extent (S1). If the node is not a singleton, then the method computes an updated block identifier for the node using the same V-Update function from the B variation (Figure 3.9). If a node is a singleton, then the node is not processed in the current iteration, and the keyword disable (S4) sets its boolean flag in the scheduler to false to skip the node in future iterations.

The correctness of our S variation follows from the correctness of the hash-based approach [19], which we show below and that says that, assuming that hash values for non-bisimilar nodes do not collide, two nodes that have different signatures will have different block identifiers. In our approach, since a singleton’s identifier will always have a distinct block identifier (no other node can be bisimilar), then we do not have to update the block identifier of the node in its extent and can disable it in the scheduler. We show this in Lemma 3.1 which captures the notion that once a block is a singleton, meaning that its block signature is distinct from all others, no other block can have the same identifier. Our implementation’s reuse of a node’s current block identifier as part of its signature ensures this further.

**Lemma 1** For \( n \geq 0 \), denote \( \text{block}_n(v) \), \( \text{sig}_n(v) \), the block and sig functions as they are after execution of L4 that executes the custom node update function V-Update-Singleton of the nth iteration of Figure 3.9 (first iteration has index 0). Then for every \( n > 0 \) and for every \( v, v' \in V_G \):

\[
\text{sig}_{n-1}(v) \neq \text{sig}_{n-1}(v') \implies \text{sig}_n(v) \neq \text{sig}_n(v')
\] (3.1)
Proof Let us first recall that hash(sort(sig_n(v))) takes care that, for every n > 0 and v, v' ∈ V_G,

\[ block_n(v) = block_n(v') \iff sig_{n-1}(v) = sig_{n-1}(v') \tag{3.2} \]

We prove the lemma by induction on n. The initial partition is \( \{V_G\} \), which means that \( sig_0(v) = sig_0(v') \), \( \forall v, v' \in V_G \). Therefore the claim is true for \( n = 1 \). Let n be > 1 and v, v' ∈ V_G such that \( sig_{n-1}(v) \neq sig_{n-1}(v') \). Then there must be a pair \((a, block_{n-1}(z))\) that (without loss of generality) occurs in \( sig_{n-1}(v) \) and does not occur in \( sig_{n-1}(v') \). We distinguish two cases for the FW bisimulation case (the FWBW case is similar): (i) There is no transition \( v' \xrightarrow{a} t \). Then \( sig_n(v') \) will not contain any pair of the form \((a, -)\), while \( sig_n(v) \) will have at least \((a, block_n(z))\). (ii) There are transitions leaving from \( v' \) and labeled with \( a \). In this case, let \( v \xrightarrow{a} t \) be any of them. \( block_{n-1}(t) \) must be \( \neq block_{n-1}(z) \), since \((a, block_{n-1}(z)) \notin sig_{n-1}(v') \). Then from (3.2) it follows that \( sig_{n-2}(t) \neq sig_{n-2}(z) \) and, by using the inductive hypothesis, that \( sig_{n-1}(t) \neq sig_{n-1}(z) \). Further, we apply (3.2) one more time, to obtain that \( block_n(t) \neq block_n(z) \). Thus, the set \( sig_n(v) \) will contain the pair \((a, block_n(z))\), but \( sig_n(v') \) not.

Given a dataset graph \( G \) with nodes \( V \) and edges \( E \), the Paige-Tarjan \[94\] partition refinement algorithm judiciously selects blocks to refine which leads to an algorithm with complexity of \( O(|E|\log |V|) \). Our B variation processes all instance nodes and edges leading to a complexity of \( O(|E| |V|) \). Like in the Paige-Tarjan algorithm, the B variation terminates when two consecutive iterations have the same number of distinct blocks, which results in the coarsest partition having the fewest possible number of blocks that have a maximal number of equivalent instance nodes in their extents. The B variation terminates since the summary can contain only as many blocks as instance nodes in the dataset graph. Optionally, if the algorithm detects an equal number of blocks and instance nodes at the end of an iteration then the algorithm can terminate since the number of blocks cannot increase; this termination condition is akin to all nodes being singleton and thus the algorithm does not require recomputing any node’s block identifier. The S variation’s loop termination condition is the same as the Paige-Tarjan and B variations because the termination condition does not distinguish between singleton blocks and non-singleton blocks. If two consecutive iterations have a constant total number of singletons and non-singletons, then the loop termination condition has been met and the algorithm terminates (lines 13, 15, 16 of Figure 3.8). Since all nodes in a dataset may be non-singleton, the S variation’s worst-case complexity is \( O(|E| |V|) \).

Linear time algorithms \[37\] for computing summaries have been proposed for directed-acyclic graphs. One optimization \[37\] relies on the computation of rank, a notion of a node’s depth in a topological sort of the tree’s strongly-connected components (which can be computed using a variant of Tarjan’s classical algorithm \[114\]), and which results in a sequential, linear-time algorithm. Computing a bisimulation relation of a graph is \( P \)-complete \[14\], implying the difficulty of parallelization despite having an efficient sequential algorithm, e.g., up to \( n \) iterations may be needed to compute the summary of a dataset graph consisting of an acyclic, non-branching path of \( n \) nodes. Notice that our singleton optimization will have less effect in sequential algorithms in cases where a node’s block identifier is not recomputed more than once. This is the case in the rank-based approach \[37\] where a singleton does not participate in a strongly-connected component since when a singleton is found, it is already too late since the optimization does not recomputate a node’s block identifier more than once if it is not in a strongly-connected component. The complexity in such optimizations is \( O(|E| \log |V|) \) even with the singleton optimization since block identifiers of singletons need to be used by their neighbours in at least
one computation of the neighbours’ block identifier. However, the singleton optimization can improve per-iteration time when processing nodes that participate in a strongly-connected component.

In this section, we described our singleton optimization that skips processing singleton blocks by disabling singleton nodes in the GraphChi scheduler. Next, we describe how singleton optimization is novel and distinct from stable block optimization \[19\].

**Comparing singleton and stability optimizations**

A block is *stable* if it does not have any block edges to a block that was split in the previous iteration (a block is split when two non-bisimilar nodes are found in the block’s extent). We compare our singleton optimization with that of the existing notion of stability \[19\], showing that it is novel and unique in two ways: we show variations in how a singleton becomes stable, and we show that a singleton can repeatedly switch between stability and instability.

**Unstable singletons.** We can easily construct examples that show a singleton that is stable, a singleton that is unstable, a singleton that is not found until an unstable block is stable, and a singleton that is found by splitting an unstable block that remains unstable. We can also verify that singletons are distinct from the notion of stability because the singleton optimization is monotonic, in that once a block is a singleton it always remains a singleton, but stability is not monotonic since a block’s state can switch between stable and unstable.

![Figure 3.10: Unstable singleton example](image)

**Unstable stability.** Figure \[3.11\] gives an example where a singleton block can switch back and forth between stability and instability. At the start, all nodes are in the same block. In iteration 1, node 1 is split into a singleton because it is the only node with an out-edge with label \(a\), and node \(n\) is in a singleton because it is the only node that does not have any out-edges. As well, \(block_1(1)\) is unstable because \((block_1(1), block_0(2)) \in E_S\) but \(block_0(2)\) was split, though \(block_1(n)\) is stable because it does not have any outgoing edges. After \(n-2\) iterations, we end up with a partition that only contains singletons.

**Unstable stability.** Figure \[3.11\] gives an example where a singleton block can switch back and forth between stability and instability. At the start, all nodes are in the same block. In iteration 1, node 1 is split into a singleton since it is the only node that has out-edges with the label \(a\); as well, nodes 2 and 3 form a block, nodes 4 and 5 form a block, node 6 forms a singleton block, and nodes 7 and 8 form a block. Notice that at the end of iteration 1, only \(block_1(7)\) is stable since it does not have any outgoing
edges. In iteration 2, since \((\text{block}_1(6), \text{block}_0(7)) \in E_S\) and \(\text{block}_0(7)\) was split, then \(\text{block}_1(6)\) is unstable and we need to process it. Since it has only 1 node, the block cannot be split further. In iteration 3, \(\text{block}_1(4)\) is split such that \(\text{block}_2(4) \neq \text{block}_2(5)\).

Determining a block’s stability is more expensive than checking whether a node is a singleton since stability requires checking whether any of the instances in a block’s extent has an edge to a block that was split in the previous iteration. In contrast, checking whether a node is a singleton requires checking a block’s size from the previous iteration. So then, only after all of a block’s edges have been examined can a block’s stability be determined; an amount of work that can be considered equivalent to examining all of a block’s edges in order to update the block’s identifier. Furthermore, since a block can switch back and forth between stability and non-stability, a block’s edges have to be checked in each iteration; in contrast, the singleton optimization is monotonic and does not require rechecking every iteration. We report anecdotal results that the majority of blocks in the datasets that we consider in Section 3.3 are singleton and stable and since the majority are singleton, and checking whether a block is a singleton is cheaper than checking a block’s stability, the singleton optimization has a clear advantage over the stability optimization. For these reasons, we focus on the singleton optimization.

In this section, we described a summary construction algorithm and a singleton optimization. In the next section, we show evidence that our summary construction optimization is effective at drastically reducing per-iteration time within the first few iterations.

### 3.3 Evaluation

In this section, we give an evaluation of our GraphChi B and S variations followed by a comparison to existing literature. Our results show that our GraphChi approach is effective at computing summaries of large graphs with multi-core architecture. In Section 3.3.1 we describe the datasets that we use in our experiments. In Section 3.3.2 we compare our GraphChi B and S variations and give evidence that we achieve our goal of constructing summaries in an amount of time equivalent to the time it takes to load the input data and output the summary. In Section 3.3.3 we show a comparison of our GraphChi
approach with Hadoop.

### 3.3.1 Datasets

In this section, we describe the datasets that we use in our experiments of Sections 3.3.2 and 3.3.3.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>N</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>lmdb</td>
<td>1,327,120</td>
<td>6,147,717</td>
</tr>
<tr>
<td>spb256</td>
<td>41,472,605</td>
<td>231,000,100</td>
</tr>
<tr>
<td>dbp</td>
<td>48,603,466</td>
<td>317,220,816</td>
</tr>
<tr>
<td>Twitter</td>
<td>52,579,678</td>
<td>1,963,263,821</td>
</tr>
</tbody>
</table>

Table 3.1: Dataset graph statistics

Table 3.1 shows the number of nodes and edges in the dataset graph of each dataset that we use in our experiments. LinkedMDB [56] describes entities related to movies including actors and directors, and is the smallest dataset graph (lmdb) we consider with 1.3M nodes, 6.1M edges, and 222 distinct edge labels. The Semantic Publishing Benchmark (SPB) [8] provides a synthetically generated dataset for use in benchmarks and which results in a dataset graph (spb256, '256' represents the scale of generation) with 41.5M nodes, 231M edges, and 16 distinct edge labels. DBpedia [13, 17] describes entities such as places and events, and its dataset graph (dbp) has 48 million nodes, 317 million edges, and 1,393 distinct edge labels; we exclude highly structured entities such as geographic coordinates to reduce structuredness. Twitter [24] is the largest dataset that we consider with 52M nodes and almost 2 billion edges unlabeled follower relationships between users, which is slightly bigger than the Twitter dataset [76] used in existing summary literature.

### 3.3.2 Constructing FWBW summaries

In this section, we evaluate FWBW summary construction of different real-life datasets using our B and S variations. Our evaluations show that our S variation constructs summaries faster than the time to load an RDF dataset plus write its RDF summary. In Section 4.2.6, we consider construction of more compact selective summaries that choose a subset of the datasets predicates according to real-life scenarios, such as query logs; our S variation constructs selective summaries faster than the time to load an RDF dataset plus write its selective RDF summary.

To compare FWBW summaries, we use M1, a single machine that has 8 Xeon X6550 2GHz 8-core CPUs and can process 64 nodes in parallel. M1 has an 80GB allocation of main-memory to the Java 8 JVM that executes unmodified GraphChi 0.2; our B variation uses around 40GB with the largest dataset and our S variation uses around 60GB with the largest dataset.

<table>
<thead>
<tr>
<th>Summary</th>
<th>N</th>
<th>E</th>
<th>Load</th>
<th>B</th>
<th>S</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>lmdb</td>
<td>844,877</td>
<td>4,311,098</td>
<td>0.12</td>
<td>8.6</td>
<td>2.8</td>
<td>0.75</td>
</tr>
<tr>
<td>spb256</td>
<td>32,927,240</td>
<td>205,431,153</td>
<td>57</td>
<td>429</td>
<td>268</td>
<td>71</td>
</tr>
<tr>
<td>dbp</td>
<td>32,274,111</td>
<td>278,182,230</td>
<td>107</td>
<td>775</td>
<td>187</td>
<td>207</td>
</tr>
<tr>
<td>Twitter</td>
<td>48,332,025</td>
<td>1,945,307,755</td>
<td>527</td>
<td>3,118</td>
<td>632</td>
<td>750</td>
</tr>
</tbody>
</table>

Table 3.2: FWBW summary statistics, with M1 times (in mins) to load input, construct summary (using B or S variation), and write summary
Table 3.2 shows the number of blocks and block edges in the FWBW summary of each dataset listed in Table 3.1. The lmdb summary, which has the greatest reduction in size with respect to the dataset graph (not counting extent edges) amongst the 3 datasets, has 0.84M blocks and 4.3M block edges, which represent 64% and 70% of the number of dataset graph nodes and edges, respectively. The spb256 summary has almost 33M blocks and 205M block edges, which represent 79% and 89% of the number of dataset nodes and edges, respectively. We notice that the spb256 dataset has 653K more blocks that the dbp summary, which has 32M blocks and 278M block edges, and which represent 66% and 88% of the number of dataset nodes and edges, respectively. The Twitter summary has the least reduction in size with 48M blocks and only 18M fewer block edges than dataset graph edges, which represent over 90% and 99% of the number of dataset nodes and edges, respectively.

Table 3.2 also shows the M1 times (in mins) to load RDF data into GraphChi, construct the summary using the B or S GraphChi variation, and write the RDF summary to disk; the times shown are the average of 5 runs excluding the min and max. Constructing FWBW summaries using the S variation is 3.0x, 1.6x, 4.1x, and 4.9x faster than the B variation for lmdb, spb256, dbp, and Twitter, respectively. The time difference between between the B variation and S variation summary construction times reveals that uniqueness of an instance node’s local description is reflected in the number of singletons found, and that having more uniqueness results in more singletons and reduces summary construction time significantly. The spb256 dataset’s 1.6x performance difference, which benefits from the singleton optimization the least amongst the FWBW summary constructions, reveals that the generated dataset has little unique locality. In contrast, we see that larger real-world datasets DBpedia and Twitter benefit from the singleton optimization, with summary construction performance improving by more than 4x due to heterogeneous local semi-structure of instance nodes.

Our results in Table 3.2 show that the time take to compute spb256’s FWBW summary using the S variation is 2.1x times slower than the time to load the dataset graph and write the summary; this synthetic dataset’s result can be explained by the fact that the majority of singletons are not found until the last few iterations, which we show later. In contrast, our results show that the time taken to compute dbp’s FWBW summary using the S variation is 1.68x faster than the time to load the dataset graph and write the summary, and the time taken to compute Twitter’s FWBW summary using the S variation is 2x faster than the time to load the dataset graph and write the summary. The time difference to construct a dataset’s summary versus the time to load the dataset plus write the summary again shows that the singleton optimization greatly reduces summary construction time as a result of heterogeneity in the local semi-structure of dataset nodes. As well, the singleton optimization’s use of GraphChi’s scheduler to skip loading and processing of nodes reveals that a reduction in the amount of data that is read from disk improves summary construction performance. We report several anecdotal results that help justify our approach. We report anecdotal results that using a RAM disk did not improve performance since all disk accesses, such as loading of a dataset graph and GraphChi’s shards or writing a summary graph, are essentially sequential disk operations. We also report anecdotal results that increasing the amount of main-memory available to the JVM, nor decreasing the number of shards GraphChi should use (so that more of the graph is stored in each shard and is read into memory), did not result in discernible performance difference since the disk is the bottleneck. We report anecdotal evidence that a variant of our approach which stores block identifiers within GraphChi edges that are serialized to disk, and which avoids storing the summary in memory, is approximately 2 times slower; this approach makes it viable for a faster disk to improve summary construction performance while reducing memory consumption.
GraphChi’s design is such that even a small amount of memory (such as on a laptop) can be used as long as the dataset is partitioned into a sufficient number of shards such that a shard can fit in main-memory. If we increase the number of cores, then we can improve summary computation performance since we can process more nodes in parallel, and also potentially improve summary write performance since this is done in parallel while processing the graph. In our current approach, increasing the number of cores will not alter dataset load time. One option is to parallelize the dataset load, which requires some form of dataset pre-partitioning and ensuring that Java-based GraphChi version supports multi-threaded load into its internal graph representation.

Figures 3.12 (a) and (b) show lmdb’s FWBW summary construction for each iteration of the B and S variations; both variations take 13 iterations. Figure 3.12 (a) shows per-iteration time (in mins). In the B variation, per-iteration time is 12 seconds in iteration 1, increases steadily to 39s by iteration 5, and then maintains that time. In contrast, the S variation begins similarly but its per-iteration time decreases to around 7s by iteration 7, and then maintains that time. Our results show that the S variation’s per-iteration time deviates to 25% of the B variation within the first few iterations.

Due to GraphChi’s way of iterating over nodes that it creates in order to balance parallelism across cores, we skip singletons as follows. During an iteration \( k \), we count the number of nodes in each block. Then during the \((k + 1)\) iteration, when we execute each node’s update method, we check whether the node is contained in the extent of a singleton block. If a node is found to be in a singleton, we do not update its block identifier. As well, during the \((k + 1)\) iteration, we identify those nodes that are in singletons, then at the end of the \((k + 1)\) iteration, we disable them in the scheduler. Thus, singletons that exist at the end of an iteration \( k \) are skipped from iteration \((k + 2)\) onwards. Next, we show the count of singletons and non-singleton blocks in each iteration during FWBW summary construction, and describe the relationship between per-iteration time and count of singletons.

Figure 3.12 (b) shows, the total number of blocks in lmdb’s FWBW summary at each iteration, divided into singletons (\(SG\)) and non-singletons (\(Non-SG\)); the total is the same as the B variation. The total number of blocks increases to 837K blocks by iteration 5, and increases in small increments thereafter. No single iteration has more than 89K non-singletons and 762K singletons, which means that after the first few iterations up to 57.4% of the dataset graph’s nodes are skipped in each iteration. As noted above, singletons that exist at the end of an iteration \( k \) are not skipped until iteration \((k + 2)\) onwards. We observe that lmdb has 6.8K singletons at the end of iteration 1, which represents only 0.51% of the dataset graph’s nodes, and that the S variation is already 15% faster than the B variation in iteration 3, the earliest moment in which the singleton optimization can skip nodes. The S variation
has the greatest reduction in per-iteration time in iteration 6 after finding an almost maximal number of singletons in iteration 4. The stark reduction in per-iteration time demonstrates the benefit of skipping singletons as an optimization.

![Figure 3.13: spb256 FWBW summary construction](image)

Figures 3.13 (a) and (b) show spb256’s FWBW summary construction for each iteration of the B and S variations; both variations take 10 iterations. Figure 3.13 (a) shows the per-iteration time (in mins). As observed earlier, the B and S variations take equal number of iterations, iterations take almost equal amounts of time until iteration 5, after which iterations in variation B take longer than variation S. Figure 3.13 (b) shows the total number of blocks, divided into singletons and non-singletons. Until iteration 7, a larger proportion of the summary’s blocks are non-singletons than singletons. There are less than 11 singletons until iteration 3, and 0.9M singletons in iteration 4, which leads to a reduction in the S variation’s per-iteration time in iteration 6. Finding an almost maximal number of singletons in iteration 7, which compose 80% of the summary’s blocks, results in almost equal per-iteration time in the last two iterations. Our results show that the S variation’s summary construction of the spb256’s dataset has a slower increase in the number of singletons, and that a majority of the blocks need to be singletons before per-iteration time reaches its lowest, around 20% of the B variation’s per-iteration time.

![Figure 3.14: DBpedia FWBW summary construction](image)

Figures 3.14 (a) and (b) show dbp’s FWBW summary construction for each iteration of the B and S variations; both variations take 17 iterations. Figure 3.14 (a) shows the per-iteration time (in mins). As with lmdb, the B and S variations take equal number of iterations, iterations in variation B take longer than variation S, and iterations tend to take equal amounts of time. We attribute the difference in time in iteration 1, before skipping any singletons, due to a difference in how the algorithm counts
blocks in the first iteration. Figure 3.14 (b) shows the total number of blocks, divided into singletons and non-singletons. There is a substantial increase in the number of singletons within the first few iterations, with slow growth after iteration 3, at which point the S variation’s per-iteration time is 80% less than the B variation. A performance improvement is visible with variation S at iteration 3, the earliest moment in which the algorithm can skip singletons, and the per-iteration time is lowest at iteration 5 onwards after finding an almost maximal number of singletons, which, like Imdb, comprises of 57.3% of the dataset graph’s nodes. Our results show that having a larger number of singletons decreases per-iteration times of FWBW summary construction.

![Figure 3.15: Twitter FWBW summary construction](image)

(a) M1 time (in mins) per iteration  
(b) Millions of singletons and non-singletons per iteration

Figures 3.15 (a) and (b) show Twitter’s FWBW summary construction for each iteration of the B and S variations; both variations take 13 iterations. Figure 3.15 (a) shows the per-iteration time (in mins). The figure shows that both the B and S variations start similarly then in the 5th iteration the B variation quickly takes almost an order of magnitude longer than the S variation, taking only around 4 minutes per iteration. Figure 3.15 (b) shows the total number of blocks, divided into singletons and non-singletons. There are only 3, 15, and 12K blocks in the first 3 iterations, respectively. No singletons exist in the first two iterations, then after only 2.4K singletons are found at the end of iteration 3, there is a remarkable reduction in per-iteration time to only 4 minutes in iteration 5. This is visible as the last 8 iterations of the S variation per-iteration time taking around only 1.3% of the B variation’s time. In later iterations, even as the number of singletons represent over 90.4% of the dataset graph’s nodes, the per-iteration time remains constant. Our results show that even if only small portion of the dataset’s nodes are singletons, then drastic reductions in per-iteration times are possible.

Our results also reveal that more heterogeneity in a node’s local semi-structure reduces the per-iteration time in the S variation earlier, which then benefits overall summary construction performance due to the monotonicity of the singleton optimization. This is evident in the Imdb, dbp, and Twitter real-world datasets for which more than half the iterations have drastically lower per-iteration time than the B variation. In contrast, the spb256 generated dataset shows little heterogeneous locality; heterogeneity in the dataset’s semi-structure only gradually starts to become apparent further than 5 navigational steps away from all of spb256’s nodes.

In this section, we described the construction of summaries for LinkedMDB, SPB, DBpedia, and Twitter with our B and S variations. Our results show that our singleton optimization’s rapid reduction in per-iteration time results in summary construction performance that is faster than loading the dataset into GraphChi and writing the summary to disk. Next, we give an experimental comparison of GraphChi and Hadoop summary construction implementations.
3.3.3 Comparing GraphChi with Hadoop

Our work in Section 3.1 is, to the best of our knowledge, the first reported Hadoop-based FWBW summary construction, but FWBW summary construction is not directly comparable to FW summary construction since the number of blocks and iterations can be different. Thus, we compare GraphChi FW summary construction with Hadoop-based FW summary construction [108]. In this section, we compare our GraphChi B and S variations with HF, our Hadoop 2.20 implementation based on [108]. Since HF which computes only FW summaries, in this section we set our B and S variations to compute FW summaries by examining only outgoing edges, i.e. by excluding line V2 in the algorithm of Figure 3.8.

For verification, we generate summaries of test datasets and compare the output summaries of HF with the output summaries of the GraphChi B and S variations, ensuring that all generate the exact same FW summary of the test data.

We use M2, a single machine with 64GB of RAM and 2 Opteron6128 2GHz 8-core CPUs, which can process 16 threads in parallel, to execute Hadoop in different modes appropriate for multi-core architecture, including local mode and pseudo-distributed mode. We assign all of the machine’s 64GB of RAM to each variation, with our GraphChi variations using about 40GB and HF using about 20GB.

We execute HF as variation HL using Hadoop’s local mode, which uses a single machine, and executes all jobs in one JVM. Each job executes as parallel, threaded map tasks, but one reduce task. Since we do not use fault-tolerance, we store data as gzip-compressed files on the local file system. In local mode, the reduce task saves its output to a single file; however, map tasks cannot read gzip-compressed files in parallel. To enable parallelism, we save the reducer’s output in multiple gzip-compressed files, each of which can then be accessed by one of the parallel map threads in the next iteration.

We also execute HF as variation HP using Hadoop’s pseudo-distributed mode, which uses parallelism on a single machine by executing each map and reduce task in its own JVM, and we enable the option to reuse JVMs for subsequent tasks. We set the number of parallel tasks that Hadoop can spawn to the max number of hardware threads. Since we consider fault-tolerance out of scope, we use the local disk for storage. Additionally, we divide the machines RAM equally amongst task JVMs. As well, we partition each reducer’s output into multiple files to avoid mapping each reducer’s lone file output as input to one mapper.

<table>
<thead>
<tr>
<th>Summary</th>
<th>N</th>
<th>E</th>
<th>B</th>
<th>S</th>
<th>HL</th>
<th>HP</th>
</tr>
</thead>
<tbody>
<tr>
<td>lmdb</td>
<td>85,714</td>
<td>999,934</td>
<td>4.2</td>
<td>4.0</td>
<td>73</td>
<td>32</td>
</tr>
<tr>
<td>dbp</td>
<td>13,429,903</td>
<td>229,490,296</td>
<td>941</td>
<td>331</td>
<td>5,832*</td>
<td>1,972</td>
</tr>
</tbody>
</table>

Table 3.3: FW summary statistics, with M2 construction times (in mins)

We now compare FW summary construction performance of our B, S, HL, and HP variations. In Table 3.3 we show the FW summary statistics for lmdb and dbp, as well as their M2 construction times (in mins) for the 4 variations; the dbp HL variation time is for the first 10 iterations. The times show that both our B and S variations are much faster than the HP variation, the fastest Hadoop variation. We see that the limitation of having just one reducer in the HL variations decreases its performance by more than 2x.

Figures 3.16 (a) and (b) show lmdb’s FW summary construction for each iteration of the B and S variations; both variations take 12 iterations. Figure 3.16 (a) shows per-iteration time (in mins).
the B and S variations start similarly, increase to 20s by the next iteration, and remain at that duration for the remainder of the construction. Figure 3.16 (b) shows the total number of blocks in the FW summary at each iteration, divided into singletons and non-singletons. The number of blocks increases to around 85K blocks by iteration 4, with small increments thereafter. The FWBW summary of lmdb has an order of magnitude more singletons than in the FW summary, and our results show that the 59K singletons in lmdb’s FW summary provide little discernible per-iteration decrease when using the S variation. The drastically lower number of singletons in lmdb’s FW summary, as compared to its FWBW summary, shows that a larger number of singletons results in a more meaningful reduction in per-iteration times.

Figures 3.17 (a) and (b) show dbp’s FW summary construction for each iteration of the B and S variations; both variations take 25 iterations. Figure 3.17 (a) shows per-iteration time (in mins). The per-iteration time of both variations start similarly, then the B variation increases to 32min while the S variation decreases to 12min. Figure 3.17 (b) shows the total number of blocks in the FW summary in each iteration, divided into the number of singletons and non-singletons. There are around 85K blocks by iteration 4, and it increases in small increments thereafter. Unlike the lmdb FW summary, 22% of the dataset graph’s nodes are singletons, which helps to improve the S variation’s per-iteration time.

We compare our work with [108], which we call MRB, whose DBpedia dataset is smaller than ours by 27M and 119M nodes and edges, respectively. MRB computes its FW summary, which contains 5 million blocks, in 459 minutes without writing the summary to disk. MRB’s advantage is that it does not need to load any data, but has disadvantages in that it duplicates block identifiers as edge properties and processes all nodes in each iteration. Summaries that MRB computes are not as costly as our FWBW summaries (each iteration in a FW summary construction only looks at outgoing edges), and our S variation’s iteration time is lower than MRB’s. On Amazon EC2, a single multi-core compute
instance has more compute power and is 2.5x times more economical than an MRB-equivalent 10-node cluster. Furthermore, our S variation requires much less parallelism, uses 40x times less RAM, and has better performance even if a dataset’s summary construction needs more iterations.

Our S variation’s per-iteration time to generate Twitter’s FWBW structural summary is faster than the FW summary per-iteration time of [79, 81], whose dataset has 500M fewer edges than ours does. Furthermore, their system focuses on optimizing the skew of edges across the few large blocks, while we focus on optimizing the many small blocks at the opposite spectrum of block sizes - where singletons are very common. The singleton optimization may be integrated into Hadoop-based summary constructions by sharing all non-singleton block identifiers with all parallel tasks using Hadoop’s distributed cache since there are fewer than 5 million non-singletons in the FWBW or FW summaries that we construct.

3.4 Summary

In this chapter, we gave several FWBW summary constructions, including for Hadoop and GraphChi. Since existing literature only describes FW summary constructions, we described Hadoop implementations to compute FWBW structural summaries. Our Hadoop approaches show variations in the number of jobs needed per iteration, and the amount of intermediate data. In particular, although our p/-p variation uses less intermediate data, it requires at least 1 more iteration than the p&-p variation since the minimum number of times that the count of distinct blocks in consecutive partitions has to be computed is 3, and only 2 for the p&-p variation. Since emitting a node’s block identifier requires traversing the same edges as those used to compute the node’s block identifier, we proposed a Hadoop approach that reduces the amount of intermediate data by using Hadoop’s resettable-iterator in the reduce task to re-stream edges.

We also described a vertex-centric approach to computing FWBW summaries using GraphChi. In addition to our GraphChi variation B, we also described a novel and effective singleton optimization, variation S, that can drastically reduce the per-iteration time within the first few iterations. Furthermore, we showed that our singleton optimization is novel and distinct from the notion of stability. We compared our GraphChi B and S variations, and gave evidence that our singleton optimization is effective for constructing FWBW structural summaries in an amount of time equivalent to the time it takes to read the input and write the summary. Furthermore, our comparison of our GraphChi B and S variations to FW summary construction on Hadoop as described in existing literature show that our variations are effective on multi-core, requiring less parallelism, RAM, and disk space.

In the next chapter, we describe how to query RDF data using structural summaries instead of the instance.
Chapter 4

Summary-based Query Optimization

In this chapter, we describe how to query RDF data using RDF structural summaries. The work shown in this chapter is part of ongoing collaboration with Consens, Fionda and Pirró [31].

As large graphs, like the social graphs of Facebook and Twitter, continue to become the norm in capturing relationships between real-world entities, challenges arise in representing, storing, and querying them, and systems for graph databases and graph query languages have received attention both from academia and the industry [128, 21, 115, 62, 111]. The recursive nature of graph queries [2] relies on navigation as a first-class citizen and, since traditional relational languages (e.g., SQL) capture only implicit relationships, several graph languages have been proposed (see [7, 8, 125] for recent surveys). From the types of graph query languages that emerge, it has become the case that certain classes of queries are not practical, such as conjunctive regular path queries (CRPQs) and their variants [28, 15, 22]. This means that CRPQs cannot be used to address the challenge of querying large graphs. Alternatives to limit the complexity of query evaluation include restricted languages like acyclic conjunctive regular path queries [16] and Nested Regular Expressions (NREs) [97]. Furthermore, the SPARQL language has added support for property paths (PPs) as a way to support navigational queries. However, PPs offer limited navigational capabilities; for instance, unlike NREs, PPs do not enable nesting of expressions.

Although efficient query evaluation is one way to address the challenge of querying large graphs, using a potentially more compact summary representation to answer queries is also of interest. An advantage of using a more compact representation is that, even faced with polynomial complexity in query languages and which is still prohibitively large in the limit for large graphs, the actual cost is reduced by answering via a more compact representation. In the case of bisimulation summaries, a summary’s size is bound by the size of the dataset it summarizes.

SchemeX [74] construct FW summaries that group nodes based on the predicates and types that describe them. In ExpLOD [67], we showed how flexible FWBW summaries can help a user understand and describe RDF datasets, such as by grouping nodes that are instances of the same set of types, or that are described by the same set of predicates. However, FW summaries do not support traversing backwards along a dataset’s edges, and neither supports navigational queries beyond a single step. Summaries based on strong simulation [82], which constructs simulation summaries based on locality, and which can provide navigations longer than one step both along and backwards along edges. However, we do not compare our approach to strong simulation since scaling strong simulation requires pre-partitioning an instance dataset into overlapping subgraphs for each node in the dataset graph, limiting
the number of steps in a navigational query and which can result in an exponential increase in space.

Our goal in this work is to show that our FWBW RDF summaries support effective query optimization across a range of navigational queries, datasets, and scenarios, including user exploration and query workloads. In this work we leverage the W3C standard SPARQL query language to address the challenge of query optimization on large graphs by using RDF summaries. To leverage optimizations based on RDF summaries in existing RDF stores, we have devised a translation for SPARQL queries that executes navigational queries over RDF summaries stored alongside datasets.

Our approach to optimize graph traversal queries on top of existing SPARQL processors consists of three components. First, we have developed and implemented a translation of navigational queries into SPARQL expressions. Second, we construct and store summaries alongside the original graph in the RDF store. Third, we build upon our translation and integrate rewriting for query optimization that converts navigational queries on a dataset graph into corresponding navigational queries on a summary. Our translation leverages the Extended Property Paths (EPPs) language, which is more expressive than SPARQL property paths and defines a concise syntax and formal semantics. EPPs allows users to write more expressive navigational queries than property paths in a succinct way; and EPPs are immediately supported by existing unmodified SPARQL processors.

In Section 4.1, we describe two translations for EPPs navigational expressions that allows using summaries alongside instance graphs: EPPsItoEPPsS, that translates a dataset query to a corresponding summary query; and EPPstoSPARQL, that translates an EPPs expression to SPARQL. These two translations can occur transparently to the user and without modification to an existing SPARQL processor. All that it requires is computing a summary and storing it alongside the dataset graph, and for a user to express their navigational query using the EPPs language. The EPPs language is available as open-source and can be reused and adapted, as we have done for this work. In essence, the EPPsItoEPPsS translation allows taking a navigational query on the dataset query and translating it to a corresponding navigational query on the summary graph instead. The navigational query is executed on the RDF store by translating the query into an corresponding SPARQL query using the EPPstoSPARQL translation. This allows the use of EPPs to express the navigational queries that can be optimized on existing unmodified SPARQL processors.

In Section 4.2, we give experimental results that validate the effectiveness of our approach by comparing query performance of different summaries. In particular, we show the effectiveness of our approach using selective summaries that can improve query performance more than a full summary. Selective summaries pick predicates according to real-world scenarios, specifically from DBpedia’s query logs, and which range in popularity in the query logs, in the number of statements in the instance dataset, and the number of block edges in the summary. We evaluate our approach in three ways. We give experimental evidence that our summaries improve query performance on a variety of unmodified SPARQL processors. We give experimental evidence that our summaries improve query performance with a variety of real-world datasets, and selective summaries that are based on real-world navigational query scenarios, including manual user exploration and user query logs. We give an experimental validation that our GraphChi implementation achieves our goal of constructing full and selective summaries of large graphs in an amount of time similar to the time required to load the dataset plus write the summary. In Section 4.3 we summarize our chapter contributions.

\[1\texttt{ftp://download.openlinksw.com/support/dbpedia/}\]
4.1 EPPs query expressions

In this section, we describe a subset of the Extended Property Paths [41] language that we use in this work, giving examples of navigational queries and their translation into queries that execute on the summary using SPARQL.

An EPPsS is an EPPs [41] to be used in conjunction with structural summaries (the S suffix denotes that it is an EPPs over a summary); we focus on the non-recursive fragment of the EPPs language. The EPPs language represents a significant enhancement of SPARQL property paths [54], a feature incorporated into the SPARQL 1.1 standard to provide navigational features. The EPPs language allows writing expressive navigational queries in a concise way thanks to their regular-expression-like syntax shown in Table 4.1. The availability of path repetitions (i.e., \texttt{epp\{l, h\}}) in EPPs allows a user to cope with the lack of transitive closure since a relatively low number of repetitions is enough to cover real world graphs usually having small diameter \footnote{As an example, the diameter of Facebook is $\sim 5$ [119]}. Our choice to focus on non-recursive EPPsS is motivated by the fact that (concise) expressions in this fragment can undergo translations into (more verbose) SPARQL queries. We want to point out that such translations allow evaluating EPPs expressions on existing SPARQL processors.

\texttt{epp} ::= ‘ˆ’ | epp/’/’ | epp ‘|’ | (‘ epp ‘) | epp\{l, h\} | pos\ test | pos’ | epp ‘&’ | epp ‘∼’ | epp
\texttt{test} ::= ‘!’ | test | test ‘&&’ | test ‘||’ | test
\texttt{base} ::= uri | TP(pos ,’ epp ‘)
\texttt{pos} ::= ‘s’[‘(def)’] | ‘o’[‘(uri)’]

Table 4.1: Syntax of EPP expressions.

As it can be observed in Table [41] besides path repetitions, the EPPs language features concatenation (‘/’), path negation (‘∼’), path disjunction (‘|’), path conjunction (‘&’). The language also enables changing the direction of a traversal (‘ˆ’) and allows specifying whether to direct traversals from a subject position (s) to the object position (o) or vice versa across an RDF triple, with the option to constrain a position to a URI. For further details, the formal semantics of EPPs, and a formalization of the translation procedure, we refer the interested reader to [41]. Next, we give a hint of EPPs syntax and outline translations that allow the evaluation of EPPs expressions on existing SPARQL query processors.

4.1.1 EPPsItoEPPsS translation

We present the EPPsI, an EPPs query on the instance dataset, of our motivating example and translate it into a query for the summary using an EPPsItoEPPsS translation.

The EPPsI expression satisfying our motivating example is:

\begin{verbatim}
:Director (o rdf:type o) / (:director \&& !TP(o, :country)
\&& TP(s, :producer))/ :country)/ :actor
\end{verbatim}
The evaluation of the above example EPPsI expression on the dataset graph of Figure 1.1 starts at the :Director node and traverses the rdf:type edge backward to reach a director node. Then, it continues by traversing :director edges to reach movie nodes, filters those movies that the director produced and also have a :country edge (thus filtering movie:2014), and finally reaches actor nodes by traversing :actor edges via those movies that do not have country information. The result from the instance dataset of Figure 1.1 contains only the node actor:45772.

Figure 1.4 shows the EPPsItoEPPsS translation of the motivating example’s EPPsI. An EPPsI-toEPPsS translation of the EPPs incorporates bc:extentContains predicate-labeled traversals in order to navigate between blocks and the instances in their extent.

### 4.1.2 EPPstoSPARQL translation

We introduce EPPstoSPARQL, another translation procedure which, given an EPPsS translates into a SPARQL expression.

The EPPstoSPARQL translation of an EPPsS is a simple extension of the EPPstoSPARQL translation of an EPPsI; they differ in the following respects: The EPPsItoEPPsS translation adds edge traversals with label bc:extentContains to navigate from blocks in the summary to those instances in their extent and vice versa; and the EPPstoSPARQL translation of an EPPsS uses a different named graph (identifying the summary graph), with the option available to use a different named graph for extent edges.

The SPARQL query obtained from the EPPstoSPARQL translation of our motivating example’s EPPS expression is the following:

```sparql
SELECT ?Actor {
  { SELECT DISTINCT ?ActorBlock FROM {
    GRAPH <http://summary> {
      ?DirectorB rdf:type ?DirectorTypeB .
      ?DirectorB :director ?MovieNoCB .
      FILTER EXISTS { ?DirectorB :producer ?MovieCB .
        ?MovieCB :country ?Country . } }
    FILTER NOT EXISTS { ?MovieNoCB :country ?aCountryB }
    ?DirectorTypeB bc:extentContains :Director
    ?ActorBlock bc:extentContains ?Actor }}
}
```

Notice the SPARQL query uses a subquery to return distinct blocks of interest, and then returns all the nodes in the subquery’s block extents. The EPPstoSPARQL translation of an EPPS leverages the notion that, since a summary query returns distinct blocks, and since each dataset node belongs in the extent of exactly one block, the union of block extents form a (distinct) set of nodes. Also notice the use of bc:extentContains both to obtain the summary block corresponding to the :Director class of the instance dataset and to retrieve nodes in the instance dataset after the evaluation of the query over the summary dataset. In the above translation, we show that extent edges (which adds an edge for each dataset node) are stored together with the summary's block edges.
4.1.3 Properties of selective summaries

In this section, we describe an interesting property of selective summaries, namely that the number of iterations to compute a dataset’s full summary may differ from the number of iterations needed to compute a selective summary.

Figure 4.1: Iterations of FW full and selective summaries on an example instance

Figure 4.1 shows the possible effects on the number of iterations of FW selective summaries by showing the summary’s blocks in each iteration. Part (a) shows the blocks in each iteration of the full summary, part (b) shows an increase in the number of iterations by selecting edges with label ’a’, and (c) shows a decrease in the number of iterations by selecting edges with label ’b’. These are visible in the real-world datasets that we process.

In Section 4.1, we described a subset of the Extended Property Paths [41] language that we use in this work. We gave an example of a navigational query and its translation into a query that executes on the summary using SPARQL. We also showed how selective summaries can require a different number of iterations from the full summary. Next, we give an experimental evaluation that shows query performance improvement from using RDF summaries in unmodified SPARQL stores.

4.2 Experimental results

In Section 4.2.4, we show how summaries can improve query performance by showing the query evaluation plans of an EPPsI and its corresponding EPPsS. In this section, we give experimental evidence that using full or selective summaries instead of the dataset graph to answer navigational queries can improve query performance. In Section 4.2.1, we describe the datasets that we use in our experiments. In Section 4.2.3, we compare query performance improvement across different SPARQL systems. In Section 4.2.5, we compare query performance improvement across full and selective summaries. In Section 4.2.6, we show an evaluation of the construction time of full and selective summaries, and show that it equivalent to loading the RDF dataset plus writing the RDF summary. We include the space and time cost of storing full and selective summaries alongside a dataset graph in the SPARQL stores we use in our experiments.
4.2.1 Datasets

In this section, we describe the datasets and queries we use in our experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Instance</th>
<th></th>
<th>FWBW Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N$</td>
<td>$E$</td>
<td>$N$</td>
<td>$E$</td>
</tr>
<tr>
<td>dbp</td>
<td>48,603,466</td>
<td>317,220,816</td>
<td>32,274,111</td>
<td>278,182,230</td>
</tr>
<tr>
<td>d30</td>
<td>43,830,125</td>
<td>259,766,877</td>
<td>27,049,212</td>
<td>227,092,988</td>
</tr>
<tr>
<td>d13</td>
<td>32,087,836</td>
<td>175,567,871</td>
<td>19,317,195</td>
<td>156,948,367</td>
</tr>
</tbody>
</table>

Table 4.2: Instance and FWBW summary statistics for DBpedia full and selective summaries

We now describe the selective summaries of DBpedia that we consider in our experiments. Table 4.2 lists the number of nodes and edges in the dataset graph, and the corresponding number of blocks and block edges in the FWBW full and selective summary graphs. The dataset and FWBW summary graphs of the LinkedMDB and DBpedia datasets that we use for our query performance evaluations are described as part of Tables 3.1 and 3.2 in Section 3.3.1; dbp is included in Table 4.2 to relate its selective summaries. The d30 and d13 datasets represent selective summaries of DBpedia which we describe in Section 4.2.5.

We construct a selective summary from the dataset by summarizing only those statements whose predicate is on a given list. As shown in Table 4.2, we develop 2 selective summaries by choosing predicates from real-world scenarios and workloads as follows: d30 picks 28 distinct predicates that are present in real-world DBSB query logs and 2 closely-related predicates that do not appear in the query logs but can be navigated to from the 28 selected query log predicates; d13 picks 13 predicates present in the longest-running instance graph queries and is a subset of the d30 selective summary’s predicates. The table shows the number of statements from the dataset graph that are summarized. For example, although d13 uses only 13 predicates out of the 1,393 predicates in the full dataset, the d13 dataset still contains over 32 million edges and its resulting summary contains around 19 million blocks and around 157 million block edges, representing a 43% reduction in block edges in comparison to the full dbp summary.

In this section, we described the datasets we use to compare query performance. In the next section, we compare query performance of summaries using different SPARQL processors.

4.2.2 Selecting queries

In this section, we describe our approach to developing and selecting the LinkedMDB and DBpedia queries that we use in our experiments.

Since there are no existing benchmarks with navigational queries, we rely on meta-features described by WatDiv in order to develop navigational queries for the LinkedMDB and DBpedia datasets. WatDiv is a benchmark that addresses stress testing of RDF data management systems by generating data and queries based on structural features.

We consider the following structural features proposed by WatDiv to develop and select the queries we use in our experiments. A query’s triple pattern count refers to the number of triples in the query pattern. A query’s join vertex count refers to the number different variables that are the subject or object of one or more triple patterns. A join vertex’s degree refers to the number of times a variable appears as a subject or object. We use a join vertex’s type to refer to whether it appears only as singly-connected
leaf, as a leaf with fan-in or fan-out, or as both subject and object in triple patterns, which we represent using \( o/s, (o/s)^+ \) and traversals \( so \), respectively.

We also consider the following data-oriented features. We write queries using predicates that occur with high frequency in the instance dataset to increase the likelihood of summary-based reduction in the number of triples considered by any query. We choose not to parametrize the predicates used in our experiments’ queries in order to reduce the likelihood of picking highly-selective predicates along navigational paths. We select queries that range in the number of results from a few tens to tens of millions, and whose execution time ranges from sub-second to hundreds of seconds.

<table>
<thead>
<tr>
<th>Query</th>
<th>structure</th>
<th>predicates</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>LQ01</td>
<td>10 3 0</td>
<td>7</td>
<td>12,089</td>
</tr>
<tr>
<td>LQ02</td>
<td>45 2 2</td>
<td>43</td>
<td>1,411</td>
</tr>
<tr>
<td>LQ03</td>
<td>46 1 2</td>
<td>43</td>
<td>2,152</td>
</tr>
<tr>
<td>LQ04</td>
<td>3 3 0</td>
<td>2</td>
<td>485,649</td>
</tr>
<tr>
<td>LQ05</td>
<td>83 4 0</td>
<td>44</td>
<td>1,308</td>
</tr>
<tr>
<td>LQ06</td>
<td>8 3 0</td>
<td>4</td>
<td>43,549</td>
</tr>
<tr>
<td>LQ07</td>
<td>86 4 0</td>
<td>44</td>
<td>10,871</td>
</tr>
<tr>
<td>LQ08</td>
<td>108 4 0</td>
<td>52</td>
<td>303,895</td>
</tr>
<tr>
<td>LQ09</td>
<td>48 2 2</td>
<td>42</td>
<td>115,442</td>
</tr>
</tbody>
</table>

Table 4.3: Structure, number of predicates, and number of results for LMDB queries

Table 4.3 shows the structure of lmdb queries we use in our experiments. The queries vary in the join vertex types (leaves as \( o/s, (o/s)^+ \) and traversals as \( so \), whereby the summary itself was explored to develop the large set of valid predicates to include, such as query LQ08 which has over 108 variables and over 50 distinct predicates, and whose answer contains over 300K dataset graph nodes. We consider lmdb queries having as few as 1 triple pattern to queries having more than 100 triple patterns. We develop navigational queries that range in join vertex count, such as a range of vertex counts given by queries with different path lengths. In the lmdb queries, we vary the use of triple patterns in navigation towards results, as well as filtering based on the presence or absence of navigational paths. The lmdb queries we select range in the number of results from a few thousands to hundreds of thousands, and whose execution time ranges from sub-second to hundreds of seconds.

For the DBpedia queries that we select, we also consider the DBpedia SPARQL benchmark (DBSPB), which selects queries based on query logs. We consider this benchmark in order to vary our use of predicates based on real-world scenarios. We picked queries that vary in their popularity both in the DBpedia instance dataset, summary, and query logs. We give additional details of the predicates in the DBpedia queries in Appendix A.2.

In Table 4.4 we show the structure of the navigational queries we use with DBpedia. The table shows that the dbp queries that we use in our experiments have a mix of structural features including leaves, shown as \( o/s \) and \( (o/s)^+ \), and traversals, shown as \( so \). We consider queries that have as few as 1 triple pattern, to queries having up to 4 triple patterns. We use navigational queries that range in
Table 4.4: Structure, number of predicates, and number of results for DBpedia queries

<table>
<thead>
<tr>
<th>Query</th>
<th>structure</th>
<th>predicates</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQ01</td>
<td>2 0 2</td>
<td>3</td>
<td>1,983</td>
</tr>
<tr>
<td>DQ02</td>
<td>2 0 2</td>
<td>3</td>
<td>1,307</td>
</tr>
<tr>
<td>DQ03</td>
<td>3 0 2</td>
<td>3</td>
<td>414</td>
</tr>
<tr>
<td>DQ04</td>
<td>4 0 2</td>
<td>4</td>
<td>138</td>
</tr>
<tr>
<td>DQ05</td>
<td>2 0 2</td>
<td>3</td>
<td>30,509</td>
</tr>
<tr>
<td>DQ06</td>
<td>2 0 2</td>
<td>3</td>
<td>23,527</td>
</tr>
<tr>
<td>DQ07</td>
<td>3 1 1</td>
<td>4</td>
<td>2,263</td>
</tr>
<tr>
<td>DQ08</td>
<td>3 1 1</td>
<td>4</td>
<td>1,939</td>
</tr>
<tr>
<td>DQ09</td>
<td>2 0 0</td>
<td>1</td>
<td>4,108,026</td>
</tr>
<tr>
<td>DQ10</td>
<td>2 0 2</td>
<td>3</td>
<td>5,083</td>
</tr>
<tr>
<td>DQ11</td>
<td>1 2 0</td>
<td>2</td>
<td>3,846,352</td>
</tr>
<tr>
<td>DQ12</td>
<td>2 0 1</td>
<td>2</td>
<td>2,244,451</td>
</tr>
<tr>
<td>DQ13</td>
<td>2 0 0</td>
<td>1</td>
<td>753,369</td>
</tr>
<tr>
<td>DQ14</td>
<td>2 1 1</td>
<td>3</td>
<td>753,359</td>
</tr>
<tr>
<td>DQ15</td>
<td>3 1 1</td>
<td>4</td>
<td>751,304</td>
</tr>
<tr>
<td>DQ16</td>
<td>2 1 0</td>
<td>1</td>
<td>5,871,143</td>
</tr>
<tr>
<td>DQ17</td>
<td>2 1 1</td>
<td>2</td>
<td>12,149,641</td>
</tr>
</tbody>
</table>

join vertex count, such as a range of vertex counts given by queries with different path lengths. Like in LinkedMDB, we vary the use of triple patterns for navigation as well as for filtering. We select queries that range in the number of results from a few tens to tens of millions, and whose execution time ranges from sub-second to hundreds of seconds.

We note that we are of (SPB) [8], a benchmark designed to address key technical challenges of RDF systems using a data publishing use case. Although SPB provides a benchmark which generates queries, and from which we can obtain navigational fragments as we have done for LinkedMDB and DBpedia, we have instead opted to use the structural features described in WatDiv. Specifically, we choose to use real-world datasets and non-parametrized queries, applying complex features for LinkedMDB and simpler features (based on query logs) for DBpedia.

### 4.2.3 Comparing SPARQL engines

In this section, we compare query performance of dataset graphs and full summaries with SPARQL processors.

We prepare different RDF systems and execute all queries on a compute server M3 having 2 E5355 (4-core) CPUs and 28 GB of main-memory. To compare query performance of structural summaries on different systems, we compare lmdb query performance on a system loaded with the dataset and summary graphs. So, we create two installs for each system: one install has only the dataset graph loaded, and the other install has both the dataset and structural summary loaded. We then compare
performance and system statistics.

We select queries taken from real-world scenarios, including user manual exploration and query logs, with a range of execution times, result cardinalities, and structural features. We express our dataset queries as EPPsI that are translated to summary queries via an EPPsItoEPPsS translation, both of which undergo an EPPstoSPARQL translation in order to execute on unmodified SPARQL processors.

Our query evaluation methodology is as follows. First, we load all dataset combinations once, e.g., dataset graph with structural summary graph. Second, we prepare a batch file containing EPPs queries, which the application translates into SPARQL queries. We execute two types of queries: dataset queries that return a single result giving the count of matched nodes, and their corresponding summary queries that also returns the count of matched nodes. We choose these two types of queries as a way to mimic measurement the time for a query to return all answers. Third, the application executes the set of prepared queries in sequence against a SPARQL processor and we collect query times and result answers. Fourth, after all the prepared queries of a given type complete execution, the triple store and benchmark application are shutdown as a way to hard reset their memory allocations (via JVM or otherwise). Next, the process repeats at the preparation for the next system.

<table>
<thead>
<tr>
<th>Query</th>
<th>v-inst</th>
<th>v-sum</th>
<th>f-inst</th>
<th>f-sum</th>
<th>s-inst</th>
<th>s-sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>LQ01</td>
<td>0.84</td>
<td>0.76</td>
<td>36.2</td>
<td>31.5</td>
<td>24.9</td>
<td>34.2</td>
</tr>
<tr>
<td>LQ02</td>
<td>1.0</td>
<td>1.1</td>
<td>6.0</td>
<td>4.9</td>
<td>5.6</td>
<td>5.5</td>
</tr>
<tr>
<td>LQ03</td>
<td>1.0</td>
<td>1.1</td>
<td>10.6</td>
<td>8.1</td>
<td>8.8</td>
<td>8.7</td>
</tr>
<tr>
<td>LQ04</td>
<td>1.9</td>
<td>0.88</td>
<td>30.1</td>
<td>27.8</td>
<td>27.6</td>
<td>25.4</td>
</tr>
<tr>
<td>LQ05</td>
<td>2.8</td>
<td>2.9</td>
<td>160</td>
<td>137</td>
<td>55.6</td>
<td>63.8</td>
</tr>
<tr>
<td>LQ06</td>
<td>3.4</td>
<td>2.0</td>
<td>193</td>
<td>161</td>
<td>132</td>
<td>126</td>
</tr>
<tr>
<td>LQ07</td>
<td>11.8</td>
<td>5.0</td>
<td>506</td>
<td>378</td>
<td>231</td>
<td>209</td>
</tr>
<tr>
<td>LQ08</td>
<td>79.1</td>
<td>62.4</td>
<td>683</td>
<td>458</td>
<td>380</td>
<td>285</td>
</tr>
<tr>
<td>LQ09</td>
<td>113</td>
<td>15.5</td>
<td>951</td>
<td>741</td>
<td>921</td>
<td>793</td>
</tr>
</tbody>
</table>

Table 4.5: Query time (in secs) with lmdb on Virtuoso (v-), Fuseki (f-), and Sesame (s-) with dataset (-inst) and summary (-sum) graphs

Table 4.5 shows the time (in secs) to execute lmdb queries using 3 SPARQL processors. We configure each SPARQL processor as follows. We use v- to refer to Virtuoso [39 [38] open-source edition version 7.10.3207 which has its numberOfBuffers set to 500,000. We use f- to refer to Jena-TDB [64] 2.12.0 with a Fuseki 1.1.0 HTTP interface with a Java runtime heap size of 12GB. We use s- to refer to Sesame [110] 2.7.13 with its Tomcat 8.0 HTTP interface and which is set to use the native storage type with a Java runtime heap size of 12GB. The table shows query times for each system when querying against the dataset (columns with suffix -i) versus querying against the summary graph (columns with suffix -s). For example, on Virtuoso, LQ09 takes around 113s with the dataset (v-i), and only 15s with the summary (v-s). On Fuseki and Sesame, the same query takes 950s and 920s, respectively, with the dataset, and 741s and 793s, respectively, with the summary graph, and constitutes a performance improvement of 28% and 16%, respectively. Our results show that executing queries with sub-second to hundreds of seconds execution time on all systems is generally faster with the summary than executing with the dataset. Our results also show that Virtuoso can be more than an order of magnitude faster than Fuseki
Figure 4.2: Query performance ratio of summary versus instance on Virtuoso \((v-)\), Fuseki \((f-)\), and Sesame \((s-)\) systems

and Sesame. Summaries can answer queries faster than on the instance since a SPARQL processor generates fewer intermediate result rows while answering a query using a summary that contains fewer possible rows on which to join; we show this in Section 4.2.4. This effect is further accentuated with different types of summaries, which we show in Section 4.2.5.

Figure 4.2 compares the ratio of query performance with \(v\)-ratio, \(s\)-ratio, and \(f\)-ratio. The ratios show that Sesame provides the least benefit from summaries (averaging less than 4% across the shown queries), and even has a 27% performance decrease in comparison to the dataset with query LQ01. We also see that Fuseki provides performance improvement for all queries. There is clear evidence that using summaries with Virtuoso results in more than 25% performance improvement for most of the queries, with queries LQ04 and LQ07 with more than 2x faster, and query LQ09 at 7.25x faster. We make note of the fact that, even though Fuseki has a better improvement ratio across all queries when using structural summaries, Sesame is often faster than Fuseki for executing queries on the dataset or summary. Our results show that Virtuoso’s stable and mature query processing framework benefits greatly from answering queries on a summary stored as rows in its underlying relational database. Virtuoso is able to leverage processing fewer intermediate result rows to improve query performance by over 7x.

<table>
<thead>
<tr>
<th>Graphs</th>
<th>Fuseki</th>
<th>Sesame</th>
<th>Virtuoso</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>2.5GB (177)</td>
<td>485MB (205)</td>
<td>314MB (49)</td>
</tr>
<tr>
<td>I+S</td>
<td>3.5GB (934)</td>
<td>864MB (665)</td>
<td>534MB (113)</td>
</tr>
</tbody>
</table>

Table 4.6: Disk space usage (with load time in secs in parentheses) of lmdb for Fuseki, Sesame, and Virtuoso with dataset \((I)\) and summary graphs \((I+S)\)

To show the impact of structural summaries on SPARQL processors, we compare the disk space and time cost of loading and storing just the lmdb dataset with the disk space and time cost of loading both lmdb dataset and FWBW summary. Table 4.6 shows the size of the lmdb dataset in each of the Fuseki, Sesame, and Virtuoso systems. For each system, two sizes are shown: the size of the system after loading just the dataset graph \((I)\), and the size with the system after loading both the dataset graph and structural summary \((I+S)\). For example, the lmdb dataset on Virtuoso takes 314MB of disk space,
and its summary increases total disk space usage (which includes the dataset) to 534MB. While Fuseki has the least increase in factor, it also uses up to 5x more disk space than its counterparts, Virtuoso and Sesame. The table shows that, regardless of dataset and system, loading the summary alongside the dataset increases disk usage by up to 1.8x. Table 4.6 also shows the time taken to load the dataset on each system and the time taken to load both the dataset graph and the summary graph on a system. For example, it takes 49s to load the lmdb dataset in Virtuoso, and an additional 64s to load the summary. The times show that the time to load a summary is within a small factor of the time to load the dataset, with Virtuoso having the smallest factor of 1.3x.

In this section, we showed that summaries can improve query performance. Our results in Table 4.6 show that Virtuoso, a stable and mature query processing framework with an underlying relational database, has more than a 7x query performance improvement, uses the least amount of secondary-storage space of the SPARQL processors we consider, and appears least likely to exhibit effects of disk access. In light of this fact, in the next section, we compare Virtuoso’s query evaluation plans of an EPPsI and EPPsS and show how summaries optimize queries.

### 4.2.4 Comparing query plan evaluation

In this section, we show how LinkedMDB’s FWBW summary can improve query performance by comparing the query evaluation plans of an EPPsI and its corresponding EPPsS. In particular, we examine the number of statements for each predicate that participates in a query, the number of intermediate rows in a query’s evaluation, and the evaluation order of predicates to show that, since a summary can only have as many block edges as its instance graph and often as little as 70% such as for LinkedMDB’s FWBW summary, the EPPsS will evaluate fewer intermediate rows than its EPPsI. Our results show that EPPsS expressions can have fewer intermediate rows during query evaluation despite changes in predicate evaluation order.

We examine LinkedMDB query LQ01 from Table 4.5 which has the following EPPsI; the query navigates from ?actor nodes to reachable ?writer nodes while considering predicates such as :rdf:type, :performance, and :page.

```sparql
?actor (:performance && TP(s,rdf:type) && TP(s,:page))/
((o :performance s) && TP(s,rdf:type) && TP(s,:page))/
( :writer && TP(s,rdf:type) && TP(s,:producer)
&& TP(s,:page) && TP(s,:actor) && TP(s,:director)) ?writer
```

We obtain query plans using Virtuoso [39, 38] open-source edition version 7.10.3207 which has its NumberOfBuffers parameter set to 5,000,000, and its QueryLog parameter set to a file to which Virtuoso appends query execution plans; we have occasionally found it helpful to examine query execution plans that give row estimates with the use of ‘SET SPARQL_TRANSLATE ON;’ and ‘SET EXPLAIN ON;’ in Virtuoso’s isql interface prior to submitting SPARQL queries as ‘SPARQL <sparqlquery>’. We load a single Virtuoso instance with the LinkedMDB instance graph and summary graph with two different graph URIs. We obtain the EPPsI and EPPsS query plans of LQ01’s evaluation on Virtuoso from the query log; in Appendix A.3, we show Virtuoso’s complete EPPsI and EPPsS query execution plans for LQ01.
Table 4.7: For each predicate of LinkedMDB query LQ01 of Table 4.5, we show the number of statements in the graph (suffix -stmts), the number of intermediate rows in Virtuoso’s query evaluation (-rows), and the order in which Virtuoso evaluates predicates (-order) in the EPPsI (prefix i-) and EPPsS (s-)

<table>
<thead>
<tr>
<th>Predicate</th>
<th>i-stmts</th>
<th>s-stmts</th>
<th>i-rows</th>
<th>s-rows</th>
<th>i-order</th>
<th>s-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>actor</td>
<td>245,252</td>
<td>177,817</td>
<td>1,377,690</td>
<td>689,525</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>director</td>
<td>70,336</td>
<td>44,582</td>
<td>178,916</td>
<td>44,582</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>page</td>
<td>512,943</td>
<td>355,600</td>
<td>268,357</td>
<td>145,703</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>page</td>
<td></td>
<td></td>
<td>268,357</td>
<td>145,703</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>page</td>
<td></td>
<td></td>
<td>414,435</td>
<td>181,346</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>performance</td>
<td>587,593</td>
<td>354,876</td>
<td>168,216</td>
<td>144,558</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>performance</td>
<td></td>
<td></td>
<td>290,678</td>
<td>180,184</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>producer</td>
<td>50,270</td>
<td>32,171</td>
<td>314,065</td>
<td>162,695</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>type</td>
<td>740,468</td>
<td>492,609</td>
<td>145,599</td>
<td>144,558</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>type</td>
<td></td>
<td></td>
<td>145,599</td>
<td>144,558</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>type</td>
<td></td>
<td></td>
<td>435,757</td>
<td>270,059</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>writer</td>
<td>45,884</td>
<td>26,687</td>
<td>28,549</td>
<td>252,645</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>TOTAL</td>
<td>2,252,746</td>
<td>1,484,342</td>
<td>4,036,218</td>
<td>2,506,116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extent</td>
<td></td>
<td></td>
<td>1,327,120</td>
<td>12,089</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7 lists, once for each distinct predicate in LQ01 (in alphabetical order), the number of corresponding statements in the instance graph (i-stmts) and summary graph (s-stmts), respectively; the summary graph reduces the number of statements as compared to the instance graph. The most reduction occurs with predicate rdf:type with a reduction of 247,859 statements, and the least reductions occur with predicates :writer and :producer with under 20K reduction in each of their respective number of statements. The difference in the total number of statements between the summary and instance graphs is more than the number statements for any individual predicate, and the total number of statements in the summary graph that can participate in any query evaluation is 35% less the total of the instance graph. This means that in principle, aside from the predicate bc:extentContains in the EPPsS, evaluating predicates in the EPPsS in the same order as in the EPPsI should reduce the number of intermediate rows in of the EPPsS. On the other hand, even though including the count of 1,327,120 extent edges (which corresponds to an extent edge for each instance node in the dataset graph) in an EPPsS evaluation increases the total number of statements that can participate in a query plan to 25% more than that of an EPPsI, since our EPPsS translation selects only distinct blocks and since each instance node belongs to exactly one block, each extent edge of a distinct block can appear as exactly one intermediate row in any execution plan.

Table 4.7 lists, for each predicate in LQ01, the number of intermediate rows in Virtuoso's evaluation of the EPPsI (i-rows) and EPPsS (s-rows). For example, the predicate :actor results in 1,377,690 intermediate rows in the EPPsI and 689,525 intermediate rows in the EPPsS. Except for :writer, all EPPsS predicates traverse fewer intermediate rows; the predicate :actor has a reduction of 688,165 rows, and which represents 17% of the total rows the EPPsI traverses. For the predicate :writer, the EPPsS traverses 9x more rows than the EPPsI; however, the 252,645 rows that the EPPsS traverses is still 33% fewer than the average number of rows the EPPsI traverses across all predicates. We calculate the total number of intermediate rows in the evaluation of the EPPsI and the EPPsS and note that the EPPsI traverses 60% more rows than the EPPsS. The EPPsI returns 12,089 distinct writers, which means that the EPPsS traverses the same number of extent edges (from distinct blocks) to the result writers. In
relation to the number of result writers, the EPPsI traverses 28x more rows and the EPPsS traverses 17x more rows.

Table 4.7 also shows the order of predicates in Virtuoso’s query plan for the EPPsI (i-order) and EPPsS (s-order). The table shows that only predicate :performance occurs in the same position in both the EPPsI and EPPsS execution plans, showing that the query plan’s predicate order between an EPPsI and EPPsS can change drastically. The table also shows that the predicate :writer predicate occurs first in the EPPsI’s execution plan and occurs last in the EPPsS’s execution plan, and which is then the last predicate evaluated by the EPPsS before evaluating predicate bc:extentContains. This example shows that the order of predicate evaluation can change between queries return the same result and can affect the number of intermediate rows during query execution.

Figure 4.3 shows the predicates of LQ01 (in alphabetical order). The figure shows, once for each distinct predicate, the ratio of number of statements in the summary graph over the number of statements in the instance graph (ratio-stmts). The figure shows that, for all predicates in LQ01, the summary graph has over 25% reduction in the number of statements compared to the instance graph. Figure 4.3 also shows, for each predicate of LQ01, the ratio of intermediate rows in Virtuoso’s query plan for the EPPsS over that of the EPPsI (ratio-rows). The figure shows that the EPPsS traverses as few as 25% of the number of rows as the EPPsI, such as with predicate :director. The figure shows that there is a reduction in the number of intermediate rows when executing the EPPsS as compared to the execution of the EPPsI.

An instance graph’s summary cannot have more block edges than the number of edges in the instance graph, and we have shown that a summary can have as few as 25% of an instance’s statements for a predicate. This leads to the possibility of using summaries to answer navigational queries instead of instance graphs since summaries offer the possibility of reducing the number of intermediate rows during query evaluation; we expect this to be the case if a SPARQL processor’s query optimizer processes an EPPsS’s predicates in the same order as its EPPsI. Even if a summary, in the extreme case, contains only singletons then the summary will have the exact same number of block edges as there are edges in its instance graph. Though we could hope that a query processor reuses a predicate evaluation
order of an EPPsI for evaluation of an EPPsS, the reality is that an EPPsS execution plan, and thus potential performance benefits of summary graphs, is dependent on a given SPARQL processor’s query optimizer, and which we have shown can choose a different plan. We report anecdotal work on different EPPstoSPARQL translations of EPPsS expressions that produce different (equivalent) query plans with different performance. Thus the benefit of using summaries relies on the effectiveness of query optimizers, and though we have described the benefits of summaries in the context of Virtuoso’s query optimizer, our work opens the opportunity for integrating further summary-based optimizations in SPARQL query processors.

In this section, we compared Virtuoso’s query evaluation plans of an EPPsI and EPPsS and showed that summaries can optimize queries by reducing the number of intermediate rows, despite a change in predicate evaluation order.

In the next section, we use Virtuoso to compare selective summary performance.

4.2.5 Comparing selective summaries

In this section, we compare query performance of a full structural summary that includes all of a dataset graph’s predicates with query performance of selective structural summaries that are smaller than a full summary since they consider fewer, more relevant, predicates. The motivation for using fewer predicates is that, since a structural summary aims to improve navigational query performance, then a summary that focuses on specific traversals, such as the predicates from a workload, can potentially provide equal or better query performance while decreasing the time and space cost of summary construction.

We construct two selective summaries of DBpedia: d30 which summarizes the dbp instance dataset using only 30 predicates, and d13 which summarizes the dbp instance using only 13 predicates (and which are subset of d30’s predicates). The predicates we pick for the selective summaries are based on DBpedia’s query logs [86]. Table A.2 of Appendix A.2 shows the predicates that we have chosen to summarize for d30 and d13. The table shows each predicate’s associated count of statements in the instance dataset (inst), the count of block edge statements in the full, d30, and d13 summary graphs, and the count of queries in the DBSB query logs that contain each predicate (qlog). The predicates picked for the d30 and d13 selective summaries differ dramatically in their popularity in the query logs and instance, ranging by orders of magnitude, in order to target query performance of real-world scenarios.

We compare performance of DBpedia selective summaries using Virtuoso on a single machine M1 that has 8 Xeon X6550 2GHz 8-core CPUs and with over 100GB of RAM. The machine’s operating system is Ubuntu 14.04.3 LTS.

To compare query performance of DBpedia’s selective summaries, we install three systems; each system contains the full dbp dataset, and one of the dbp, d30, or d13 summaries. In order to consider the impact of querying the dataset with each type of summary, and since storing a summary alongside a dataset may incur a penalty such as due to increased index storage, we load the full dbp dataset alongside each summary in the SPARQL processor.

Table 4.8 shows the time (in secs) of querying DBpedia’s full (dbp) and selective (d30,d13) datasets and summaries on Virtuoso. For example, query DQ17 completes in 506s on the system that contains the dbp dataset alone, and 367s with the full summary on a system containing both the full summary and the dataset. As well, the same query completes in 354s with the d30 selective summary and 368s with the d13 selective summary. Our results show that selective summaries improve performance of
Table 4.8: Query time (in secs) of DBpedia instance, dbp summary, and d30 and d13 selective summaries

<table>
<thead>
<tr>
<th>Query</th>
<th>dbp</th>
<th></th>
<th></th>
<th></th>
<th>dbp</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inst</td>
<td>sum</td>
<td>inst</td>
<td>sum</td>
<td>inst</td>
<td>sum</td>
<td>inst</td>
<td>sum</td>
</tr>
<tr>
<td>DQ01</td>
<td>1.1</td>
<td>0.85</td>
<td>1.1</td>
<td>0.90</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ02</td>
<td>1.4</td>
<td>0.53</td>
<td>1.3</td>
<td>0.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ03</td>
<td>1.5</td>
<td>1.0</td>
<td>1.5</td>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ04</td>
<td>1.9</td>
<td>1.0</td>
<td>1.7</td>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ05</td>
<td>1.9</td>
<td>0.85</td>
<td>1.0</td>
<td>0.54</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ06</td>
<td>2.4</td>
<td>0.90</td>
<td>1.7</td>
<td>0.82</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ07</td>
<td>2.6</td>
<td>2.1</td>
<td>2.5</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ08</td>
<td>3.4</td>
<td>2.7</td>
<td>3.4</td>
<td>2.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ09</td>
<td>7.4</td>
<td>4.8</td>
<td>7.9</td>
<td>4.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ10</td>
<td>9.0</td>
<td>2.0</td>
<td>8.9</td>
<td>1.9</td>
<td>9.1</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ11</td>
<td>12.7</td>
<td>8.7</td>
<td>11.0</td>
<td>6.7</td>
<td>13.6</td>
<td>8.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ12</td>
<td>15.4</td>
<td>6.6</td>
<td>15.1</td>
<td>5.7</td>
<td>14.7</td>
<td>4.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ13</td>
<td>15.5</td>
<td>7.6</td>
<td>16.1</td>
<td>7.4</td>
<td>16.2</td>
<td>7.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ14</td>
<td>49.3</td>
<td>36.3</td>
<td>50.8</td>
<td>36.8</td>
<td>50.6</td>
<td>35.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ15</td>
<td>64.5</td>
<td>49.8</td>
<td>64.6</td>
<td>49.5</td>
<td>68.3</td>
<td>53.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ16</td>
<td>69.3</td>
<td>6.4</td>
<td>61.9</td>
<td>3.8</td>
<td>69.3</td>
<td>4.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ17</td>
<td>506</td>
<td>367</td>
<td>506</td>
<td>354</td>
<td>526</td>
<td>368</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

queries that have a range of execution times, from sub-second times to hundreds of seconds.

We exclude queries with spurious query performance. In one case, the time to query the dataset more than doubles when we load a summary alongside the dataset. In another case, an dataset query takes only tens of milliseconds with the dbp-p13 summary but takes 2min otherwise. Deeper analysis into the query plans of affected dataset and summary queries reveals that query plans can change between systems, likely due to changes in statistics on which Virtuoso’s query optimizer relies.

In Figure 4.4 we show the query performance ratio of DBpedia’s full (dbp-ratio) and selective summaries (d30-ratio, d13-ratio) in comparison to the dataset. The query performance of the full and selective summaries are almost the same. We also see one case (DQ16) where the d13 summary provides 17x performance improvement in comparison to 10x for the full summary. The reason why selective summaries improve query performance more than the full summary is because a selective summary has potentially fewer block edges, and thus fewer rows in the underlying relational database, on which to join and generate intermediate results during query processing as compared to a full summary. We have shown in Figure 4.2 that the d13 summary has less than 60% of the number of edges of the full summary, and this in turn corresponds to the observed improvement in query performance with the d13 summary. Our results show that selective summaries targeting different workloads are effective at improving query performance the same or better than the full summary.

Table 4.9 shows the disk usage (in GB) and load time (in mins) for the dbp dataset graph (I), as well as full DBpedia dataset with each of the full (full) and selective (d30, d13) summary graphs (I+S)
Chapter 4. Summary-based Query Optimization

Figure 4.4: Query performance ratio of summary versus instance for DBpedia dbp summary, and d30 and d13 selective summaries

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I</td>
<td>I+S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst</td>
<td>full</td>
<td>d30</td>
<td>d13</td>
<td></td>
</tr>
<tr>
<td>9.2 (50)</td>
<td>19.8 (380)</td>
<td>17.8 (189)</td>
<td>15.8 (99)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.9: Virtuoso disk space usage (in GB) with load time (in mins in parentheses) of dbp dataset graph (I), plus full and selective summaries (I+S)

with Virtuoso. We notice that the time taken to load the dataset graph and full summary takes 7.6x longer than to load only the dataset graph, even though disk space usage is doubled; however, this implementation uses a single-thread to load similar to other systems, and loading block edges and extent edges in parallel is much faster. On the other hand, the d13 selective summary is an attractive summary to use since loading the dataset graph and the d13 summary alongside each other barely doubles the load time of just the dataset graph, and uses only 71% more disk space.

In this section, our results show that it is possible to use selective summaries, at less than double the cost of using the dataset alone, with the potential of improving query performance by more than 17x.

4.2.6 Full and selective summary construction

In Section 3.3.2, we showed that our S variation constructs full summaries faster than the time to load an RDF dataset plus write its full RDF summary. In this section, we give evidence that shows our S variation constructs selective summaries faster than the time to load an RDF dataset plus write its selective RDF summary.

We execute our experiments using M1, a single machine that has 8 Xeon X6550 2GHz 8-core CPUs, and which can process 64 vertices in parallel. We allocate 80GB of main-memory to the Java 8 JVM and execute our summary construction algorithm using unmodified GraphChi 0.2.
Table 4.10: Time (in mins) for RDF dataset load, full and selective summary construction with B and S variations, and RDF summary write

<table>
<thead>
<tr>
<th>Summary</th>
<th>Load</th>
<th>B</th>
<th>S</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbp</td>
<td>107</td>
<td>775</td>
<td>187</td>
<td>207</td>
</tr>
<tr>
<td>d30</td>
<td>160</td>
<td>611</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>d13</td>
<td>131</td>
<td>411</td>
<td>95</td>
<td>95</td>
</tr>
</tbody>
</table>

Table 4.10 shows the time (in mins) to load the RDF dataset into GraphChi’s internal format, construct its FWBW summary using the B or S variation, and then write the summary as RDF; the times shown are the average of 5 runs excluding the min and max. The S variation is 4.1x, 4.8x and 4.3x faster than B variation for the full dbp, d30, and d13 summary constructions, respectively. Our results also show that computing the summary is 1.7x, 2.3x, and 2.4x faster than loading the RDF dataset plus writing the RDF summary for the full, d30, and d13 summaries, respectively. The load time for the d30 and d13 summaries are higher than the full summary due to additional processing to filter statements based on a given predicate list. The results show the S variation’s effectiveness as a result of skipping a majority of the summary’s blocks, which are singletons.

Figures 4.5 and 4.6 show the FWBW summary constructions for the d13 and d30 selective summaries; the FWBW summary construction for the full dbp summary is shown in Figure 3.14. We observe that the d13 selective summary takes 12 iterations, while the d30 summary takes 11 iterations, both in contrast to the 17 iterations needed to compute the dbp full summary. The B variation’s per-iteration time peaks for the full summary at iteration 4, while it peaks earlier at iteration 2 for d13 and d30. The S variation’s per-iteration time for all three summaries reaches its highest at iteration 2 and decreases substantially...
in iteration 3, the earliest moment in which the singleton optimization can skip singletons. We notice that in all three summaries the majority of singletons are found by iteration 3 and that this matches the low per-iteration times starting in iteration 5, and continuing for the remainder of the construction. For all iterations, each per-iteration time of d13 is lower than for the same iteration on d30, and similarly, each per-iteration time of d30 is lower than for the same iteration on dbp full. The figures also show the number of singleton blocks in each iteration of the summary’s construction. We notice that within the first few iterations of the d13 and d30 constructions, the number of non-singleton blocks is dwarfed by the number of singletons. In the last iteration, over 82% of the each of dbp, d30, and d13 summary’s blocks are singletons. This has the impact of reducing, for all three summaries, the S variation’s per-iteration to under 16% of the B variation’s per-iteration time. We also note that d30’s B and S variations take longer to compute than d13 with the B and S variations, respectively, even though d30 requires fewer iterations. Notice that a selective summary is not guaranteed to require fewer iterations to compute. In particular, recall that d13’s predicates are a subset of d30’s but that d13 requires more iterations to compute.

Figure 4.7 (i), (ii), and (iii) show the block sizes in the DBpedia full, and d30 and d13 selective summaries, respectively. Each figure shows the number of blocks having a given block size, ordered by increasing block size. In each summary, there are an order of magnitude more singleton blocks than the block size with the second-largest count. The full summary has one block with the largest block size of 83,389, d30 has one block with the largest block size of 260,864, and d13 has one block with the largest block size of 83,391. Overall, despite the fact that more than half of the dataset graph’s nodes are singletons, the remainder are non-singleton blocks and so our summaries can still optimize queries.

Our results show that our singleton optimization is effective at computing summaries of large graphs, and is particularly efficient with selective summaries, in terms of computations, storage, and query performance.

4.3 Summary

In this chapter, we described our S+EPPs architecture that allows answering navigational queries using structural summaries instead of dataset graphs on existing unmodified SPARQL stores. We gave an example of our S+EPPs approach that answers navigational queries using EPPsItoEPPsS and EPPstoSPARQL translations. Our reported evaluations showed that construction time of selective summaries is faster than that of the full summary, and we also showed that our S variation continues to construct selective summaries in an amount of time equivalent to input load plus summary write. Furthermore, we showed evaluations of the query performance improvement of our FWBW summaries with a variety of unmodified RDF stores, real-world big graphs, and real-world query scenarios. We further showed that storing summaries alongside dataset graphs increases load and storage costs by a small factor. We describe, and experimentally show, that selective summaries have the property of possibly requiring a different number of iterations to compute. Overall, our results show that our S+EPPs approach can improve query performance with selective summaries more than a full summary while being faster and smaller to compute.
Chapter 4. Summary-based Query Optimization

(a) Count of block sizes in dbp full FWBW summary

(b) Count of block sizes in d30 FWBW selective summary

(c) Count of block sizes in d13 FWBW selective summary

Figure 4.7: Blocks sizes in DBpedia full and selective summaries
Chapter 5

Ontology for Structural Summaries

The work of Chapter 4 shows the benefit of using RDF summaries to answer navigational queries translated to SPARQL. Having a summary ontology adds value to our approach of using RDF summaries such as for verification of logical constraints. In this chapter, we describe an ontology for representing structural summaries using the OWL 2 format. The OWL 2 summaries can work with existing, unmodified semantic web tools, such as for verification using automated reasoning.

Billions of descriptions are generated by companies, individuals, and recent paradigms, such as Internet-of-Things, and made available online for linking and reuse. For example, there are billions of descriptions in the LOD Cloud, a community-contributed semantic web that is being touted as a next step of the online web with the use of URIs to uniquely address nodes and their semi-structured descriptions. Even DBpedia, one of its larger datasets, contains around 40 million nodes describing real-world nodes such as people and places. Furthermore, LinkedMDB, which contains descriptions about movies, actors and directors, has over 30K links to DBpedia and represents only a portion of LinkedMDB’s 700K interlinks to more than 10 other datasets in the LOD Cloud.

As the number of descriptions continues to grow, it is a challenge to understand how to interlink and reuse large datasets. Predicates and classes that are used in descriptions are often prescribed by schemas but provide little insight to how they are used in practice, particularly with many choices of schema. One way for a user to understand descriptions for reuse or interlinking is to manually explore a dataset, but using an RDF browser to explore linked data can be a tedious, time-consuming exercise for large datasets.

Queries can be used to match and return subgraphs of interest within instance data but this often involves a trial-and-error approach on a schema-conformant dataset that tests for permissible structures but quickly becomes complicated if a dataset uses multiple schemas. Even if a user knows how a schema is used, it is a daunting task to distinguish millions of potentially equivalent descriptions returned by a query. In the case of writing queries over an instance dataset, a group-by operation could be used to avoid exploring non-unique semi-structures – a complex task that, at the implementation-level, also involves a columnar representation which forgoes easy reuse by semantic web tools. In the LOD Cloud, since descriptions can be complex due to a lack of fixed schema and freedom to interlinks datasets, a scalable and effective approach is needed to understand and describe unique descriptions of large datasets.

An easier way to understand a large dataset is to group all nodes that have equivalent descriptions,
a *structural summary*. For example, a user interested in unique descriptions of musicians can use a structural summary to group all those musicians that are described using exactly the same predicates and classes. An advantage of structural summaries is that since they capture only distinct descriptions they are often smaller than the original instance, and thus easier to explore, formulate queries, and understand their results.

Our ontology captures unique semi-structures of datasets regardless of which schemas are in use and enables the use of structural summaries with existing semantic web tools. One application of using a structural summary within existing tools is for summary-based optimizations to improve query performance [30]. Structural summaries were constructed and stored as semantic web datasets, and their use in semantic web store allowed query improvements of up to 17x in comparison to using the instance dataset. In the next section, we describe features that our structural summary ontology is to designed to support, followed by a description of our ontology that is used to represent structural summaries of large datasets, in particular, those from the LOD Cloud.

In Section 5.1, we describe the requirements for our summary ontology using competency questions. Our requirements for our summary ontology include describing a summary, its relationships to an instance dataset, as well as relationships between summaries. In Section 5.2, we define the summary’s classes, properties, and constraints that are needed in order to express a summary that can satisfy our requirements. Specifically, the ontology we give in this chapter allows describing a summary’s blocks and block edges, the extent relationships between blocks and instances, as well as relating multiple summaries together such as during iterative summary construction. In Section 5.3, we describe how our summary ontology satisfies our competency questions (Section 5.3.1 specifically); verification that a structural summary correctly groups nodes having equivalent descriptions is beyond the scope of this work. In particular, we give two examples in which we express and verify logical constraints of a summary represented using OWL 2 notation. The first example verifies the relationship between a summary and its original instance by verifying that a summary partitions an instance dataset’s nodes. The second example verifies the relationship between summaries by verifying that one summary is a refinement of another. In Section 5.4, we characterize the ontology’s development along the ontology’s semantic and pragmatic dimensions [92], and these describe the ontology using notions such as the ontology’s use in reasoners, a semantic dimension, as well as the ontology’s degree of granularity, a pragmatic dimension. In Section 5.5, we end the chapter with a review of our contributions.

## 5.1 Competency Questions

We now motivate our proposed summary ontology by considering several applications based on our experiences [30] with a focus on two particular scenarios: exploration, and query optimization.

**Exploration use case.** A user interested in understanding the unique descriptions of a dataset can write queries for summaries in order to find and understand unique descriptions, and avoids the exploration of potentially many equivalent descriptions.

**Query optimization use case.** Queries over the instance are answered using the summary instead by using simple translations. In existing work, we use queries that are expressed using a custom path language which are then translated to SPARQL queries to execute over a summary expressed as a se-
mantic web dataset. By storing a summary in such a way, summaries act as an efficient index within existing systems by applying simple translations to queries.

To determine the scope and requirements of our ontology, we have develop a set of competency questions \[49\] that our ontology should be able to answer.

CQ1. Do two instances have equivalent descriptions?

CQ2. What are all of the unique descriptions that exist in the instance dataset?

CQ3. What is the relationship between different descriptions from the same summary?

CQ4. Is one summary a refinement of another?

CQ5. Which nodes have a given description?

CQ6. How are descriptions in different summaries related?

In earlier work, we have shown that ExpLOD’s \[67\] usage summaries are a flexible FWBW summary for RDF data. A usage summary groups equivalent entity descriptions such as by schemas or attributes used in an entity’s description. For example, one usage summary can group entities by the types that they are an instance of, as well as attributes that describe them. Another type of usage summary can group entities by types and attributes based on the domain that contributed them, showing the unique ways that providers describe entities. Usage summaries are an accurate way to identify all the unique descriptions and is an effective tool for analyzing how semantic web datasets are interlinked.

To our knowledge SchemeX \[74\] is the only system that uses ontology-based summaries for understanding and querying RDF datasets. In particular, SchemeX uses VoID \[3, 4\], a vocabulary that allows to formally describe RDF datasets and linking between RDF datasets, and constructs FW summaries based on combinations of classes and predicates that nodes are directly connected to. That is, SchemeX’s VoID-based summaries represent groups of instances having equivalent description based on the set of classes that they are an instance of and the sets of properties that describe them.

The ExpLOD and SchemeX applications demonstrate the need for a summary representation to group instances with equivalent descriptions (competency question CQ1). ExpLOD addresses the need to identify all unique descriptions in an instance dataset, while SchemeX focuses on capturing all the unique descriptions in batches of streaming data. However, both ExpLOD and SchemeX group instances based on their description that essentially does not traverse more than one step away from an instance. Furthermore, the VoID ontology itself does not have a way to represent summaries that traverse more than one step away from a node, thus we propose a summary ontology that can express all possible unique (structural) descriptions in a dataset (competency question CQ2). As well, SchemeX’s VoID-based representations of FW summaries can only answer queries that have exactly one step in a navigational query. Since both ExpLOD and SchemeX do not describe equivalence of instances beyond more than one step we seek a summary representation that can perform answer queries that have navigations of more than one step as well as navigations that traverse edges in both forward and backward directions (competency question CQ3).

As we have described in Chapter 3 and as seen in existing approaches \[19, 65, 94\], summary construction can occur as an iterative computation; essentially, each iteration increases the number of steps away from instance nodes by which a summary determines node equivalence. For this reason, we desire a
summary ontology that can represent summaries that group instances based on some form of refinement (competency question [CQ4]. We have shown that summaries can improve query performance of navigational queries on unmodified SPARQL processors (competency question [CQ5]. Also, we have shown in Chapter 4 that selective summaries, which summarize a subset of a dataset’s predicates and that can then improve query performance more than a full summary, can relate different summaries based on the selection of summarized predicates (competency question [CQ6].

5.2 Ontology

In this section, we describe our proposed ontology for structural summaries. Our ontology is geared towards grouping millions of nodes in large datasets by their description as a way to capture unique descriptions. Additionally, the ontology captures the relationship between unique descriptions by allowing the reuse of predicates and classes from instance data. Such an ontology can then benefit exploration of unique descriptions to interlink and reuse.

5.2.1 Classes and Properties

The following URIs are typed as rdfs:Class,

- Summary
- Block
- BlockEdge

The following URIs are typed as rdf:Property,

- hasBlock; domain is Summary, range is Block.
- extentContains; domain is Block and its range is URI (a subject or object from the instance).
- refinesTo; domain and range is Block.

![Graphical view of summary ontology](image)

Figure 5.1: Graphical view of summary ontology
5.2.2 Constraints

Bisimulation-based constraints will be expressed using functions over models since bisimulation is defined over sets. Example constraints:

- The nodes in the extents of summary’s blocks must form a partition
- A block that is refined will form a partition of its instances, i.e. each instance node will be in the extent of exactly one of its child blocks
- Each block/block edge has at least one instance in its extent (there are no empty blocks or block edges).

5.3 Implementation

Our summary ontology can describe summaries for use with existing semantic web tools, such as semantic web stores and query engines.

Example of our proposed ontology are given in RDF for a dataset and its summary. Here is an example instance about a person who directed and produced two movies:
As described in Chapter 1, a RDF summary partitions the nodes of an instance dataset into groups of nodes that have equivalent descriptions; the notion of equivalence is based on bisimilarity. An RDF summary expresses its blocks and block edges as statements; the FWBW summary RDF summary of the above example instance is expressed as follows.

-- summary meta-data
:summary1 rdf:type bc:Summary .
:summary1 bc:hasBlock :05e36522d .
:summary1 bc:hasBlock :061abd6c .
:05e36522d rdf:type bc:Block .
:061abd6c rdf:type bc:Block .
-- block edges
:05e36522d :director :061abd6c .
:05e36522d :producer :061abd6c .
-- block extents
:05e36522d bc:extentContains dir:8487 .
:061abd6c bc:extentContains movie:10559 .

The example’s FWBW summary describes a block with IRI :05e36522d that contains the lone instance dir:8487 in its extent, using the predicate bc:extentContains, and block :061abd6c that contains two instances in its extent, movie:10559 and movie:2014. The summary groups the two movies into the same block because they have equivalent descriptions – both movies are produced and directed by the same person (competency question [CQ1]). The summary shown partitions all of the dataset’s instances into blocks containing equivalent instances (competency question [CQ2]).

Additionally, for each statement in the instance dataset, a new statement representing a block edge is generated with the same predicate but using the subject’s block and the object’s block as the new statement’s subject and object, respectively (competency question [CQ3]). Block edges are reused from the instance data in order to maintain the relationship between groups of nodes. For example, the two edges labeled :director and :producer from block :05e36522d to block :061abd6c indicate that the extent of :05e36522d both directed and produced both movies in the extent of :061abd6c. Notice in our example that the number of block edges in :summary1 is fewer than the number of edges in the instance dataset, making it easier to explore and less expensive to query (in terms of number of block edge statements) than the original instance.

In Chapter 3, we show how a structural summary is generated using an iterative summarization algorithm (competency question [CQ4]). We show that, given an input semantic web dataset, all the instance’s nodes start in an initial partition with exactly one block and the partition is then iteratively refined. The partition that cannot be further refined signals that the partition is the coarsest possible.
A refinement from the initial partition to that of the structural summary, based on the above example, can be expressed as follows:

```sparql
-- initial summary, summary0
:summary0 rdf:type bc:Summary .
:summary0 bc:hasBlock :ab0000 .

-- summary1 produced in next iteration
:summary1 rdf:type bc:Summary .
:summary1 bc:hasBlock :05e36522d .
:summary1 bc:hasBlock :061abd6c .

-- meta-data expressing refinement of summary0 to summary1
:ab0000 bc:refinesTo :05e36522d .
:ab0000 bc:refinesTo :061abd6c .
```

These statements describe two summaries. The first is summary0, the initial partition which contains a single block, :ab0000, that groups all instances together without considering any part of their description. The second summary, summary1, is a refinement of summary0, that refines the single block of summary0 to two blocks, :05e36522d and :061abd6c.

The summary ontology we propose supports different types of RDF summaries. Chapter 4 shows that the summary ontology supports full and selective summaries to improve query performance. The summary ontology also supports intermediate summaries generated by each successive iteration during construction, summaries based on simulation, approximate summaries.

### 5.3.1 Verification of Structural Summaries

In this section, we describe how we verify that our summary ontology satisfies the competency questions given in Section 5.1.

The relationship between instance data and a summary, as well as between summaries, is a relationship between partitions.

**Proposition 2** Using a semantic web reasoner, we verify that summaries represented in OWL 2 are partitions, thereby satisfying competency questions CQ1, CQ2, CQ4, and CQ6.

We focus on the verification of partitions feature since it occurs in several ways that relate to describing relationships between summaries and original instance data, as well as relationships between summaries. First, a summary’s blocks partition an instance dataset’s nodes. Second, two summaries resulting from successive iterations of a summary construction algorithm form a partition of their blocks since the algorithms splits blocks. Third, a summary based on a superset of predicates of a selective summary forms a partition of the selective summary since it has predicates by which to split blocks. Fourth, a dataset’s FWBW summary forms a partition the dataset’s FW summary since there are more instance dataset edges by which to split blocks. We give two examples of summary verification. The first example verifies the relationship between a summary and its original instance by verifying that a summary partitions an instance dataset’s nodes. The second example verifies the relationship between summaries by verifying that one summary is a refinement of another. Verification that a structural summary correctly groups nodes having equivalent descriptions is beyond the scope of this work.
A partition can be verified by using OWL 2's `disjointUnion` axiom with a semantic web reasoner. To use this axiom, each instance is assigned to a distinct block extent class, and the set of block extent classes of a summary are made disjoint from each other. This can be done by first creating a new class to represent each block’s extent then, for each statement in the structural summary having the predicate bc:extentContains, creating a new statement where the instance is typed with the class indicating membership in its block extent. To reduce the number of statements included in the structural summary, these statements can be generated as needed for verification.

For our first example, to verify that the above summary forms a partition of a dataset graph’s nodes, each instance is typed using a distinct class representing the block of which it is an extent, and the summary is then the disjoint union of block extent classes.

```
-- create extent member classes
:summary1/Instances rdf:type rdfs:Class .
:05e36522d/ExtentMember rdf:type rdfs:Class .
:061abd6c/ExtentMember rdf:type rdfs:Class .
-- set extent member classes to be disjoint from one another
:summary1/Instances owl:disjointUnionOf :05e36522d/ExtentMember .
:summary1/Instances owl:disjointUnionOf :061abd6c/ExtentMember .
-- classify each node as a member of it respective block extent
dir:8487 rdf:type :05e36522d/ExtentMember .
movie:10559 rdf:type :061abd6c/ExtentMember .
```

In addition to verifying that a structural summary’s blocks form a partition, the OWL `disjointUnion` axiom can also be used to verify that one summary is a refinement of another. A summary is a refinement of another when blocks are split by some condition (such as one node being described by a predicate and another node not being described by that predicate). This approach can verify intermediate summaries generated by our iterative partition refinement algorithm, which successively refines the initial partition until the resultant summary cannot be further refined.

For our second example, a summary can be verified to be a refinement of another by taking each statement with the predicate `bc:refinesTo`, and generating a new statement having the predicate `owl:disjointUnion` but using the `ExtentMember` class of each subject and object block. This way, the extent member classes for a set of blocks become disjoint unions of the block from which they were refined from. For the above example, the resultant summary can be verified to be a refinement of the initial summary (that contains a single block), using the following statements in addition to the above verification statements:

```
-- declare parent block’s extent member class
:ab000/ExtentMember rdf:type rdfs:Class .
-- parent block is disjoint union of blocks to which it refines to
:ab000/ExtentMember owl:disjointUnion :05e36522d/ExtentMember .
:ab000/ExtentMember owl:disjointUnion :061abd6c/ExtentMember .
```

In addition to the above two verifications, involving a summary and an instance as well as between summaries, this approach is applicable in other scenarios. Two other scenarios that are applicable to
our work include verifying partitions between FWBW summaries and FW summaries, and verifying partitions between summaries and selective summaries (we do not show these examples).

The work of Chapter 4, which describes how a query on the instance undergoes a translation to a corresponding query on the summary to return the same results, verifies that our RDF summaries satisfy competency questions CQ3 and CQ5.

In this section, we have given examples of how a structural summary can be expressed using our ontology. In addition, we have shown how a summary can be verified to be a partition in its relationships with instance datasets as well as other summaries. In the next sections, we give some high-level comments about our ontology’s development.

5.4 Ontology Development Discussion

In this section, we describe semantic and pragmatic dimensions that show how our ontology’s development achieves our goal of using structural summaries within existing unmodified semantic web tools such as for exploration, query optimization, and verification.

5.4.1 Semantic Dimensions

An ontology’s semantic dimension aims to characterize the way in which the ontology’s keywords are selected and defined. We describe three semantic dimensions of our summary ontology.

First, an ontology’s degree of formality and structure describes an ontology’s choice of keywords ranging from informal to very formal. Furthermore, a highly-structured ontology describes its support for computation. The summary ontology we describe is based on bisimulation, a well-structured notion of equivalence, thus our ontology relies on formal notions. Furthermore, our summary supports some degree of informality by summarizing arbitrary sets of dataset’s predicates for selective summaries. The summary ontology’s structuredness supports computation, and this is shown in the use of summaries with unmodified RDF tools including existing SPARQL processors and automated reasoners.

Second, an ontology’s expressiveness of the knowledge representation language characterizes the language and framework that is used to express the ontology and its content such as its ability to express formal notions. This dimension generally considers ontologies that are logic-based to be machine-interpretable, and this is the direction we consider with our ontology. Our summary ontology aims to be logic-based in order to allow machine-interpretable applications, as we demonstrate by using RDF summaries in a SPARQL processor and verifying OWL 2 summaries with an automated reasoner.

Third, an ontology’s representational granularity characterizes the breadth in terminology, such as using a few informal terms or using a more formal terms with their relationships expressed using many constraints. The third dimension can sometimes be evaluated using quantifiable metrics based on ontological substructures, and can thus be seen to have a dependency on the second dimension. Our summary ontology provides the ability to connect ontological classes having a strict relationship (such a lattice that arises from graph partitioning) with quantifiable metrics that can be used to describe different summaries. One example of this in our work is the use of selective summaries that are based on a subset of a dataset’s predicates; we envision that it is possible to pair structural and value-based summaries using the RDF Data Cube Vocabulary.
5.4.2 Pragmatic Dimensions

An ontology’s pragmatic dimensions aim to characterize the ontology’s application and use. We describe four pragmatic dimensions of our summary ontology.

First, an ontology’s intended use and application focus aims to characterize the application and role of the ontology and its instances. In this work, we promote our ontology as a way to use summaries within existing semantic web tools, allowing exploration and query optimization. Second, an ontology’s role of automated reasoning describes the expected role the ontology plays in automated reasoning applications and our work demonstrates verification using an automated reasoner; specifically, we represented and verified a summary partition represented in OWL 2 notation.

Third, an ontology’s descriptive and prescriptive dimensions aim to describe an ontology’s role in enabling a range of, possibly unused, substructures, as in a descriptive role, or a more stricter reductionist view which tries to reduce the number of concepts to use to express an instance. Our summary ontology prescribes the use of a minimal number of formal classes to describe an instance’s summary, potentially using the same or a subset of its predicates. Fourth, an ontology’s design methodology characterize its development, such as whether it used a top-down approach, whereby development proceeds starting from notions about the real-world, or bottom-up, whereby development proceeds considering the data and application first. Our hybrid approach mixes a bottom-up approach, whereby an RDF summary can improve query performance while it does not explicitly rely on the ontology’s semantics, and also uses a formal approach by verifying logical constraints using OWL 2 summaries with an automated reasoner.

In this section, we have described our summary ontology’s development along semantic and pragmatic dimensions. The dimensions help describe our choices that support our goal of providing summaries that can be used within existing semantic web tools, such as for query optimization in unmodified SPARQL processors and for logical verification using OWL 2 summaries in unmodified automated reasoners.

5.5 Summary

In this chapter, we have described an ontology for representing summaries. With a need to satisfy the competency questions given in Section 5.1, we have developed a summary ontology whose classes and properties are given in Section 5.2. Specifically, the ontology we give in this chapter serves the need to describe a summary’s blocks and block edges, as well as relationships between summaries, such as during an iterative construction. Furthermore, the competency questions define constraints such as the relationship between blocks and their instances, such as that each instance may appear in exactly one block. In Section 5.3, we use OWL 2 notation to verify that the summary can satisfy logical constraints and give an example that shows a summary partitions a dataset’s nodes. We discuss the ontology’s semantic and pragmatic dimensions in Section 5.4 and these relate the ontology with notions such as its use in reasoners, a semantic dimension, as well as its degree of granularity, a pragmatic dimension.
Chapter 6

Conclusions

This chapter contains a brief review of this thesis’ contributions followed by some possible future research directions.

6.1 Summary of contributions

The goal of this thesis is to show that our multi-core approach to constructing and using RDF summaries for query optimization in existing unmodified SPARQL processors is novel and effective for large graphs.

With organizations generating large volumes of semi-structured data, it becomes increasingly challenging to explore and understand. Examples of large graphs include DBpedia’s RDF dataset that contains tens of millions of nodes and hundreds of millions of edges to describe entities such as people and locations, and Twitter’s follower graph amongst over 50 million users that contains close to 2 billion edges.

Although there exists several summary construction algorithms, starting from a hash-based approach that was parallelized into an MPI implementation and then implemented in a distributed setting using MapReduce, there remains a need to support multi-core architecture. In Chapter 3, we described an efficient optimization that dramatically reduces the summary construction time of both real-world and synthetic datasets. To improve query performance, summaries can be used as a graph-based index, an approach that has seen use for multiple formats including XML and RDF. In Chapter 4, we propose a novel approach of using RDF summaries within existing unmodified SPARQL processors and show that they are effective at improving query performance. In Chapter 5, we describe a summary representation that allows the use of summaries in unmodified semantic web tools such as for verification using an automated reasoner.

6.1.1 Summary construction

In Chapter 3 we use GraphChi [77], a multi-core graph processing framework, to construct bisimulation summaries with the goal of constructing summaries in roughly the same amount of time that it takes to input the RDF data into the framework (for a range of real world data graphs) and output the summary as RDF. In order to achieve our goal, we leverage the following four features of our summary construction approach [71]. First, we describe a summary construction implementation on a parallel graph processing framework (GraphChi). Second, we present a novel and very effective singleton optimization that allows
us to achieve the goal of drastically reducing per-iteration times after only a few iterations. Third, we give an experimental validation that our GraphChi implementation achieves our goal of constructing summaries in an amount of time similar to the time required to load the dataset and then write the summary; in our experiments, we use graphs ranging from millions to billions of edges. Fourth, we compare our multi-core approach to Hadoop and give evidence to support GraphChi’s use on multi-core systems.

### 6.1.2 Summary-based query optimization

In Chapter 4, we have devised a translation that outputs SPARQL queries that execute over summaries directly represented in RDF in order to make both graph traversal queries and summary-based optimizations readily available in existing RDF systems. Our approach to optimize graph traversal queries on top of existing SPARQL processors consists of three components. First, we have developed and implemented an algorithm that translates graph traversal queries into SPARQL expressions. Second, we construct and store summaries alongside the original graph in the RDF store. Third, we build upon our translation algorithm to develop a rewriting optimization that converts graph traversal queries into equivalent SPARQL queries that execute over RDF graphs of both the original data and the summary. Our translation leverages Extended Property Paths (EPPs) [41], a language more expressive than SPARQL property paths with a concise syntax and a formal semantics. The significant enhancements that EPPs bring include: (i) users can leverage their syntax to write more expressive navigational queries than property paths in a succinct way; (ii) EPPs are immediately available into existing SPARQL processors. We have devised 2 translations from EPPs: EPPstoSPARQL, to translate EPPs expressions to SPARQL; and EPPsItoEPPsS that allows using summaries alongside instance graphs. The advantage of the translation is that it can be executed on existing SPARQL processors. From the syntactic point of view, the advantage of using EPPs to write navigational queries instead of writing them directly into pure SPARQL is that the same request can be expressed more succinctly and without the need to deal with intermediate variables.

### 6.1.3 Ontology for structural summaries

In Chapter 5, we gave an ontology that supports representing bisimulation summaries in existing RDF systems. The summary ontology’s representation describes a summary’s blocks and block edges, and also supports relating multiple summaries, such as subsequent summaries from an iterative construction algorithm. We have given an implementation of a summary based on a mapping of OWL 2 into RDF [87], followed by a verification that our ontology satisfies logical constraints, such as the requirement that a summary’s block extents form a partition of a dataset’s nodes. We end the chapter with a discussion of our ontology’s semantic and pragmatic dimensions that guided its design and development, such as its degree of formality and structure as a semantic dimension, and the its role within automated reasoning as a pragmatic dimension.
6.2 Future work

In this section, we give several possible directions for future work.

Our work of Chapter 3 describes and validates the benefits of using our proposed singleton optimization to reduce the number of nodes processed in each iteration. By default, GraphChi has a deterministic execution order that is dependent on conflict nodes, nodes that have an edge between them and are in the same interval. GraphChi’s conflict-resolution approach is to serialize the execution of conflict nodes. GraphChi also supports non-deterministic execution, such as through internal determinism [18], which represents the use of fine-grained determinism within a non-deterministic setting. Since our approach uses a BSP model whereby nodes in the current iteration use values from the previous iteration, we execute our GraphChi implementation using non-determinism. GraphChi also supports execution of asynchronous updates such that node or edge values become visible to other nodes immediately. Recent literature on asynchronous parallelism shows that it can improve performance despite an increase in implementation complexity [123, 78, 44, 53]. The internal determinism and asynchronous update features may provide further benefits to our summary construction approach. For example, using our singleton optimization in an asynchronous manner in our approach could potentially allow blocks in an iteration to skip processing of a singleton as soon as it is found in the same iteration. The challenge in this approach is to determine whether a singleton was found in between processing of nodes in a non-singleton neighbour’s extent, since the block identifiers of two nodes in a block’s extent may differ if one node did not skip the singleton (because the singleton was not yet found) and if the other node did skip the singleton (after the singleton was found). One potential solution is to keep track of such blocks since there are far fewer non-singleton blocks than singletons. Another option is to process blocks in order so that non-singleton blocks are processed after blocks that are likely to become singleton.

The query optimization of Chapter 4 shows that summaries are effective at answering queries faster than on an instance dataset. The Semantic Publishing Benchmark (SPB) [8] has identified key choke points of SPARQL systems, including join ordering, aggregation, duplicate elimination, and complex filter conditions. Our query performance evaluations can be extended with more detailed analysis of cost estimates from query planners and optimizers of SPARQL processors in order to help determine which queries are worthwhile to optimize. For example, in Section 4.2.4 we show that Virtuoso’s query plan changes between a query on the instance and on the summary. Since Virtuoso allows specifying an explicit join order, via the clause ‘define sql:select-option ”order”’, we can examine whether summary queries can improve query performance even further by using summarization statistics, or by comparing that the same join order is used for an instance query and for its corresponding summary query. Furthermore, in this work we have examined query optimization only in the context of FWBW bisimulation summaries. It may be worthwhile to consider how to obtain query optimizations with a mix of different summaries, such as using a FWBW summary to answer navigational queries that traverse edges in a forward and backward manner, and using a FW summary to answer portions of a navigational query that only traverses edges in a forward direction.

The structural summary ontology given in Chapter 5 can include support for meta-data about blocks, such as extent size, and we have leveraged this ability to aid understanding and exploring of large datasets, and to generate queries. However, we do not use the summary with a reasoner in order to find additional query optimizations. We envision leveraging summary statistics using a reasoner in order to reveal previously unknown optimizations, such as recognizing that if a block has only two instances in its extent, then if it is split it will result in two singletons while reducing the number of non-singletons,
and so it has a higher likelihood of contributing to the singleton optimization than a block with many nodes in its extent. Another potential optimization that could be automatically revealed is knowing that a singleton will never split and so can reduce the workload of determining another block’s stability. Furthermore, integrating logical constraints that are defined for related data structures, such as for graph topology or partition lattices from the COLORE \[50\] ontology repository, could also aid finding additional optimizations. To our knowledge, there are no SPARQL processors that leverage flexible RDF summaries of large datasets with statistics and reasoners in order to improve query performance.
Bibliography


Appendix A

Appendix

A.1 Computing Bisimulation Relations in Pig Latin

In this section, we give an example Pig Latin script to compute bisimulation summaries of class usage of an RDF dataset.

Using Pig Latin’s nested-record data model, we present a script in Figure A.1 to compute two usage summaries of the LOD cloud. The first usage summary groups entities based on the set of classes that it is a type of, and the second usage summary groups entities on the sets of classes that it is a types of, as well as the graph URI making those statements; notice that the second summary is a refinement of the first. We first explain the script and then describe some of its limitations.

The input to the script in Figure A.1 are files on HDFS containing quads that are loaded into a relation having a column for subject, predicate, object, and graph URIs (line 1). Line 3 examines each quad to determine the class and predicate usage. Notice that the class and predicate usage neighbourhood can be decided at a statement level and so does not require a self-join to examine longer paths. In this example, since we are interested only in class usages (and the URI of the provider that states an instance’s class usage), usages other than class usage are filtered on line 8.

In order to assign a bisimulation identifier based on usages, each instance’s usages are grouped. Line 10 groups each instance’s class usages by placing all tuples having the same suri into a bag with the field name usagestmtsclass. The foreach operation from lines 12 to 24 iterates over each instance’s record and computes two bisimulation identifiers. The first identifier is computed from the set of distinct classes (line 13-16), and the second identifier is computed from the set of distinct (class usage, guri) pairs (lines 18-21) which captures the sets of classes usages and the providers that expressed them. Each identifier is computed as an MD5 hash of the distinct and ordered set of usages; assuming there are no hash collisions, sets of usages containing the same members will have the same hash value. The foreach outputs records containing a distinct instance URI, its bisimulation identifiers and the (distinct) usages that contributed to each bisimulation identifier. In essence, this creates an initial partition where each block contains a single instance, and which is more refined than the coarsest, stable partition of both summaries. So to combine instances having the same bisimulation identifier into the same block, instances are grouped by bisimulation identifier (lines 35 and 36).

The output of the Pig Latin script are files containing nested-records that are stored on HDFS. Distinct bisimulation identifiers and their usages are stored to HDFS on lines 26 to 32 for later reference.
Appendix A. Appendix

The two usage summaries are stored (lines 34-38) as relations containing the bisimulation identifier and a bag of instance URIs that have the same bisimulation identifier. Notice that identical bisimulation identifiers computed from class usage and provider guri will also have an identical bisimulation identifier computed from just class usage, since the first is a refinement of the latter.

The Pig Latin script in Figure A.1, which expresses one iteration of a summary construction, suffers from four main problems. First, ExpLOD’s mechanism for creating usage summaries requires random access to nodes and their neighbours, a feature that is not available in shared-nothing frameworks such as MapReduce, though time-forward processing techniques as leveraged in [57] can be of use. Second, the Pig Latin script in Figure A.1 requires that a set of usages from which a bisimulation identifier is to be computed fits in memory (all other works involving construction of bisimulations are similarly affected). As indicated in [122], since hash-based partitioning that uses the entire set is not sufficient in some cases (since the set does not fit in memory), set-similarity joins may be executed by transforming each input record into multiple records based on a partition of its fields. We envision that similar techniques may be beneficial to calculating participation amongst blocks, as is done in summaries that introduce error as described above ([116, 90]).

The third problem of the above script is that Pig Latin lack of support for loops makes it difficult to support flexible usage summaries based on arbitrary paths (a feature of ExpLOD) and reasoning-like tasks, such as finding all equivalent instances based on owl:sameAs interlinks. [120] overcome this challenge by precomputing a synonyms table based on equivalent instances that are replicated to each cluster node as part of a broadcast join; however, the synonym table’s size may still exceed main-memory. Finally, although a group by operation is used to combine blocks (a bottom/up approach), it is difficult to perform refinements (considered a top/down approach) since Pig Latin does not support producing multiple tuples during foreach iterations. We workaround this problem by creating each partition from the same initial partition (lines 35-36) where although the partition based on class usage and graph URI is a refinement of the partition based on class usage alone, the two partitions are created by performing two separate group by operations. This is similar to the advice given by the authors of [116] but due to framework limitations rather than computational complexity. These four challenges highlight the need for a combination of features for efficient, flexible construction of bisimulation summaries of large graphs.
mydata = load './data/btc2011.*' using NQuadLoadFunc() as (suri:chararray, puri:chararray, ouri:chararray, guri:chararray);

usagestmts = FOREACH mydata {
  p = (INDEXOF(puri, '22-rdf-syntax-ns#type') > -1 ? CONCAT('C+', ouri) : CONCAT('P+', puri));
  GENERATE suri as suri:chararray, p as unuri:chararray, guri as guri:chararray;
}

usagestmtsclass = FILTER usagestmts BY unuri matches 'C.*';
suriguri0 = GROUP usagestmtsclass BY (suri);
suriguri1 = FOREACH suriguri0 {
  b00 = FOREACH usagestmtsclass GENERATE unuri;
  b0 = DISTINCT b00;
  un = ORDER b0 BY *;
  unhash = edu.toronto.cs.trellis.pig.MD5Func(un);
}

c00 = FOREACH usagestmtsclass GENERATE unuri, guri;
c0 = DISTINCT c00;
ung = ORDER c0 BY *;
unhashg = edu.toronto.cs.trellis.pig.MD5Func(ung);

GENERATE group as suri:chararray, unhash, un, unhashg, ung;
storeclassusage = FOREACH suriguri1 GENERATE unhash, un;
storeclassusage0 = DISTINCT storeclassusage;
STORE storeclassusage0 INTO './output/classusageinfo';

storeclassusagebeg = FOREACH suriguri1 GENERATE unhashg, ung;
storeclassusage0g = DISTINCT storeclassusagebeg;
STORE storeclassusage0 INTO './output/classgurusageinfo';

classusagesummary = FOREACH suriguri1 GENERATE suri, unhash, unhashg;
classusagesummary0 = GROUP classusagesummary BY unhash;
classusagesummary0g = GROUP classusagesummary BY unhashg;
STORE classusagesummary0 INTO './output/classusagesummary';
STORE classusagesummary0g INTO './output/classusagegsummary';

Figure A.1: Pig Latin summary construction
### A.2 DBpedia Queries

In this section, we describe the DBpedia queries and the predicate statistics used to select the queries. In [A.1] we show the DBpedia queries that we have developed for our experiments.

<table>
<thead>
<tr>
<th>Query</th>
<th>S+EPPs expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQ01</td>
<td>(_s((\text{owl:#sameAs})_o/_s((\text{dbpedia:wikiPageRedirects})_o/_s((\text{dbpedia:neighboringMunicipality})_o))</td>
</tr>
<tr>
<td>DQ02</td>
<td>(_s((\text{foaf:primaryTopic})_o/_s((\text{dbpedia:family})_o/_s((\text{rdf:#type})_o))</td>
</tr>
<tr>
<td>DQ03</td>
<td>(_s((\text{owl:#sameAs})_o/_s((\text{dbpedia:isPartOf}) &amp;&amp; !(\text{EPPTest}(_o,_s(\text{owl:#sameAs})_o)) &amp;&amp; (\text{EPPTest}(_o,_s(\text{dbpedia:type})_o))_o))</td>
</tr>
<tr>
<td>DQ04</td>
<td>(_s((\text{owl:#sameAs})_o/_s((\text{dbpedia:isPartOf}) &amp;&amp; !(\text{EPPTest}(_o,_s(\text{owl:#sameAs})_o)) &amp;&amp; (\text{EPPTest}(_o,_s(\text{dbpedia:type})_o)) &amp;&amp; (\text{EPPTest}(_o,_s(\text{foaf:homepage})_o))_o))</td>
</tr>
<tr>
<td>DQ05</td>
<td>(_s((\text{foaf:primaryTopic})_o/_s((\text{dbpedia:isPartOf})_o/_s((\text{foaf:isPrimaryTopicOf})_o)_o))</td>
</tr>
<tr>
<td>DQ06</td>
<td>(_s((\text{dbpedia:country}) &amp;&amp; !(\text{EPPTest}(_o,_o(\text{dc:subject})_o)) &amp;&amp; !(\text{EPPTest}(_s,_o(\text{dbpedia:TimeZone})_s)) &amp;&amp; !(\text{EPPTest}(_s,_o(\text{dbpedia:isPartOf})_s))_o))</td>
</tr>
<tr>
<td>DQ07</td>
<td>(_s((\text{dbpedia:country}) &amp;&amp; !(\text{EPPTest}(_o,_o(\text{dc:subject})_s)) &amp;&amp; !(\text{EPPTest}(_s,_o(\text{dbpedia:TimeZone})_s)) &amp;&amp; !(\text{EPPTest}(_s,_o(\text{dbpedia:isPartOf})_s))_o))</td>
</tr>
<tr>
<td>DQ08</td>
<td>(_s((\text{dbpedia:genre})_o/_s((\text{dbpedia:phylum})_o/_s((\text{dbpedia:wikiPageWikiLink})_o))</td>
</tr>
<tr>
<td>DQ09</td>
<td>(_s((\text{dbpedia:hasPhotoCollection}) &amp;&amp; !(\text{EPPTest}(_s,_s(\text{dbpedia:TimeZone})_o)))_o))</td>
</tr>
<tr>
<td>DQ10</td>
<td>(_s((\text{dbpedia:wikiPageWikiLink})_o/_s((\text{dbpedia:wikiPageWikiLink})_o))</td>
</tr>
<tr>
<td>DQ11</td>
<td>(_s((\text{dc:subject})_o))</td>
</tr>
<tr>
<td>DQ12</td>
<td>(_s((\text{dc:subject}) &amp;&amp; !(\text{EPPTest}(_o,_o(\text{dbpedia:country})_s)) &amp;&amp; !(\text{EPPTest}(_s,_o(\text{dbpedia:TimeZone})_s))_o))</td>
</tr>
<tr>
<td>DQ13</td>
<td>(_s((\text{dc:subject}) &amp;&amp; !(\text{EPPTest}(_o,_o(\text{dbpedia:country})_s)) &amp;&amp; !(\text{EPPTest}(_o,_o(\text{dbpedia:TimeZone})_s))_o))</td>
</tr>
<tr>
<td>DQ14</td>
<td>(_s((\text{dbpedia:wikiPageRedirects})_o/_o((\text{dbpedia:wikiPageRedirects})_o))</td>
</tr>
<tr>
<td>DQ15</td>
<td>(_s((\text{dbpedia:wikiPageRedirects})_o/_o((\text{dbpedia:wikiPageRedirects})_o))</td>
</tr>
<tr>
<td>DQ16</td>
<td>(_s((\text{dbpedia:wikiPageRedirects})_o/_o((\text{dbpedia:wikiPageRedirects})_o))</td>
</tr>
<tr>
<td>DQ17</td>
<td>(_s((\text{dbpedia:wikiPageWikiLink}) &amp;&amp; !(\text{EPPTest}(_o,_o(\text{dbpedia:country})_s)) &amp;&amp; !(\text{EPPTest}(_o,_o(\text{dbpedia:country})_s))_o))</td>
</tr>
</tbody>
</table>

Table A.1: DBpedia queries

Table A.2 shows statistics of the predicates in the DBpedia instance, and full and selective summaries, listing the 30 predicates picked to construct the d30 selective summary. The table shows each predicate’s associated count of statements in the instance dataset (inst), the count of block edge statements in the full, d30, and d13 summary graphs, and the count of queries in the DBSB query logs that contain each predicate (qlog). The table lists predicates by decreasing popularity in the d13 summary. The predicates picked for the d30 and d13 selective summaries differ dramatically in their popularity in the query logs and instance, ranging by orders of magnitude. We also note that although the full summary reduces the number of statements of a predicate by several million, there is much less difference in the number of statements between the full, d30, and d13 summaries.

---

<table>
<thead>
<tr>
<th>predicate</th>
<th>inst</th>
<th>full</th>
<th>d30</th>
<th>d13</th>
<th>qlogcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbpedia:wikiPageWikiLink</td>
<td>136,425,275</td>
<td>126,718,320</td>
<td>126,500,600</td>
<td>126,392,477</td>
<td>120</td>
</tr>
<tr>
<td>dc:subject</td>
<td>16,598,103</td>
<td>16,355,976</td>
<td>16,321,182</td>
<td>16,313,124</td>
<td>1,235,586</td>
</tr>
<tr>
<td>dbpedia:hasPhotoCollection</td>
<td>4,040,107</td>
<td>3,900,267</td>
<td>3,891,611</td>
<td>3,886,625</td>
<td>762</td>
</tr>
<tr>
<td>owl:#sameAs</td>
<td>4,111,659</td>
<td>3,400,475</td>
<td>3,387,173</td>
<td>3,372,297</td>
<td>24,172</td>
</tr>
<tr>
<td>dbpedia:wikiPageExternalLink</td>
<td>6,620,610</td>
<td>3,297,447</td>
<td>3,294,805</td>
<td>3,046,284</td>
<td>389</td>
</tr>
<tr>
<td>dbpedia:wikiPageRedirects</td>
<td>5,873,900</td>
<td>2,442,121</td>
<td>2,438,214</td>
<td>2,389,841</td>
<td>322,774</td>
</tr>
<tr>
<td>dbpedia:country</td>
<td>624,327</td>
<td>553,856</td>
<td>547,097</td>
<td>543,096</td>
<td>2,223</td>
</tr>
<tr>
<td>dbpedia:isPartOf</td>
<td>675,566</td>
<td>538,004</td>
<td>524,027</td>
<td>514,464</td>
<td>39</td>
</tr>
<tr>
<td>dbpedia:timeZone</td>
<td>196,855</td>
<td>158,706</td>
<td>156,655</td>
<td>154,717</td>
<td>23</td>
</tr>
<tr>
<td>dbpedia:phylum</td>
<td>178,473</td>
<td>147,087</td>
<td>145,957</td>
<td>145,834</td>
<td>151</td>
</tr>
<tr>
<td>dbpedia:genus</td>
<td>152,218</td>
<td>129,412</td>
<td>128,323</td>
<td>128,212</td>
<td>394</td>
</tr>
<tr>
<td>dbpedia:division</td>
<td>55,176</td>
<td>46,322</td>
<td>46,069</td>
<td>46,048</td>
<td>73</td>
</tr>
<tr>
<td>dbpedia:spokenIn</td>
<td>15,602</td>
<td>15,348</td>
<td>15,348</td>
<td>15,348</td>
<td>1</td>
</tr>
<tr>
<td>rdf:#type</td>
<td>60,149,040</td>
<td>54,925,967</td>
<td>53,126,668</td>
<td>1,615,746</td>
<td></td>
</tr>
<tr>
<td>foaf:primaryTopic</td>
<td>10,030,260</td>
<td>6,550,254</td>
<td>6,537,663</td>
<td>2,662</td>
<td></td>
</tr>
<tr>
<td>foaf:isPrimaryTopicOf</td>
<td>10,030,260</td>
<td>6,550,254</td>
<td>6,537,663</td>
<td>2,524</td>
<td></td>
</tr>
<tr>
<td>dbpedia:team</td>
<td>1,366,541</td>
<td>1,222,557</td>
<td>1,116,993</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>foaf:homepage</td>
<td>532,529</td>
<td>523,810</td>
<td>523,523</td>
<td>3,037</td>
<td></td>
</tr>
<tr>
<td>dbpedia:careerStation</td>
<td>577,196</td>
<td>526,305</td>
<td>499,652</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dbpedia:wordnet_type</td>
<td>437,766</td>
<td>427,692</td>
<td>427,428</td>
<td>381,705</td>
<td></td>
</tr>
<tr>
<td>dbpedia:type</td>
<td>338,407</td>
<td>308,503</td>
<td>303,789</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>dbpedia:family</td>
<td>298,319</td>
<td>239,502</td>
<td>237,092</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>dbpedia:kingdom</td>
<td>220,096</td>
<td>181,555</td>
<td>180,225</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>dbpedia:binomialAuthority</td>
<td>69,371</td>
<td>67,276</td>
<td>67,180</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>dbpedia:district</td>
<td>54,171</td>
<td>51,639</td>
<td>51,346</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>dbpedia:department</td>
<td>36,485</td>
<td>29,870</td>
<td>28,874</td>
<td>288</td>
<td></td>
</tr>
<tr>
<td>dbpedia:parent</td>
<td>22,757</td>
<td>22,639</td>
<td>22,635</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>dbpedia:leaderName</td>
<td>17,302</td>
<td>17,035</td>
<td>17,026</td>
<td>4,445</td>
<td></td>
</tr>
<tr>
<td>dbpedia:neighboringMunicipality</td>
<td>16,655</td>
<td>16,653</td>
<td>16,653</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>dbpedia:hybrid</td>
<td>1,851</td>
<td>1,521</td>
<td>1,517</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Statistics of predicates in instance graph, FWBW full, d30, and d13 summary graphs, and count of distinct queries in DBSB logs
A.3 Virtuoso LinkedMDB query LQ01 execution plan

In this section, we show Virtuoso’s execution plans for an EPPsI and its corresponding EPPsS, giving additional details related to the work presented in Section 4.2.4.

Below, we show the instance and summary query execution plans of query LQ01 of Table 4.5 using Virtuoso open-source edition version 7.10.3207. In Section A.3.1, we show Virtuoso’s query execution plan for the instance query, and in Section A.3.2 we show Virtuoso’s execution plan for the summary query. As we describe in Section 4.2.4, these execution plans show that there are opportunities for summary-based optimizations.

We highlight the vastly different query plans that can occur with the same SPARQL processor as we observe that the order of almost all predicates differs and the number of intermediate rows for the predicate writer in the summary query’s execution plan is an order of magnitude more than in the instance query. While we focus this example on a specific query optimizer, namely Virtuoso, the rationale also applies to different SPARQL processors and optimizations to reduce intermediate rows on summary graphs and that can provide further opportunities for query performance improvement.

A.3.1 LQ01 on instance graph

```{fork
time 21% fanout 28549 input 1 rows
RDF_QUAD 2.6e+04 rows(s_44_30_t11.S, s_44_30_t11.O)
inlined P = #/writer G = #/inst
time 0.26% fanout 5.89219 input 28549 rows
RDF_QUAD 1.8 rows(s_44_30_t10.S, s_44_30_t10.O)
inlined P = #/performance , S = k_s_44_30_t11.S G = #/inst
time 9.1% fanout 0.865548 input 168216 rows
END Node
After test:
0: if ({{
time 0.012% fanout 1 input 168216 rows
time 0.52% fanout 1.86703 input 168216 rows
RDF_QUAD 1.6 rows()
inlined P = #/producer , S = k_s_44_30_t10.S G = #/inst
time 0.018% fanout 0 input 314065 rows
Subquery Select( <none> )
})
) then 4 else 29 unkn 29
4: if ({{
time 0.015% fanout 0.873573 input 168216 rows
time 0.81% fanout 9.37529 input 146949 rows
RDF_QUAD 4.4 rows()
inlined P = #/actor , S = k_s_44_30_t10.S G = #/inst
time 0.073% fanout 0 input 1.37769e+06 rows
Subquery Select( <none> )
})
) then 8 else 29 unkn 29
8: if ({{
time 0.039% fanout 0.873437 input 168216 rows
time 0.38% fanout 1.21773 input 146926 rows
RDF_QUAD 1.1 rows()
inlined P = #/director , S = k_s_44_30_t10.S G = #/inst
time 0.013% fanout 0 input 178916 rows
Subquery Select( <none> )
})
) then 12 else 29 unkn 29
12: if ({{
time 0.019% fanout 0.865548 input 168216 rows
```
time 0.52% fanout 1.84312 input 145599 rows
RDF_QUAD 1 rows()
inlined P = #/page , S = k_s_44_30_t10.S G = #/inst
time 0.018% fanout 0 input 268357 rows
Subquery Select( <none> )

) then 16 else 29 unkn 29
16: if ({
time 0.015% fanout 0.865548 input 168216 rows
time 0.59% fanout 1.84312 input 145599 rows
RDF_QUAD 1 rows()
inlined P = #/page , S = k_s_44_30_t10.S G = #/inst
time 0.016% fanout 0 input 268357 rows
Subquery Select( <none> )
}
)
) then 20 else 29 unkn 29
20: if ({
time 0.014% fanout 0.865548 input 168216 rows
time 0.58% fanout 1 input 145599 rows
RDF_QUAD 1 rows()
inlined P = ##type , S = k_s_44_30_t10.S G = #/inst
time 0.012% fanout 0 input 145599 rows
Subquery Select( <none> )
}
)
) then 24 else 29 unkn 29
24: if ({
time 0.016% fanout 0.865548 input 168216 rows
time 6.7% fanout 1 input 145599 rows
RDF_QUAD 1 rows()
inlined P = ##type , S = k_s_44_30_t10.S G = #/inst
time 0.1% fanout 0 input 145599 rows
Subquery Select( <none> )
}
)
) then 28 else 29 unkn 29
28: BReturn 1
29: BReturn 0

time 1.9% fanout 1.99643 input 145599 rows
RDF_QUAD_POGS 1.3 rows(s_44_30_t9.S)
P = #/performance , O = k_s_44_30_t10.O G = #/inst
time 22% fanout 0 input 290678 rows
END Node

After test:
0: if ({
time 0.02% fanout 1 input 290678 rows
time 11% fanout 1.42575 input 290678 rows
RDF_QUAD 1 rows()
inlined P = #/page , S = k_s_44_30_t9.S G = #/inst
time 0.12% fanout 0 input 414435 rows
Subquery Select( <none> )
}
)
) then 4 else 9 unkn 9
4: if ({
time 0.069% fanout 1 input 290678 rows
time 15% fanout 1.49911 input 290678 rows
RDF_QUAD 1 rows()
inlined P = ##type , S = k_s_44_30_t9.S G = #/inst
time 0.14% fanout 0 input 435757 rows
Subquery Select( <none> )
}
)
) then 8 else 9 unkn 9
8: BReturn 1
9: BReturn 0

After code:
0: count ress_44_30_t11.0set no set_ctrdistinct DISTINCT HASH
Appendix A

A.3.2 LQ01 on summary graph

{ fork
Subquery 27
{ Subquery 30
{ time 0.056% fanout 44582 input 1 rows
RDF_QUAD_POGS 4.1e+04 rows(t22.S)
inlined P = #/director G = #/sum
time 1% fanout 0.92082 input 44582 rows
Distinct (t22.S)

After code:
0: S := := artm t22.S
4: BReturn 0

Subquery Select(S)
{ time 0.0076% fanout 0 input 41052 rows
Subquery Select( <none> )
}

RDF_QUAD 2.9 rows(s_44_30_t10.S, s_44_30_t10.0)
inlined P = #/performance , S = k_S G = #/sum
time 21% fanout 0.624725 input 144558 rows

END Node

After test:
0: if ({
time 0.12% fanout 1 input 144558 rows
RDF_QUAD 1 rows()
inlined P = #/page , S = k_s_44_30_t10.S G = #/sum
time 0.12% fanout 0 input 145703 rows
Subquery Select( <none> )
}
) then 4 else 25 unkn 25
4: if ({
time 0.13% fanout 1 input 144558 rows
RDF_QUAD 1 rows()
inlined P = #/page , S = k_s_44_30_t10.S G = #/sum
time 0.13% fanout 0 input 145703 rows
Subquery Select( <none> )
}
) then 8 else 25 unkn 25
8: if ({
time 0.17% fanout 1 input 144558 rows
RDF_QUAD 1.1 rows()
inlined P = ##type , S = k_s_44_30_t10.S G = #/sum
time 0.14% fanout 0 input 144558 rows
Subquery Select( <none> )
}
) then 12 else 25 unkn 25
12: if ({
time 0.14% fanout 1 input 144558 rows
RDF_QUAD 1.1 rows()
inlined P = ##type , S = k_s_44_30_t10.S G = #/sum
time 0.13% fanout 0 input 144558 rows
}
Appendix A. Appendix

Subquery Select( <none> )
}
) then 16 else 25 unkn 25
16: if {
    time 0.14% fanout 1 input 144558 rows
    time 4.4% fanout 1.12547 input 144558 rows
    RDF_QUAD 1.5 rows()
    inlined P = #/producer , S = k_s_44_30_t10.S G = #/sum
    time 0.13% fanout 0 input 162695 rows
    Subquery Select( <none> )
}
) then 20 else 25 unkn 25
20: if {
    time 0.12% fanout 0.625873 input 144558 rows
    time 5.8% fanout 7.62117 input 90475 rows
    RDF_QUAD 4.6 rows()
    inlined P = #/actor , S = k_s_44_30_t10.S G = #/sum
    time 0.49% fanout 0 input 689525 rows
    Subquery Select( <none> )
}
) then 24 else 25 unkn 25
24: BReturn 1
25: BReturn 0

END Node

After test:
0: if {
    time 0.17% fanout 1 input 180184 rows
    time 7.3% fanout 1.00645 input 180184 rows
    RDF_QUAD 1 rows()
    inlined P = #/page , S = k_s_44_30_t9.S G = #/sum
    time 0.36% fanout 0 input 181346 rows
    Subquery Select( <none> )
}
) then 4 else 9 unkn 9
4: if {
    time 0.16% fanout 1 input 180184 rows
    time 10% fanout 1.4988 input 180184 rows
    RDF_QUAD 1.1 rows()
    inlined P = #/type , S = k_s_44_30_t9.S G = #/sum
    time 0.56% fanout 0 input 270059 rows
    Subquery Select( <none> )
}
) then 8 else 9 unkn 9
8: BReturn 1
9: BReturn 0

time 3.9% fanout 1.40215 input 180184 rows
RDF_QUAD 1.6 rows(s_44_30_t11.0)
inlined P = #/writer , S = k_s_44_30_t10.S G = #/sum
time 1.5% fanout 0.0430327 input 252645 rows
Distinct (s_44_30_t11.0)

After code:
0: endS := := artm s_44_30_t11.0
4: BReturn 0

time 0.027% fanout 0 input 10872 rows
Subquery Select(endS)
}
time 2.5% fanout 0 input 10872 rows
RDF_QUAD 2.5 rows(s_45_34_t12.0)
inlined P = #/hasExtent , S = k_endS G = #/ext
After code:
0: isnotnull := Call isnotnull (s_45_34_t12.0)
5: sum resisnotnullset no set_ctr
10: BReturn 0
}
time 0.0024% fanout 0 input 1 rows
Select (res)
}