UCLA
UCLA Electronic Theses and Dissertations

Title
Integration, Provenance, and Temporal Queries for Large-Scale Knowledge Bases

Permalink
https://escholarship.org/uc/item/67x5d0n2

Author
Gao, Shi

Publication Date
2016

Peer reviewed|Thesis/dissertation

Integration, Provenance, and Temporal Queries for Large-Scale Knowledge Bases

Shi Gao

ABSTRACT OF THE DISSERTATION

Integration, Provenance, and Temporal Queries for Large-Scale Knowledge Bases

by

Shi Gao

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2016

Professor Carlo Zaniolo, Chair

Knowledge bases that summarize web information in RDF triples deliver many benefits, including support for natural language question answering and powerful structured queries that extract encyclopedic knowledge via SPARQL. Large scale knowledge bases grow rapidly in terms of scale and significance, and undergo frequent changes in both schema and content. Two critical problems have thus emerged: (i) how to support temporal queries that explore the history of knowledge bases or flash-back to the past; (ii) how to integrate knowledge from difference sources and improve the quality of integrated knowledge base while preserving the provenance information. In this dissertation, we propose a framework that supports knowledge integration, temporal query evaluation and user-friendly interfaces for large-scale knowledge bases. Towards this goal, we make the following contributions:

(i) We propose SPARQL^T , a temporal extension of structured query language SPARQL based on a point temporal model which simplifies the expression of temporal joins and eliminates the need for temporal coalescing. This approach makes possible an end-user interface HKB (Historical Knowledge Browser) where users can browse the evolution history of knowledge bases and express historical queries via simple by-example conditions in the infoboxes of Wikipedia pages.

(ii) We have designed and implemented RDF-TX (RDF Temporal eXpress), an efficient system for managing temporal RDF data and evaluating SPARQL^T queries. RDF-TX takes advantage

of compressed Multiversion B+ trees to achieve fast evaluation of temporal queries. The experimental result demonstrates that our indexing and query optimization techniques deliver superior performance over other systems.

(iii) We propose a framework for knowledge extraction and integration. We first introduce IBMiner, a novel NLP-based system that derives knowledge bases from free text and preserves the provenance of extracted triples. IBminer uses a deep NLP-based approach to extract subject-attribute-value triples from free text, and maps the attributes to those introduced in existing knowledge bases. Then we integrate public knowledge bases with the knowledge base generated by IBMiner into one of superior quality and coverage, called IKBStore. User-friendly interfaces are provided to manage the knowledge in IKBStore while maintaining provenance information.

The dissertation of Shi Gao is approved.

Yingnian Wu

Wei Wang

Junghoo Cho

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2016

To my family and friends

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

VITA

2009   Summer Exchange Student, UCLA

2010   Bachelor of Engineering in Software Engineering, Zhejiang University, Hangzhou, China.

2011 – 2012   Teaching Assistant, Computer Science Department, UCLA.

2011   Summer Intern, Teradata Optimization Group, El Segundo.

2012   Summer Intern, Teradata Optimization Group, El Segundo.

2010 – 2015   Research Assistant, Computer Science Department, UCLA.

CHAPTER 1

Introduction

Knowledge bases that summarize valuable information in the RDF format are rapidly growing in terms of scale and significance and playing a crucial role in many important applications such as text summarization, semantic search, and question answering. Significant research progress on text mining and crowdsourcing has lowered the barrier to constructing and accessing knowledge bases. While the volume of knowledge bases keeps growing, large-scale knowledge base undergo frequent changes in both schema and content. New challenges are brought to researchers: (i) how to support temporal queries that explore the history of knowledge bases or flash-back to the past; (ii) how to integrate knowledge from difference sources and improve the quality of integrated knowledge base while preserving the provenance information. The study of knowledge integration and temporal query support for large-scale knowledge bases has not yet received full attention it deserves. Next we discuss the unsolved issues in these aspects and the solutions we proposed to address the issues.

1.1   Temporal Query over the History of Knowledge Bases

As the real world evolves, the information in knowledge bases inevitably changes over time. Large knowledge bases undergo frequent changes. When the information in the real world evolves, the RDF triples stored in the knowledge bases are updated correspondingly by human editors and knowledge discovery systems. Table 1.1 lists the statistics of Wikipedia Infobox edit history, which shows that updates are quite common in many properties: e.g., on average each value in the population property of the city pages is updated more than 7 times. This is not specific to

1

Category	Property	Average Number of Updates
Software	Release	7.27
Player	Club	5.85
Musician	Genre	7.24
Country	GDP(PPP)	11.78
City	Population	7.16

Table 1.1: Statistics about the updates of Wikipedia Infobox

Wikipedia, but also happens in other knowledge repositories.

The management of historical information has emerged as a critical problem for knowledge bases, motivating the launching of projects such as DBpedia Live [dbp]. In fact, timestamping is an important part of the provenance information that is associated with each RDF triple in the knowledge base. The evolution history of knowledge bases captures and describes the change of real world entities and properties, and thus is of great interest to users. However, the size of the evolution history is very large and the schema of knowledge base is also under evolution, which poses challenges in query language, indexing and query processing.

As the RDF model for representing knowledge bases is gaining great popularity, the importance of managing and querying the evolution history of knowledge bases is also recognized. Gutierrez et al. [GHV07] extended the RDF model with time elements and several approaches [Gra10, PSH07, PUS08, TB09, KPG12] have been proposed to support the queries on temporal RDF datasets. Most previous works employ relational databases and RDF engines to store temporal RDF triples and rewrite temporal queries into SQL/SPARQL for evaluation. The languages proposed in these works use an interval-based temporal model which leads to complex expressions for temporal queries, e.g., those requiring joins and coalescing [CZ99, Tom96].

At the physical level, previous works rely on relational databases/RDF engine to store and query temporal RDF triples, which results in complex SPARQL and SQL queries in evaluation. Although some works exploit indices such as tGrin [PUS08] and keyTree [TB09] to accelerate the processing of temporal queries, these indices only support a limited set of temporal queries.

Queries that involve temporal join are not supported. On the other hand, as the size of historical knowledge base is rapidly growing, comprehensive indexing of temporal RDF data becomes prohibitively expensive. Existing temporal index suffers from its high space overhead and slow index scan. At last, none of previous works explore the query optimization techniques for temporal queries. These issues in existing systems limits their scalability on large knowledge bases and complex queries.

1.2 Knowledge Integration

The extraordinary success of Wikipedia and DBpedia [BLK09] shows that large-scale knowledge bases and powerful structure queries can achieve accurate knowledge retrieval and provide richer information comparing with normal web documents. In recent years, a tremendous amount of knowledge bases haven been released, including some general knowledge bases such as DBpedia and Yago [HSB13], and domain specific ones such as Geonames [GEO] and MusicBrainz [MUS]. Besides being very valuable for human readers, these knowledge bases have many applications in various information systems [MRS11, ATS11, JT10].

However, there remain the obstacles that existing knowledge bases are quite incomplete and inconsistent. For example, about 40% of Wikipedia pages are missing their InfoBoxes (*subject-attribute-value* triple) entirely, and the others contain InfoBoxes with missing entries. The problem of incompleteness is largely due to the crowdsourcing process used to populate knowledge bases, which is often compounded by the lack of general ontologies and the pervasive use of synonyms and coreferences to denote the same concept. Inconsistency is caused by the same reasons. When the information is present, it might be represented using synonymous attributes, such as *birth date*, *date of birth*, and *born*, as it is in fact the case for DBpedia and many manually created knowledge bases.

These problems limit the use of knowledge bases in the applications that require complete and robust results. There have been some works on knowledge extraction [HCH08, BEP08, SKW08, ECD04, PGR10, LBN10] to solve these problems. These works either consider structured data

sources [HCH08, BEP08, SKW08] or use limited forms of NLP-based techniques to extract knowledge from text [ECD04, PGR10, LBN10]. None of existing approaches (i) take full advantage of the linguistic morphologies of sentences to drive high-quality structured information from text; (ii) integrate the knowledge from text documents, structured knowledge bases, and crowdsourcing interfaces into a super knowledge base which covers more structured summaries with consistent terminology and ontology.

A general knowledge base with superior coverage and accurate information benefits a large set of semantic applications such as SWiPE [AZ12] and Facet Search [HBS10] by ensuring the quality of facts and providing more complete results. In the integrated knowledge bases, it is important to capture the provenance of knowledge. The provenance of knowledge is very valuable for human readers to identify the causes of changes and trace the evolution history. It is also important for semantic applications since provenance information can be used to ensure the reproducibility and quality of data. However, most works for provenance management in semantic web focus on the provenance of SPARQL queries [GKC13, DAA12]. New techniques are needed for managing provenance in knowledge integration.

1.3 Overview and Contributions

In this dissertation, we aim at addressing the issues of knowledge integration and temporal query support for large-scale knowledge bases. Towards this goal, we made the following contributions.

1.3.1 Query Language, Interface, and System for Querying the History of Knowledge Bases

We first introduce SPARQL^T, a temporal extension of SPARQL that can express powerful structured queries on temporal RDF triples. SPARQL^T is based on a point temporal model which simplifies the expression of temporal joins and eliminates the need for temporal coalescing. Then we demonstrate a query interface HKB (Historical Knowledge Browser) [GCA15] that allows users who are unfamiliar with knowledge base schema and SPARQL^T syntax to query the history of knowledge bases. In HKB, the Wikipedia Infoboxes are extended with temporal fields, where the

user can enter temporal query conditions. From the modified Infoboxes and query conditions, our interface derives equivalent queries that are optimized and executed in our query engine. It also allows users to browse the historical knowledge base by specifying a previous version or a period.

Then we implement a fast query engine RDF-TX (RDF Temporal eXpress) [GGZ15, GCA15] that efficiently supports the data management and query evaluation of large temporal RDF datasets while simplifying the temporal queries for SPARQL programmers and consequently, for end-user interfaces facilitating the expression of the same queries.

To achieve fast query evaluation, we use efficient storage and index schemes for temporal RDF triples based on Multi-Version B+ Tree (MVBT) [BGO96], and then build a query processor [GGZ15] that takes full advantage of such comprehensive indices to process SPARQL^T queries. RDF-TX also features a query optimizer that uses the statistics of temporal RDF graphs to find the efficient join orders for complex SPARQL^T queries.

1.3.2 Knowledge Integration

We tackled the problem of incompleteness with the use of IBMiner [MAG14, MKI13a], a text-mining system that derives knowledge bases from free text. IBminer first extracts semantic links from text using NLP framework Semscape [MGK14]. Then, by learning from the current InfoBoxes in Wikipedia, and relying on a large body of categorical information, IBminer converts the semantic links into the final InfoBox triples. IBminer also introduces a type-checking mechanism that automatically infers the acceptable value domains for each attribute, a piece of information that contributes greatly to the accuracy of final results.

Furthermore, we integrate public knowledge bases with the knowledge base generated by IB-Miner into one of superior quality and coverage, called IKBStore [MGZ13a]. To eliminate the duplicates and improve the inconsistency in the integrated knowledge base, we propose new techniques to generate context-aware synonyms for the entities and attributes that are used to reconcile knowledge extracted from various sources. IKBStore archives the provenance of knowledge for debugging and verification. Two user-friendly interfaces InfoBox Knowledge-Base Browser (IBKB)

and InfoBox Editor (IBE) [MGZ13b] are provided to manage the knowledge in IKBStore while maintaining provenance information.

In summary, in this dissertation we have made the following contributions:

(i) We propose SPARQL$^\text{T}$, a simple and effective extension of SPARQL for querying the history of RDF knowledge base. SPARQL$^\text{T}$ employs a point temporal model which simplifies the expression of temporal joins and eliminates the need for temporal coalescing.

(ii) We designed and implemented an efficient system RDF-TX for managing temporal RDF data and evaluating SPARQL$^\text{T}$ queries. Our system exploits MVBT to store and index temporal RDF triples. Then a hybrid method combining dictionary based compression and prefix encoding is adopted to reduce the storage overhead of indices. The algorithms on MVBT are extended and optimized to exploit the characteristics of the compression scheme and query patterns. The experimental result demonstrates superior performance and scalability of our query engine compared with other approaches.

(iii) We designed and implemented Semantic Web Information System, which consists of a set of tools for knowledge extraction and integration: IBMiner, CS^3, IBKB and IBE. IBminer uses a deep NLP-based approach to extract subject-attribute-value triples from free text, and maps the attributes to those introduced in existing knowledge bases. We then integrated public knowledge bases with the knowledge base generated by IBMiner into one of superior quality and coverage, called IKBStore. We also propose Context-Aware Synonym Suggestion System (CS^3) to generate contextaware synonyms for the entities and attributes that are used to reconcile knowledge extracted from various sources. Two interfaces IBKB and IBE were developed for browsing and editing the knowledge base, and archiving the provenance for verification.

The rest of this dissertation is organized as two main parts: (i) temporal query support over the history of knowledge bases in Chapter 2 and (ii) knowledge extraction and integration framework in Chapter 3 .

In Chapter 2, we provide an overview of RDF-TX system and temporal RDF model in Section 2.1. Then we present the design and syntax of SPARQL$^\text{T}$ in Section 2.2. Section 2.3 describes

the storage model and index compression techniques. We explain our query evaluation techniques in Section 2.4. Section 2.5 introduces a query optimizer for join order optimization in complex SPARQL^T queries. We evaluate our approach on real world datasets in Section 2.6, followed by interface in Section 2.7 and related work in Section 2.8.

In Chapter 3, we begin with an overview of the Semantic Web Information Management System for knowledge extraction and integration in Section 3.1. Then we introduce our knowledge integration techniques in Section 3.2, and present two subsystems: the text-mining system IBMiner which is described in Section 3.3 and the Context-aware Synonym Suggestion System described in Section 3.4. Section 3.5 describes our work on provenance management and Section 3.6 shows our crowdsourcing tools for knowledge browsing and editing. The evaluation results of SWIM system are presented in Section 3.7. We discuss the related work on knowledge extraction and integration in Section 3.8.

At last, we conclude this dissertation and discuss further work in Chapter 4.

CHAPTER 2

RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases

There is a growing interest in large scale knowledge bases such as DBpedia and Yago2, which play a key role in semantic applications. Many important properties such as occupation, role, and marriage status are time-dependent. As the real world evolves, the knowledge base is updated and the evolution history of entities and their properties becomes of great interest to users. Thus, users need query tools of comparable power and usability to explore such evolution histories or flash-back to the past. However, the size of the evolution history is very large and the schema of knowledge base is also under evolution, which poses challenges in query language, indexing and query processing.

As the RDF model for representing knowledge bases is gaining great popularity, the importance of managing and querying the evolution history of knowledge bases is also recognized. Gutierrez et al. [GHV07] extended the RDF model with time elements and several approaches [Gra10, PSH07, PUS08, TB09] have been proposed to support the queries on temporal RDF datasets. Most previous works employ relational databases and RDF engines to store temporal RDF triples and rewrite temporal queries into SQL/SPARQL for evaluation. The languages proposed in these works use an interval-based temporal model which leads to complex expressions for temporal queries, e.g., those requiring joins and coalescing [CZ99, Tom96]. At the physical level, previous approaches exploit indexes such as tGrin [PUS08] to accelerate the processing of simple temporal queries, but they do not explore the use of general temporal indices and query optimization techniques. This limits their scalability and performance on large knowledge bases and complex queries.

In this chapter, we describe a vertically integrated system RDF-TX (RDF Temporal eXpress)

that efficiently supports the data management and query evaluation of large temporal RDF datasets while simplifying the temporal queries for SPARQL programmers and consequently, for end-user interfaces facilitating the expression of the same queries.

To support the queries over the evolution history of knowledge bases, we develop efficient storage and index schemes for temporal RDF triples using multiversion B+ tree (MVBT) [BGO96] and implement a query engine which achieves fast query evaluation by taking advantage of comprehensive indices. We also develop a query optimizer that generates efficient join orders for complex queries using a cost-based model and the statistics of temporal RDF graphs.

RDF-TX provides a general and scalable solution for the problem of managing and querying the evolution history of RDF knowledge bases based on three main contributions:

1. We propose SPARQL^T , a temporal extension of structured query language SPARQL based on a point-based temporal model which simplifies the expression of temporal joins and eliminates the need for temporal coalescing. This approach makes possible end-user interfaces, such as those in [AZ12, GCA15], where queries are entered via simple by-example conditions in the infoboxes of Wikipedia pages.

2. We present an efficient system for managing temporal RDF data and evaluating SPARQL^T queries. Our system uses MVBT to store and index temporal RDF triples. An effective delta encoding scheme is introduced to reduce the storage overhead of indices. The algorithms on MVBT are extended and optimized to exploit the characteristics of the compression scheme and query patterns. The experimental result demonstrates superior performance and scalability of SPARQL^T query engine compared with other approaches.

3. We develop a query optimizer that finds the efficient join orders of SPARQL^T query patterns using the statistics of temporal RDF graphs. To manage temporal statistics, we introduce compressed Multi-Version SB Trees (MVSBT) that provides highly accurate estimation of statistics with a small storage overhead.
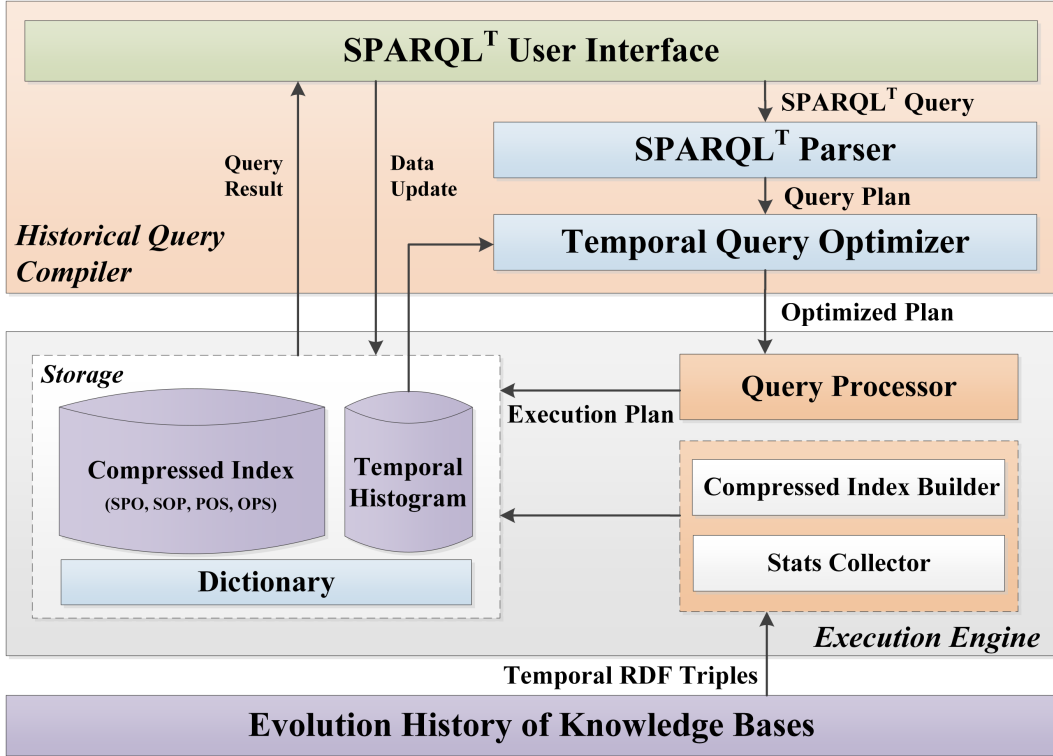
Figure 2.1: RDF-TX Architecture

2.1 Overview and Data Model

In this section, we first provide a general overview of RDF-TX system and overall workflow. Then we discuss the temporal RDF model introduced in [GHV07] and illustrate how we model the evolution history of knowledge bases using the temporal RDF model.

2.1.1 System Architecture

We design and implement from scratch a comprehensive system RDF-TX for managing and querying the evolution history of knowledge bases. Figure 2.1 shows the high level architecture of RDF-TX , which can be divided into two main components: (i) *Historical Query Compiler* that transparently compiles SPARQL^T queries and optimizes the query plans; (ii) *Execution Engine* that manages temporal data and evaluates the SPARQL^T queries.

Historical Query Compiler. The goal of SPARQL^T is to allow users to express a wide variety of historical queries in a simple way. To this end, we introduce SPARQL^T , a temporal extension of SPARQL. Users can write and submit SPARQL^T queries through a user-friendly interface. Our interface extends the *By-Example Structured Query (BEStQ)* to help casual users formulate historical queries without having to learn the schema of knowledge base and the SPARQL^T syntax. Since the usability is beyond the scope of this paper, interested readers are referred to [GCA15] for more details about the by-example query interface.

The SPARQL^T queries are compiled to query plans represented as graphs of query patterns and passed to temporal query optimizer to improve the order of joins. The optimized query plans are submitted to *Execution Engine* for evaluation.

Execution Engine. In our engine, the historical information is represented as temporal RDF triples and stored using MVBT indices that support fast query processing. We propose an effective delta encoding scheme to reduce the storage overhead of indices. The query processor transforms the query plans from compiler to execution plans expressed in query operators (e.g. temporal join) and executes them on the compressed MVBT indices.

2.1.2 Temporal RDF Graph

Knowledge bases such as DBpedia [BLK09] and Yago [HSB13] can be represented as RDF graphs which consist of a set of RDF triples in the format (*subject*, *predicate*, *object*) [1]. The subject and predicate of a RDF triple are elements from the set of Uniform Resource Identifiers $\mathcal{U}$, while the object is a URI from $\mathcal{U}$ or a value from the set of literals $\mathcal{L}$. For example, the RDF triple for "a university identified by http://www.w3.org/edu/University_of_California has 18,896 academic staff" is:

---

[1]For the sake of simplicity, we do not discuss the concept of blank nodes in this paper.
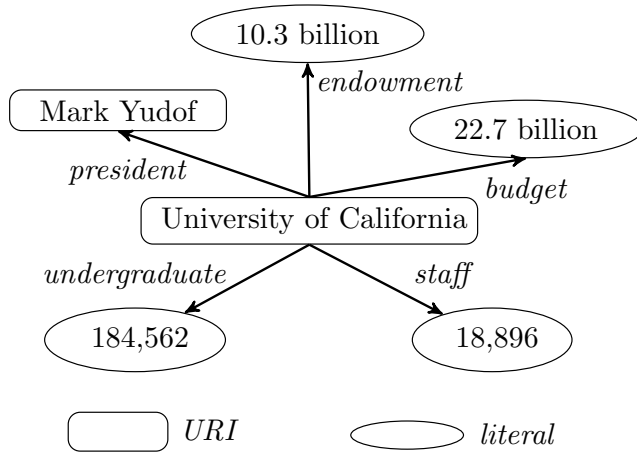
Figure 2.2: RDF graph based on the Wikipedia Infoboxes of University of California on 09/01/2013

- *subject*: http://www.w3.org/edu/University_of_California

- *predicate*: http://www.w3.org/elements/Staff

- *object*: 18896

In the rest of this paper, we assume the prefix parts of URI (e.g. http://www.w3.org/edu/) are given and simplify the representation of above RDF triple as (*University of California*, *staff*, *18896*). Figure 2.2 shows an RDF graph based on the Infoboxes of subject *University of California*, which is taken from the Wikipedia page on 09/01/2013. The predicate names are slightly modified to better illustrate our problem. In the RDF graph, each rectangle corresponds to a URI, and each ellipse corresponds to a literal value. Each edge represents an RDF triple and is labeled with a predicate name.

In reality, the RDF triples in the knowledge bases are updated frequently by human editors and knowledge discovery systems. The historical triples are usually archived in the backend databases or system logs. There is no support in existing knowledge management systems for reasoning such historical information. For example, when we check the Wikipedia page of University of California on 03/01/2015, most Infoboxes are different from the knowledge in Figure 2.2 due to

Predicate	Object	Timestamp
president	Mark Yudof	06/16/2008 . . . 09/29/2013
	Janet Napolitano	09/30/2013 . . . *now*
endowment	10.3	07/01/2013 . . . 06/30/2014
(billions)	13.1	07/01/2014 . . . *now*
undergraduate	184,562	05/14/2013 . . . 01/29/2015
	188,300	01/30/2015 . . . *now*
staff	18,896	08/29/2013 . . . 01/29/2015
	19,700	01/30/2015 . . . *now*
budget	22.7	01/30/2013 . . . 01/29/2015
(billions)	25.46	01/30/2015 . . . *now*

Table 2.1: The evolution history of subject *University of California* from 09/01/2013 to present, represented as temporal RDF triples.

the change of facts in the real world. In many knowledge bases such as DBpedia, the history is stored as many snapshots, which is inefficient in storage and difficult to search.

To make the evolution history queriable, we store it in the temporal RDF model [GHV07] that extends the RDF Graph with temporal elements. Each RDF triple is annotated with a temporal element to represent the time when this triple is valid. Formally, given a temporal element set $\mathcal{T}$, a *Temporal RDF Graph* consists of a set of temporal RDF triples that each temporal RDF triple is a RDF triple (s, p, o) annotated with a temporal element $t \in \mathcal{T}$. $\mathcal{T}$ is a point-based temporal domain. A set of temporal RDF triples with consecutive time points $\{(s, p, o) [t] \mid t_s \leq t \leq t_e\}$ can be encoded using interval-based expression as: $(s, p, o) [t_s \ldots t_e]$.

The evolution history of subject *University of California* is represented as a set of temporal RDF triples, as shown in Figure 2.1.2. All the triples share the same subject *University of California*. We use DAY as the granularity of time and *now* as the current time. Typically, one triple is valid over several days, which we represent with . . . between the start day and the end day (start and end included): e.g. [07/01/2013 . . . 06/30/2014] represents all the days between 07/01/2013 and 06/30/2014.

13

2.2 SPARQL^T Query Language

To query the temporal RDF graphs, we propose a temporal extension of SPARQL called SPARQL^T. In the design of SPARQL^T, the first issue to be addressed is the choice of the temporal model. Many existing works [PSH07, PUS08, TB09] use the interval-based model because of efficiency considerations. However, to express temporal queries, the interval-based representation requires additional operators such as temporal interval overlap, intersect and coalesce, which introduce complications and difficulties [CZ99, ZWZ06], particularly for casual users working with friendly wysiwyg interfaces. Therefore, we use a point-based temporal model that resolves these problems at the logical level; however at the physical level we retain the interval representation for efficiency reasons. We combine the benefits of both models, exploiting the fact that queries expressed on the point-based model can be easily mapped into equivalent queries on the interval-based model for execution.

SPARQL^T preserves all the standard syntax of SPARQL and has additional temporal patterns and constructs to express temporal que-ries. The input of a SPARQL^T query is a temporal RDF graph and the output is a set of mappings that replace the variables in SPARQL^T queries with values from the input temporal RDF graph. Next we explain the semantics and syntax of SPARQL^T queries via simple examples.

SPARQL^T Query Pattern. In SPARQL, a query consists of a set of graph query patterns $\{s\ p\ o\}$. To express a query, users specify the known parts with literals and the unknown parts with variables. For example, the SPARQL query pattern that finds the budget of University of California is: $\{University\_of\_California\ \ budget\ \ ?o\}$ in which subject and predicate are literals and object is a variable. Matching this query pattern against the RDF graph in Figure 2.2 returns the literal *22.7 billion*.

While SPARQL is powerful but it was not designed for temporal reasoning, thus we extend SPARQL query patterns with temporal elements to match the triples in the temporal RDF graphs. The syntax of SPARQL^T query pattern is: $\{s\ \ p\ \ o\ \ t\}$. Each element of a SPARQL^T query pattern

is a literal or a variable.

SPARQL^T query. Given the set of variables $\mathcal{V}$, a SPARQL^T query is a set of SPARQL^T query patterns $\{s\ p\ o\ t\}$ with optional FILTER clause f where $s \in \mathcal{U} \cup \mathcal{V}$, $p \in \mathcal{U} \cup \mathcal{V}$, $o \in \mathcal{U} \cup \mathcal{L} \cup \mathcal{V}$, $t \in \mathcal{T} \cup \mathcal{V}$, and f is a set of constraints with the elements from $\mathcal{U} \cup \mathcal{L} \cup \mathcal{T} \cup \mathcal{V}$.

In SPARQL, there are 8 types of query patterns as: S, P, O, SP, SO, PO, SPO, and full scan. For example, SP refers to a query pattern in which subject and predicate are literals and object is a variable. Full scan refers to the query pattern in which all three elements are variables. SPARQL^T supports 16 types of query patterns [2], which enable the expression of many interesting queries over temporal RDF graphs, as shown in following examples.

Temporal Selection. We first discuss *temporal selection* queries that have one SPARQL^T query pattern. An example of temporal selection query is the "when" query that retrieves the valid timestamps of given facts. Users only need to specify the values for RDF elements (*s, p, o*) and a variable for the temporal element.

EXAMPLE 1. When did Janet Napolitano served as the president of University of California.

SELECT [?t]

{University_of_California president Janet_Napolitano ?t.}

The square brackets in the SELECT clause indicate that the timestamps will be displayed in the compact format $[t_s \ldots t_e]$. Running Example 1 against the temporal RDF graph in Figure 2.1.2 returns [09/30/2013 . . . *now*].

One common type of temporal selection queries retrieves information from a previous version of the knowledge base. The temporal constraints (at a time point or within a period) can be easily specified in FILTER clause.

---

[2]The list of all query patterns in SPARQL^T : S, ST, P, PT, O, OT, SP, SPT, SO, SOT, PO, POT, SPO, SPOT, T, and full scan.

EXAMPLE 2. Find the budget of University of California in 2013. [3]

SELECT ?budget

{University_of_California budget ?budget ?t .

FILTER(YEAR(?t) = 2013) . }

Temporal Join. More complex queries often use temporal joins which, in SPARQL^T , are expressed by multiple query patterns that share the same temporal element. General temporal join may involve both key and temporal dimensions.

EXAMPLE 3. Find the name of the university in which Mark Yudof served as the president and the number of undergraduate students when he was in office.

SELECT ?university ?number [?t]

{?university undergraduate ?number ?t .

 ?university president Mark_Yudof ?t . }

Queries using multiple temporal joins are rather simple to express in SPARQL^T, whereas in languages based on interval-based temporal model, such queries tend to be much more complex. For example, if users want to search the number of undergraduate and graduate students when Mark Yuodf was in office, we only need to add one more query pattern: {*?university graduate ?number2 ?t*} to Example 3. If we switch to interval-based model, the query will consist of three query patterns and three temporal conditions: *?I$_1$ overlap ?I$_2$, ?I$_1$ overlap ?I$_3$, ?I$_2$ overlap ?I$_3$* where *?I$_1$, ?I$_2$, ?I$_3$* are three variables for intervals.

Besides temporal join, the point-based query patterns also support the expression of other temporal operations such as MEET and CONTAIN and more flexible temporal constraints. We define two built-in functions to facilitate the expression of complex constraints: *TSTART* and *TEND*. Given a temporal variable ?t, TSTAR-T(?t) refers to the earliest time point; TEND(?t) refers to the latest time point. With these two functions, SPARQL^T can readily express Allen's interval

---

[3]YEAR is a built-in function that returns the year of a date.

16

operators [All83].

EXAMPLE 4. Find who succeeded Mark Yudof as the president of University of California .

SELECT ?successor

{ University_of_California president Mark_Yudof ?t_1 .

University_of_California president ?successor ?t_2 .

FILTER(TEND(?t_1) = TSTART(?t_2)) . }

Temporal Aggregates. Temporal aggregation has been widely studied in the field of temporal databases. In SPARQL$^\text{T}$, temporal aggregates allow users to aggregate over a group of temporal RDF triples using "GROUP BY *?t*". The semantics of temporal aggregate is defined by assuming that (i) aggregate is first computed on all the snapshots, which results in a set of aggregate values annotated with timestamps (e.g. {100, 01/01/2008}, {100, 01/02/2008} ... {100, 03/01 /2008}); (ii) then the results that have identical non-temporal values and adjacent timestamps are merged (e.g. {100, [01/01/2008 ... 03/01/2008]}). In our system, this naive implementation has been replaced with the optimized techniques discussed in [YW01, ZMT08].

EXAMPLE 5. Find the total number of undergraduate students in the universities in California.

SELECT SUM(?student) [?t]

{ ?university category University_In_California ?t .

?university undergraduate ?student ?t . }

GROUP BY ?t

Duration. Many temporal queries involve the reasoning of duration. For instance, find the person who served as the president of University of California for more than one year. We define a built-in function *LENGTH* that counts the number of time points within the same consecutive period of time. If there are multiple valid intervals for a fact, we return the length of max duration. Another similar function *TOTAL_LENGTH* is defined to compute the total length of all valid intervals.

EXAMPLE 6. Find each person who served as the president of University of California for more than one year.

SELECT ?person [?t]

{ University_of_California president ?person ?t .

FILTER(LENGTH(?t) > 365 DAY) . }

2.3   Storage and Index

Since the performance of the query engine is heavily influenced by data index, it is important to choose appropriate index structure and storage schema for temporal RDF data.

A natural approach followed by previous works [PSH07] consists in managing temporal RDF triples using existing RDBMS. However, for searching both RDF information and temporal information, two sets of indices are required, and this can result in significant costs in storage and retrieval time, which are shown in Section 2.6. A second natural approach will be using RDF engines, such as Jena and Virtuoso, which have seen recent improvements in performance and functionality. However, this requires the standard RDF reification approach in which a temporal RDF triple is represented as an entity instance with five properties: subject, predicate, object, start time, and end time. Thus we need to use five triples for each temporal fact, whereby the space cost increases along with complexity of the queries and time required to optimize and execute them.

Therefore, rather than modifying and extending existing systems we design and build a new system that integrates advanced indexing and data compression techniques into an architecture conceived for efficient support of SPARQL^T queries

2.3.1   Index Scheme

Many index structures [BGO96, JSL00, LHN08, ND] have been proposed for temporal data. Each index has its own strength and they have shared issues such as space overhead and limited support

for general temporal queries.[4] After studying the literature and applications, we employ Multi-version B+ Tree (MVBT) to index temporal RDF data for the following reasons. First, MVBT is a bi-dimensional index with asymptotic worst-case guarantee and delivers good performance in real world datasets. Second, we propose an effective approach to compress MVBT which greatly reduces the space cost. The algorithms [BS96, ZTS02] are extended and optimized on compressed MVBT to improve the performance of index scan and join. Next we will briefly review the structure of MVBT and discuss the index scheme in RDF-TX system.

Multiversion B+ Tree. Multiversion B+ Tree [BGO96] is a temporal index structure with optimal worst case guarantees for data insert, update, and delete. Suppose the number of existing data items in MVBT is N. The space complexity of MVBT is $O(N)$. The complexity of a temporal query in version i is asymptotically equal to the complexity of the query on a B+ tree that maintains all the data valid in version i.

Rather than a single tree, an MVBT is actually a forest of trees. It has multiple root nodes and each of them corresponds to a temporal partition of data. Comparing with the entries in B+ tree, the entries in MVBT are extended with start version and end version to represent the live period of data. Thus the MVBT entry can be represented as (*key*, *start version*, *end version*, *data value/pointer*). An entry that stores data inserted in version i carries a valid period of (i, *now*). Delete operation modifies the end version (t_e) of a live period. The entries are sorted first by t_s and then by key. When there are too many entries or not enough live entries in an MVBT node, node structure changes (split or merge) are triggered. A simple example of MVBT is shown in Figure 2.3. We first insert five values into an empty MVBT in Version 1, which results in an MVBT tree shown in Figure 2.3 (a). Then we insert key 14 in Version 2 and delete key 46 in Version 3. The MVBT index becomes (b) with <14, 2, *> added to Node A and <46, 1, *> changed to <46, 1, 3> in Node B.

Indexing Temporal RDF. We implement in-memory MVBT and store all the temporal RDF triples in the MVBT indices. The insertion of an interval-encoded RDF triple $\{(s, p, o) [t_s, t_e]\}$ on MVBT index M is decomposed into two operations : (i) insert data item (s, p, o) into M at time

---

[4]We leave the detailed discussion of temporal index in Section 2.8.
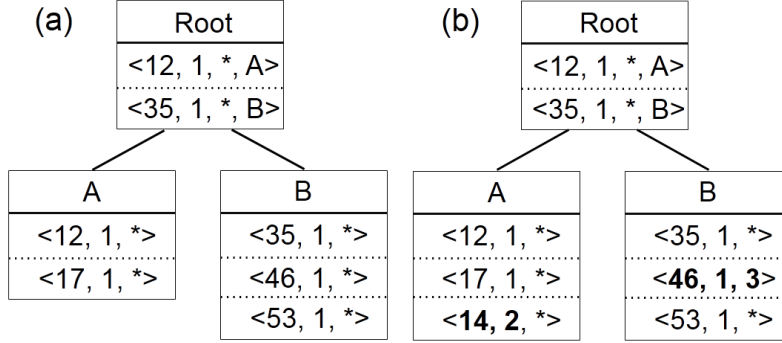
Figure 2.3: An Example of MVBT (***** denotes *now*)

t_s; (ii) delete data item (s, p, o) at time t_e. We use the unix time format for temporal elements (t_s, t_e) and store them as 64-bit integers.

Since the variable may be located in any position of (s, p, o), we create four MVBT indices (SPO, SOP, POS, OPS) for different orders of keys (s, p, o). These MVBT indices cover all 16 SPARQL$^{\text{T}}$ query patterns discussed in Section 2.2. For example, the MVBT index for temporal RDF triples in POS order can cover four patterns: P, PT, PO, POT. In query evaluation, the query engine parses the SPARQL$^{\text{T}}$ query and its prefix pattern to identify the corresponding MVBT index.

We employ dictionary encoding in the index construction. The use of dictionary reduces the index size and avoids the slow comparison between long string literals. Thus, before constructing the MVBT indices, our system scans the temporal RDF triples and replaces the literals with dictionary IDs. Then the triples that consist of IDs and timestamps are inserted into our indices. The mapping relations are maintained in our in-memory dictionary for index update and query evaluation. Since the main space cost in our indices is the large number of MVBT entries, dictionary encoding only reduces space cost by 10% - 20%. After dictionary encoding, we exploit delta compression which significantly reduces the space cost of MVBT indices, as shown in next section.

2.3.2 Index Compression

For the Wikipedia Infobox History, the size of one standard MVBT index implemented in Java is 1.5–2.2 times of the raw data. Moreover, a temporal RDF graph requires four MVBT indices. Therefore, if a naive approach is used, comprehensive indexing of temporal RDF data becomes prohibitively expensive. We next discuss effective compression techniques that address this problem.

We observe two characteristics of MVBT. First, the entries in MVBT node are sorted and neighboring entries often share the same prefix, which could be utilized to reduce space cost. Second, in MVBT all node structure operations start from an operation called *version split* that copies all the live entries to a new node. This guarantees the query performance but leads to a lot of long intervals. Given these characteristics, we introduce an effective delta encoding method to compress MVBT indices.

2.3.2.1 Compression Techniques

We design a compression scheme for variable delta encoding of MVBT entry. An MVBT entry for temporal RDF data consists of five values: $(v_1, v_2, v_3, t_s, t_e)$ where v_1, v_2, v_3 are elements in RDF triples. We store the minimum values for keys and timestamps in each node as base values. Since the data entries are sorted first by start version (t_s) then by key, most entries have very close start versions (t_s). Therefore for start versions, we only keep the minimal value of each node, and compute and store the delta start versions. For t_e, the compression rules are as follows: (i) if the valid interval (t_s, t_e) is a short interval, t_e is stored as the length of intervals; (ii) if the valid interval is long, t_e is stored as the delta value between t_e and minimum t_e in the node; (iii) if the valid interval is a live interval (t_e is *now*), a special flag is set and t_e is stored as empty. Other values (v_1, v_2, v_3) are compressed as the delta values (i) between current value and the value in neighbor entry or (ii) between current value and minimum value in leaf node.

The compressed values are stored in a compact byte array. Figure 2.4(a) illustrates the format of compressed MVBT entry. Every entry consists of three parts: header, key block (v_1, v_2, and v_3),

| Uncompressed | Value1, Value2, Value3 | | Ts, Te | |
| Compressed | Header (2 bytes) | Key Block | Time Block | |

| H Flag | Payload | Te Flag | | V1 Flag | V1 Data | ... ... |

| Compact Header (1byte) | H Flag | Payload |

(a)

# of triples (million)	5	10	15	20	25	30
Time (seconds)	1.36	2.65	3.87	4.81	6.18	7.25

(b)
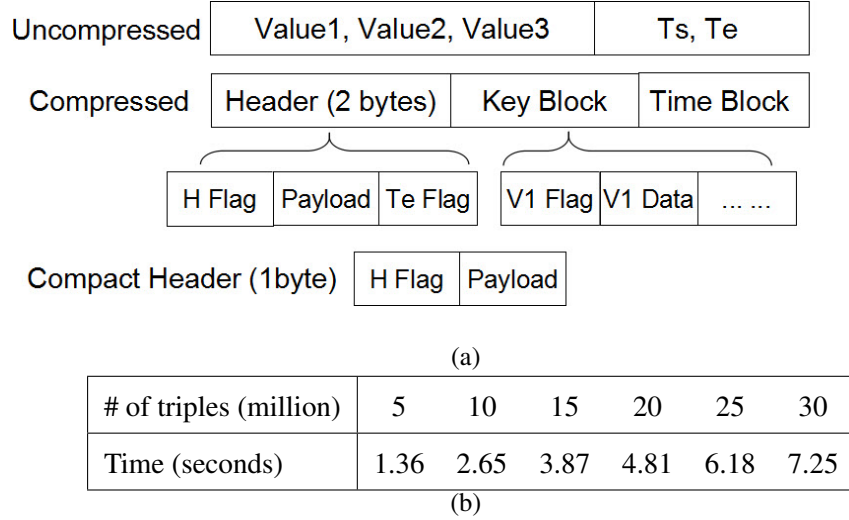
Figure 2.4: (a) Compressed MVBT Entry Format (b) Compression Time

and time block (t_s and t_e). A normal header (2 bytes) contains a flag (H Flag, 1 bit) for header type (normal/compact), a payload (13 bits in total, 7 bits for key block and 6 bits for time block) that stores the number of bytes for each delta value, and the t_e flag (2 bits) that records the compression rule for t_e. For the delta values in key block, we use 1 bit to record how the delta is computed (with neighbor or with node minimum value).

We observe that in large datasets, it is very common that two neighboring MVBT entries (i) share at least one element in key block; (ii) have very close t_s (delta size $\leq$ 4 bytes) (iii) both t_e are *now*. Case (iii) is common in MVBT due to *version split* operation. Thus for these entries, we propose a compact header which consists of 1 bit header type and 7 bits payload (for two delta values in key block and t_s delta value).

There is a trade-off between the compression ratio and query performance. Since the number of index nodes is much smaller than the number of leaf nodes and the index nodes are accessed more frequently than leaf nodes, we only compress the leaf nodes of MVBT indices. As shown in evaluation (Section 2.6), the size of compressed MVBT is about 24% of standard MVBT.

We build MVBT indices for different subsets of Wikipedia dataset and then test the time of compressing MVBT entries, as shown in Figure 2.4(b). The result shows that it takes very little

22

time to apply the delta encoding technique. Given an MVBT index built from 30 million temporal RDF triples, the time for compressing all the leaf nodes is only 7.25 seconds.

2.3.2.2   Index Maintenance and Search

An important principle of index compression is to reduce the storage overhead while maintaining the performance of index update and search. For data insertion, we first look up index nodes and identify the leaf node to be updated. In the leaf node, we decompress the start versions (t_s) to find the position of input start version i and compute the delta values of input data. Then we modify the $(i + 1)$th entry if its delta values are changed. One issue is that we need to scan from the beginning of all entries. To address this issue, we add a checkpoint in each node that stores the position of MVBT entry with largest t_s. Then in data insertion, since the t_s of input data must be larger than existing t_s, we only decompress the entries after checkpoint. Deletion in MVBT only updates the end version of a live entry. Thus we simply scan all the entries and modify the t_e of matched entry. As shown in Section 2.6, insertion/deletion on compressed MVBT only takes 5% more time than standard MVBT.

In query evaluation, instead of decompressing the whole tree, the query engine only decompresses the entries that are accessed for input query. When a leaf node is visited, query engine first reads the timestamps to see if the node region intersects the query region. Then partial or complete key blocks of temporally overlapped entries are decompressed and checked for other conditions.

2.4   Query Processing

In this section, we present the design and implementation of RDF-TX query engine, which makes use of MVBT to process the temporal operations of the language.

2.4.1   Compiling SPARQL$^\text{T}$ Query

The overall evaluation of SPARQL$^\text{T}$ queries consists of four steps:

- Parse the input query and translate point-based query patterns to interval-based query patterns.

- Construct a query plan. The plan is represented as a graph in which each node is an interval-based query pattern.

- When the query contains multiple temporal joins, optimize the query plan to improve the join order.

- Translate the query plan to an execution plan that is evaluated on compressed MVBT indices.

Next we elaborate each step with more details.

Translating Query Patterns. Since the temporal RDF graph is stored as interval-based temporal RDF triples, we translate the point-based SPARQL^T query patterns to the interval-based patterns that can be converted to range queries and executed on MVBT. For key elements, we take the literals as prefix and convert the unknown parts to key ranges. For temporal element (t), if there exist temporal constraints in the FILTER clause, we generate time ranges based on the constraints; otherwise, the default range is [*0*, *now*] where *0* refers to the minimum time point. Consider the query pattern {*University of California budget ?budget ?t*} (*YEAR(?t) = 2013*) in Example 2. The interval-based query pattern can be described as a query region with key range and time range as follows:

- key range: (*University of California*, *budget*, _) – (*University of California*, *budget*, ∞)

- time range: *01/01/2013 – 12/31/2013*

Here _ and ∞ denote the extrema of the string domain.

Constructing and Optimizing Query Plan. The query engine generates a query plan that consists of interval-based query patterns from the first step. This query plan can be represented as a graph in which the edges between the nodes are added when two query patterns share the same variable. If there are multiple join operations, the query optimizer (discussed in Section 2.5) is

called to find efficient query plans using the statistics of temporal RDF graphs.

Executing the query plan on MVBT. Lastly, the optimized plan is translated to an execution plan which is similar to the query plan in relational databases. Every query pattern is converted to an index scan operator on MVBT indices. For instance, the SPARQL^T query in Example 2 is executed by performing an index scan on the MVBT for SPO order since it only contains one SP query pattern. Then the join operators are added based on the optimized join order. Finally, appropriate filter operators are added using the FILTER clause of SPARQL.

2.4.2   Executing Query Plan

Next we describe the implementation of index scan and temporal join in our query processor. Other operators such as filter and aggregate are implemented similar to their counterparts of existing engines thus omitted.

2.4.2.1   Index Scan

We perform an index scan for each interval-based query pattern. For the index scan on MVBT, we employ the link-based range-interval algorithm [BS96] which introduces *Backward Link* in MVBT to process the range queries. The MVBT leaf nodes are equipped with backward links that point to the temporal predecessors. The index scan is performed as: (i) search all the nodes that intersect the right border of query region; (ii) follow the backward links of the nodes to find all the nodes that intersect query region; (iii) scan the leaf nodes found in the first two steps to retrieve the entries. An example of linked index scan is shown in Figure 2.5. The shadowed rectangle represents a query. MVBT nodes D and E are first visited. Then as the predecessor of D and E, node B is checked. Lastly, node A is visited. Note that index scan is performed on compressed indices, thus step (iii) is implemented based on our compressed index search in Section 2.3.2.2.
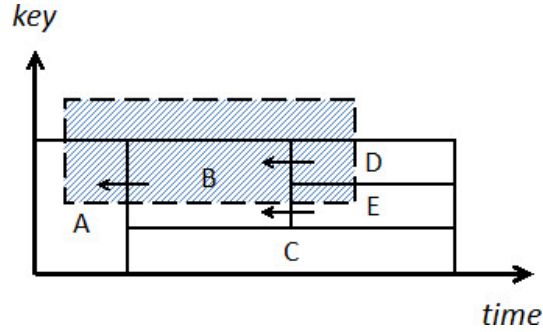
Figure 2.5: An Example of MVBT Backward Link

2.4.2.2 Temporal Join

Temporal join represents one of the most expensive operations in the temporal query language, especially when the size of knowledge base is very large. In our implementation, we explore three types of joins: *Merge Join*, *Hash Join*, and *Synchronized Join*.

Merge join is very popular and widely used in existing SPARQL engines [NW10, YLW13]. These systems build indices for all permutations so that the optimizer leverages the indices to perform order-preserving merge joins. However, this does not work for MVBT index since the entries are sorted by time. With MVBT, we need to materialize and sort the intermediate result to do merge join, which is much slower than hash join. For temporal join, we first build hash tables based on the joint key variables (s, p, o). Then for matched triples, we check if their valid intervals are overlapped.

When the size of result is large, the cost of building a hash table may be very expensive. Thus we extend the synchronized join [ZTS02]. The basic idea of synchronized join is as follows: (i) synchronously find the set of all MVBT node pairs (e_1, e_2) that intersect each other and the right border of query region; (ii) join e_1 and e_2; (iii) join the predecessors of e_1 and e_2 by following the backward links. This algorithm avoids materializing the intermediate result, but it is much slower than hash-based join since one page and its predecessors are visited many times. So we optimize this algorithm by caching recently visited records; that is, given a page e from step (i), we cache the records in e and its predecessors, and perform joins between e and other pages. This optimized synchronized join is used when the query pattern in the join accesses a large portion of index (e.g.

find all the triples valid in a certain period).

2.5 Optimization

In RDF-TX , improper join orders may generate large intermediate results and slow down execution. Therefore, a natural step is to optimize complex SPARQL$^\text{T}$ queries by finding efficient join orders. The key of join optimization is to efficiently estimate the costs of different join orders, which is not a trivial task for temporal queries. In this section, we present a query optimizer that uses estimated statistics of temporal RDF graph to optimize the orders of temporal joins in SPARQL$^\text{T}$ queries.

2.5.1 RDF-TX Query Optimizer

For queries that involve multiple temporal joins, we implement a query optimizer that uses bottom-up dynamic programming strategy [MN06] to find the cost-optimal query plans. Our optimizer generates multiple query plans and finds the plan with lowest estimated cost. A large query plan is generated by joining two small optimal query plans. The cost is computed based on the cardinalities of query patterns and intermediate results.

The cardinality estimation is a well-studied problem in relational databases and SPARQL engines [NM11, NW10, SSB08]. To estimate the cardinality of join result, an effective approach is characteristic set [NM11]. In a RDF graph R, the characteristic set $S_C(s)$ of a subject s is the set of related predicates: $S_C(s) = \{p | \exists o, (s, p, o) \in R\}$.

The idea of characteristic set is that semantically similar subjects (e.g. University of California and University of Michigan) usually have the same characteristic set. For every characteristic set, the number of distinct subjects that belong to the characteristic set, and the number of occurrences of the predicates in these subjects are recorded and used to estimate the cardinality. For example, given a characteristic set {*president, undergraduate*}, there are 100 distinct subjects belong to this characteristic set. The numbers of occurrences for *president* and *undergraduate* are 150 and 110

respectively. Consider a SPARQL query with two query patterns:

SELECT ?s ?o1 ?o2 .

{?s president ?o1 .

?s undergraduate ?o2 . }

Suppose that only this characteristic set contains both predicates in the query. Then the result cardinality is estimated as: $100 \times \frac{150}{100} \times \frac{110}{100} = 165$.

Characteristic sets provide highly accurate estimation of cardinality. But it can not be used to estimate the cardinality of SPARQL$^\text{T}$ queries since the statistics of temporal RDF graph vary on different time points. Consider following SPARQL$^\text{T}$ query:

SELECT ?s ?o1 ?o2 ?t

{?s president ?o1 ?t.

?s undergraduate ?o2 ?t.

FILTER(?t ≤ 01/01/2013) . }

To estimate the cardinality of this SPARQL$^\text{T}$ query, we need to know the number of subjects that (i) belong to the set of temporal RDF triples valid in the period [0, 01/01/2013] and (ii) share the the characteristics set {*president, undergraduate*}. These statistics change with time and none of existing data structures can provide estimation of these statistics. Thus we introduce a temporal histogram to maintain the statistics of temporal RDF data in next Section. With temporal histogram, the characteristics sets can be easily integrated into our query optimizer. For each partial query plan, we compute the time ranges and search the histogram to retrieve the statistics of characteristic sets. Then the cardinalities are used to compute the cost and find efficient join orders.

2.5.2 Temporal Histogram

Join order optimization for SPARQL$^\text{T}$ query needs the total number of subjects and predicates for a given characteristic set in the time range specified by input queries. This problem is similar to the temporal aggregation that computes the aggregate value in a certain period. Thus we propose
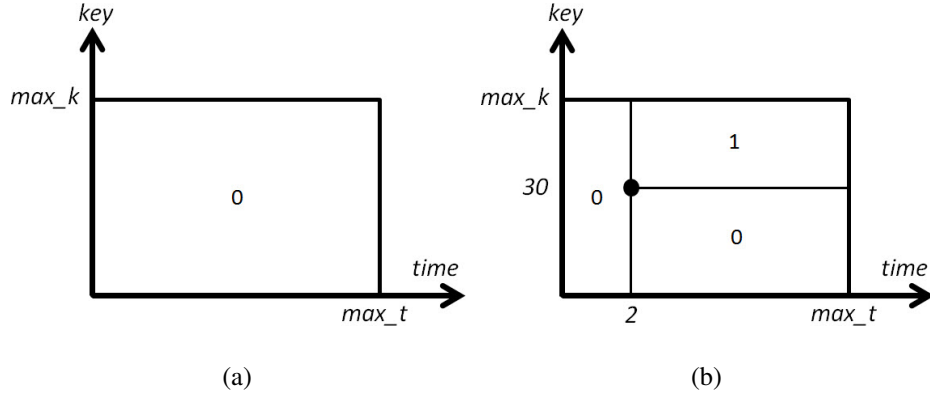
Figure 2.6: An Example of MVSBT Record Split

Compressed MVSBT that extends the temporal aggregate index Multiversion SB Tree [ZMT08] to maintain the statistics of characteristic sets. Next we review the structure of Multiversion SB Tree, and then introduce compressed MVSBT.

Multiversion SB Tree (MVSBT). MVSBT [ZMT01, ZMT08] is a data structure that combines the features of MVBT and SB Tree [YW01] to compute temporal aggregates. Similar to MVBT, MVSBT is a forest of trees with multiple root nodes and each of them points to an SB Tree for a temporal partition of data. Each data record in MVSBT has a key range (k_s, k_e), an interval (t_s, t_e), and a value v. The key range and the interval represent the rectangle covered by this record in key-time space. v maintains the aggregate value. The key-time rectangles of the records are mutually disjoint and the union of all the rectangles is equal to the whole key-time space. The split operation splits the rectangle on the new point. Given a query (k, t), MVSBT computes the aggregate values in the query region (key range: $[0, k]$, time range: $[0, t]$). An example of MVSBT for aggregate COUNT is shown in Figure 2.6. Figure 2.6(a) shows the initial record of an empty MVSBT. The aggregate value is 0 since no point is inserted. Then one point with key 30 and timestamp 2 is inserted. The initial record is split into three records as shown in Figure 2.6(b). The record on the top right corner has aggregate value 1 and other records has aggregate value 0 since all the points in top right record are larger than split point (30, 2). Suppose we have two queries (10, 1) and (40, 5). The first query point falls in the left record (time range is $[0, 2]$) and returns 0. The second query points falls in the top right record and thus gets the result 1.

29

Compressed MVSBT. Although MVSBT has good performance on temporal aggregation, it takes too much space for storage. Since query optimization does not require very accurate estimates, we can trade accuracy with efficiency. Compressed MVSBT (CMVSBT) is based on the idea that instead of accurately recording the points and splitting rectangles for every new point, a CMVSBT record can contain m ($m \geq 1$) points. Then we can estimate the aggregate value using the ratio of covered space to full space. Instead of storing the exact values of points, we store the statistic values such as total number of points and max value. The structure of leaf and index CMVSBT records is as follows:

- Leaf Record: $< k_s, k_e, t_s, t_e, k_m, t_m, v, c >$
- Index Record: $< k_s, k_e, t_s, t_e, list, ptr, c >$

where k_s and k_e are the start value and end value of key range; t_s and t_e are the start value and end value of interval; k_m and t_m store the max key value and time value of the points located in the rectangle of this record; $list$ is a list of points; ptr is the pointer to a CMVSBT node; v and c are *fixed* and *current statistic values*. *fixed statistic value* refers to the aggregate value, while *current statistic value* refers to the aggregate value computed over the points contained in the current record. The final statistic value is estimated by combining both values (discussed in Section 2.5.3).

Since the index nodes are visited more frequently than leaf nodes, we store the exact values of points in a list in the index nodes, while in leaf nodes we only maintain three statistics (k_m, t_m, c). The algorithm for data insertion in CMVSBT is similar to the one for MVSBT. Instead of splitting the record for every input point, CMVSBT record is split when the number of points in a record is larger than the threshold. The split point is (k_m, t_m). The algorithm for record splitting in CMVSBT for COUNT is shown in Figure 2.7. Let c_m and l_m denote the thresholds for the number of points in leaf nodes and index nodes. When a new point p (k, t) is inserted into compressed MVSBT, we look up the index nodes to find a set of nodes N whose rectangles cover this point. In a leaf node n_f, if p falls in the rectangle of record, c is increased by 1, and the max values (k_m, t_m) are updated if $p.k > k_m$ or $p.t > t_m$. Then if $c = c_m$, we split the record based on the position (k_m, t_m). After split, the statistical values (v) of new records will be equal to (i) $c/2 + v$

if the new record has the same k_s with old one; (ii) $c/2$ otherwise (based on the logical splitting in MVSBT [ZMT08]). We use $c/2$ since we assume the points are uniformly distributed in the record. The c of new records are initialized to be 0. In an index node $n_i \in N$, if r_i is the lowest record that fully covers p, p is appended to the end of $list$. Like MVSBT, compressed MVSBT also assumes that the data items come in nondecreasing time order. Thus $list$ is automatically sorted by time. If length($list$) $= l_m$, the record is split on $p.t$ and c is copied to new record. If $r.k_m = r.k$ or $r.t_m \neq r.t_s$, the split point ($r.k_m$, $r.t_m$) falls on the borders of rectangle. Then r is split into two records. When we set $c_m = 1$ and $l_m = 1$, the algorithm in Figure 2.7 is the same with the split algorithm of MVSBT. More details about CMVSBT construction are available in our technical report [GGZ15].

Algorithm: leafRecordSplit(n_f, r, q)

Input: CMVSBT leaf node n_f, record r, new point p (k, t)

1: $r.c = r.c + 1$

2: **if** $p.k > r.k_m$ **then**

3:      $r.k_m = p.k$

4: **end if**

5: $r.t_m = p.t$

6: **if** $r.c = c_m$ **then**

7:      **if** $r.k_m \neq r.k_s$ and $r.t_m \neq r.t_s$ **then**

8:          // Split r to three records

9:          $v' = r.c/2 + r.v$

10:          $r_1 =$ new record($r.k_s$,$r.k_m$,$r.t_m$,$r.t_e$,k_s,t_m,v',0)

11:          $r_2 =$ new record($r.k_m$,$r.k_e$,$r.t_m$,$r.t_e$,k_m,t_m,$r.c/2$,0)

12:          n_f.add(r_1); n_f.add(r_2)

13:          $r.t_e = r.t_m$

14:      **else**

15:          // Split r to two records, similar to step 7 - 11, omit

16:      **end if**

17: **end if**

Algorithm: indexRecordSplit(n_i, r, p)

Input: CMVSBT index node n_i, record r, new point p (k, t)

1: $r.list$.add(p)

2: **if** length($r.list$) $== l_m$ **then**

3:      $r_1 =$ new record($r.k_s$,$r.k_e$,$p.t$,$r.t_e$,$new\ list()$,$r.ptr$,$r.c$)

4:      n_i.add(r_1)

5:      $r.t_e = p.t$

6: **end if**

Figure 2.7: Algorithms for Record Split in CMVSBT
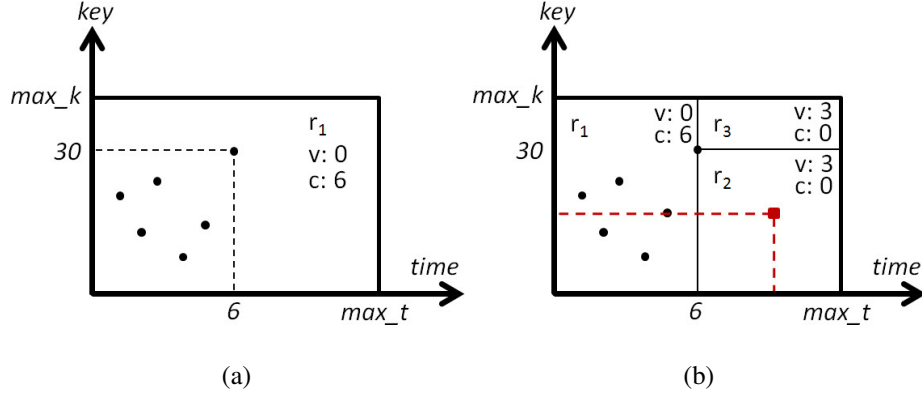
Figure 2.8: An Example of CMVSBT Leaf Record Split

Here we prepare a simple example to illustrate the process of record split in CMVSBT. We assume that we have inserted six points into a compressed MVSBT, as shown in Figure 2.8(a). The threshold c_m is set to be 6. The max key $r_1.k_m$ is 30 and the max time $r_1.t_m$ is 6. Although the rectangle of r_1 is the whole space, all the points fall in the effective rectangle rec (key range:[0, k_m], time range:[0, t_m]). Since $r_1.c \geq 6$, we split it into three rectangles as shown in Figure 2.8(b). The values of r_1 are not changed since all the points still fall in r_1. $r_2.v$ is approximated by the number of points covered by the virtual center of r_2 (the red point). As we can see, the red rectangle covers half of rec, so $r_2.v = r_1.c/2 + r_1.v = 3$, $r_3.v = r_1.c/2 = 3$.

For each characteristic set, we need to maintain: (i) the number of distinct subjects and (ii) the number of predicate occurrences. As discussed in next section, each type of statistic values requires two CMVSBTs: one for start points and one for end points. Thus our temporal histogram consists of four CMVSBTs and the schema of characteristic sets. In RDF-TX , we set the max size of temporal histogram as 10% of raw data. If the size of temporal histogram is larger than the threshold, we increase c_m and l_m and merge the neighbor records until the temporal histogram is small enough.

2.5.3 Statistics Estimation

Given a query q (k, t), compressed MVSBT estimates the statistics v_a in the query region (key range: $[0, k]$, time range: $[0, t]$). The algorithm consists of two main steps: (i) starting from root node, we look up the CMVSBT nodes whose rectangle covers q; (ii) in each node, we find all the rectangles whose time range contains t and $k_s \leq k$, and accumulate the approximate statistic value v_a of these rectangles. In a CMVSBT record, v_a equals to the sum of fixed statistics value v and current statistic value v_e. v_e is approximated by multiplying c by the proportion of query region in the rectangle $ratio$, as $c \times ratio$ where $ratio = ratio_k \times ratio_t$. If $q.k \geq r.k$, $ratio_k = 1$; otherwise, $ratio_k = (r.k_m - q.k)/(r.k_m - r.k_s)$. And $ratio_t$ can be computed in a similar way.

CMVSBT supports the point-based query that estimates the number of points in the query rectangle between (0,0) and query point (k, t). However, the query pattern in SPARQL$^{\text{T}}$ is translated to a range query whose start version and key are not necessarily 0. Thus we use the query reduction approach [ZMT08] which reduces one range query into four point queries. In this approach, we need two CMVSBTs for the start points and end points of temporal RDF triples. Then the statistics in the query region (key range: $[k_1, k_2]$, time range: $[t_1, t_2]$) is calculated as:

$Q_s(k_2, t_2)$ - $Q_e(k_2, t_1)$ - $Q_s(k_1, t_2)$ + $Q_e(k_1, t_1)$

where $Q_s(k, t)$ and $Q_e(k, t)$ refer to the point queries on the CMV-SBT of start points and end points respectively.

An example of query reduction is shown in Figure 2.9. The shadowed rectangle represents the interval query. Then the value is: $Q_s(k_C, t_C)$ - $Q_e(k_A, t_A)$ - $Q_s(k_D, t_D)$ + $Q_e(k_B, t_B)$. Assume that CMVSBT returns accurate value, then the result is 5 -2 -2 + 1 = 2.

During query optimization, we cache all the statistics to reduce the time on scanning CMVS-BTs. When one statistic value is required, we first search the statistics cache. If it is not cached, then we use the CMVSBTs to estimate the statistic value.
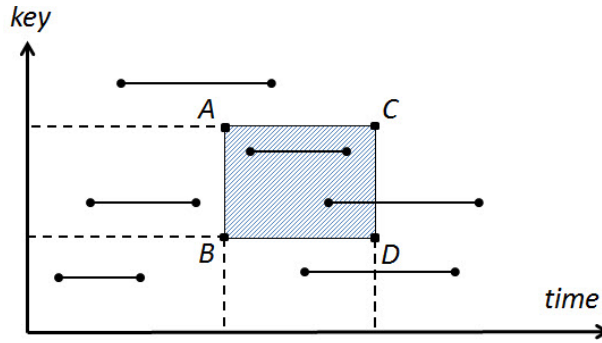
Figure 2.9: Example of Query Reduction

2.6 Experimental Evaluation

RDF-TX is implemented in Java as a sequential main memory query engine. To evaluate the performance of our system, we conduct experiments on two real world datasets and compare results with other approaches including RDF Reification, RDBMS-based approach, and Named Graph.

2.6.1 Experiment Setup

2.6.1.1 Dataset

Wikipedia. Wikipedia [AGD13] is a real world dataset extracted from the edit history of English Wikipedia (2003–2012). The dataset is in JSON format. We parse the raw file and generate 38.5 million temporal RDF triples as our test benchmark. This dataset contains the edit history of 1.8 million subjects and 110,000 predicates. The number of predicates is very large since many predicates are used only once in the edit history (some of them are spelling mistakes).

GovTrack. GovTrack [gov] is a public dataset about US congress. It contains the information of congressman/house of representative, bills, and votes. We parse the XML source files and generate 22 million temporal RDF triples. There are 0.4 million subjects and 60 predicates in this dataset.

2.6.1.2 Implementation and Configuration

RDF Reification. RDF reification provides a way to store RDF triple and its meta knowledge in standard RDF model by representing annotated RDF triple as an entity with following properties: *subject, predicate, object, meta knowledge*. Similarly, we represent a temporal RDF triple as an entity with five properties: *subject, predicate, object, start time, end time*. Then SPARQL^T queries are easily rewritten to SPARQL queries. We evaluate the reification approach in three well known RDF engines: Jena v2.13 [WSK03], Virtuoso v7.20 [vir] and RDF-3X v0.3.8 [NW10].

RDBMS-based Approach. An alternative approach to store temporal RDF data is to use relational databases. Temporal RDF triples can be stored in a relational table with five columns *subject, predicate, object, start time, end time*. We choose MySQL memory engine (v5.5) in our evaluation since it supports in-memory B+ tree index, which makes it a good competitor for our compressed MVBT implementation. We build four B+ tree indices on SPO, SOP, PSO, and OPS, which are similar to the index design of RDF-TX query engine. Additionally, we build two B+ tree indices on start time and end time for the evaluation of temporal constraints. The SPARQL^T queries are rewritten to SQL queries with self joins.

Named Graphs. Named graph [CBH05] is an extension of RDF model that identifies graphs with URLs and allows graph metadata such as provenance and trust. We implement the approach described in [TB09] that stores temporal information as graph metadata using Jena Named Graph implementation. We also test Ng4j v0.9.3 implementation [BCW05], but it is much slower than Jena and other approaches. Thus we only report the results on Jena Named Graph. The results on Ng4j are available in our technical report [GGZ15]. In the rest of this paper, we use "**Jena Ref**" and "**Jena NG**" to denote Jena Reification and Jena Named Graph respectively.

RDF-TX . Our query engine is a single-thread implementation using compressed MVBT as indices. Only the construction of compressed MVBT is paralleled (using at most four threads). The node capacity of MVBT is 200 for storage efficiency.[5]

We also test STUN system [KPG12] that supports queries on annotated RDF. Building STUN

---

[5]The relationship between node capacity and MVBT size has been discussed in previous works [AS13, BGO96].
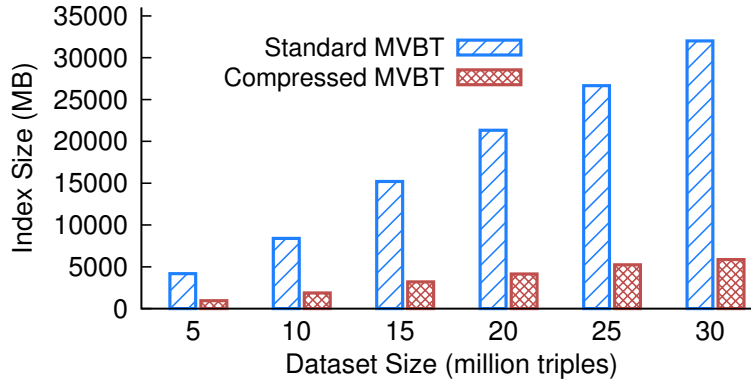
Figure 2.10: Compression Saving for MVBT Index

index for large temporal datasets takes a lot of time. For a subset of Wikipedia with 10 million triples, index construction is not finished within 12 hours. Thus it is not reported in our evaluation.

All the experiments are performed on a machine with 4 AMD Opteron 6376 CPUs (64 cores) and 256GB RAM running Ubuntu 12.04 LTS 64-bit. The index decompression time is included in the query execution time. The execution time reported is calculated by taking the average of 5 runs.

2.6.2 Index Space

We first investigate the effectiveness of our delta encoding techniques (Section 2.3.2). We implement the standard MVBT indices (4 indices: SPO, SOP, POS, OPS) with numeric keys as baseline. Figure 2.10 shows the space costs of standard MVBT and compressed MVBT in Wikipedia dataset. On average, our delta encoding technique reduces the space cost of MVBT by 76%.

Then we compare the space overhead of compressed MVBT and other types of index in Wikipedia, as shown in Figure 2.11. Since Wikipedia has a large number of unique timestamps, most named graphs are very small ($\leq$ 5 triples). Thus indexing named graph incurs a lot of overhead and Jena Named Graph takes far more space than other approaches. The space cost of MySQL memory engine and Jena Reification are similar, which are 3-4 times of raw data. RDF-3x has a size of only twice the raw data due to its implementation of runtime execution engine (almost no
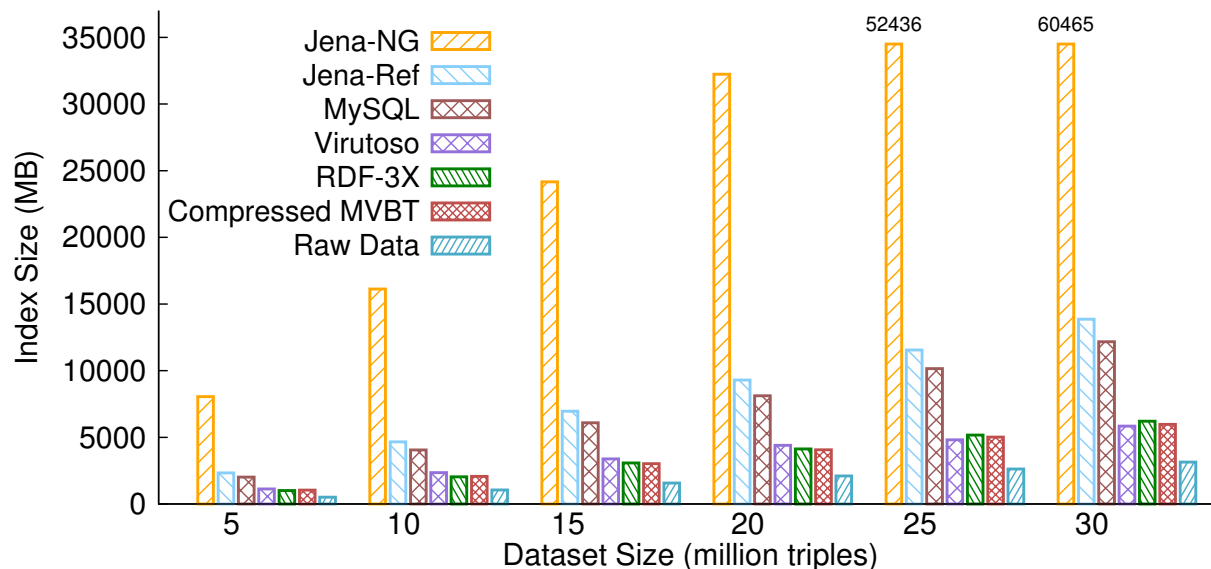
Figure 2.11: Index Size Comparison. The size of dictionary is included in the results.

extra indices need to be created in advance). For Virtuoso, index size is similar because of its column-wise compressed storage and a clustered index scheme for both row and column-wise tables introduced in recent versions. The index space of our implementation is almost the same with Virtuoso and RDF-3X, while the query performance is much better as shown in Section 2.6.3. The space of our comprehensive indices (4 compressed MVBT + dictionary) is about 1.3 - 1.8 times of raw data. The results for GovTrack dataset are similar and thus omitted.

2.6.3 Query Performance

To evaluate the query performance of our system, we created three sets of queries: (a) Temporal selection queries. Each query consists one query pattern and several temporal constraints, as in Example 2 discussed in Section 2.2; (b) Temporal join queries. Each query has 2 query patterns (1 temporal join), as in Example 4; (c) Complex queries which have 3 or more query patterns (2 or more temporal joins). We use the first two query sets to evaluate the performance as the dataset size increases, and the third query set to evaluate the performance as the query pattern size increases. The test queries are rewritten to standard SPARQL and SQL for execution in MySQL and RDF engines. For all the implementations, we report the average warm-cache query execution time.

38
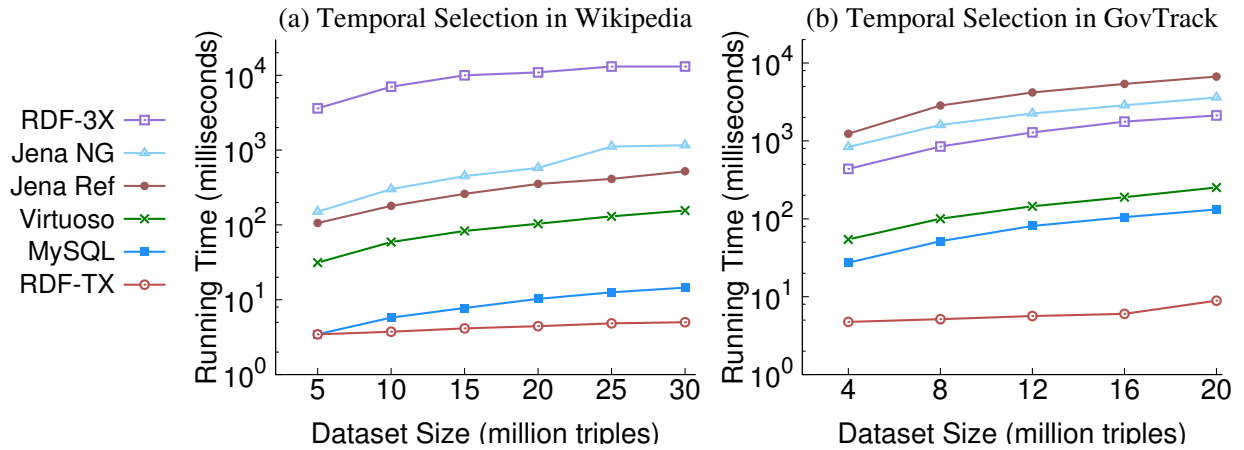
Figure 2.12: Time Performance for Temporal Selection in Wikipedia and GovTrack
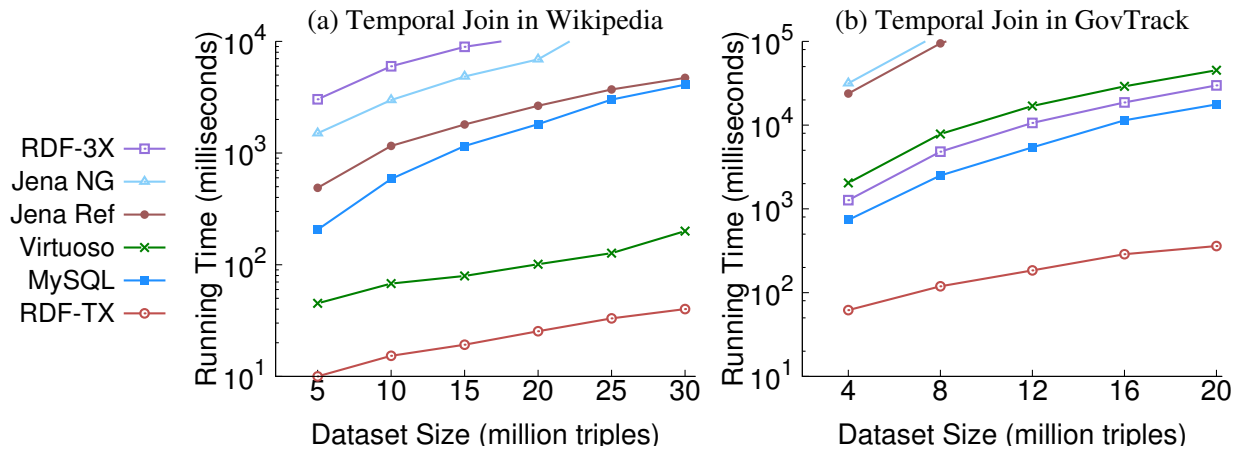
Figure 2.13: Time Performance for Temporal Join in Wikipedia and GovTrack

Temporal Selection and Join. We create 10 temporal selection and 10 temporal join queries for each dataset and conduct the experiments as the size of dataset N increases (N: 5-30 million in Wikipedia and 4-20 million in GovTrack).

Figure 2.12(a) shows the query execution time for temporal selection in Wikipedia. RDF-TX and MySQL show similar performance in small datasets. As the size of dataset increases, RDF-TX shows better performance than MySQL. In the largest dataset (30 million), RDF-TX is about 3X faster than MySQL and 10X faster than Virtuoso. Jena Named Graph and Reification are

2 orders of magnitude slower than SPARQL$^\text{T}$ engine due to the slow index scan.

RDF-3X is much slower than other systems due to its poor support of constraints. Most historical queries involve temporal constraints. For instance, consider Example 2 in Section 2.2 that searches the budget of University of California in 2013. This query has one temporal constraint that the valid period of temporal RDF triple should overlap (01/01/2013, 12/31/2013). This constraint can be expressed as: $?t_s \leq 12/31/2013$ && $?t_e \geq 01/01/2013$. In RDF-3X, the numbers are encoded as strings. So for temporal constraints, RDF-3X converts strings back to integers at running time to evaluate the constraints, which is inefficient.

The results of temporal join in Wikipedia are shown in Figure 2.13(a). RDF-TX is about 2 orders of magnitude faster than MySQL and Jena, and 6X faster than Virtuoso. RDF-3X is still slow since the condition of temporal join (e.g. OVERLAP and MEET) is expressed as constraints in FILTER clause.

Figure 2.12 (b) and Figure 2.13 (b) show the query execution time for temporal selection and join in GovTrack. These approaches use more time to execute the queries on GovTrack than Wikipedia because the number of distinct predicates in GovTrack is 60, which is very small comparing with Wikipedia ($\sim$ 110000 predicates). Thus the query patterns (e.g. P and PT) return much more results in GovTrack. The RDF-3x performs better than Jena on this dataset since it has a smaller number of distinct time periods ($\sim$ 10000) and predicates. MySQL and Virtuoso are about 1 order of magnitude slower than RDF-TX on selection and 2 orders of magnitude slower on Join.

RDF-TX performs 1-2 orders of magnitude faster than most competitors for selection and join. An important reason behind this is that MVBT can process two-dimensional (key and time) range query in one operation, while SPARQL and SQL engines need additional join and index scan.

Complex Queries. We generate 25 complex queries for each dataset with increasing query pattern size (3-7). The generation process is as follows: a set of 5 queries is created initially, and each query has 3 query patterns; then we incrementally add query patterns to existing queries until the size of query patterns reaches 7. The experiment is conducted on two datasets (each has 20 million
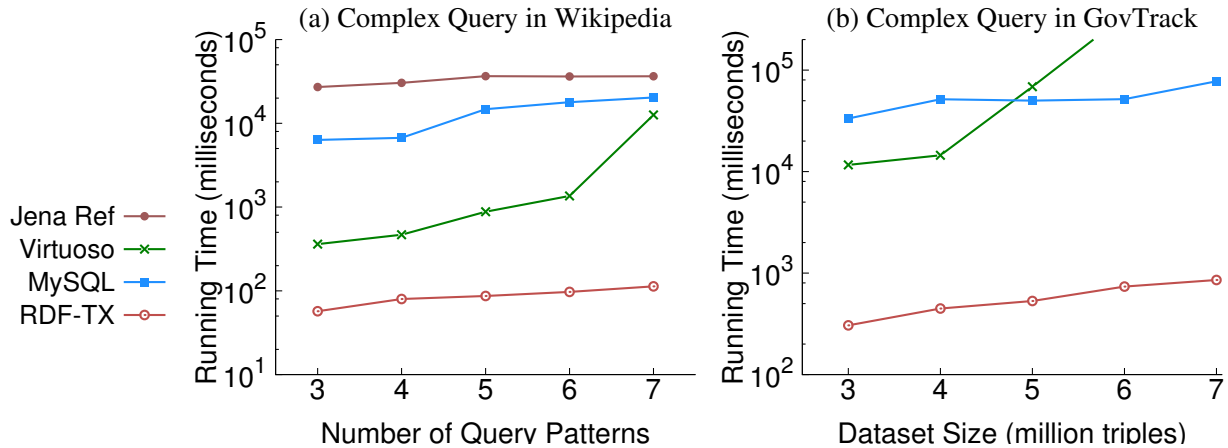
Figure 2.14: Time Performance for Temporal Join in Wikipedia and GovTrack

triples) and the optimizers are enabled in all compared approaches.

The evaluation results in Wikipedia are shown in Figure 2.14 (a)[6]. Jena Named Graph and RDF-3X are not reported since they are much slower than other approaches so we omit them. For RDF engine and RDBMS, a query with more patterns is translated to more joins, which increases the complexity of parsing and optimization. On average, RDF-TX is 2 orders of magnitude faster than MySQL and Jena, and 1 order of magnitude faster than Virtuoso.

The evaluation results for GovTrack are shown in Figure 2.14 (b). A notable change in this graph is that Jena is not reported in this experiment. Jena is too slow compared to other approaches on GovTrack since the query patterns usually cover a large portion of dataset, which leads to slow execution time if an inefficient join order is generated; meanwhile, the column-store traits of Virtuoso excel in this small predicate cardinality case. On average, our system is still about 2 orders of magnitude faster.

As the size of query pattern increases, the query execution time does not change much because in most queries, the optimizers can generate the query plans that start from the query patterns with small cardinalities.

---

[6]The query running time of Virtuoso on pattern size 6/7 is averaged over four queries since Virtuoso generates a very inefficient join order for one query which takes more than 1 hour to finish.
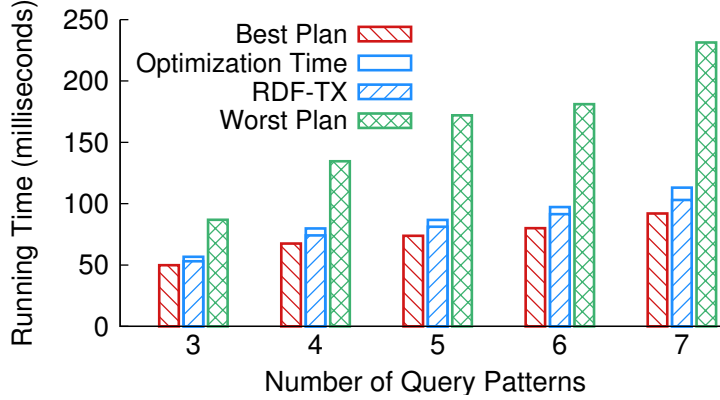
Figure 2.15: Query Execution Time of the best/worst plans, and the plan generated by SPARQL^T optimizer for complex queries in Wikipedia

2.6.4 Effectiveness of Query Optimizer

In this section, we explore the impact of query optimizer in the query evaluation. We enumerate all the possible query plans of the complex queries (Section 2.6.3) in Wikipedia and find the best and worst execution times. Figure 2.15 shows the query execution times of best/worst plans and the plan generated by RDF-TX query optimizer (blue bar) in a Wikipedia set with 20 million triples. The result shows that the plan generated by our query optimizer is very close to the best performing execution plan. On average, the execution time of optimized query plan is about half of the time used by worst plan. In the relatively simple queries (3 query patterns), the difference between the best plan and the worst plan is small. As the number of query patterns increases, the difference becomes much larger. Thus, the optimizer is important for scaling up towards complex queries with a lot of query patterns. We also measure the time used for query optimization, which varies from 3.5 to 10 milliseconds as the size of query increases.

Then we measure the storage overhead of temporal histogram. The CMVSBTs for temporal statistics are built using the dictionary IDs. As discussed in Section 2.5.2, we merge CMVSBT records and increase c_m and l_m until the size is small enough. In this experiment, the size of temporal histogram is 177.5 MB, which is about 8.5% of raw data size.

42
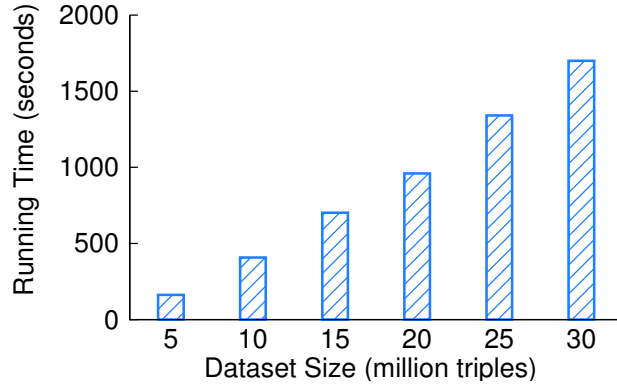
Figure 2.16: Index Construction Time
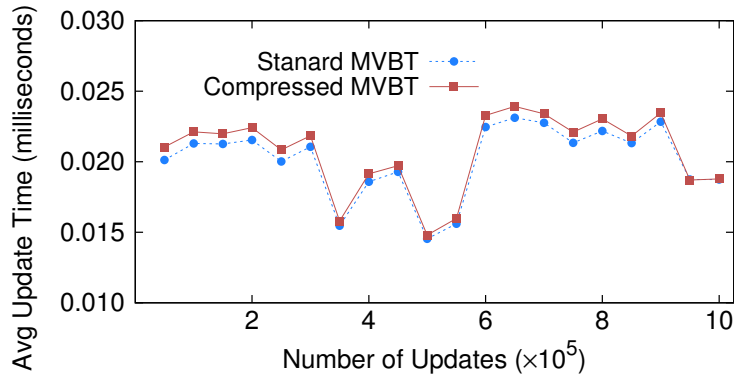
Figure 2.17: Index Maintenance Time

2.6.5 Index Construction & Maintenance

For large datasets, we first build standard MVBT and then compress the MVBT indices. In RDF-TX , the process of index construction is paralleled using at most 4 threads. We evaluate the index construction time for compressed MVBT time on different sizes of subsets of Wikipedia in Figure 2.16 (compression time included). The time for index construction is approximately linear with the size of datasets, and it increases slightly faster in the datasets with 25 million and 30 million triples due to degraded performance caused by JVM garbage collection.

RDF-TX also supports the index update on compressed MVBT, which is important for real-time applications. Thus we further measure the average index maintenance time on a compressed MVBT index built from a 25 million subset of Wikipedia. We perform 1 million updates (68%

43

insert, 32% delete) which simulates the changes in real Wikipedia edit history. Figure 2.17 shows the results by comparing maintenance time of compressed MVBT with the time used on standard MVBT. Our compression technique shows a decent performance. Comparing with the update on MVBT, the update on compressed MVBT only takes 5% more time. This little overhead is negligible w.r.t. 76% space saved using compression.

2.7   Historical Knowledge Browser

The goal of SPARQL^T is to express a wide variety of temporal queries with minimum extension of SPARQL. However, for users who are unfamiliar with the schema of knowledge bases and the syntax of SPARQL^T , it is still a difficult task to write a SPARQL^T query. Thus we develop a query interface called Historical Knowledge Browser that extends the By-Example Structured Query (BEStQ) approach introduced in the SWiPE system [AZ12, AZ14]. The interface uses the Wikipedia Infoboxes extended with temporal fields, where the user can enter temporal query conditions. From the modified Infoboxes and query conditions, our system derives equivalent queries that are optimized and executed in our query engine.

The interface supports (i) querying the current knowledge base and its history and (ii) browsing the history of entities and properties. Our user who wants to find the population of San Diego when Bob Filner served as the mayor, might start by loading subject San Diego in our interface, as shown in Figure 2.18. Since all the fields are editable, he enters "Bob Filner" in the *Government Mayor* and variable "?pop" in *City Population* and these two InfoBoxes are set with the same temporal variable "?t" to indicate the temporal join. Then our system generates and executes SPARQL^T query following SPARQL^T query.

SELECT ?pop [?t]

{ **San_Diego Mayor Bob_Filner ?t .**

San_Diego Population ?pop ?t . }

Our system also provides navigation toolbars so that users can browse the history of entities
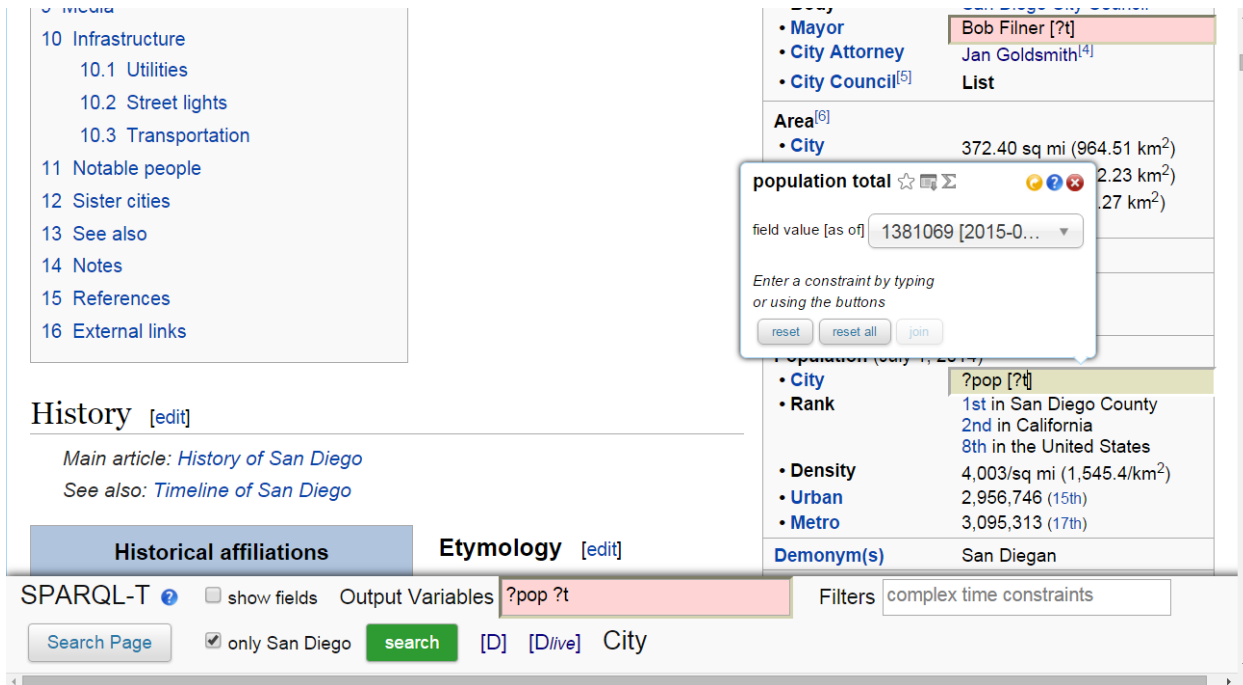
Figure 2.18: Historical Knowledge Browser Interface

Figure 2.19: Navigation Window for Property Mayor

and properties. For example, if the user clicks on a property, the interface will show a navigation window where the user can see the property type and its history and specify temporal conditions, as shown in Figure 2.19. The interface is implemented as middleware systems on Wikipedia and

demonstrated in International Semantic Web Conference 2015.

2.8   Related Work

Temporal Index.   There has been a large body of research on temporal index in the literature [BGO96, JSL00, LHN08, ND, ST99, TML99].   MAP21 [ND] is an index over B+Tree by mapping time ranges to one dimensional points, thus time intervals/points can be used as keys and queried in a B+Tree. OB+tree [TML99] organizes B+Trees in a versioned way with shared nodes whose contents do not change over versions. However, MAP21 and OB+tree only support single dimension query. BT-tree [JSL00] enables branched versions along with the temporal index, while the time in our system is linear, i.e. no branching. MVBT [BGO96] and TSB-Tree [LHN08] are bi-dimensional indices, which satisfy our requirements exactly. TSB-Tree is a temporal index very similar to MVBT and implemented in Immortal DB [LBM05] on Microsoft SQL Server, with better integration to SQL Server's existing index structures. The major difference between these two is that TSB-Tree migrates old data to a historical store during node splitting, while MVBT moves new data. Since MVBT is a general approach which is not targeted on specific platforms, we adopt and extend it in RDF-TX .

Temporal RDF Model. Temporal RDF is first studied in [BC02] in which Buraga et al. introduced the XML-based *Temporal Relation Specification Language* (TRSL) to express temporal relations in RDF. Later, Gutierrez et al. [GHV05, GHV07] provide complete semantics for temporal RDF model and a query language for reasoning temporal RDF graph. The discussion of query language is limited to query examples and evaluation complexity. No implementation or experimental evaluation is provided. The temporal RDF model is used in many works on temporal RDF reasoning [PSH07, PUS08, TB09] and temporal knowledge discovering [HSB11, HSB13]. The SPARQL^T language proposed in this paper is also based on this model.

Query Languages and Systems for Temporal RDF. The progress of knowledge discovery has enabled us to construct large temporal knowledge bases [AGD13, HSB13] which thus need powerful query language and efficient query processor for reasoning. Several query lan-

guages [Gra10, MB09, PSH07, PUS08, TB09] are proposed for temporal RDF triples. T-SPARQL [Gra10] is a temporal extension of SPARQL based on a multi-temporal RDF model. The RDF triple is annotated with a temporal element that represents a set of temporal intervals. Thus a temporal join is expressed using additional functions (e.g. OVERLAP). At the best of our knowledge, no actual implementation of T-SPARQL is available. The τ-SPARQL system reported in [TB09] uses the temporal RDF model [GHV07] and augments SPARQL query patterns with two variables ?*s* and ?*e* to bind the start time and end time of temporal RDF triples and express temporal queries. The evaluation is done by rewriting τ-SPARQL queries to standard SPARQL queries. Perry et al. [PSH07] propose a framework to support temporal and spatial semantic queries. Simple selection and join queries are expressed using two temporal operators. These operators are implemented in Oracle by extending Oracle Semantic Data Sore and SQL functions. These works rely on relational databases/RDF engine to store and query temporal RDF triples, which results in complex SPARQL and SQL queries in evaluation.

The tRDF system [PUS08] extends the temporal RDF model [GHV07] with indeterminate temporal annotations. The temporal elements could be timestamps or indeterminate temporal values. The temporal queries are evaluated using tGrin index that clusters the temporal RDF triples based on graphical-temporal distance. However, tRDF only supports a subset of temporal queries discussed in this paper. Most significantly, temporal joins are not supported since tGrin index relies on the temporal distance to filter the triples, while the temporal distance between two temporally joined query patterns can not be determined. STUN [KPG12] system supports queries on annotated RDF, but it is not scalable for large temporal datasets as discussed in Section 2.6.1.2.

CHAPTER 3

SWIM: A Framework for Knowledge Extraction and Integration

The importance of knowledge bases in semantic-web applications has motivated the endeavors of several important projects that have created the public-domain knowledge bases shown in Table 3.1. In this chapter, we present the Semantic Web Information Management system (SWIM), which consists of an integrated set of powerful tools [MGZ13b, MGZ13a, MAG14, MGK14] for knowledge extraction and integration. The goal of SWIM system is to (i) extract structured summaries from free text, (ii) integrate the knowledge in web documents and existing knowledge bases into a more complete and consistent repository, and (iii) preserve provenance information for verification. SWIM provides much better support for advanced web applications, and in particular for user-friendly search systems that support Faceted Search [HBS10] and By-Example Structured Queries [AZ12]. Our approach in achieving this ambitious goal involves the five main tasks of:

Name	Size (MB)	Number of Entities (10^6)	Number of Triples (10^6)
ConceptNet [LS04]	3075	0.30	1.6
DBpedia [BLK09]	43895	3.77	400
Geonames [GEO]	2270	8.3	90
MusicBrainz [MUS]	17665	18.3	131
NELL [CBK10]	1369	4.34	50
OpenCyc [OPE]	240	0.24	2.1
YaGo2 [HSB13]	19859	2.64	124

Table 3.1: Some of the publicly available Knowledge Bases

1. Integrating existing knowledge bases by converting them into a common internal representation and store them in IKBStore.

2. Completing the integrated knowledge base by extracting more facts from free text using text-mining system IBMiner.

3. Generating a large corpus of context-aware synonyms that can be used to resolve inconsistencies in IKBStore and to improve the robustness of query answering systems.

4. Archiving the provenance of knowledge bases for debugging and verification.

5. Developing user-friendly interfaces for browsing and editing IKBStore.

The first step was greatly simplified by the fact that many projects, including DBpedia [BLK09] and YaGo [HSB13], represent the information derived from the structured summaries by RDF triples of the form $<subject, attribute, value>$, which specifies the $value$ for an $attribute$ (property) of a $subject$. This common representation facilitates the use of these knowledge bases by a roster of semantic-web applications, including queries expressed in SPARQL, and user-friendly search interfaces [AT10, AZ12]. In this project, we convert all the knowledge bases into RDF format and employ the interlinking information and semantic similarity to align the entities, attributes, and categories in knowledge bases.

The integrated knowledge base so obtained will represent a big step forward, since it will (i) improve coverage and quality of the knowledge available to semantic web applications and (ii) provide a common ground for different contributors to improve the knowledge bases in a more standard and effective way. However, since the process of generating structured summaries is usually manual and a standard ontology is often not used, the resulting knowledge bases are prone to the issue of incompleteness. For example in Wikipedia, around 40% of pages are missing their entire InfoBox. Many other pages are also missing part of their InfoBoxes. This mainly indicates Wikipedia's coverage is quite incomplete. Other similar knowledge bases suffer from the same problem as well.

To address this issue, we proposed InfoBox Miner (IBMiner). IBMiner employs an NLP-based text mining framework, called SemScape, to extract initial triples from the text. Then using

a large body of categorical information and learning from matches between the initial triples and existing InfoBox items in the current knowledge base, IBMiner translates the initial triples into more standard InfoBox triples. IBMiner significantly improves the coverage of the integrated knowledge base.

Another obstacle in achieving the goal of SWIM is that different systems do not adhere to a standard terminology to represent their knowledge, and instead use plethora of synonyms and polysemy. Thus, we need to resolve *synonyms* and *polysemy* for the entity names as well as the attribute names. For example, by knowing "*Johann Sebastian Bach*" and "*J.S. Bach*" are synonyms, the knowledge base can merge their triples and associate them with one single name.

For synonym and polysemy issues, we proposed Context-aware Synonym Suggestion System (CS^3). CS^3 first extracts context-aware attribute and entity synonyms, and then uses them to improve the consistency of IKBStore. CS^3 learns attribute synonyms by matching morphological information in free text to the existing structured information. Similar to IBMiner , CS^3 takes advantage of a large body of categorical information available in Wikipedia, which serves as the contextual information. Then, CS^3 improves the attribute synonyms so discovered, by using triples with matching subjects and values but different attribute names. After unifying the attribute names in different knowledge bases, CS^3 finds subjects with similar attributes and values as well as similar categorical information to suggest more entity synonyms.

The process of knowledge integration is complex and time-consuming which makes the provenance management of integrated data set to be a non-trivial task. Since knowledge integration involves multiple steps, it is necessary to record the provenance of knowledge for debugging and verification. Existing semantic web systems lack efficient provenance management to ensure the quality and reproducibility of integrated knowledge base. In our system, we archive the provenance of every fact, including source repositories, contributors, and integration operations.

At last, usability remains as a major problem since the knowledge base is usable only by people who can write SPARQL queries thus casual users are excluded. To this end, we propose two user-friendly interfaces InfoBox Knowledge-Base Browser (IBKB) and InfoBox Editor (IBE). These interfaces support knowledge browsing and editing, and query functions needed for managing and

upgrading knowledge bases.

In the rest of this chapter, we first introduce the various intertwined aspects of SWIM system, while the remaining sections provide an in-depth coverage of the techniques used for knowledge extraction and integration.

3.1 Overview

As already mentioned, our goal is to address the issues of incompleteness and inconsistency and integrate existing knowledge bases into a more consistent and complete one, in which the knowledge provenance is well preserved. SWIM performs five tasks to achieve this goal. Here we elaborate these five tasks in more details:

Task A: Collecting publicly available knowledge bases, unifying knowledge representation format, and integrating knowledge bases using existing interlinks and structured information. Creating the initial knowledge base is actually a straightforward task, since many of the existing knowledge bases are representing their knowledge in RDF format. Moreover, they usually provide information to interlink a considerable portion of their subjects to those in DBpedia. Thus, we use such information to create the *initial integrated knowledge base*. We refer to *initial integrated knowledge base* as *initial KB*. However, the coverage and consistency provided by each individual system remain limited. To overcome these problems, we propose new techniques for merging, completing, and integrating these knowledge bases at the semantic level, as discussed in Section 3.2.

Task B: Completing knowledge base using accompanying text. In order to do so, we employ the IBMiner system [MAG14, MGK14] to generate structured data from the free text available at Wikipedia or similar resources. IBMiner first generates semantic links between entity names in the text using recently proposed text mining SemScape [MGK14]. Then, IBMiner learns common patterns called *Potential Matches* (*PMs*) by matching the current triples in knowledge base to the semantic links derived from free text. It then employs the *PMs* to extract more InfoBox triples from text. These newly found triples are then merged with knowledge base to improve its coverage. More details of IBMiner are discussed in Section 3.3.

Task C: Generating a large corpus of context-aware synonyms. Since IBMiner learns by matching structured data to the morphological structure in the text, it may find more than one acceptable matching attribute name for a given link name from the text. This in fact implies possible attribute synonyms, and it is the main intuition that CS^3 uses to learn attribute synonyms. Based on *PM*, CS^3 creates the *Potential Attribute Synonyms* (*PAS*) which is in nature similar to *PM*. However instead of mapping link names into attribute names, *PAS* provides mapping between different attribute names based on the categorical information of the subject and the value. Similar to the case of IBMiner , the categorical information serves as the contextual information, and improves the final results of the generated attribute synonyms. As described in Section 3.4, CS^3 improves *PAS* by learning from the triples with matching subjects and values but different attribute names in the current knowledge base. CS^3 also recognizes context-aware entity synonyms by considering the categorical information and InfoBoxes of the entities. These synonyms help realign attributes and entity names to construct the final IKBStore.

Task D: Archiving the provenance of knowledge. In many knowledge bases, every piece of knowledge is assigned with a value indicating the confidence on the correctness of this information [CBK10, MGZ13b]. The use of such confidence value and other similar values (e.g. evidence value [MGZ13b]) can help determine the quality of knowledge. In SWIM, we use provenance semiring to annotate the fact with knowledge lineage and confidence value to preserve these important properties of knowledge. We will talk more about knowledge provenance in Section 3.5.

Task E: In order to browse into our final knowledge base, we propose the InfoBox Knowledge-Base Browser (IBKB) which provides structured summaries and their originating sources. We also demonstrate our editing tool called InfoBox Editor (IBE), which is able to provide relevant attributes for a user-specified subject, so the user can easily improve the knowledge base without needing to know the underlying terminology of the system.

Applications: SWIM can benefit a wide variety of applications, since it covers a large number of structured summaries represented with a standard terminology. Knowledge extraction and population systems such as *IBminer* and *OntoMiner* [MKI13b], knowledge browsing tools such as DBpedia Live [dbp] and InfoBox Knowledge-Base Browser (IBKB)[MGZ13b], and semantic web

search such as *Faceted Search* [AT10] and By-Example Structured queries [AZ12] are three prominent examples of such applications. In particular for semantic web search, SWIM improves the coverage and accuracy of structured queries due to superior quality and coverage with respect to existing knowledge bases. Moreover, SWIM can serve as a common ground for different contributors to improve the knowledge bases in a more standard and effective way. Using the knowledge provenance, SWIM is also a good mean for verifying the correctness of the current structured summaries as well as those generated from the text.

3.2 IKBStore: Integrated Knowledge Base

SWIM allows us to integrate the existing knowledge bases into one of superior quality and coverage. In this section, we elaborate the steps in the process of knowledge base integration.

3.2.1 Data Gathering

We process the knowledge bases listed in Table 3.1, which include some domain specific knowledge bases (e.g. MusicBrainz [MUS], Geonames [GEO], etc.), and some domain independent ones (e.g. DBpedia [BLK09], YaGo2 [HSB13], etc.). Although most knowledge bases such as DBpedia and YaGo already provide there knowledge in RDF, some of them may use other representations. Thus, for all knowledge bases, we convert their knowledge into <*Subject*, *Attribute*, *Value*> triples and store them in IKBStore. IKBStore is currently implemented over Apache Cassandra which is designed for handling very large amount of data. IKBStore recognizes three main types of information:

- *InfoBox triples:* These triples provide information on a known subject ($subject$) in the <$subject$, $attribute$, $value$> format. E.g. <*J.S. Bach*, *PlaceofBirth*, *Eisenach*> which indicates the birthplace of the subject *J.S.Bach* is *Eisenach*.

- *Subject/Category triples:* They provide the categories that a subject belongs to in the form of *subject/link/category* where, *link* represents a taxonomical relation. E.g. <*J.S.Bach*, *is*

in, *Cat:German Composers*> which indicates the subject *J.S.Bach* belongs to the category *Cat:German Composers*.

- **Category/Category triples:** They represent taxonomical links between categories. E.g. <*Cat:German Composers*, *is in*, *Cat:German Musicians*> which indicates the category *Cat:German Composers* is a sub-category of *Cat:German Musicians*.

3.2.2   Initial Knowledge Integration

The aim of this phase is to find the initial interlinks between subjects, attributes, and categories from the various knowledge sources to eliminate duplication, align attributes, and reduce inconsistency using only the existing interlinks. At the end of this phase, we have an initial knowledge base which is not quite ready for structured queries, but provides a better starting point for *IBminer* to generate more structured data and for CS^3 to resolve attribute and entity synonyms.

- **Interlinking Subjects:** Fortunately, many subjects in different knowledge bases have the same name. Moreover, DBPedia is interlinked with many existing knowledge bases, such as YaGo2 and FreeBase, which can serve as a source of subject interlinks. For the knowledge bases which do not provide such interlinks (e.g. NELL), in addition to exact matching, we parse the structured part of knowledge base to derive candidate interlinks for existing entities, such as *redirect* and *sameAs* links in Wikipedia.

- **Interlinking Attributes:** As we mentioned previously, attributes interlinks are completely ignored in the current studies. In this phase, we only use exact matching for interlinking attributes.

- **Interlinking Categories:** In addition to exact matching, we compute the similarity of the categories in different knowledge bases based on their common instances. Consider two categories c_1 and c_2, and let $S(c)$ be the set of subjects in category c. The similarity function for categories interlink is defined as $Sim(c_1, c_2) = |S(c_1) \cap S(c_2)|/|S(c_1) \cup S(c_2)|$. If the $Sim(c_1, c_2)$ is greater than a certain threshold, we consider c_1 and c_2 as aliases of each other,

which simply means that if the instances of two categories are highly overlapping, they might be representing the same category.

After retrieving these interlinks, we merge similar entities, categories, and triples based on the retrieved interlinks. The provenance information is generated and stored along with the triples.

3.2.3   Further Knowledge Integration

Once the initial knowledge base is ready, we first employ IBMiner to extract more structured data from accompanying text and then utilize CS^3 to resolve synonymous information and create the final knowledge base. More specifically, we perform the following steps in order to complete and integrate the final knowledge base:

- **Improving Knowledge base coverage:** The web documents contain numerable facts which are ignored by existing knowledge bases. Thus, we first enrich the knowledge base by employing our knowledge extraction system IBMiner . IBMiner learns *PM* from free text and the initial knowledge base to derive more triples which will greatly improve the coverage of existing knowledge bases. These new triples are then added to IKBStore. For each generated triple, we also update the confidence and evidence frequency in IKBStore. That is if the triple is already in IKBStore, we only increase and update its confidence and evidence frequency.

- **Realigning attribute names:** Next we employ CS^3 to learn *PAS* and generate synonyms for attribute names and expand the initial knowledge base with more common and standard attribute names.

- **Matching entity synonyms:** This step merges the entities base on the entity synonyms suggested by CS^3. For the suggested synonym entities such as s_1, s_2, we aggregate their triples and use one common entity name, say s_1. The other subject (s_2) is considered as a possible alias for s_1, which can be represented by RDF triple $<s_1$, *alias*, $s_2>$.

3.3 IBMiner: Deriving Structured Summaries from Text

Current knowledge bases suffer from the issue of incompleteness, largely due to the fact that they are created manually. An important information source in the web is the free text in web pages. Previous research addressing such problems has relied on approaches that exploit the structured information in web documents, but do not fully exploit the morphological information in the text. In this section, we present a new system, called IBMiner that uses deep NLP to extract structured summaries. IBMiner uses morphological information in the text to infer graph-based structures called TextGraphs, which summarize the information extracted from text as links representing grammatical relations between single- or multi-word terms. Then, IBMiner generates semantic links between words and terms in TextGraphs by using a set of predefined SPARQL-like patterns. Finally, by learning from the current InfoBoxes in existing knowledge bases, and relying on a large body of categorical information, IBMiner converts the semantic links into the final RDF triples, and also infers new attribute synonyms. This deep text-mining process produces more complete knowledge base, significantly improving the recall for structured queries. In the rest of this section, we use InfoBox to denote the final RDF triples used in existing knowledge bases, such as Wikipedia.

Although IBMiner 's process is quite complex, we can divide it into three high-level steps which are elaborated in this section. The first step is to parse the sentences in text and convert them to a more machine friendly structure called TextGraphs which contain grammatical and semantic links between entities mentioned in the text. As discussed in subsection 3.3.1, this step is performed by the NLP-based text mining framework *SemScape* [MKI11a]. The second step is to extract semantic links from TextGraph. As explained in Subsection 3.3.2, using SPARQL-like graph patterns on the TextGraphs, IBminer generates a set of initial triples called *semantic links* in the form of $<subject, link, value>$ where $subject$ is an entity, $value$ is either an entity or a value, and $link$ is a semantic relation between $subject$ and $value$. Each semantic link represents a small piece of information that is potentially useful to generate an InfoBox triple (fact in RDF format $<subject, predicate, object>$). As the third step, IBminer generates an intermediate

structure called *Potential Matches* (*PM*). *PM* is a very large set of automatically generated patterns which will be used to translate link names in the *semantic links* into appropriate attribute names and generate new InfoBox triples. This step is described in Subsection 3.3.3.

3.3.1 From Text to TextGraphs

To have a deeper understanding of the knowledge in the text, IBMiner converts the sentences in the text into a weighted graph structure called *TextGraphs*. This task is performed by employing the NLP-based text mining framework *SemScape* [MKI11b, MKI14]. *TextGraphs* are machine-friendly weighted graph structures, that represent grammatical connections between words and terms in the sentence, where terms are single- or multi-word phrases identifying an entity or a concept. Each link in the TextGraph is assigned a *confidence value* (weight) indicating our confidence on the correctness of the link and an *evidence count* indicating the frequency. TextGraphs generated in this way provide a semantic representation of the connections between words, terms, and phrases through labeled and weighted links. For instance, Figure 3.1 shows the TextGraph for following sentence:

Example Sentence: *Johann Sebastian Bach (31 March 1685 - 28 July 1750) was a German composer, organist, harpsichordist, violist, and violinist of the Baroque Period.*

For the sake of brevity, we did not include all links and weights in this graph. This graph connects words and terms to each other trough grammatical links such as '*subj_of*', '*obj_of*', '*prop_of*', '*det_of*', etc. The graph also identifies multi-word terms (shown in dashed boxes) and their links to other components of the sentence. For instance, consider the following possible subjects for the main verb: '*Johann Sebastian Bach*', '*Sebastian Bach*', and '*Bach*'. These terms are referred to as *candidate terms*.

3.3.2 Generating Semantic Links

The next step to generate InfoBoxes or RDF-like triples is to find the semantic connections between terms used in the TextGraphs. These connections are referred to as *semantic links* throughout the
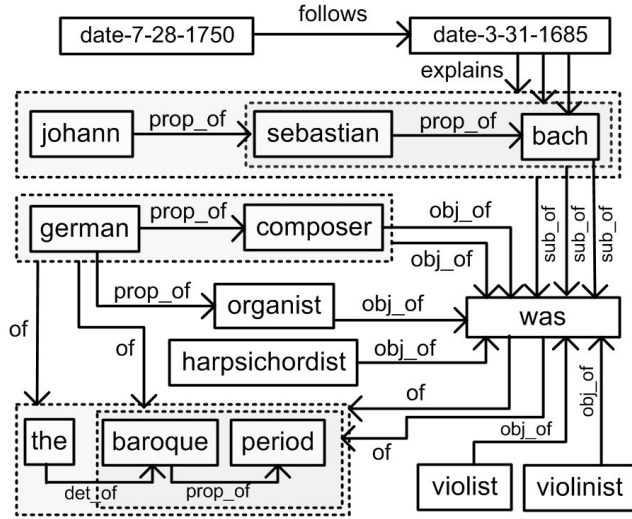
Figure 3.1: Part of the TextGraph for our example sentence.

paper, and will be used later to generate the final InfoBox triples. Before continuing, let us first define semantic links:

Semantic Link. Semantic link is a triple, denoted as $<subject, link, value>$ where $subject$ and $value$ are two candidate terms in a TextGraph connected through the phrase $link$. For instance, *<Bach, was, composer>*, *<Bach, was, German>*, and *<Bach, was, organist>* are three possible semantic links in our running example.

It is important to understand that with the above definition some of the links in the TextGraph are also semantic links (e.g. *<Date-3-31-1685, was, Bach>*). However, there are many more semantic links that can be extracted from the TextGraphs. This section explains how IBMiner extracts such links with a pattern-based approach.

3.3.2.1 Extracting Semantic Links

To generate semantic links such as the ones introduced above, IBMiner uses a limited set of carefully created graph-based patterns. These patterns, which are also called Graph Domain rules, capture common structures in the TextGraphs that indicate possible semantic links between candidate terms. Then from the matching sub-graphs for these patterns, IBMiner constructs the semantic

58

links. Semantic links that are created more than once from different patterns or different sentences are also merged as explained later in this section.

Graph Domain Rules. Given a TextGraph T, Graph Domain (GD) rule is a graph query executed on T which returns semantic links matching certain conditions. The format of GD rules is similar to SPARQL, with some extended key words for TextGraphs. To understand our GD rules and the process of using them to generate semantic links, we provide several examples in this section. We should stress that the GD rules in IBMiner are completely generic and domain independent. In the current implementation of IBMiner , we have created 59 such GD rules which are available for online access [Sem].

As shown in Figure 3.1, terms are represented by nodes in the graph and connected in various ways. For instance, the term '*Johann Sebastian Bach*' is connected to the term '*composer*' though a connecting node (verb) '*was*'. This is actually a very common pattern that can generate several small facts such as $<$*Johann Sebastian Bach*, *was*, *composer*$>$. Generating such a fact in bag of keyword approaches or even in shallow NLP-based techniques is indeed very challenging. IB-Miner uses the following GD rule to extract similar facts to the mentioned one:

—————————————————— Rule 1. ——————————————————

SELECT (?1 ?3 ?2)
WHERE {
    ?1 "subj_of" ?3.
    ?2 "obj_of" ?3.
    NOT("not" "prop_of" ?3).
    NOT("no" "det_of" ?1).
    NOT("no" "det_of" ?2).   }

——

As depicted in the part a) of Figure 3.2, the pattern graph (WHERE clause) of Rule 1, specifies two nodes with names ?1 and ?2 which are connected to a third node (?3) respectively with 'subject_of' (sub_of) and 'object_of' (obj_of) links. One possible match for this pattern in the TextGraph

59
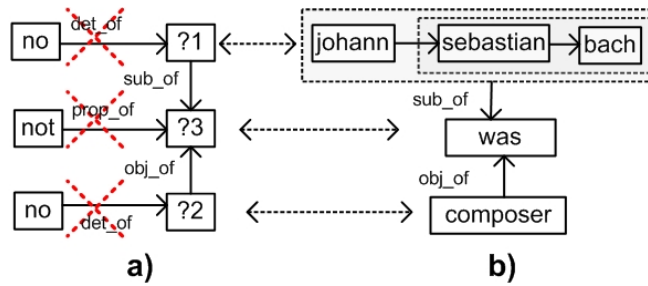
Figure 3.2: Part a) shows the graph pattern for Rule1, and b) depicts one of the possible matches for this pattern.

of our running example is shown in part b) of Figure 3.2. Due to the structure of the TextGraphs, matching multi-word terms (hyper nodes in the TextGraphs) to the nodes in the patterns is an easy task for IBMiner whereas this is actually a challenging issue in works such as [WW10] which are based on dependency parse trees. Using the SELECT clause in Rule 1, the rule returns several triples for our running example such as:

- *<Johann Sebastian Bach, was, composer>*,

- *<Sebastian Bach, was, composer>*,

- *<Bach, was, composer>*,

- *<Johann Sebastian Bach, was, organist>*,

- *<Sebastian Bach, was, organist>*, etc.

Subsection 3.3.2.2 discusses how IBMiner assigns confidence value to each of the above triples. As can be seen, we have made the syntax of the GD rules similar to that of SPARQL [SPA]. The SELECT clauses in GD pattern may indicate more than one triple. However, we added the notation of 'NOT' to indicate the absence of some links in the pattern, which requires a more complex expression in SPARQL.

As a more complex example, consider Rule 2 which captures *Bach*'s nationality in our running example. This rule looks for a '*to be*' verb (?3) with an object (?2), where the object has an adjective (?4). If such a pattern is found, the subject (?1) of the verb is connected to that adjective (?4) via the verb (?3). The following results are generated by this rule for our running example:

- *<Johann Sebastian Bach, was, German>*,

- *<Sebastian Bach, was, German>*,

- *<Bach, was, German>*, etc.

————————————————— Rule 2. —————————————————

SELECT (?1 ?3 ?4)

WHERE {

    ?1 "subj_of" ?3.

    ?2 "obj_of" ?3.

    ?4 "prop_of" ?2.

    ?4 "POS_Tag" ?5.

    NOT("not" "prop_of" ?3).

    NOT("no" "det_of" ?1).

    FILTER (regex(?3, "^am^|^is^|^are^|^was^|^were^|^be|^...", "i"))

    FILTER (regex(?4, "^JJ^|^ADJP^", "i"))   }

——-

Note that by using POS-Tag information which carried up from parse trees to the TextGraph and the FILTER keyword which accepts *Regular Expressions*, it is easy to verify that the node matching to the variable ?4 is an adjective. Without this filter, Rule 2 will generate wrong triples such as *<Bach, was, music>* for sentence "*Bach was a music composer.*".

Many of the semantic links in text can be found in prepositional phrases. In order to capture this type of information, we have considered rules such as the following one[1].

————————————————— Rule 3. —————————————————

---

[1]Similar rules are generated for cases such as passive sentences, sentences including verbs with modifier, prepositions that are connected to nouns, etc.

SELECT (?1 ?3+?4 ?2)

WHERE {

    ?1 "subj_of" ?3.

    ?2 ?4 ?3.

    NOT("no" "det_of" ?1).

    NOT("not" "prop_of" ?3).

    FILTER (regex(?4, "^to^|^on^|^in^|^with^|^from^|^at^|...", "i")) }

This rule captures semantic links in prepositional phrases that are connected to a verb. The link in this case is specified by ?3+?4 which indicates the concatenation of the verb and the preposition.

For the rest of the paper, we refer to this set of semantic links generated by IBminer as T_n. To unify similar link names, IBMiner uses their stems as new link names for triples in T_n. To this end, IBMiner replicates each triple by replacing the link name with its stem. The stem and synonym information is provided by WordNet [Wor12]. As an example for the semantic link *<Johann Sebastian Bach, was, composer>*, IBMiner also generates the triple *<Johann Sebastian Bach, be, composer>*. This simple expansion technique improves the process of interpreting attributes names (Section 3.3.3) by populating verb tenses and word variations.

3.3.2.2   Computing Links Confidence

As discussed in Section 3.3.1, every edge in TextGraph is assigned with *confidence* values. To compute the confidence value of the generated semantic links, we simply use the minimum confidence among the matching edges of TextGraph as the final confidence of each triple. The intuition behind using the minimum is that if one of the links in the matching subgraph is of low confidence, it makes the entire match low confident.

Since the same semantic link may be generated more than once from different rules or TextGraphs, we need to combine its evidence count e and confidence value c. Similar to [LWW11], the only

assumption for the combination process is that evidences of the same piece of information are independent from each other. Thus, if a piece of information has been generated twice, once with evidence count and confidence of e_1 and c_1, and once with e_2 and c_2, we combine the evidence count to $e_1 + e_2$ and the confidence to $1-(1-c_1)(1-c_2)= c_1+(1-c_1)c_2$. This new confidence is higher than both c_1 and c_2 which indicates the link's correctness probability is now higher.

3.3.3 Mapping Links to Attributes

In this step, IBMiner uses contextual information about subjects and values in the semantic links to map the link names in T_n to attribute names used in the original InfoBoxes. In this process, many of the inaccurate or irrelevant triples in T_n will be dropped since no good maps for their links can be found. For the rest of this section, we refer to the set of triples taken from the original InfoBoxes in WikiPedia as T_i.

The key idea in mapping the link name of a semantic link (T_n) to the attribute name in current InfoBoxes (in T_i) is to learn the attribute mapping by example. To understand the concept of attribute mapping and the intuition behind IBMiner approach to create this mapping, we continue with some examples. Consider the following two semantic links which have been generated from the TextGraphs in the previous section:

- *<Johann Sebastian Bach*, *was*, *composer>*

- *<Johann Sebastian Bach*, *was*, *German>*

Although the attribute names in both triples denote the same term, '*was*', they connect the subjects and values with completely different relations. As the attribute names in the actual InfoBox triples (T_i) suggest, the first '*was*' is mostly interpreted as '*occupation*' while the second one is usually called '*nationality*'. This kind of interpretation for the link names in semantic links based on the best match in existing InfoBoxes is referred to as *attribute mapping* throughout this paper. The difference in subject part of the triples also implies different mappings. For instance, consider the following triples:

- *<Benz, was, German>*

- *<Johann Sebastian Bach, was, German>*

Although the title '*nationality*' for both attributes is not completely wrong, a better interpretation for the first attribute could be '*made-in*' or '*founded-in*' since the subject is not a person.

In other words, the meaning or interpretation of a link in semantic link $<s, l, v>$ depends not only on the link name (l) but also on the taxonomical/categorical information of the subject (s) and the value (v). Under this intuition, to correctly map link name l to attribute names used in T_i, IBMiner considers the categorical information of the subject and value provided by Wikipedia under *categories* section. For instance, in the triple $<$*Johann Sebastian Bach*, *was*, *composer*$>$, knowing that 1) '*Johann Sebastian Bach*' is in category '*people*', 2) '*composer*' is in category '*occupations in music*', and 3) according to the matching examples, link '*was*' between these two categories is mostly called '*occupation*' in T_i, we can infer the new InfoBox triple $<$*Johann Sebastian Bach*, *occupation*, *composer*$>$. Next, we formally explain this technique for attribute mapping.

3.3.3.1 Generating Potential Matches

To map links in T_n to attributes in T_i, IBMiner constructs a structure called *Potential Matches* (*PM*) by learning from the existing examples. To understand this structure, let us start with the definition of triple match.

Matching triples: Given two triples $t_n = < s_1, l, v_1 >$ and $t_i = < s_2, \alpha, v_2 >$ where $t_n \in T_n$ and $t_i \in T_i$, we say t_n and t_i match each other iff $s_1 = s_2$ and $v_1 = v_2$.

For instance, $<$*Johann Sebastian Bach*, *was*, *German*$>$ from semantic links (T_n) matches $<$*Johann Sebastian Bach*, *nationality*, *German*$>$ from InfoBoxes triples (T_i), since both subjects and both values in the two triples are the same. The set of all triples in T_n that match with at least one triple in T_i is referred to as T_m in this paper. Using such matching triples between T_i and T_n, IBMiner automatically generates a set of patterns in a structure called *Potential Matches* or *PM* which is defined

as follows:

Potential Matches: Potential Matches are records in the form of $<c_s, l, c_v>: \alpha$, each associated with a confidence value c and an evidence frequency e. Each record in *PM* indicates that if a subject from category c_s is connected to a value from category c_v with link l in T_n, α may be a map for l with confidence c and evidence e.

As an example, consider *PM* record *<people, was, occupations in music>* : *occupation*. This record specifies a simple pattern indicating the link name '*was*' between a subject from '*people*' category to a value from '*occupations in music*' category may be interpreted as the attribute '*occupation*'. Next we explain how IBMiner used triples in T_m to generate these patterns and build the *PM* structure.

Assume $t_m = <s, l, v>$ ($t_m \in T_m$) with confidence c_m matches $t_i = <s, \alpha, v>$ ($t_i \in T_i$). In the following, $C_s = \{c_{s1}, c_{s2}, ...\}$ and $C_v = \{c_{v1}, c_{v2}, ...\}$ denote the categorical information respectively for s and v. This categorical information is retrieved from Wikipedia. If no category is found for a subject (or a value), we will consider them to be in the most general categories (e.g., '*Category:Things*'). Moreover, in addition to the direct categories, C_s and C_v may contain indirect (more general) categories that will be described in next subsection. We say attribute α is a *potential match* for link l from any category in C_s to any category in C_v with confidence c_m and evidence count 1. To construct the final list of potential matches (*PM*), for each $c_s \in C_s$ and $c_v \in C_v$, we add the following record to the *PM* structure:

$$<c_s, l, c_v>: \alpha$$

For each newly generated *PM* record, confidence value c is initiated by c_m and evidence frequency e is initiated by 1. It is worthy to mention that the number of potential matches for t_m is $|C_s| \times |C_v|$ where $|C_x|$ is the number of categories for the entity x. If we encounter more evidence for the same record in *PM*, IBMiner increases its confidence and evidence by combining the records as explained in Subsection 3.3.2.2. At the end of this step, *PM* will contain a big list of potential matches with their confidence values and evidence frequencies. In Section 3.3.3.3, we explain how *PM* is used to generate the final InfoBox triples by attribute mapping.

65

3.3.3.2 Selecting Best Categories

A very important issue in generating potential matches is the quality and quantity of the categories for the subjects and values. The direct categories provided for most of the subjects are too specific and only a few subjects are listed in each of these categories. Generating the potential matches over direct categories does not generalize the matches for newly observed subjects. On the other hand, exhaustively adding all the indirect (or ancestor) categories will generate too many inaccurate potential matches and significantly increase the processing time. For instance considering only four levels of categories in Wikipedia's taxonomy, the subject '*Johann Sebastian Bach*' belongs to 422 categories. In this list, there are some useful indirect categories such as '*German Musicians*' and '*German Entertainers*', as well as many categories which are either too general or inaccurate (e.g. '*People by Historical Ethnicity*' and '*Centuries in Germany*'). Considering the same issue for the value part, hundreds of thousands of potential matches may be generated for a single triple in T_m. This issue not only wastes our resources, but also impacts the accuracy of the final results.

To address this issue, we use a flow-driven technique to rank all the categories to which entity s belongs, and then select the best N_C categories. The main intuition is to propagate flows or weights through different paths from s to each of its categories. The categories receiving more weights are considered to be more related to s. Now, L being the number of allowed ancestor levels, we create the categorical structure for s up to L levels. Starting with node s as the root of this structure and assigning weight 1.0 to it, we iteratively select the closest node to s, which has not been processed yet, propagate its weight to its parent categories, and mark it as processed. To propagate weights of node c_i with k ancestors, we increment the current weights of each k ancestors with w_i/k, where w_i is the current weight of node c_i. Although w_i may change even after c_i is processed, we will not re-process c_i after any further updates on its weight to facilitate the algorithm. After propagating the weight to all the nodes, we select the N_C categories with the highest weight for generating potential matches and final InfoBox triples.

3.3.3.3 Generating InfoBox Triples

Once *PM* is generated, IBMiner uses it to map the link names of semantic links (T_n) into the attribute names of InfoBoxes (T_i). Let $t_n = <s, l, v>$ ($t_n \in T_n$) be the triple whose link (l) needs to be mapped, and s and v are listed in category sets $C_s = \{c_{s1}, c_{s2}, ...\}$ and $C_v = \{c_{v1}, c_{v2}, ...\}$ respectively. The key idea to map l is to take a consensus among all pairs of categories in C_s and C_v and decide which attribute name is the best possible match.

To this end, for each $c_s \in C_s$ and $c_v \in C_v$, IBMiner finds all potential matches such as $<c_s, l, c_v>$: α_i. The resulting set of potential matches are then grouped by the InfoBox attribute names, α_i's, and for each group we compute the average confidence and the aggregate evidence frequency of the matches. IBMiner uses two thresholds at this point to remove low-confidence (named τ_c) or infrequent (named τ_e) potential matches. Next, IBMiner filters the remaining results by a very effective type-checking technique explained in the next subsection. At this point, there should be only a few matches left in the set, which are called *ranked matches*. If there exists any match in this list, we consider the one with the largest evidence count, say pm, as the only attribute map and report the new InfoBox triple $<t_n.s, pm.a_i, t_n.v>$ with confidence $t_n.c \times pm.c$ and evidence $t_n.e$.

3.3.3.4 Type-checking

Attributes may take values from specific domains. For instance, attribute '*origin*' may accept only values from '*geopolitical locations*' domain. This sort of information on domain of values for attributes can be used as a simple, yet effective type-checking technique which in turn improves accuracy of the final results. For example, the value '*German*' fails the type-checking for attribute '*origin*' since in the original InfoBoxes they are not used together at all. On the other hand, '*German*' passes the check for attribute '*nationality*' since in several instances these two are used in the same triple in the original InfoBoxes. To automatically identify the correct domain for values of a given attribute, currently IBMiner learns from T_i triples. That is, for each attribute α, IBMiner counts the number of times that any value, say v, is used in a triple in T_i with attribute α. This number is called the *value rank* of v in α's domain. Value ranks are then used to verify the

correctness of the generated results.

One of the drawbacks of type-checking is that if we eliminate all the triples with wrong value types, we will never find new values for some of the attributes. For instance, attribute *short description* accepts a variety of values which might not be listed in its domain. To avoid this issue, IBMiner performs type checking on ranked matches for t_n, only if $t_n.v$ belongs to the domain of one (or more) attribute in the ranked matches. For instance assume for link '*was*' in semantic link *<Johann Sebastian Bach, was, organist>*, two attributes '*occupation*' and '*instrument*' are suggested using potential matches. Since value '*organist*' is frequently used with attribute '*occupation*' but not with attribute '*instrument*', our type checking will cancel the latter and only accepts '*occupation*' as the final attribute map. On the other hand, assume for link '*was*' in semantic link *<Johann Sebastian Bach, was, organist of Baroque Period>*, two attributes '*short description*' and '*occupation*' are suggested. This time value '*organist of Baroque Period*' has not co-occurred with any of the two attributes, and thus the type checking will not eliminate any of the maps.

Finally for generic data types such as integers, floating points, dates, and URL addresses, IBMiner records the number of times any value from each data type is used with an attribute to do a more effective type-checking. For instance, if for an attribute, say height, mostly integer or floating point values are used, the type checking will cancel any other value types (e.g. strings, dates, etc.) for this attribute.

3.3.3.5   Suggesting Secondary Matches

As mentioned earlier, IBMiner may find more than one ranked match for a triple in T_n. Intuitively, such cases are implying a possible synonym for the mapped attributes. Synonyms play a crucial role in unifying different terminologies used in Wikipedia or other date sources. As an example, consider the triple *<Johann Sebastian Bach, was born in, Eisenach>* in T_n. The technique mentioned in Subsection 3.3.3.1 finds three ranked matches for link '*was born in*'. These matches are '*born*', '*birthPlace*', and '*origin*'. Although all of these are correct matches, IBMiner only picks the most evident one ('*born*' in this case) since for many cases the less evident ranked matches are wrong.

To verify which of the secondary ranked matches are actually correct, IBMiner uses a very similar idea to that for *PM*. The idea is that if two attributes are synonyms, they are usually expressed in similar ways in the text (e.g. attributes '*birthdate*' and '*dateOfBirth*' are synonyms and they are both presented in the text with links such as '*was born on*', '*born on*', or '*birthdate is*'). Thus, IBMiner constructs a structure, called *Potential Attribute Synonyms* (*PAS*), to count the number of times each pair of attributes in the InfoBoxes are presented in the text in the same form (i.e. with the same link). These numbers are then used to compute the probability of that any given two attributes are synonyms. Since *PAS* is also used in Context-aware Synonym Suggestion System (CS^3), thus we leave the detailed discussion of *PAS* in Section 3.4.1.

3.4 CS^3: Discovering Attribute and Entity Synonyms

Synonyms are terms describing the same concept, which can be used interchangeably. According to this definition, no matter what context is used, the synonym for a term is fixed (e.g. '*birthdate*' and '*date of birth*' are always synonyms). However, the meaning or semantic of a term usually depends on the context in which the term is used. The synonym also varies as the context changes. For instance, in an article describing IT companies, the synonym of the attribute name '*wasCreatedOnDate*' most probably is '*founded date*'. In this case, knowing that the attribute is used for the name of a company is a contextual information that helps us find an appropriate synonym for '*wasCreatedOnDate*'. However, if this attribute is used for something else, such as an invention, one can not use the same synonym for it.

Being aware of the context is even more useful for resolving polynymous phrases, which are in fact much more prevalent than exact synonyms in the knowledge bases. For example, consider the entity/subject name '*Johann Sebastian Bach*'. Due to its popularity, a general understanding is that the entity is describing the famous German classical musician. However, what if we know that for this specific entity the birthplace is in '*Berlin*'. This simple contextual information will lead us to the conclusion that the entity is refereing to the painter who was actually the grandson of the famous musician Johann Sebastian Bach. A very similar issue exists for the attribute synonyms.

For instance considering attribute '*born*', '*birthdate*' can be a synonym for '*born*' when it is used with a value of type '*date*'; but if '*born*' is used with values which indicate places, then '*birthplace*' should be considered as its synonym.

CS^3 constructs a structure called *Potential Attribute Synonyms* (*PAS*) to extract attribute synonym. In the generation of *PAS*, CS^3 essentially counts the number of times each pair of attributes are used between the same subject and value and with the same corresponding semantic link in the TextGraphs. The context in this case is considered to be the categorical information for the subject and the value. These numbers are then used to compute the probability that any given two attributes are synonyms. Next subsection describes the process of generating *PAS*. Later in Subsection 3.4.2, we will discuss our approach to suggest entity synonyms and improve existing ones.

3.4.1 Generating Attribute Synonyms

Intuitively, if two attributes (say '*birthdate*' and '*dateOfBirth*') are synonyms in a specific context, they should be represented with the same (or very similar) semantic links in the TextGraphs (e.g. with semantic links such as '*was born on*', '*born on*', or '*birthdate is*'). In simpler words, we use text as the witness for our attribute synonyms. Moreover, the context, which is defined as the categories for the subjects (and for the values), should be very similar for synonymous attributes.

More formally, let attributes α_i and α_j be two matches for link l in initial triple $<s, l, v>$. Let $N_{i,j} (= N_{j,i})$ be the total number of times both α_i and α_j are the interpretation of the same link (in the initial triples) between category sets C_s and C_v. Also, let N_x be the total number of time α_x is used between C_s and C_v. Thus the probability that α_i (α_j) is a synonym for α_j (α_i) can be computed by $N_{i,j}/N_j$ ($N_{i,j}/N_i$). Obviously this is not always a symmetric relationship (e.g. '*born*' attribute is always a synonym for '*birthdate*', but not the other way around, since '*born*' may also refer to '*birthplace*' or '*birthname*' as well). In other words having N_i and $N_{i,j}$ computed, we can resolve both synonyms and polynyms for any given context (C_s and C_v).

With the above intuition in mind, the goal in *PAS* is to compute N_i and $N_{i,j}$. Next we explain how CS^3 constructs *PAS* in one-pass algorithm which is essential for scaling up our system. For

each two records in *PM* such as $<c_s, l, c_v>$: α_i and $<c_s, l, c_v>$: α_j respectively with evidence frequency e_i and e_j ($e_i \leq e_j$), we add the following two records to PAS:

$$<c_s, \alpha_i, c_v>: \alpha_j$$

$$<c_s, \alpha_j, c_v>: \alpha_i$$

Both records are inserted with the same evidence frequency e_i. Note that, if the records are already in the current *PAS*, we increase their evidence frequency by e_i. At the very same time we also count the number of times each attribute is used between a pair of categories. This is necessary for estimating N_i and computing the final weights for the attribute synonyms. That is for the case above, we add the following two *PAS* records as well:

$$<c_s, \alpha_i, c_v>: \text{``'} \text{ (with evidence } e_i)$$

$$<c_s, \alpha_j, c_v>: \text{``'} \text{ (with evidence } e_j)$$

Improving *PAS* with Matching InfoBox Items: Potential attribute synonyms can be also derived from different knowledge bases which contain the same piece of knowledge, but in different attribute names. For instance let $<J.S.Bach, birthdate, 1685>$ and $<J.S.Bach, wasBornOnDate, 1685>$ be two InfoBox triples indicating $bach$'s birthdate. Since the subject and value part of the two triples matches, one may say $birthdate$ and $wasBornOnDate$ are synonyms. To add these types of synonyms to the *PAS* structure, we follow the exact same idea explained earlier in this section. That is, consider two triples such as $<s, \alpha_i, v>$ and $<s, \alpha_j, v>$ in which α_i and α_j may be a synonym. Also, let s and v respectively belong to category sets $C_s = \{c_{s1}, c_{s2}, ...\}$ and $C_v = \{c_{v1}, c_{v2}, ...\}$. Thus, for all $c_s \in C_s$ and $c_v \in C_v$ we add the following triples to *PAS*:

$$<c_s, \alpha_i, c_v>: \alpha_j \text{ (with evidence 1)}$$

$$<c_s, \alpha_j, c_v>: \alpha_i \text{ (with evidence 1)}$$

This intuitively means that from the context (category) of c_s to c_v, attributes α_i and α_j may be synonyms. Again more examples for these categories and attributes increase the evidence which in turn improve the quality of the final attribute synonyms. Much in the same way as learning from initial triples, we count the number of times that an attribute is used between any possible pair of categories (c_s and c_v) to estimate N_i.

Generating Final Attribute Synonyms: Once *PAS* structure is built, it is easy to compute attribute synonyms as described earlier. Assume we want to find best synonyms for attribute α_i in InfoBox Triple $t=<s,\ \alpha_i,\ v>$. Using *PAS*, for all possible α_j, all $c_s \in C_s$, and all $c_v \in C_v$, we aggregate the evidence frequency (e) of records such as $<c_s,\ \alpha_i,\ c_v>$: α_j in *PAS* to compute $N_{i,j}$. Similarly, we compute N_j by aggregating the evidence frequency (e) of all records in form of $<c_s,\ \alpha_i,\ c_v>$: ''. Finally, we only accept attribute α_j as the synonym of α_j, if $N_{i,j}/N_i$ and $N_{i,j}$ are respectively above predefined thresholds τ_{sc} and τ_{se}. We study the effect of such thresholds in Section 3.7.

3.4.2 Generating Entity Synonyms

There are several techniques to find entity synonyms. Approaches based on the string similarity matching [Nav01], manually created synonym dictionaries [SR98], automatically generated synonyms from click log [CLP10, CLP12], and synonyms generated by other data/text mining approaches [Tur01, MKI13b] are only a few examples of such techniques. Although performing very well on suggesting context-independent synonyms, they do not explicitly consider the contextual information for suggesting more appropriate synonyms and resolving polynyms.

Very similar to context-aware attribute synonyms in which the context of the subject and value used with an attribute plays a crucial role on the synonyms for that attribute, we can define context-aware entity synonyms. For each entity name, CS^3 uses the categorical information of the entity as well as all the InfoBox triples of the entity as the contextual information for that entity. Thus to complete the exiting entity synonym suggestion techniques, for any suggested pair of synonymous entities, we compute entities context similarity to verify the correctness of the suggested synonym.

It is important to understand that this approach should be used as a complementary technique over the existing ones for two main reasons. First, context similarity of two entities does not always imply that they are synonyms specially when many pieces of knowledge are missing for most of entities in the current knowledge bases. Second, it is not feasible to compute the context similarity of all possible pairs of entities due to the large number of existing entities. In this work, we use the OntoMiner system [MKI13b] in addition to simple string matching techniques (e.g. Exact string matching, having common words, and edit distance) to suggest initial possible synonyms.

Let '*Johann Sebastian Bach*' and '*J.S. Bach*' be two synonyms that two different knowledge bases are using to denote the famous musician. A simple string matching would offer these two entity as being synonyms. Thus we compare their contextual information and realize that they have many common attributes with similar values for them (e.g. same values for attributes *occupation*, *birthdate*, *birthplace*, etc.). Also they both belong to many common categories (e.g. *Cat:German musician*, *Cat:Composer*, *Cat:people*, etc.). Thus we suggest them as entity synonyms with high confidence. However, consider '*Johann Sebastian Bach (painter)*' and '*J.S. Bach*' entities. Although the initial synonym suggestion technique may suggest them as synonyms, since their contextual information is quite different (e.i. they have different values for common attributes *occupation*, *birthplace*, *birthdate*, *deathplace*, etc.) our system does not accept them as being synonyms.

3.5   Knowledge Provenance Management

The quality of knowledge in semantic web is very important for semantic applications, especially for semantic search systems. To access the quality of knowledge generated by a knowledge discover process, we need to archive the provenance for every fact in knowledge base. In reality, different applications may require different types of qualities, such as the confidence value, evidence frequency, lineage, etc. Therefore, a general model is needed for knowledge provenance management.

There have been several models for capturing provenance in semantic web [TFK11, MWF07]. In SWIM, we extend *the abstract provenance model* discussed in [TFK11] to archive knowledge

provenance. The process of capturing knowledge provenance consists of two main phases: (i) constructing the provenance polynomial for every fact using semiring; (ii) materializing the polynomial for each type of provenance. As we discussed earlier, the advantage of such model is the generality. It does not only work for specific provenance, but also any provenance that can be expressed using semiring. In this way, we just need to do single round evaluation to construct the expression of provenance process (*how provenance*). Then SWIM can evaluate various types of provenance by computing the provenance value based on corresponding semirings.

In order to support different applications on knowledge base, SWIM archives two types of knowledge provenance: knowledge lineage and confidence level. Knowledge lineage of a fact f is the set of source facts which involves in the generation of f. The knowledge lineage can help us identify the source facts of questionable results. Confidence level is a number that indicates the confidence on the correctness of this fact. It can be generated by applications or manually specified by users. Confidence level is important to reflect the reliability of knowledge.

Knowledge provenance has several important applications such as restricted search on specific sources, tracking erroneous knowledge pieces to the emanating source, and better ranking techniques based on reliability of the knowledge in each source. In SWIM, some provenance-based functions are being investigated. First, provenance-based knowledge maintenance. For some facts in knowledge base, the generation process may be very complex, which means that it may be time-consuming to regenerate the knowledge and related provenance when the source knowledge bases are changed. To facilitate the maintenance of knowledge, we can materialize the provenance expression to a graph structure. Then when we need to update the knowledge and its provenance value, we can directly use the intermediate results in materialized nodes. Second, provenance based debugging. In SWIM, we support semantic search using SPARQL. Then the knowledge provenance of SPARQL queries can be captured to debug the suspicious results.

3.6   User-Friendly Interfaces for Browsing and Editing Knowledge

Usability represents a major problem in semantic web, since the knowledge base is now usable only by people who can write SPARQL queries thus casual users are excluded. Even expert programmers will need to spend a fair amount of time to learn DBpedia and thousands of names of entities and properties there used (e.g., names such as: foaf:givenName and dbpprop:populationTotal). On the other hand, it is a painful task for users to prepare structured summaries for a given document since he/she needs to remember the schema and pick correct attribute names. Thus, we propose two interfaces: InfoBox Knowledge-Base Browser (IBKB) and InfoBox Editor (IBE).

The InfoBox Knowledge-Base Browser (IBKB): IBKB is implemented to let users browse the current knowledge base. For a given subject, the tool provides i) structured summary items in user specified order, ii) the synonyms found by IBminer for the attributes used in the summaries, and iii) wrong summary items recognized by IBminer. The tool also includes the sources of each piece of information which can be one or more of the initial knowledge-bases and/or IBminer. By clicking on each source name, the user will be provided with the original form of the triple in that source. Each entity in the result pages is also connected to its own page to make the browsing easier for the users.

Using IBKB, users can select one or more summary items from the user interface, and provide their feedback on the correctness, relevance, and significance of the items. In addition to using such feedback to improve IBminer's performance and to tune its patterns, users' feedback will be used to ranking the structured summaries.

The InfoBox Editor (IBE): In addition to the browsing tool for the current knowledge base, we provide an easy-to-use tool, referred to as IBE, for enhancing the manual process of generating structured information by the users such as in Wikipedia. For the existing subjects, IBE allows users to add more textual information and structured summaries. To create a new subject, users are asked to enter the name (a descriptive subject), one or more categories for the subject, and a descriptive text for it. They can optionally add as many structured summaries as they desire. The tool suggests a domain, a InfoBox template, and some structured summaries extracted from

75

the entered text. In this way, users can easily edit the summaries and fill the missing spots in the suggested templates without worrying about the underlying attribute names. As a result, the manually generated summaries will follow a more standard terminology; this will improve the quality of the final knowledge base.

IBKB and IBE outline a new level of functionality required for curated Web corpora to take a central role in advanced applications. Thus the new responsibility of curators will go beyond that of enabling the creation of textual documents and supervising their contents. They will also be responsible for promoting and supervising the process of knowledge creation and integration, crucial for the many applications that rely on the knowledge bases created from Web document corpora. Recent developments, including Wikidata [Wik], underscore the significance of this trend.

More details about IBKB and IBE are available in [MGZ13b].

3.7 Experimental Evaluation

In this section, we test and evaluate different steps of creating IKBStore in terms of precision and recall. To this end, we create an initial knowledge base using subjects listed in Wikipedia for three specific domains (Musicians, Actors, and Institutes). For these subjects, we add their related structured data from DBpedia and YaGo2 to our initial knowledge bases. Then, to learn *PM* and *PAS* structures, we use the entire Wikipedia's long abstracts provided by DBpedia for the mentioned subjects. We should state that IBMiner only uses the free text and thus can take advantage of any other source of textual information.

All the experiments are performed in a single machine with 16 cores of 2.27GHz and 16GB of main memory. This machine is running Ubuntu12. On average, SemScape spends 3.07 seconds on generating initial semantic links for each sentence on a singe CPU. That is, using 16 cores, we were able to generate initial semantic links for 5.2 sentences per second.

Dataset Name	Subjects #	InfoBox Triples #	Subjects with Abstract	Subjects with InfoBox	Sentences per Abstract
Musicians	65835	687184	65476	52339	8.4
Actors	52710	670296	52594	50212	6.2
Institutes	86163	952283	84690	54861	5.9

Table 3.2: Description of data sets used in experiments.

3.7.1 Data Sets

To perform our evaluation studies, we created three data sets for the domains of Musicians, Actors, and Institutes from the subjects and their accompanying abstracts in Wikipedia. These data sets do not share any subject, and in total they cover around 7.9% of Wikipedia subjects. To build each data set, we started from a general WikiPedia category describing the domain of the data set (e.g. *Category:Musicians* for Musicians data set) and collected all the Wikipedia pages in this category or any of its descendant categories up to four levels. Table 3.2 provides more details on each data set.

As for our initial knowledge base, we used *DBpedia* and Yago2, which provide a better starting point for our text mining tools. Then we used IBminer to mine the text of the entire long abstract of each article to create our data set. We should stress that no structured data was associated with the text in our experiments. In other words, for each subject one can also collect text from different sources, and employ IBMiner and CS^3 to generate InfoBox triples and synonyms from that text.

To evaluate the quality of existing knowledge base, we randomly selected 50 subjects (which have both abstract and InfoBox) from the Musicians data set. For each subject, we compared the information provided in its abstract and in its InfoBox. For these 50 subjects, 1155 unique InfoBox triples (and 92 duplicate triples) are listed in DBpedia. Interestingly, only 305 of these triples are inferable from the abstract text (only 26.4% of the InfoBoxes are covered by the accompanying text.) More importantly, we have found 47 wrong triples (4.0%) and 146 (12.6%) triples which are not useful in terms of main knowledge (e.g. triples specifying upload file name, image name, format, file size, or alignment). Thus an optimistic estimation is that at most 76% of facts provided

in DBpedia are useful for semantic-related applications. We may call this portion of the knowledge base as *useful* in this section. Moreover, the accuracy of DBpedia in this case is less than 96%.

3.7.2 Completing Knowledge by IBminer

3.7.2.1 Precision/Recall Performance

In our experiment, we built the Potential Match from 80% of the Musicians data set, using 50 categories ($N_C = 50$) and 4 levels ($L = 4$). The total number of initial triples for this chunk of data set is $|T_n| = 3.7M$, while $52.9K$ of them match with original InfoBoxes ($|T_m| = 52.9K$). Using these *PM*s and the initial triples generated from the remaining 20% of the abstracts, we generated our InfoBox triples without any frequency and confidence constrains (i.e. $\tau_e = 0$ and $\tau_c = 0.0$). Later in this section, we analyze the effect of N_C and L selection on the results.

To estimate the accuracy of the final triples, we randomly selected 5% of the generated triples ($\approx 42K$) and carefully graded them by matching against their abstracts. Many similar systems such as [CBK10, HSB11, WLW12] have also used manual grading due to the lack of good benchmarks. Recall estimation is also very hard since we again do not know what portion of the InfoBoxes in Wikipedia are covered or mentioned in its accompanying text (long abstract for our case). Thus, we only used those InfoBox triples which match at least one of our initial triples, and compute how many of them are also generated by IBMiner . In this way, we make sure that the resulting InfoBox triples were most likely to be mentioned in the text.

To have a better understanding of the issue, we studied the relation between the abstracts and existing InfoBox triples in DBpedia for 50 randomly selected subjects (which have both abstract and InfoBox) from the Musicians domain. Interestingly, out of 1155 unique[2] InfoBox triples, only 305 are covered by their abstracts (i.e. only 24.4%). We should also add that many of the covered cases need extra or common-sense knowledge to be converted to the exact InfoBox format (e.g. different formats for names or dates). More importantly, we have found 47 *wrong triples* (3.8%) and 146 *unimportant triples* (11.7%) (e.g. file names, image names, formats, size, or alignment).

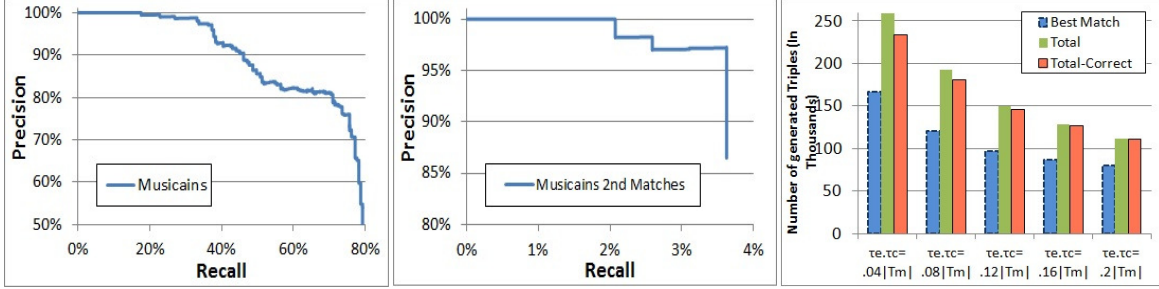[2]We had encountered 92 (7.4%) duplicate triples.

Figure 3.3: Results for Musicians data set: a) Precision/Recall diagram for best matches, b) Precision/Recall diagram for attribute synonyms, and c) the size of generated results for the test data set.

Next we will present our results on the Musicians data set.

Best Matches: Part a) in Figure 3.3 depicts the precision/recall diagram for best matches. As we decrease our thresholds on potential match evidence count (τ_e) and confidence (τ_c), we generate more triples with lower recall and precision. To ease our analysis, we study the effect of both threshold in the same experiment by multiplying them. As can be seen, for the first 20% coverage, IBMiner is generating only correct information. For these cases, τ_e is very high. To reach 97% precision which is more than what DBpedia offers, one should set $\tau_c.\tau_e$ to 6,300 ($\approx .12|T_m|$). For this case as shown in Part c) of the figure, IBMiner generates around $96.6K$ triples with 37.3% recall.

Secondary matches: We also generate the secondary matches or what we refer to as *attribute synonyms* for all the InfoBox triples generated in the previous step. Precision and recall are computed similarly and depicted in part b) of Figure 3.3 (while the potential attribute synonym evidence count (τ_{se}) decreases from right to left). For instance, to have 97% accuracy one need to set $\tau_{sc}.\tau_{se}$ to 24,000($\approx .9|T_m|$). Although synonym results indicate only 3.6% improvement on the overall recall, the number of correct new triples that they generate are relatively large ($53.6K$ triples).

Part c) of Figure 3.3 summarizes the number of best matching triples, the total number of generated items, and the total number of correct items for various τ_e and τ_{se}. It shows that for $\tau_e=.12|T_m|$, we reach up to 97% accuracy while the total number of new InfoBox triples is $146.3K$. If we do not consider duplicate, unimportant, and wrong triples in the original InfoBoxes, then the
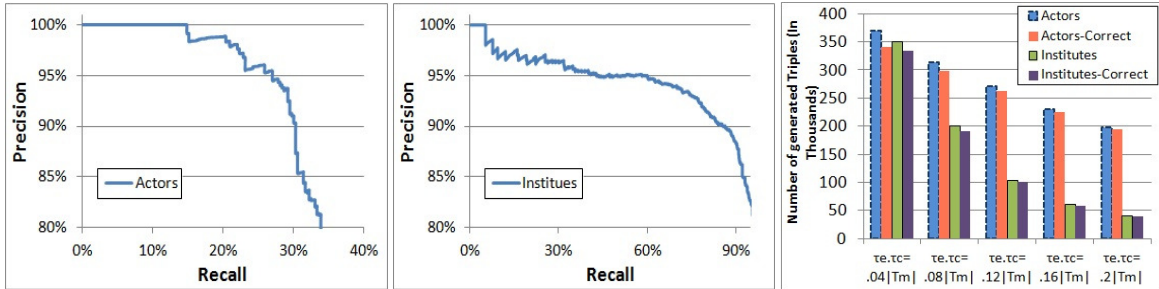
Figure 3.4: a) Precision/Recall diagram for best matches (Actors), b) Precision/Recall diagram for best matches (Institutes), and c) the size of generated results for Actors and Institutes.

coverage of the current InfoBoxes is improved by at least 27.6%. This is actually very impressive considering that IBMiner only uses the long abstract of the page.

To study IBMiner 's performance on different domains, we ran it on two other data sets: i) Actors data set which has similar attributes with the Musicians, and ii) Institutes data set which has completely different set of attributes from the other two. For these experiments, we used N_C=50 and L=4. The PM for each case is constructed considering the entire data set. As shown in Table 3.2, the average number of sentences in these two data sets is less than that for Musicians, however the total number of resulted triples are still comparable to that for Musicians data set.

As in the case of Musicians data set, we manually graded triples from the resulted triples for Actors and Institutes for minimum possible thresholds (5000 triples from each). Part a) and b) in Figure 3.4 depict the precision/recall diagrams for Actors and Institutes data sets respectively when $\tau_c.\tau_e$ decreases from left to right. Although we did not create any specific GD rule for these two data sets, their accuracy can still reach high values. This simply shows that our technique is domain-independent.

Part a) of Figure 3.4 depicts the precision in Actors results drops early. The main reason for this outcome is that many triples with same attribute and same value (more than 300 in some cases) in the original InfoBoxes are simply wrong (e.g., values *TV*, *Film*, and *Television* are frequently listed as the *occupation* of a person). A very similar problem exists for the Musicians data set (e.g. *singer* is listed as the *instrument* for a person in more than 300 evidences). Since the the same errors are repeated in several examples, IBMiner generates some high-confidence wrong triples.
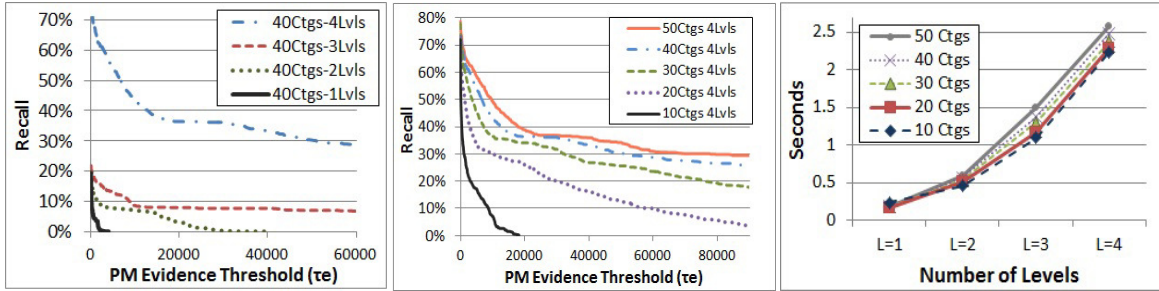
80

Figure 3.5: a) The impact of increasing level number on Recall, b) The impact of increasing Categories number on Recall, and c) InfoBox generation delay per abstract.

Evenso, IBMiner is still able to generate $270, 8K$ results with 97.0% accuracy and 25.6% recall.

On the other hand, the InfoBox triples in the Institutes data set are organized very loosely, with different names used for similar attributes. Thus many more attribute names are used in this data set than the other two, and as a result, for many attributes, IBMiner is given very few example to learn from,. This directly affects the number of final results (Part b in Figure 3.4). Even so, IBMiner manages to generate $102.9K$ results for this data set, with 97.0% accuracy and 25.4% recall. Interestingly, if one can tolerate 95% accuracy, the recall for this case can reach up to 56.1% which is quite impressive.

In Part c) of Figure 3.4, we provide the total number of generated results for various evidence and confidence thresholds. Although better tuned thresholds could be used, we kept the same thresholds used for Musicians (scaled by the size of T_m) and report the total number of generated results and the number of accurately generated ones for each data set in this figure. Although better coverage is reported for the Institutes data set, for accuracy more than 96% much fewer results were generated. This is due to the large number of attribute names and few examples for each of the attributes in the original InfoBoxes in Wikipedia. IBMiner also generates more results for Actors data set than Musicians data sets, which is mainly due to higher popularity of pages in the Actors data set which contains many contemporary people.

3.7.2.2 The Impact of Category Selection

To understand the impact of category selection technique, we repeat the previously mentioned experiments for different levels (L) and category numbers (N_C). Here, we only compare the results based on the recall estimation. First, we fix the N_C at 40 and change L from 1 to 4. Part a) of Figure 3.5 depicts the estimated recall (only for the best matches) while *PM*'s frequency threshold (τ_e) increases. Not surprisingly, as L increases, the recall improves significantly, since more general categories are considered in PM construction. On the other hand, using more categories also improves the recall as depicted in Part b) of the figure. In this part, we fix L at 4 and change N_C from 10 to 50. The results indicate that even with 10 categories, we may obtain good recalls for lower frequency thresholds (τ_e). This manly proves the efficiency of our category selection technique, since even with few categories the system is able to reach high coverage.

In Part c) of Figure 3.5, we compare the time performance of all the above cases. This diagram shows the average time spent on generating final InfoBoxes for each abstract. As we increase the levels, the processing time increases exponentially since IBminer performs more database accesses to generate the categories structure. However, we observe very small changes with the increase in N_C, since it does not require any additional database accesses.

3.7.2.3 Application-based Evaluation

In previous sections, we showed that the recall of the generated information by IBminer can reach high values. However, to understand how this improvement in recall actually helps the applications of knowledge bases, we carried out an application based experiment. In this experiment, we create a set of popular queries (explained next) and compare the search results obtained after DBpedia is extended by IBminer with the original ones. We also compare these with the results produced by the iPopulator [LBN10] system. Although other similar systems have been proposed recently, we selected iPopulator for two main reasons: i) it is designed for generated information from free text in Wikipedia pages, and more importantly ii) the iPopulator's results are publicly available.

Creating the query set: In order to create a set of popular queries for our evaluation, we used
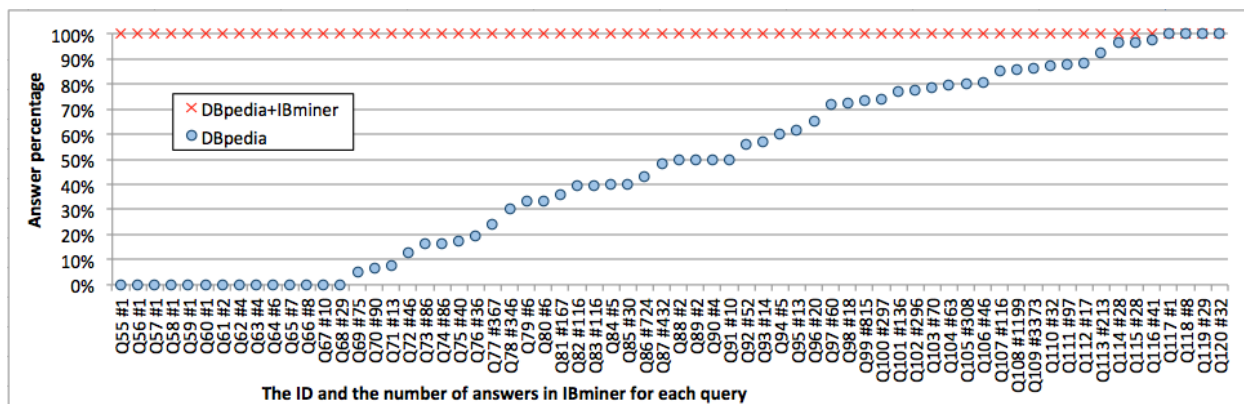
Figure 3.6: Number of results generated for different queries using DBpedia and IBminer knowledge bases.

Google Search Auto-Complete system, and found around 150 keyword queries suggested by this system to complete two phrases: "*musicians who*" and "*actors who*". We were able to translate 120 of these keyword-based queries to SPARQL. The remaining keyword queries, (e.g., "*Actors who are tall*", "*Musicians who married normal people*", etc.) are too vague for a precise translation and quantification and were thus ignored.

Knowledge Bases: Three different knowledge bases (KBs) are used in this evaluation. As for the baseline KB, we use DBpedia's InfoBox triples. Note that both IBminer and iPopulator use DBpedia as their initial knowledge bases. Since the goal is to measure how much IBminer's result improves DBpedia, we combine the triples in DBpedia and IBminer into our second KB called IBminer+DBpedia. We create the third KB similarly for iPopulator by adding its results to those of DBpedia.

After preparing the queries and the knowledge bases, we employed Apache Jena [Jen], and ran the queries using the three knowledge bases. For more than 44% of the queries, no answer is found from any of the knowledge bases. This very clearly proves the incompleteness of the current knowledge bases. Nevertheless, for the remaining queries, the IBminer case almost always (except 4 queries) finds more answers than the baseline case. Figure 3.6 shows what portion of the answers (for each query) that are found using IBminer+DBpedia is also inferable using

only DBpedia's knowledge base. To better present the results, we sort the queries based on the percentage introduced above. We also exclude the queries with no answer in any of the knowledge bases from the figure.

As Figure 3.6 indicates, there are several cases (11.6%) in which DBpedia is not able to provide any answer for the queries as opposed to IBminer. Using IBminer, we are actually able to find between 1 to 29 answers for these queries. The number of found results using IBminer+DBpedia is included in the horizontal axis in addition to the query IDs in Figure 3.6. In total for all queries, IBminer improves original DBpedia by 53.3% on answering structured queries. This value for iPopulator is less than 1%. In fact, iPopulator was able to slightly improve DBpedia in query answering only for 6.6% of the queries. This is mainly because of three reasons: i) the recall of iPopulator is much lower than IBminer (at least 10 times). ii) iPopulator does not recognize synonymous attributes, and iii) some of the generated values in iPopulator are not accurate and in addition to the correct values, they usually contain other unrelated textual phrases as well.

All queries and their answers from the introduced knowledge bases are available at SWIMS website [Sem]. In this website, users can also use our recently proposed demo at PVLDB13 [MGZ13b] and find the list of InfoBox triples for the answers provided for each query. We provide the provenance of each triples, which indicates if the triple is taken from DBpedia, IBminer, etc. Thus verifying the quality of the answers for these queries is quite easy.

3.7.3   Completing Knowledge by Attribute Synonyms

In order to compute the precision and recall for the attribute synonyms generated by CS^3, we use the Musicians data set and construct the PAS structure as described in section 3.4. Using PAS, we generated possible synonyms for $10,000$ InfoBox items in our initial knowledge base. Note that these synonyms are for already existing InfoBoxes which differentiate them from secondary matches evaluated in previous section. This has generated more than $14,900$ synonym items. Among the $10,000$ InfoBox items, 1263 attribute synonyms were listed and our technique generated 994 of them. We used these matches to estimate the recall value of our technique for
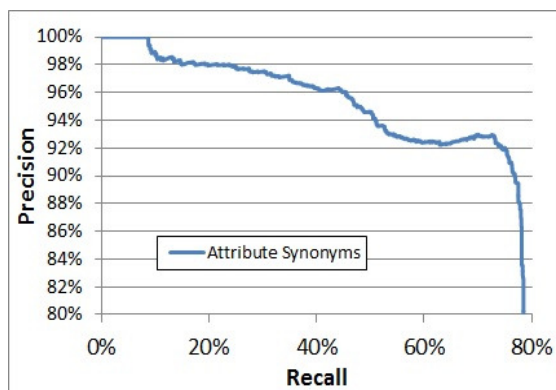
Figure 3.7: The Precision/Recall diagram for the attribute synonyms generated for existing InfoBoxes in the Musicians data set.

different frequency thresholds (τ_{cs}) as shown in Figure 3.7. As for the precision estimation, we manually graded the generated synonyms.

As can be seen in Figure 3.7, CS^3 is able to find more than 74% of the possible synonyms with more than 92% accuracy. In fact, this is a very big step in improving structured query results, since it increase the coverage of the IKBStore by at least 88.3%. This in some sense improves the consistency of the knowledge base by providing more synonymous InfoBox triples. In aggregate with the improvement we achieved by IBMiner , we can state that our IKBStore doubles the size of the current knowledge bases while preserving their precision (if not improving) and improving their consistency.

3.7.4 Summary

After initial integration, knowledge completion and alignment, IKBStore contains 9.2 million English subjects and 105.4 million triples. All triples in our integrated KB are assigned accuracy, confidence, and frequency values, as explained in previous Sections. The sources from which the triples are generated are also stored to support provenance auditing in our KB. We are also processing the whole wikipedia text, which will produce more Infoboxes for IKBStore.

3.8 Related Work

Knowledge Integration There are several attempts to generate large-scale knowledge bases [OPE, CBK10, LS04, BEP08, BLK09, GEO, MUS]. However, none of them are providing any automatic integration technique among existing knowledge bases. There are only a few recent efforts on integrating knowledge bases in both domain-specific and general topics. GeoWordNet [GMF10] is an integrated knowledge base of GeoNames [GEO], WordNet [SR98], and MultiWord-Net [PBG02]. However, the integration is based on the manually created concept mapping which is time-consuming and error-prone for large scale knowledge base integration. In [HSB13, HSB11], Yago2 is integrated with GeoNames and structured information in Wikipedia. The category (class) mapping in Yago2 is performed by simple string matching which is not reliable for large taxonomies such as the one in Wikipedia. Recently, Wu et al. proposed Probase [WLW12] which aims to generate a general taxonomy from web documents. Probase merged the concepts and instances into a large general taxonomy. In taxonomy integration, the concepts with similar context properties (e.g. derived from the same sentence or with similar child nodes) are merged.

Knowledge Extraction A large body of research currently focuses on pattern-based knowledge generation from free text. Etzioni et al. in KnowItAll created a general purpose ontology by finding unary and binary predicate instances from the text [ECD04]. Later, Yates et al. proposed a similar system called TextRunner which improves the accuracy of KnowItAll by considering all meaningful relations in the text [YCB07]. Despite the impressive accuracy in their evaluations, they both suffer from limited coverage issues.

In [NMI07], Nguyen et al. used both semantic and syntactic information to extract relations from Wikipedia. However, their approach focuses on the relations between existing entities in Wikipedia, which is not easily applicable on InfoBox generation and population. Other similar systems such as PROSPERA [NTW11] also suffer from the very same problem. Wu and Weld used an automatic technique to create a training set [WW10]. Their main idea, which was originally proposed in [HZW10], is to assign each attribute/value pair in the InfoBoxes to a sentence in the page using some straightforward heuristics. Then, these assignments are used to learn extractors

which in turn generate more attribute/values. Although it is shown impressive improvement with respect to TextRunner, their accuracy is still limited to less than 90% for most cases.

NELL [CBK10] performs a pattern-based fact generation on a large-scale text set and iteratively improves their knowledge base. In 66 days experiment, NELL generated about 230,000 unary relations (Subject/Category triples) and 12,000 binary relations (InfoBox-like triples) with 74% accuracy. Comparing with the size of the input data set which contains 500 million web pages, the coverage seems too low. Another approach which attempts to generate a large scale commonsense knowledge base is CYC [EG06]. Although, CYC has been recently equipped with several NLP-based and automatic techniques, the main core still relies on supervised techniques on semi-structured data sets.

Another large scale pattern-based knowledge construction system is Probase [WLH11, WLW12], which iteratively induces taxonomical information from large text. At each iteration, Probase uses existing taxonomies to identify new '*isA*' pairs from text and integrates the new pairs into the taxonomies. It also proposes an algorithm to merge and connect concepts. Probase produces a general taxonomy with more than 2.7 million concepts from the textual web documents. However, Probase is only able to generate taxonomical information which is not the focus of this paper.

Recently, two similar approaches to IBminer are proposed by Lange et al. called iPopulator [LBN10] and Liu et al. called DeepDive [NZR12]. iPopulator presents a new technique which uses existing InfoBox triples in Wikipedia to discover structures in the text, that represent possible attribute/value for the subjects. However, iPopulator also suffers from low coverage issue. DeepDive, on the other hand, first employs NLP tools to extract relations from raw text. Then, the relations are used to train statistical models which convert the initial relations into a knowledge base. Although this approach in nature is very similar to IBminer, it does not provide any evaluation or experiment on the quality of the generated triples.

Synonym Suggestion One of the early work in this area is WordNet [SR98] which provides manually defined synonyms for general words. Words and very general terms in WordNet are grouped to set of synonyms called *Synset*. Although WordNet contains 117,000 high quality Synsets, it

is still limited to only general words which miss most of multi-word terms, and as a result it is not effective to be used in large scale knowledge base integration systems. Another approach to extract synonyms is based on approximate string matching (fuzzy string searching) [Nav01]. This technique is effective in detecting certain kinds of format-related synonyms, such as normalization and misspelling. However, this technique can not discover semantic synonyms.

To overcome the shortcomings of the aforementioned approaches, more automatic approaches for generating entity synonyms from web scale documents were presented in recent years. In [CGX09a, CGX09b], Chaudhuri et al. proposed an approach to generate a set of synonyms which are substrings of given entities. This approach exploits the correlation between the substrings and the entities in the web documents to identify the candidate synonyms. In [CLP10, CLP12], Cheng et al. used query click logs in web search engine to derive synonyms. First, according to the click logs, every entity is associated with a set of relevant web pages. Then, the queries which result in certain web pages are considered as possible synonyms for the associated entities of those pages. The synonyms are then refined by evaluating the click patterns of queries. Later in [CCC12], Chakrabarti et al. refined this approach by defining similarity functions, which solves click log sparsity problem and verifies that candidate synonym string is of the same class of the entity. In [Tur01], the author proposed a machine learning algorithm, called PMI-IR to mine synonyms from web documents. In PMI-IR, if two sets of documents are correlated, the words associated with the two sets are taken as candidate synonyms.

CHAPTER 4

Conclusion and Future Work

The success of semantic applications such as semantic search and question answering systems shows that major advances in knowledge discovery and retrieval can be achieved once large-scale knowledge bases are available and the quality of knowledge is guaranteed. Most of efforts in this research area focus on constructing knowledge bases using structured information, whereas knowledge integration techniques are neglected, and so was the use of deep NLP techniques for extracting knowledge from free text. On the other hand, the information in the knowledge base is evolving over time. The history of knowledge base is of great interest to users, who thus need powerful tools to explore it. In this dissertation, we tackled these problems and made significant progress as follows:

(i) We proposed SPARQL^T and implemented RDF-TX which supports powerful queries over the evolution history of RDF knowledge bases. SPARQL^T enables the expression of a wide variety of temporal queries via simple extension of SPARQL query pattern and built-in functions. The evolution history is stored as temporal RDF triples in compressed MVBT. Temporal queries over temporal RDF graphs are efficiently evaluated in the backend query engine that achieves excellent performance by exploiting MVBT as index and leveraging fast algorithms for range selection and temporal join on MVBT. RDF-TX also features a query optimizer that uses the statistics of temporal RDF graphs to find the efficient join orders for complex SPARQL^T queries. Extensive experiments on real world datasets show that RDF-TX outperforms other approaches that use state-of-art RDF engines and relational databases in all kinds of queries and delivers 1 - 2 orders of magnitude performance improvement in complex queries. This confirms the effectiveness and superior performance of our approach.

(ii) We proposed SWIM framework, which consists of a set of tools for knowledge extraction and integration. We first integrate currently existing knowledge bases into a more complete and consistent knowledge base IKBStore. IKBStore merges existing knowledge bases using the interlinks they provide. Then, it employs the text-based knowledge discovery system IBMiner to improve the coverage of the initially integrated knowledge bases by extracting knowledge from free text. Finally, we utilize the CS^3 to extract context-aware attribute and entity synonyms and use them to reconcile among different terminologies used by different knowledge bases. In this way, we create an integrated knowledge base which outperforms existing knowledge bases in terms of coverage, accuracy, and consistency. The provenance information is well preserved in the integrated knowledge base. Our preliminary evaluation also proves this claim as it shows that the integrated knowledge base greatly improves the coverage of the existing knowledge base while slightly improving accuracy and resolving many inconsistencies.

The work presented in this dissertation introduces many opportunities for further studies and improvements. For temporal queries over the history of knowledge bases, we plan to improve RDF-TX on the parallel computing over MVBT indices. The structure of MVBT makes it easy to be partitioned based on the timestamps. The temporal operations are executed in parallel by first computing on different partitions and then merging the results. For many large knowledge bases, the size of history may grow very fast. Thus we also plan to explore more efficient index scheme for temporal RDF data.

As for knowledge integration, IKBStore will be extended and improved with knowledge taken from Wikidata and Freebase, and our interface will be extended with multilingual query capabilities. Moreover, we plan to apply our tools and approach to document corpora other than Wikipedia, e.g., medical and technical encyclopedias.

REFERENCES

[AGD13]   Enrique Alfonseca, Guillermo Garrido, Jean-Yves Delort, and Anselmo Penas. "WHAD: Wikipedia historical attributes data." *LRE*, p. 28, 2013.

[All83]   James F. Allen. "Maintaining Knowledge About Temporal Intervals." *Commun. ACM*, **26**(11):832–843, 1983.

[AS13]   Daniar Achakeev and Bernhard Seeger. "Efficient Bulk Updates on Multiversion B-trees." *PVLDB*, **6**(14):1834–1845, 2013.

[AT10]   Witold Abramowicz and Robert Tolksdorf, editors. *Business Information Systems, 13th International Conference, BIS 2010, Berlin, Germany, May 3-5, 2010. Proceedings*, volume 47 of *Lecture Notes in Business Information Processing*. Springer, 2010.

[ATS11]   P. Adolphs, M. Theobald, U. Schafer, H. Uszkoreit, and G. Weikum. "YAGO-QA: Answering questions by structured knowledge queries." In *Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on*, pp. 158–161. IEEE, 2011.

[AZ12]   Maurizio Atzori and Carlo Zaniolo. "SWiPE: searching wikipedia by example." In *WWW (Companion Volume)*, pp. 309–312, 2012.

[AZ14]   Maurizio Atzori and Carlo Zaniolo. "Expressivity and Accuracy of By-Example Structure Queries on Wikipedia." CSD Technical Report #140017, UCLA, 2014.

[BC02]   Sabin C. Buraga and Gabriel Ciobanu. "A RDF-based Model for Expressing Spatio-Temporal Relations Between Web Sites." In *WISE*, pp. 355–361, 2002.

[BCW05]   Christian Bizer, Richard Cyganiak, and E Rowland Watkins. "Ng4j-named graphs api for jena." In *ESWC*, 2005.

[BEP08]   Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. "Freebase: a collaboratively created graph database for structuring human knowledge." In *SIGMOD*, pp. 1247–1250, 2008.

[BGO96]   Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. "An Asymptotically Optimal Multiversion B-tree." *VLDB*, **5**(4):264–275, 1996.

[BLK09]   Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. "DBpedia - A crystallization point for the Web of Data." *J. Web Sem.*, **7**(3):154–165, 2009.

[BS96]   Jochen Van den Bercken and Bernhard Seeger. "Query Processing Techniques for Multiversion Access Methods." In *VLDB*, pp. 168–179, 1996.

[CBH05]   Jeremy J. Carroll, Christian Bizer, Patrick J. Hayes, and Patrick Stickler. "Named graphs, provenance and trust." In *WWW*, pp. 613–622, 2005.

[CBK10]   Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. "Toward an Architecture for Never-Ending Language Learning." In *AAAI*, 2010.

[CCC12]   Kaushik Chakrabarti, Surajit Chaudhuri, Tao Cheng, and Dong Xin. "A framework for robust discovery of entity synonyms." In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '12, pp. 1384–1392, New York, NY, USA, 2012. ACM.

[CGX09a]   Surajit Chaudhuri, Venkatesh Ganti, and Dong Xin. "Exploiting web search to generate synonyms for entities." In *Proceedings of the 18th international conference on World wide web*, WWW '09, pp. 151–160, New York, NY, USA, 2009. ACM.

[CGX09b]   Surajit Chaudhuri, Venkatesh Ganti, and Dong Xin. "Mining document collections to facilitate accurate approximate entity matching." *Proc. VLDB Endow.*, **2**(1):395–406, August 2009.

[CLP10]   Tao Cheng, Hady Wirawan Lauw, and Stelios Paparizos. "Fuzzy matching of Web queries to structured data." In *ICDE*, pp. 713–716, 2010.

[CLP12]   Tao Cheng, Hady Wirawan Lauw, and Stelios Paparizos. "Entity Synonyms for Structured Web Search." *IEEE Trans. Knowl. Data Eng.*, **24**(10):1862–1875, 2012.

[CZ99]   Cindy Xinmin Chen and Carlo Zaniolo. "Universal Temporal Extensions for Database Languages." In *ICDE*, pp. 428–437, 1999.

[DAA12]   Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. "Provenance for SPARQL Queries." In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, pp. 625–640, 2012.

[dbp]   "DBpedia Live." http://live.dbpedia.org.

[ECD04]   Oren Etzioni, Michael J. Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. "Web-scale information extraction in knowitall: (preliminary results)." In *WWW*, pp. 100–110, 2004.

[EG06]   C. Elkan and R. Greiner. "Building large knowledge-based systems: representation and inference in the Cyc project." *Artificial Intelligence*, **61**(1):41–52, 2006.

[GCA15]   Shi Gao, Muhao Chen, Maurizio Atzori, Jiaqi Gu, and Carlo Zaniolo. "SPARQLT and its User-Friendly Interface for Managing and Querying the History of RDF Knowledge Bases." In *ISWC*, 2015.

[GEO]   "GeoNames." http://www.geonames.org.

[GGZ15]   Shi Gao, Jiaqi Gu, and Carlo Zaniolo. "RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases." *UCLA CS Tech Report 150004*, 2015.

[GHV05]   Claudio Gutiérrez, Carlos A. Hurtado, and Alejandro A. Vaisman. "Temporal RDF." In *ESWC*, pp. 93–107, 2005.

[GHV07]   Claudio Gutierrez, Carlos A. Hurtado, and Alejandro A. Vaisman. "Introducing Time into RDF." *TKDE*, **19**(2):207–218, 2007.

[GKC13]   Floris Geerts, Grigoris Karvounarakis, Vassilis Christophides, and Irini Fundulaki. "Algebraic structures for capturing the provenance of SPARQL queries." In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pp. 153–164, 2013.

[GMF10]   Fausto Giunchiglia, Vincenzo Maltese, Feroz Farazi, and Biswanath Dutta. "Geo-oWordNet: A Resource for Geo-spatial Applications." In *ESWC (1)*, pp. 121–136, 2010.

[gov]   "GovTrack Dataset." https://www.govtrack.us/.

[Gra10]   Fabio Grandi. "T-SPARQL: a TSQL2-like temporal query language for RDF." *GraphQ*, pp. 21–30, 2010.

[HBS10]   Rasmus Hahn, Christian Bizer, Christopher Sahnwaldt, Christian Herta, Scott Robinson, Michaela Bürgle, Holger Düwiger, and Ulrich Scheel. "Faceted Wikipedia Search." In *BIS*, 2010.

[HCH08]   Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors. *WWW 2008, Beijing, China, April 21-25, 2008*. ACM, 2008.

[HSB11]   Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo, and Gerhard Weikum. "YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages." WWW, pp. 229–232, 2011.

[HSB13]   Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. "YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia." *Artif. Intell.*, **194**:28–61, 2013.

[HZW10]   Raphael Hoffmann, Congle Zhang, and Daniel S. Weld. "Learning 5000 Relational Extractors." In *ACL*, pp. 286–295, 2010.

[Jen]   "Apache Jena." http://jena.apache.org/.

[JSL00]   Linan Jiang, Betty Salzberg, David B. Lomet, and Manuel Barrena García. "The BT-tree: A Branched and Temporal Access Method." In *VLDB*, pp. 451–460, 2000.

[JT10]   Xing Jiang and Ah-Hwee Tan. "CRCTOL: A semantic-based domain ontology learning system." *JASIST*, **61**(1):150–168, 2010.

[KPG12]   Chanhyun Kang, Andrea Pugliese, John Grant, and V. S. Subrahmanian. "STUN: Spatio-Temporal Uncertain (Social) Networks." In *ASONAM*, pp. 543–550, 2012.

[LBM05]   David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. "Immortal DB: transaction time support for SQL server." In *SIGMOD*, 2005.

[LBN10]   Dustin Lange, Christoph Böhm, and Felix Naumann. "Extracting structured information from Wikipedia articles to populate infoboxes." In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pp. 1661–1664, New York, NY, USA, 2010. ACM.

[LHN08]   David B. Lomet, Mingsheng Hong, Rimma V. Nehme, and Rui Zhang. "Transaction time indexing with version compression." *PVLDB*, **1**(1):870–881, 2008.

[LS04]   H. Liu and P. Singh. "ConceptNet: A Practical Commonsense Reasoning Tool-Kit." *BT Technology Journal*, **22**(4):211–226, October 2004.

[LWW11]   Taesung Lee, Zhongyuan Wang, Haixun Wang, and Seung won Hwang. "Web Scale Taxonomy Cleansing." *PVLDB*, **4**(12):1295–1306, 2011.

[MAG14]   Hamid Mousavi, Maurizio Atzori, Shi Gao, and Carlo Zaniolo. "Text-Mining, Structured Queries, and Knowledge Management on Web Document Corpora." *SIGMOD Record*, **43**(3):48–54, 2014.

[MB09]   Brian McBride and Mark Butler. "Representing and Querying Historical Information in RDF with Application to E-Discovery." HP Laboratories Technical Report HPL-2009-261, 2009.

[MGK14]   Hamid Mousavi, Shi Gao, Deirdre Kerr, Markus Iseli, and Carlo Zaniolo. "Mining Semantics Structures from Syntactic Structures in Web Document Corpora." *Int. J. Semantic Computing*, **8**(4):461–490, 2014.

[MGZ13a]   Hamid Mousavi, Shi Gao, and Carlo Zaniolo. "Discovering Attribute and Entity Synonyms for Knowledge Integration and Semantic Web Search." *SSW*, 2013.

[MGZ13b]   Hamid Mousavi, Shi Gao, and Carlo Zaniolo. "IBminer: A Text Mining Tool for Constructing and Populating InfoBox Databases and Knowledge Bases." *PVLDB*, **6**(12):1330–1333, 2013.

[MKI11a]   Hamid Mousavi, Deirdre Kerr, and Markus Iseli. "A New Framework for Textual Information Mining over Parse Trees." In *(CRESST Report 775). University of California, Los Angeles*, 2011.

[MKI11b]   Hamid Mousavi, Deirdre Kerr, and Markus Iseli. "A New Framework for Textual Information Mining over Parse Trees." In *ICSC*, 2011.

[MKI13a]   Hamid Mousavi, Deirdre Kerr, Markus Iseli, and Carlo Zaniolo. "Deducing InfoBoxes from Unstructured Text in Wikipedia Pages." In *CSD Technical Report #130001, UCLA*, 2013.

[MKI13b]   Hamid Mousavi, Deirdre Kerr, Markus Iseli, and Carlo Zaniolo. "OntoHarvester: An Unsupervised Ontology Generator from Free Text." In *CSD Technical Report #130003, UCLA*, 2013.

[MKI14]   Hamid Mousavi, Deirdre Kerr, Markus Iseli, and Carlo Zaniolo. "Mining Semantic Structures from Syntactic Structures in Free Text Documents." In *CSD TR #140005, UCLA*, 2014.

[MN06]   Guido Moerkotte and Thomas Neumann. "Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products." In *VLDB*, pp. 930–941, 2006.

[MRS11]   Y. Mass, M. Ramanath, Y. Sagiv, and G. Weikum. "Iq: The case for iterative querying for knowledge." In *CIDR*, 2011.

[MUS]   "MusicBrainz." http://musicbrainz.org.

[MWF07]   Simon Miles, Sylvia C. Wong, Weijian Fang, Paul Groth, Klaus-Peter Zauner, and Luc Moreau. "Provenance-based validation of e-science experiments." *Web Semant.*, **5**(1), March 2007.

[Nav01]   Gonzalo Navarro. "A guided tour to approximate string matching." *ACM Comput. Surv.*, **33**(1):31–88, March 2001.

[ND]   Mario A. Nascimento and Margaret H. Dunham. "Indexing Valid Time Databases via B$^+$-Trees." *TKDE*, **11**(6):929–947.

[NM11]   Thomas Neumann and Guido Moerkotte. "Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins." ICDE, pp. 984–994, 2011.

[NMI07]   Dat P. T Nguyen, Yutaka Matsuo, and Mitsuru Ishizuka. "Exploiting syntactic and semantic information for relation extraction from wikipedia." In *IJCAI07-TextLinkWS*, 2007.

[NTW11]   Ndapandula Nakashole, Martin Theobald, and Gerhard Weikum. "Scalable knowledge harvesting with high precision and high recall." WSDM '11, pp. 227–236, New York, NY, USA, 2011. ACM.

[NW10]   Thomas Neumann and Gerhard Weikum. "The RDF-3X Engine for Scalable Management of RDF Data." *VLDBJ*, **19**(1):91–113, 2010.

[NZR12]   Feng Niu, Ce Zhang, Christopher Re, and Jude W. Shavlik. "DeepDive: Web-scale Knowledge-base Construction using Statistical Learning and Inference." In *VLDS*, pp. 25–28, 2012.

[OPE]   "OPENCYC." http://www.cyc.com/platform/opencyc.

[PBG02]   Emanuele Pianta, Luisa Bentivogli, and Christian Girardi. "MultiWordNet: developing an aligned multilingual database." In *Proceedings of the First International Conference on Global WordNet*, 2002.

[PGR10]   Aditya G. Parameswaran, Hector Garcia-Molina, and Anand Rajaraman. "Towards The Web of Concepts: Extracting Concepts from Large Datasets." *PVLDB*, **3**(1):566–577, 2010.

[PSH07]   Matthew Perry, Amit P. Sheth, Farshad Hakimpour, and Prateek Jain. "Supporting Complex Thematic, Spatial and Temporal Queries over Semantic Web Data." In *GeoS*, pp. 228–246, 2007.

[PUS08]   Andrea Pugliese, Octavian Udrea, and V. S. Subrahmanian. "Scaling RDF with Time." In *WWW*, pp. 605–614, 2008.

[Sem]   "Semantic Web Information Management System (SWIMS)." http://semscape.cs.ucla.edu.

[SKW08]   Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. "YAGO: A Large Ontology from Wikipedia and WordNet." *J. Web Sem.*, **6**(3):203–217, 2008.

[SPA]   "SPARQL Query Language for RDF." http://www.w3.org/TR/rdf-sparql-query/.

[SR98]   Michael M. Stark and Richard F. Riesenfeld. "WordNet: An Electronic Lexical Database." In *Eurographics Workshop on Rendering*. MIT Press, 1998.

[SSB08]   Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. "SPARQL basic graph pattern optimization using selectivity estimation." In *WWW*, pp. 595–604, 2008.

[ST99]   Betty Salzberg and Vassilis J. Tsotras. "Comparison of Access Methods for Time-Evolving Data." *ACM Comput. Surv.*, **31**(2):158–221, 1999.

[TB09]   Jonas Tappolet and Abraham Bernstein. "Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL." In *ESWC*, pp. 308–322, 2009.

[TFK11]   Yannis Theoharis, Irini Fundulaki, Grigoris Karvounarakis, and Vassilis Christophides. "On Provenance of Queries on Semantic Web Data." *IEEE Internet Computing*, **15**(1), 2011.

[TML99]   Theodoros Tzouramanis, Yannis Manolopoulos, and Nikos A. Lorentzos. "Overlapping B$^+$-Trees: An Implementation of a Transaction Time Access Method." *DKE*, **29**(3):381–404, 1999.

[Tom96]   David Toman. "Point vs. Interval-based Query Languages for Temporal Databases." In *PODS*, pp. 58–67, 1996.

[Tur01]   Peter D. Turney. "Mining the Web for Synonyms: PMI-IR versus LSA on TOEFL." In *Proceedings of the 12th European Conference on Machine Learning*, EMCL '01, pp. 491–502, London, UK, UK, 2001. Springer-Verlag.

[vir]   "Virtuoso." https://github.com/openlink/virtuoso-opensource.

[Wik]   "Wikidata." http://www.wikidata.org.

[WLH11]   W.Wu, H. Li, H.Wang, and K. Zhu. "Towards a probabilistic taxonomy of many concepts." Technical report, 2011.

[WLW12]   Wentao Wu, Hongsong Li, Haixun Wang, and Kenny Q. Zhu. "Probase: a probabilistic taxonomy for text understanding." SIGMOD '12, pp. 481–492, New York, NY, USA, 2012. ACM.

[Wor12]   "WordNet." http://wordnet.princeton.edu/, 2012.

[WSK03]   Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. "Efficient RDF Storage and Retrieval in Jena2." In *SWDB*, pp. 131–150, 2003.

[WW10]   Fei Wu and Daniel S. Weld. "Open Information Extraction Using Wikipedia." In *ACL*, pp. 118–127, 2010.

[YCB07]   Alexander Yates, Michael Cafarella, Michele Banko, Oren Etzioni, Matthew Broadhead, and Stephen Soderland. "TextRunner: open information extraction on the web." In *Proceedings of Human Language Technologies*, pp. 25–26, Stroudsburg, PA, USA, 2007.

[YLW13]   Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. "TripleBit: a Fast and Compact System for Large Scale RDF Data." *PVLDB*, **6**(7):517–528, 2013.

[YW01]   Jun Yang and Jennifer Widom. "Incremental Computation and Maintenance of Temporal Aggregates." In *ICDE*, pp. 51–60, 2001.

[ZMT01]   Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. "Efficient Computation of Temporal Aggregates with Range Predicates." In *PODS*, 2001.

[ZMT08]   Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. "On Computing Temporal Aggregates with Range Predicates." *TODS*, **33**(2):12:1–12:39, 2008.

[ZTS02]   Donghui Zhang, Vassilis J. Tsotras, and Bernhard Seeger. "Efficient Temporal Join Processing Using Indices." In *ICDE*, pp. 103–113, 2002.

[ZWZ06]   Xin Zhou, Fusheng Wang, and Carlo Zaniolo. "Efficient Temporal Coalescing Query Support in Relational Database Systems." In *DEXA*, pp. 676–686, 2006.