UCLA
UCLA Electronic Theses and Dissertations

Title
Wikipedia Infobox Temporal RDF Knowledge Base and Indices

Permalink
https://escholarship.org/uc/item/7kc476n9

Author
Song, Aige

Publication Date
2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Wikipedia Infobox Temporal RDF Knowledge Base and Indices

A thesis submitted in partial satisfaction

of the requirements for the degree Master of Science

in Computer Science

by

Aige Song

2015

Copyright by

Aige Song

2015

ABSTRACT OF THE THESIS

Wikipedia Infobox Temporal RDF Knowledge Base and Indices

by

Aige Song

Master of Science in Computer Science

University of California, Los Angeles, 2015

Professor Carlo Zaniolo, Chair

As real world evolves, Infoboxes for Wikipedia subjects are updated to reflect the information changes in the real world, and there is a growing interest in the evolution history of subjects in the Wikipedia. Thus, the management of historical information and the efficiencies of queries for these temporal information have become the major concern.

In this paper, we introduce the Wikipedia Infobox temporal RDF knowledge base that constructed from the Wikipedia Infobox history dump, and evaluate the efficiencies of temporal queries based on the temporal knowledge base. Specifically, we evaluate temporal selection and temporal join queries based on different database systems with different indices, including MySQL B+ Tree, PostgreSQL B-Tree, and Interval Tree.

The thesis of Aige Song is approved.

Junghoo Cho

Douglas S. Parker

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2015

Table of Contents

1. Introduction 1
2. Overview 2
3. Background 2
    3.1 Temporal RDF 2
    3.3 SPARQL^T 3
4. Wikipedia Infobox temporal RDF knowledge base 5
    4.1 Data Generating Process 5
    4.2 Statistics 9
5. Temporal Databases 10
    5.1 Data Insertion 10
    5.2 Index 10
    5.3 Query Evaluation 15
6. Experiments 23
7. Previous Work 26
8. Future Work 27
9. Conclusion 28
10. Acknowledgements 29

References 30

Appendix 31

1. Introduction

As there is a growing interest in today's large knowledge bases, the entity information in these knowledge bases are updated frequently as the real world evolves. Thus, the evolution history of information in these knowledge bases has been brought into people's concerns. What are the previous properties the entities had? When did the change occur? How to manage those historical information? How to query those historical information effectively and efficiently? These are the problems that are of great interest to users. Thus, the management of the historical information has become more and more crucial in the large scale knowledge bases.

Wikipedia, for example, as one of the most popular large scale knowledge base in the world, experiences a great amount of updates every day. The history of Infobox of an entity clearly describes how the entity's properties have been evolved. Thus, the Wikipedia Infobox history becomes an excellent dataset to study the management of historical information. The entity, along with the parameters and values in the Infobox fit into the RDF model. In this paper, we collect the evolution history of Wikipedia Infobox from Wikipedia history dump and store them in temporal RDF model. We first analyze the data generating process. Then we evaluate the efficiency of the queries with time dimension involved. We use different databases and data structures to store the temporal RDF triples, including MySQL, PostgreSQL, and Interval Tree. By comparing the temporal query time with different indices, MySQL B+ Tree, PostgreSQL B-Tree, and Interval Tree, we also evaluate how these indices can help optimizing the performance of temporal queries.

2. Overview

Previous work on temporal Wikipedia Infobox is very sparse. Most of the work focuses on current Infobox Information instead of Infobox history. No complete dataset of Wikipedia Infobox history knowledge base exists. However, the Wikipedia Infobox history is, in fact, very useful to study the evolution history of information. Thus, in this paper, we introduce our complete Wikipedia Infobox knowledge base built on Wikipedia Infobox history dump based on temporal RDF structure. In later sections, we will discuss the temporal RDF data model for the Wikipedia Infobox history, and will also present the data collecting and generating processes.

In their paper, Gao and Chen et al. [11] introduce $SPARQL^T$ which extends SPARQL with a point-based explicit time model that eliminated the need for time coalescing. They also introduce their user friendly interface[2] for queries. However, they speed up the query using in memory Multi-Version B+ Tree (MVBT) [12] indices only. This paper is based on Gao et al.'s research, but we further explore the performances of Postgres SQL B-Tree, MySQL B+Tree, and Interval Tree to support temporal RDF queries in more details. Specifically, we compare the creation time of these indices, how much storage to store the indices, and measure their query run time for eleven queries, including traditional and temporal selection, and temporal and hybrid join queries.

3. Background

3.1 Temporal RDF

The Resource Description Framework (RDF) is a metadata model from World Wide Web Consortium (W3C) that has become more and more popular today. RDF models the data into

triples *<s, p, o>,* where *s* is the subject, *p* is the predicate, and *o* is the object. The predicate refers to the property of a subject, and the object is the value that the property p has for that given subject. For example, the statement “Jennifer Lawrence’s profession is an actress” can be converted into a triple where the subject is “Jennifer Lawrence”, the predicate is “profession” and the object is “actress”.

As real-world information changes over the time, many values in RDF triples can have temporal annotations associated with them. Temporal RDF[6] is a RDF model with temporal context, which allows navigating data across time. Besides traditional RDF with *<s,p,o>*, temporal RDF triples contains the start time and the end time when the triple exists. The triple format for interval based temporal RDF is *<s,p,o>: st-ed*. For example, the following interval based temporal RDF triples can be used to record the population of San Diego between 12/19/2012 to 05/21/2015:

- *<San Diego, population, 1322553>: 12/19/2012 - 10/01/2013*
- *<San Diego, population, 1307402>: 10/02/2013 - 04/29/2014*
- *<San Diego, population, 1345895>: 04/30/2014 -05/21/2015*

3.3 SPARQL^T

SPARQL^T [11] is an extension of SPARQL based on a point-based explicit time model that supporting time coalescing. In SPARQL^T, temporal RDF is constructed by adding an additional temporal column to RDF, in *<s,p,o>:t* format. The basic granularity for the time in [2] is DAY. For example, the interval-based temporal RDF triple

<San Diego, population, 1307402>: 10/02/2013 - 04/29/2014

can be expressed as following:

<San Diego, population, 1307402>: 10/02/2013

<San Diego, population, 1307402>: 10/03/2013

…

<San Diego, population, 1307402>: 04/29/2014

In SPARQL^T, Gao et al. expresses the constraints on the RDF graph pattern as *?s ?p[?t] ?o*, where *?t* is the explicit time point binding for the triples. Since SPARQL^T uses a point-based explicit time model, it eliminates the need for temporal coalescing.

In[2], Gao et al. introduce their user-friendly query interface that solves Usability problem for knowledge base queries. They addressed this problem by extending the By-Example Structured Query(BEStQ)[14]. The interface they implemented supports (i) querying the current knowledge base and entities' histories and (ii) browsing the history of entities and properties. The interface enables queries by allowing users to enter conditions into Wikipedia pages with extended temporal fields. Figure 1 is an example for the query interface.

Figure 1. Query Interface

4. Wikipedia Infobox temporal RDF knowledge base

4.1 Data Generating Process

This section describes the process to generate the. The follow figure (Figure 1) shows the workflow for data generating. The pseudo codes for generating process are attached in Appendix.

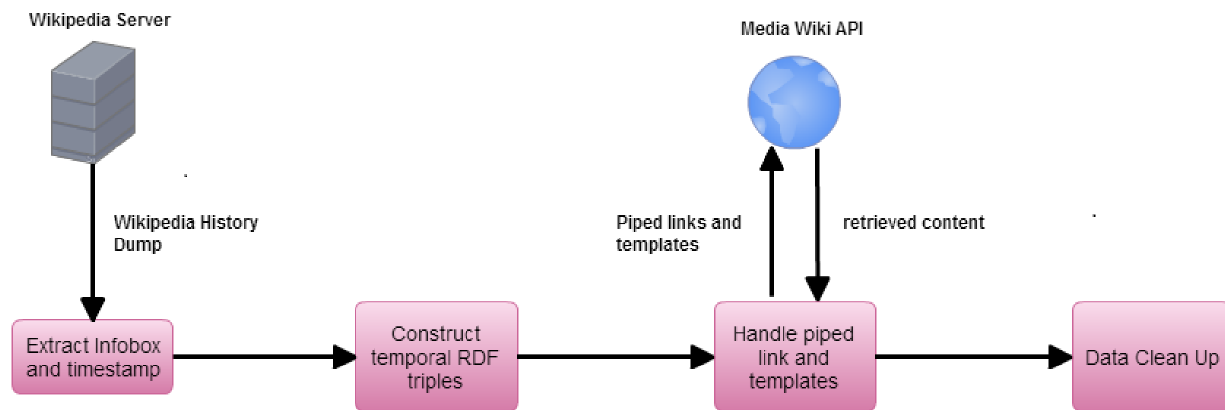


Figure 1. Data Generating Process Flowchart

4.1.1 Extract Infobox and timestamp

The Wikipedia Infobox temporal RDF triples are generated from the Wikipedia history dumps, which are in XML format. After downloading the history dump from the Wikipedia Server, we first scan through each of the entity in the dump, and extract the Infoboxes and their timestamps. Incomplete Infoboxes and Infoboxes with bad formats are dropped during processing. Figure 2 is a sample Infobox for Jennifer Lawrence in the Wikihistory dump.

```
{{Infobox person
| name        = Jennifer Lawrence
| image       = Jennifer Lawrence SDCC 2015 X-Men.jpg
| caption     = Lawrence at the 2015 [[San Diego Comic-Con International]]
| birth_name   = Jennifer Shrader Lawrence
| birth_date   = {{birth date and age|1990|8|15}}
| birth_place  = [[Louisville, Kentucky]], U.S.
|residence = [[Beverly Hills, California]], U.S.<ref>{{cite web|url=http://variety.com/2014/dirt/real-estalker/jennifer-lawrence-snags-celebrity-pedigreed-pad-in-beverly-hills-1201337732/|title=Jennifer Lawrence Snags Celebrity Pedigreed Pad in Beverly Hills|accessdate=April 25, 2015|date=October 23, 2014|work= [[Variety (magazine)|Variety]]}}</ref>
| occupation   = Actress
| years_active = 2006–present}}
```

Figure 2. Wikipedia Infobox Sample

4.1.2 Construct Temporal RDF Triples

Before we construct the triples, we first clean the data to remove unnecessary information. These information include reference to other articles, file links, image links, urls to other websites, and unnecessary formatting tags.

The Infobox starts with keyword “Infobox” and is followed by the template name for that entity. An Infobox contain several key - value pairs, where the key must match with the key specified in the Wkipedia template. When converting these key-value pairs into RDF triples, the entity name maps to the subject, the key maps to the predicate, and the value maps to the object. For each triples in the Infoboxes we extracted for a particular entity, we first check whether the key exists before. If the key is a new key, we initialize an entry for that key, and set its initial value with a timestamp starting with the timestamp for that Infobox and ending with a timestamp of ‘9999-12-31 23:59:59’. If the key exists, we then compare the value with the latest value for that key in the system. If the values match, then there is no need to update the value and

timestamps for this key. If they do not match, then we set the end time of the latest value to the timestamp of current Infobox and create a new value entry for that key with a starting timestamp equal to the current Infobox timestamp and an ending timestamp of ‘9999-12-31 23:59:59’.

4.1.3 Handle Piped Links and Templates

The next step is to retrieve the contents for piped links and templates in the triples. Piped links[15] in Wikipedia are the hyperlinks whose text displayed is different to the title of the page it links to. For example, [[train station|station]] links to a page with title “train station”, but the link displays as “station” on the page. Templates[16] in Wikipedia are texts in the content page that contains a reference to the target. For example, "{{birth date and age|1990|8|15}}” actually refers to the “birth date and age template” and is displayed as “(1990-08-15) August 15, 1990 (age 25)” on the webpage. We use MediaWiki API[7] to retrieve the real content that is displayed on the page for piped links and templates.

4.1.4 Data Clean Up

The last step is data cleaning. The data has to be cleaned up because most of the Wikipedia pages have been edited by normal users, resulting in a lot of noisy data. Final data cleaning includes removing consecutive duplicate triples, removing triples with inconsistent starting and ending timestamps, and fill in the gap between the consecutive triples with the same subject and key but are not continuous in their timestamps.

Another challenge that we faced during data clean-up is that when two users modifying the Infobox at the same time, the Wikidump alternatively records the two versions of Infobox repeatedly until one user finishes editing. This results in the case that the history of Infobox alternatively shows the two triples, and transaction time for each triple only lasts for a short time. Below is an example for this problem.

Table 1. Repeated Triples Sample

Subject	Key	Value	Starttime	Endtime
Hunnic language	fam1	possibly Altaic languages	2010-06-30 11:16:58	2010-11-26 13:22:56
Hunnic language	fam1	Altaic	2010-11-26 13:22:56	2010-11-26 16:09:31
Hunnic language	fam1	possibly Altaic languages	2010-11-26 16:09:31	2011-01-31 00:32:50
Hunnic language	fam1	Altaic	2011-01-31 00:32:50	2011-01-31 07:23:16
Hunnic language	fam1	possibly Altaic languages	2011-01-31 07:23:16	2011-01-31 10:50:40
Hunnic language	fam1	Altaic	2011-01-31 10:50:40	2011-01-31 10:53:13
Hunnic language	fam1	possibly Altaic languages	2011-01-31 10:53:13	2011-01-31 17:33:34

In order to solve this issue, we set a threshold for the occurrence of the repeated triples. If that pattern occurs more than 3 times consecutively, we pick the last triple as the valid triple and change its start time to the start time of the first triple where the repeated triples start.

Due to the large amount of data in Wikipedia history dump, in order to maximize CPU and memory utilization, we used parallel programming in the whole process. Table 2 is part of RDF triples extracted from Jennifer Lawrence's Infoboxes.

Table 2. Infobox Temporal RDF Triples Sample

Subject	Key	Value	Starttime	Endtime
Jennifer Lawrence	location	Louisville, Kentucky	2007-08-08 23:45:12	9999-12-31 12:59:59
Jennifer Lawrence	birth_name	Jennifer Shrader Lawrence	2012-09-20 02:08:00	2013-01-10 19:58:34
Jennifer Lawrence	birth_name		2013-01-10 19:58:34	2013-01-10 20:22:58
Jennifer Lawrence	birth_name	Jennifer Shrader Lawrence	2013-01-10 20:22:58	2013-10-20 10:48:47
Jennifer Lawrence	birth_name	Jennifer BrookeScott Lawrence	2013-10-20 10:48:47	2013-10-20 11:05:27
Jennifer Lawrence	birth_name	Jennifer Shrader Lawrence	2013-10-20 11:05:27	9999-12-31 12:59:59

Jennifer Lawrence	other name	Jenn	2011-05-05 01:43:20	2011-05-22 14:12:32
Jennifer Lawrence	other name	Jen	2011-05-22 14:12:32	2011-07-16 01:39:22
Jennifer Lawrence	other name		2011-07-16 01:39:22	2011-07-16 02:13:52
Jennifer Lawrence	other name	Jen	2011-07-16 02:13:52	9999-12-31 12:59:59
Jennifer Lawrence	home_town	Louisville, Kentucky	2012-09-20 02:08:00	2014-01-26 17:31:01
Jennifer Lawrence	home_town	Louisville, Kentucky, U.S.	2014-01-26 17:31:01	9999-12-31 12:59:59

4.2 Statistics

Wikipedia Infobox Temporal RDF Knowledge Base contains 170,941,613 temporal triples in total with a temporal range from 01/2004 to 08/2015. Figure 3 shows the number of updates of Wikipedia Infobox for each month between 01/2004 and 08/2015. From the chart, we can see that the number of update starts to grow significantly starting from 2006, and reaches its peak on 07/2011 with a maximum amount of 3,220,921.

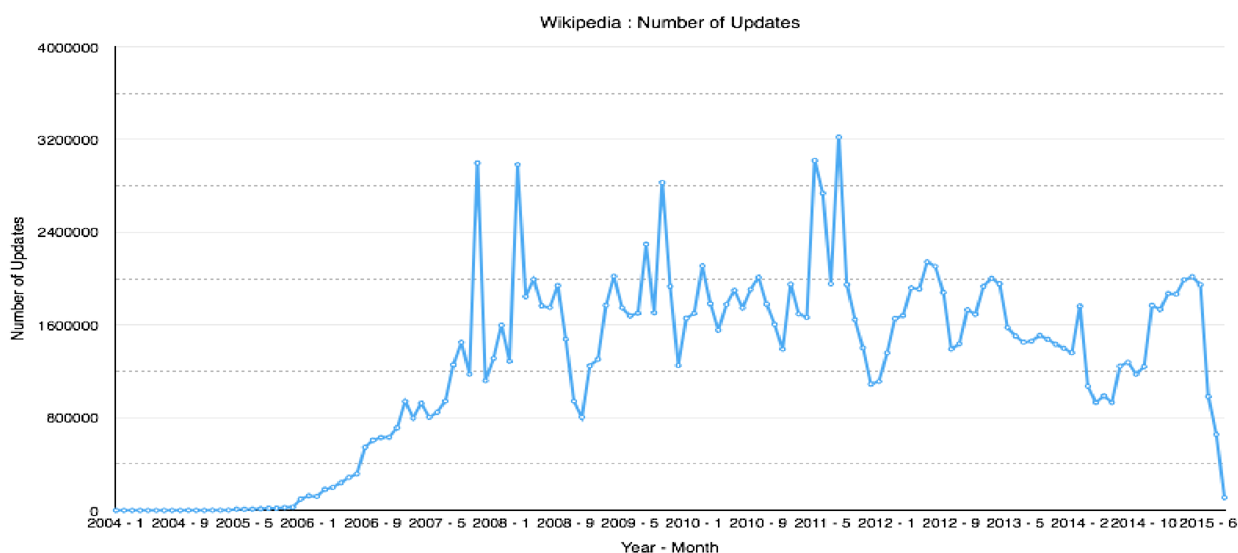


Figure 3. Wikihistory Dump Number of Updates per month

5. Temporal Databases

5.1 Data Insertion

Due to the memory limitation of the server, we randomly selected 29,966,700 triples from the Wikipedia Infobox Temporal RDF Knowledge Base as our test dataset. We used MySQL and PostgreSQL databases for query evaluation. The following statement is used to create table in both MySQL and PostgreSQL:

```
CREATE TABLE dataset (s VARCHAR(128), p VARCHAR(128), o VARCHAR(128), ts TIMESTAMP, te
TIMESTAMP);
```

s, p, and *o* refers to subject, predicate and object in RDF, and *ts* and *te* are the start time and end time for each triple. Table 3 is the data insertion time for MySQL and PostgreSQL with 29,966,700 triples. It shows that the insertion time for MySQL is about half of the insertion time of PostgreSQL.

Table 3. Insertion Time (PostgreSQL vs. MySQL)

	Insertion Time
MySQL	5m 54s 727ms
PostgreSQL	13m 13s 681ms

5.2 Index

5.2.1 B+ Tree and B-Tree

B+ Tree is the default index method in MySQL, and B-Tree is the default index method in PostgreSQL. The difference of B+ Tree and B-Tree is that the internal nodes in B+ Tree store key and pointers only, and leaf nodes store data, while B-Tree store keys and data in internal

nodes. The leaf nodes in B+ Tree are linked, so a full scan through the leaf nodes would be faster.

In PostgreSQL and MySQL, we created six indices on subject, predicate and object. They are *spo, pso, osp, ops, pos,* and *sop*. These indices are created for different combinations of conditions. Two more indices *ts* and *te* are created for the start and end timestamps individually. The index creation time for the two databases are shown in Table 4.

Table 4. Index Creation Time (PostgreSQL vs. MySQL)

Index	PostgreSQL	MySQL
spo	3m 51s 217ms	12m 8s 180ms
pso	4m 38s 652ms	11m 30s 250ms
osp	5m 39s 755ms	12m 21s 480ms
ops	7m 17s 637ms	12m 41s 900ms
pos	5m 11s 339ms	13m 28s 540ms
sop	4m 52s 842ms	14m 21s 800ms
ts	5m 25s 471ms	5m 16s 410ms
te	1m 25s 981ms	3m 31s 310ms

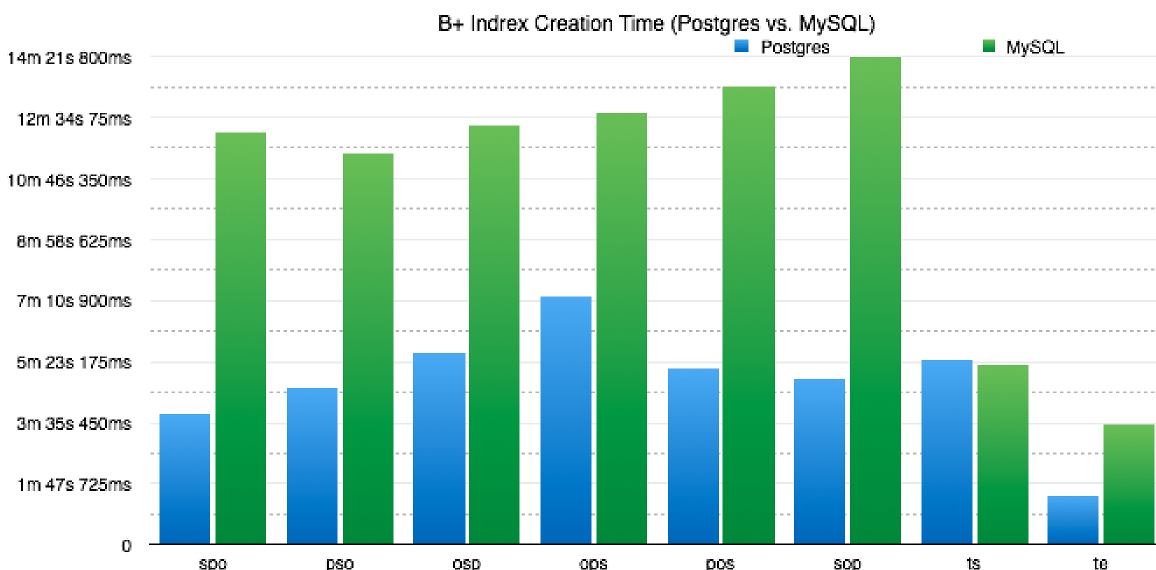


Figure 4. Index Creation Time Comparison (PostgreSQL vs. MySQL)

Figure 4 is a comparison chart for the two databases according to Table 4. Overall, PostgreSQL creates indices much faster than MySQL, except for *ts*, which takes about the same time to create the index as MySQL. In addition, the creation time for B+ index on multiple attributes in MySQL is much slower than index on single attribute. Table 5 below is are disk storages the two databases use. Apparently, PostgreSQL uses more storage that MySQL in total. The storages to store all the triples are very similar, but the B- tree index size for PostgreSQL is around 1.3GB larger than MySQL.

Table 5. Disk Storages (PostgreSQL vs. MySQL)

	Total Table Size with Indices (GB)	Table Size (GB)	Index Size (GB)
MySQL	16	2.939	13.71
PostgreSQL	18	3.055	15

5.2.2 Interval Tree

Interval Tree is a binary tree structure that holds intervals. It optimizes the query that finds the intervals that overlaps with any given point or intervals. Its leaf nodes contain the elementary intervals. Each of its internal node stores:

- a center point as its key, that separates the intervals in its left and right subtrees (usually median value is used).
- a pointer pointing to its left subtree, which stores all the intervals that are smaller than the center point.
- a pointer pointing to its right subtree, which stores all the intervals that are larger than the center point.
- a sorted list of intervals that overlap with the center point.

To construct an Interval Tree, we start from taking in the whole range of all the interval and get the median value of that interval. A node is created for the median value.We sort all the intervals before the median value to the left set of the node, and all the intervals after the median value to the right set of the node. The left and right set are divided in the same manner. All the intervals that overlap with the median value is stored in a sorted list belong to the node.

The interval tree is constructed based on the time intervals of the triples. In order to support RDF, leaf nodes in the interval tree is modified so that they can store (subject, predicate, object) pairs. Each internal node contains the median value of the time interval as its key. The left subtree of an internal node contains all the time intervals before the key of the current node, and its right subtree contains all the time intervals after the key of the current node.

Since interval tree stores all the triples in its leaf nodes and interval indices in its internal nodes, memory usage increases significantly as the number of triples inserted increases. Due to the memory limitation, our server can only support an interval tree with maximum number of triples of 29,966,700 in memory, and triples are inserted iteratively instead of recursively to avoid stack overflow issue. Figure 5 and 6 are the construction time of interval tree and memory used as the number of triples increases.

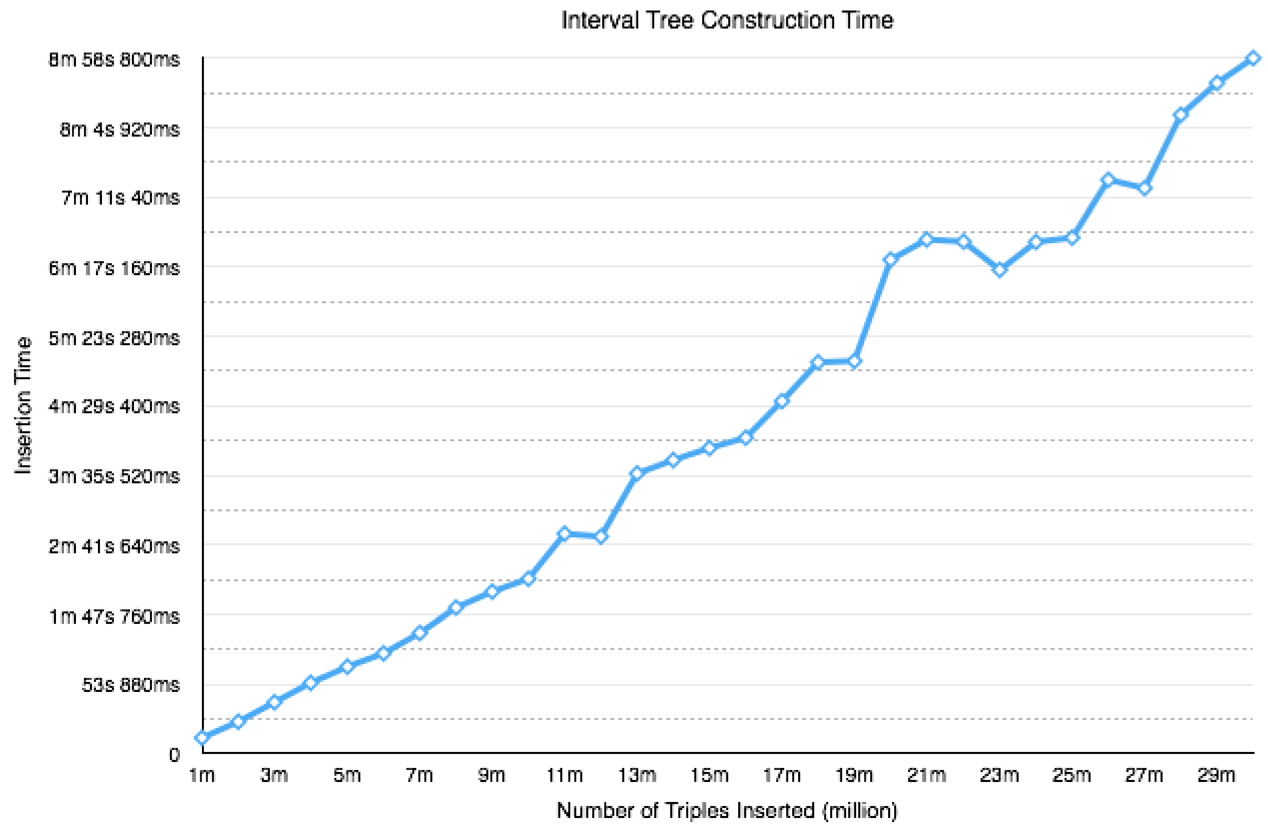


Figure 5. Interval Tree Construction Time

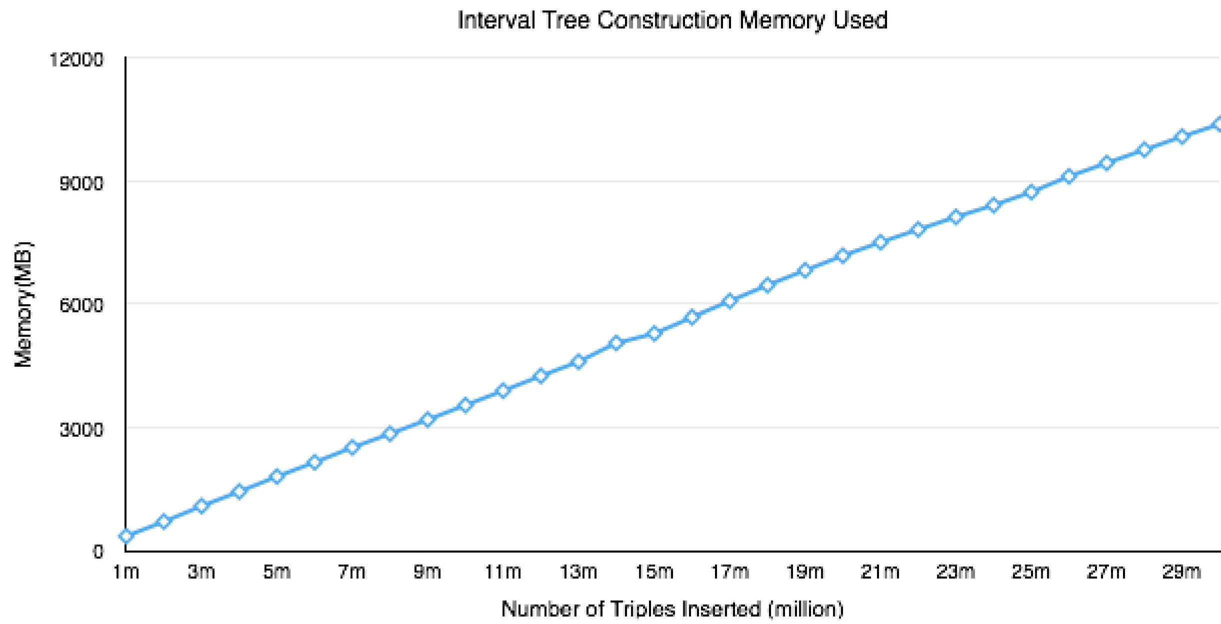


Figure 6. Interval Tree Memory

To construct an interval tree with 29,966,700 triples, it takes about 10.387 GB in memory, including data and indices. From Figure 6, we can see that the size of the interval trees almost grows linearly, since the interval tree contains exactly one leaf node for each triple. Comparing with the B+ Tree with PostgreSQL and MySQL, Interval Tree takes fewer storage in memory. However, the time to build index is much slower than the other two structures.

5.3 Query Evaluation

Queries on Wikipedia Infobox temporal RDF knowledge base includes filtering of time interval, subject, predicate and object. We classify the queries on the Wikipedia Infobox temporal RDF knowledge base into four scenarios: traditional selection, which only compares non-temporal fields (*subject, predicate, object*); temporal selection, which filters the temporal fields (*starttime, endtime*); temporal join, which joins the tables on the temporal fields; and hybrid join, which joins on both non-temporal and temporal fields. In this section, we listed the SQL query implementation for PostgreSQL and MySQL. Queries in Interval Tree are implemented in Java.

Scenario 1. Traditional Selection

Traditional selection matches the attributes with specific value. In RDF model, traditional selection only consider the value condition of subject, predicate and object. For example, ***Q1*** returns the results that match the conditions of subject and predicate.

Q1. Find the total number of students at University of California, San Diego during each period
PostgreSQL & MySQL
*select distinct * from dataset*
where subject='University of California, San Diego'

Interval Tree (Pseudo)
This query is completed with B+ Tree.

Scenario 2. Temporal Selection

Temporal selection queries are the queries whose predicates involve temporal conditions. To satisfy temporal selection query, the results have to meet the value conditions if exist and the temporal conditions. The temporal conditions usually fall into two categories:

1. Point overlapping. Check whether the time interval overlaps with a point in time. In our paper, the point granularity is DAY. (***Q2,Q3***)

2. Time interval overlapping. Check whether the time interval overlaps with another time interval. (***Q4, Q5***)

We design the Interval Tree to first retrieve all the triples that meet the temporal conditions since Interval Tree only stores intervals, and scan through the result set to find the triples that meet the non-temporal conditions. For example, in ***Q2***, the Interval Tree first retrieve all the triples that overlap with 2011-12-31, and scan through those triples to find if their subject is 'Japan' and predicate is 'GDP_PPP'.

However, this can be a problem for Interval Tree because the size of the result set is unknown. For a given time interval, the triples that overlaps that interval might be very large. In that case, it might cost more time to filter the temporal condition first than to filter the non-temporal conditions with B+ Tree first. For instance, in ***Q2***, triples whose subject is 'Japan' and key is 'GDP_PPP' might be a smaller set than triples that overlap with 2011-12-31, which means that using B+ Tree would be faster than Interval Tree.

Overlap In Time Point

Q2.Find the total GDP of Japan on 2011-12-31

PostgreSQL & MySQL

```
select distinct * from dataset
where subject='Japan' and key='GDP_PPP'
and starttime<=timestamp'2011-12-31' and endtime>='2011-12-31';
```

Interval Tree(Pseudo)

```
Input: (I) where I is the Interval Tree
Initialize Resultset, Finalresult that stores triples.
Resultset=query ("2011-12-31 00:00:00") in I
For each triple t in Resultset
        If t.subject="Japan" and t.predicate="GDP_PPP"
                Add t to Finalresult
Return Finalresult
```

Q3. Who are the key people at Bloomberg L.P. on 2010-10-01

PostgreSQL & MySQL

```
select distinct value from dataset
where subject='Bloomberg L.P.' and key='key_people'
and starttime<=timestamp'2010-10-01 00:00:00' and endtime>=timestamp'2010-10-01
00:00:00';
```

Interval Tree(Pseudo)

```
Input: (I) where I is the Interval Tree
Initialize Resultset, Finalresult that stores triples.
Resultset=query ("2010-10-01 00:00:00") in I
For each triple t in Resultset
        If t.subject="Bloomberg L.P."  and t.predicate="key_people"
                Add t to Finalresult
Return Finalresult
```

Overlap in Time Interval

Q4. Find companies that are owned by Time Warner before 2009

PostgreSQL & MySQL

```
select distinct subject from dataset
where key='owner' and value='Time Warner'
and starttime<=timestamp'2009-01-01 00:00:00';
```

Interval Tree(Pseudo)

```
Input: (I) where I is the Interval Tree
Initialize Resultset, Finalresult that stores triples.
Resultset=query interval (“2004-01-01 00:00:00”, “2009-01-01 00:00:00”) in I
For each triple t in Resultset
        If t.predicate=”owner”  and t.value=”Time Warner”
                Add t to Finalresult
Return Finalresult
```

Q5. Find the governor of Missouri between 2010-01-01 and 2010-05-01

PostgreSQL & MySQL

```
select distinct * from dataset
where subject='Missouri' and key='Governor'
and (starttime, endtime) overlaps
(timestamp'2010-01-01 00:00:00', timestamp'2010-05-01 00:00:00');
```

Interval Tree(Pseudo)

```
Input: (I) where I is the Interval Tree
Initialize Resultset, Finalresult that stores triples.
Resultset=query interval (“2010-01-01 00:00:00”, “2010-05-01 00:00:00”) in I
For each triple t in Resultset
        If t.subject='Missouri' and t.predicate='Governor'
                Add t to Finalresult
Return Finalresult
```

Scenario 3. Temporal Join

A temporal join query performs join on both value and temporal attribute. In addition to the value condition in traditional query, time dimension is added so that the two tables have to match in value and have to overlap in time. For example, following conditions exist in ***Q6:***

1. Value conditions in Table A's predicate, value, and Table B's predicate.
2. Table A's subject matches B's subject.
3. Table A's time interval overlaps B's time interval and overlaps 2007.

Q6. Find the players who were at Los Angeles Lakers during 2012 and their positions when they were at Los Angeles Lakers.

PostgreSQL

```
Select distinct A.subject from dataset as A, dataset as B
where (A.starttime, A.endtime) OVERLAPS (B.starttime, B.endtime)
and (B.starttime, B.endtime) OVERLAPS
(timestamp'2010-01-01 00:00:00', timestamp'2010-12-31 23:59:59')
and A.subject=B.subject and B.key='team' and B.value='Los Angeles Lakers' and A.key='position';
```

MySQL

```
Select distinct A.s from dataset as A, dataset as B
where ((A.st between B.st and B.ed) or (A.ed between B.st and B.ed)
or (B.st between A.st and A.ed))
and ((B.st between timestamp'2010-01-01 00:00:00' and timestamp'2010-12-31 23:59:59')
or (B.ed between timestamp'2010-01-01 00:00:00' and timestamp'2010-12-31 23:59:59')
or( timestamp'2010-01-01 00:00:00' between B.st and B.ed))
and A.s=B.s and B.p='team' and B.o='Los Angeles Lakers' and A.p='position';
```

IntervalTree(pseudo)

```
Input: (I) where I is the Interval Tree
Initialize Resultset1, Resultset2, Finalresult that stores triples
Resultset1 = query interval ('2010-01-01 00:00:00', '2010-12-31 23:59:59')  in I
For each triple t in Resultset1
        If t.predicate='team' and t.value='Los Angeles Lakers'
                Let st=t.starttime, ed=t.endtime, subject=t.subject
                Resultset2= query interval (st, ed)
                For each triple r in Resultset2
                        If r.subject=subject and r.predicate='position'
                                Add r to Finalresult
Return Finalresult
```

Q7. Find team name and the owner of the team Brandon Jennings belongs to when he is at that team

PostgreSQL

```
Select distinct B.subject, B.value from dataset as A , dataset as B
where A.subject='Brandon Jennings' and A.key='team'
and A.value=B.subject and B.key='owner'
and (A.starttime, A.endtime) OVERLAPS (B.starttime, B.endtime);
```

MySQL

```
Select distinct B.s, B.o from dataset as A , dataset as B
where A.s='Brandon Jennings' and A.p='team' and A.o=B.s and B.p='owner'
and ((A.st between B.st and B.ed) or (A.ed between B.st and B.ed)
or (B.st between A.st and A.ed))
```

IntervalTree(pseudo)

```
Input: (I, SP) where I is the Interval Tree, SP is the B+ Tree built on (subject, predicate)
Initialize Resultset1, Resultset2, Finalresult that stores triples
Resultset1 = query (subject="Brandon Jennings" and predicate="team") in SP
For each triple t in Resultset1
        Let teamname=t.value, st=t.starttime and ed=t.endtime
        Resultset2= query interval(st, ed) in I
        For each triple r in Resultset2
                If r.subject=teamname and r.predicate=owner
                        Add r to Finalresult
Return Finalresult
```

Q8. Find population in Los Angeles when Eric Garcetti is the Mayor of Los Angeles

PostgreSQL

```
Select distinct A.value from dataset as A, dataset as B
where A.subject='Los Angeles' and B.subject=A.subject
and B.key='leader_name' and B.value='Eric Garcetti' and A.key='population_total'
and (A.starttime, A.endtime) OVERLAPS (B.starttime, B.endtime);
```

MySQL

```
Select distinct A.o from dataset as A, dataset as B
where A.s='Los Angeles' and B.s=A.s and B.p='leader_name'
and B.o='Eric Garcetti' and A.p='population_total'
and ((A.st between B.st and B.ed) or (A.ed between B.st and B.ed)
or (B.st between A.st and A.ed))
```

IntervalTree(pseudo)

```
Input: (I, PV) where I is the Interval Tree, PV is the B+ Tree built on (predicate, value)
```

```
Initialize Resultset1, Resultset2, Finalresult that stores triples
Resultset1 = query (predicate="leader_name" and value=" Eric Garcetti") in PV
For each triple t in Resultset1
        Let st=t.starttime and ed=t.endtime
        Resultset2= query interval(st, ed) in I
        For each triple r in Resultset2
                If r.subject="Los Angeles" and r.predicate="population"
                        Add r to Finalresult
Return Finalresult
```

Scenario 4. Hybrid Join

The results for hybrid join has to match its value conditions and temporal attribute binds to a specific time point or time range. The two tables have to be joined by their attribute value conditions and their time intervals have to overlap with specific time ranges. For example, following conditions exist in ***Q9***:

1. Table A's subject matches Table B's subject.
2. Value conditions in Table A's predicate, value, and Table B's predicate, value.
3. Time condition for Table B's start time.

This kind of join queries are completed by first finding the triples within the query time range using Interval Tree and the triples that meet the value conditions using B+ Tree, and then get the intersection set of these triple results. For instance, in order to find resultset for ***Q9***, we first use Interval Tree to filter out all the records that are before 2012 with key equals to 'location_city' and 'value' equals to 'New York'. Then we use B+ Tree to find all the companies that have financial service as its industry. Finally, we take the intersection of these two result sets.

Q9. Find financial services companies whose headquarters are in New York before 2012.

PostgreSQL

```
Select distinct A.subject from dataset as A, dataset as B
```

```
where A.key='industry' and A.value like '%Financial services%'
and B.key='location_city' and B.value like '%New York%'
and A.subject=B.subject and B.starttime<timestamp'2012-01-01 00:00:00';
```

MySQL

```
Select distinct A.s from dataset as A, dataset as B
where A.p='industry' and A.o like '%Financial services%'
and B.p='location_city' and B.o like '%New York%' and A.s=B.s
and B.st<timestamp'2012-01-01 00:00:00';
```

IntervalTree(pseudo)

```
Input: (I, PV) where I is the Interval Tree, PV is the B+ Tree built on (predicate, value)
Initialize Resultset1, Resultset2, Tempset, Finalresult that stores triples
Resultset1 = query interval (“2004-01-01 00:00:00”, “2012-01-01 00:00:00) in I
For each triple t in Resultset1
        If t.predicate=”location_city” and t.value contains “New York”
                Add t to Tempset
Resultset2= query (predicate=”industry”, value=”Financial services”) in PV
For each triple r in Resultset2
        For each triple t in Tempset
                If r.subject=t.subject
                        Add r to Finalresult
Return Finalresult
```

Q10. Find distributor of the film directed by Peter Jackson before 2010

PostgreSQL

```
Select distinct A.subject, A.value from dataset as A, dataset as B
where A.subject=B.subject and B.key='director' and B.value='Peter Jackson'
and A.key='distributor' and B.starttime<timestamp'2010-01-01 00:00:00';
```

MySQL

```
Select distinct A.s, A.o from dataset as A, dataset as B where A.s=B.s and B.p='director' and
B.o='Peter Jackson' and A.p='distributor' and B.st<timestamp'2010-01-01 00:00:00';
```

IntervalTree(pseudo)

```
Input: (I, SP) where I is the Interval Tree, SP is the B+ Tree built on (subject, predicate)
Initialize Resultset1, Resultset2, Finalresult that stores triples
Resultset1 = query interval (“2004-01-01 00:00:00”, “2010-01-01 00:00:00) in I
For each triple t in Resultset1
        If t.value=”Peter Jackson” and t.predicate=”director”
        Let film=t.subject
        Resultset2= query (subject=film, predicate=”distributor”) in SP
        For each triple r in Resultset2
                Add r to Finalresult
```

```
Return Finalresult
```

Q11. Find all albums produced by a rock singer released in 2013

PostgreSQL

```
Select distinct A.subject, B.subject from dataset as A, dataset as B
where B.key='genre' and B.value like '%rock%' and A.value=B.subject
and A.key='Artist' and (A.starttime, A.endtime)
OVERLAPS(timestamp'2013-01-01 00:00:00', timestamp'2013-12-31 23:59:59');
```

MySQL

```
Select distinct A.s, B.s from dataset as A, dataset as B
where B.p='genre' and B.o like '%rock%' and A.o=B.s and A.p='Artist'
and ((B.st between timestamp'2013-01-01 00:00:00' and timestamp'2013-12-31 23:59:59')
or (B.ed between timestamp'2013-01-01 00:00:00' and timestamp'2013-12-31 23:59:59')
or( timestamp'2013-01-01 00:00:00' between B.st and B.ed))
```

IntervalTree(pseudo)

```
Input: (I, PV) where I is the Interval Tree, PV is the B+ Tree built on (predicate, value)
Initialize Resultset1, Resultset2, Tempset, Finalresult that stores triples
Resultset1 = query interval ("2013-01-01 00:00:00", "2013-12-31 23:59:59) in I
For each triple t in Resultset1
        If t.predicate="Artist"
                Add t to Tempset
Resultset2= query (predicate="genre", value contains "rock") in PV
For each triple r in Resultset2
        For each triple t in Tempset
                If r.subject=t.subject
                        Add r to Finalset
Return Finalresult
```

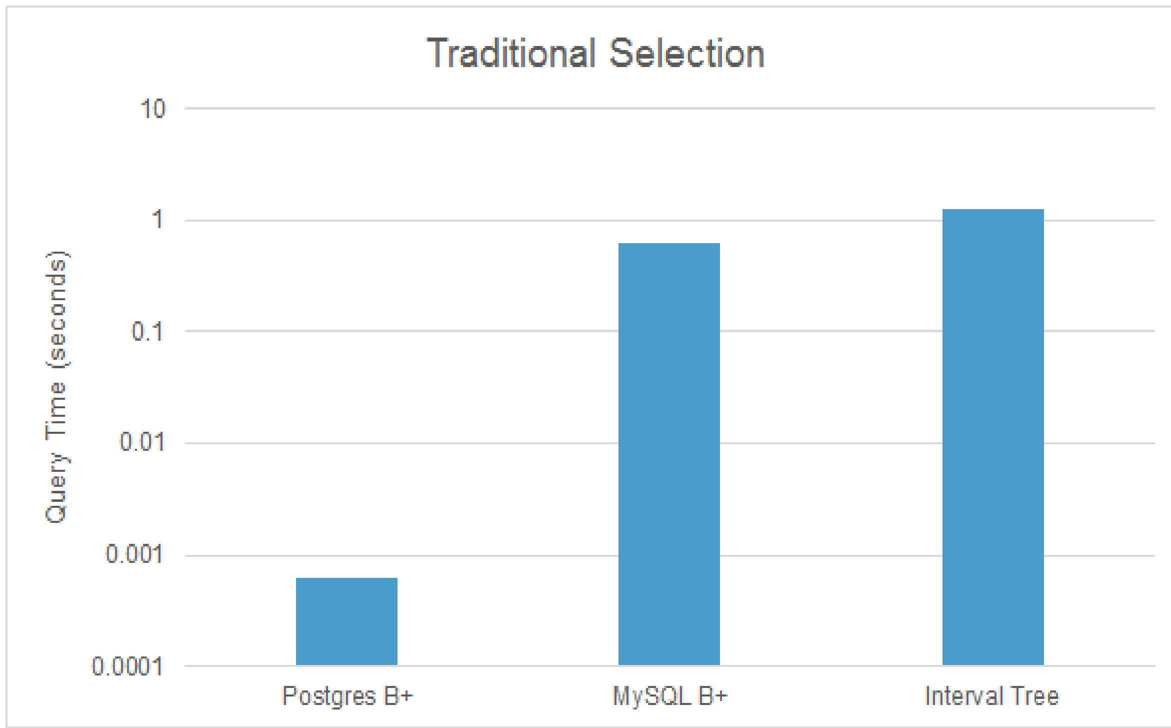
---

6. Experiments

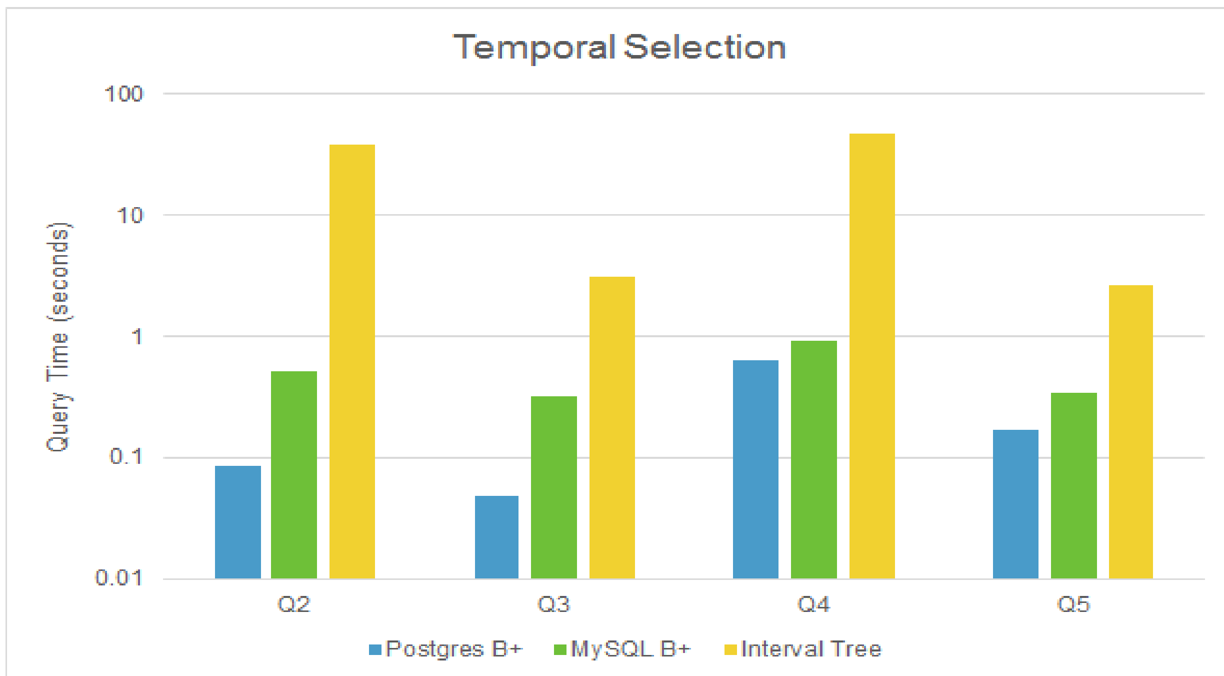
We executed queries ***Q1-11*** on PostgreSQL, MySQL and Interval Tree. Since Interval Tree only stores interval values, for queries that do not involve any time interval, an exhaustive search has to be performed, which is very time-consuming. In order to support traditional selection queries, we built a B+ tree on *s,p,* and *o* attributes before building Interval Trees to speed up queries that involve comparing other non-temporal fields.

Below are the runtime results for each scenario.

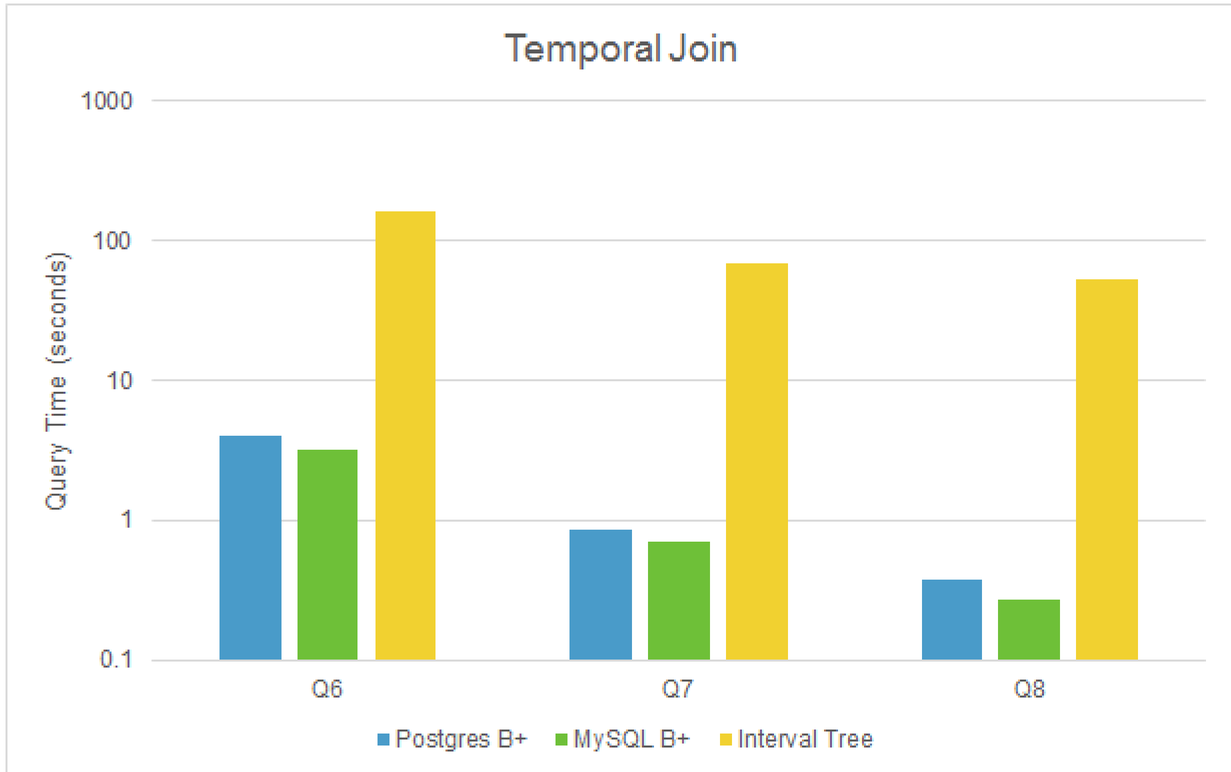
Scenario 1. Traditional Selection (Q1)



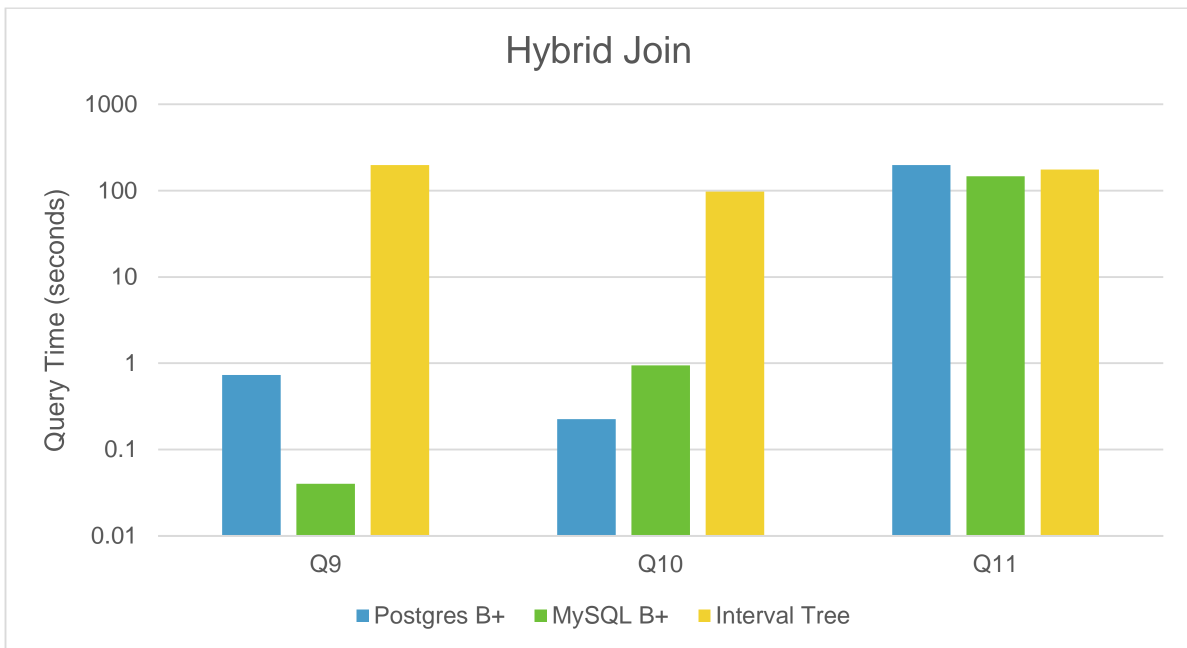
Scenario 2. Temporal Selection (Q2-Q5)



Scenario 3. Temporal Join



Scenario 4. Hybrid Join



Overall, Interval Tree is significantly slower than the other two indices. As explained earlier, for queries that have both temporal and non-temporal conditions, Interval Tree might be very inefficient if the result set for the triples that overlap with the temporal conditions are very large.

In addition, since Interval Tree are built in memory, the index has already taken a lot of memory. The memory limitation also affects the query time of Interval Tree. In addition, the implementation of temporal queries for Interval Tree is a lot more complicated than temporal queries for PostgreSQL and MySQL.

Apparently, for selection queries, PostgreSQL is faster than MySQL. But for join queries, the query time for PostgreSQl and MySQL are very similar, and sometimes, in fact, MySQL is slightly faster than PostgreSQL.

7. Previous Work

Previous work on temporal RDF data model and queries are very sparse. In [6], Gutierrez et al. presents the temporal RDF graphs, which incorporate temporal labels into RDF. It gives a very robust temporal RDF model. There are several approaches that have been proposed to support temporal RDF queries. Pugliese and Udrea et al. [1] propose tGRIN, which is a specialized balanced tree built on the temporal RDF triples based on undirected RDF graph distance and temporal distances. They use traditional databases but use extended indices to support temporal queries. [3] and [4] introduce t-SPARQL and SPARQL-st, which store temporal RDF in RDBMS and convert the temporal queries into SPARQL. These two approaches use explicit time model and interval operators to construct simple temporal selection

and join queries. However, most of these approaches either are not able to support complex queries, or have performance issues.

8. Future Work

There are some challenges we met that may need future works.

- While parsing the Wikipedia history dump, due to the large size of the dump, reading dump was a very significant bottleneck that slows down the performance. Even though multi-thread was implemented, the limit of threads was not able to improve the performance significantly.
- While parsing Infoboxes, we found that some of the keys in Wikipedia History dump are not the same as what are shown on the Wikipedia website, even though they refer to the same thing. For example, the key “mayor” on Wikipedia website shows as “leader_name” in the Wikidump. This is because Wikidump is in XML format, and Wikipedia has an internal method (expand template) to convert the key to the presented key on the website. Thus, we will need to develop a method to convert these keys to their presented key on the website to avoid inconsistency of triples in the knowledge base with the website contents.
- As mentioned earlier, the noise of Wikihistory dump is a very large issue during data processing. Many of the Infoboxes are incomplete, and the formats are very random. This makes it hard to determine the start and end pattern of the Infoboxes and the pattern of the triples. Triples that are repeated alternatively are also a big problem for data clean-up.

We need to a better way to eliminate the Infoboxes with bad format, and further work is required to more accurate data clean-up to improve the data quality.

- Since we used parallel programming to speed up Wikidump processing, the limitation of the memory makes it very hard to process the large volume of data. Processing all the Wikihistory dump took around 3 weeks. If we can expand the memory, it would be a lot easier to process all the Wikihistory dump and will save more time.
- Due to the memory limitation, we are not able to build Interval Tree on the whole Infobox knowledge base. If we can expand memory, in memory Interval Tree would be more efficient and we would be able to test Interval Tree on the whole Wikipedia Infobox history dataset.

9. Conclusion

In this paper, we have introduced the Wikipedia Infobox temporal knowledge base built on Wikipedia history dump using temporal RDF model. The knowledge base contains 170,941,613 temporal triples. We then selected around 30M triples for query testing. We created indices including PostgreSQL B-Tree, MySQL B+Tree, and in-memory Interval Tree. Overall, PostgeSQL had the fastest index creation time, and Interval Tree was the slowest. MySQL took the fewest storage to store the B+ Tree index. Four kinds of queries were used to measure the query run time with the three indices: traditional selection, temporal selection, temporal join, and hybrid join. For traditional and temporal selection, PostgreSQL B- Tree showed the best performance, while Interval Tree was the slowest among all. However, for temporal join and

hybrid join, the performances for PostgreSQL B-Tree and MySQL B+Tree were very similar, where PostgreSQL B- Tree was sometimes a little bit slower than MySQL B+Tree.

Future work such as improving the data quality in the knowledge base and increase the memory for Interval Tree can improve the performance of the queries.

10. Acknowledgements

I would like to thank Prof. Zaniolo and Mr. Gao, PHD at UCLA, for guiding me through the thesis, providing me environments for experiments, solving my questions and helping me complete the thesis.

References

[1]Pugliese, Andrea, Octavian Udrea, and V. S. Subrahmanian. "Scaling RDF with time." *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008.

[2]Gao, Chen, Atzori, Gu, and Zaniolo. “SPARQLT and its User-Friendly Interface for Querying the History of RDF Knowledge Bases.” *ISWC 2015, Bethlehem, Pennsylvania.* 2015.

[3] Grandi, F. T-sparql: a tsql2-like temporal query language for rdf. In International Workshop on on Querying Graph Structured Data (2010), 21–30.

[4] Perry, M., Sheth, A. P., Hakimpour, F., and Jain, P. Support- ing complex thematic, spatial and temporal queries over semantic web data. In Proceedings of the 2nd International Conference on GeoSpatial Semantics (2007), pp. 228–246.

[5] Tappolet, J., and Bernstein, A. Applied temporal rdf: Efficient temporal querying of rdf data with sparql. In ESWC (2009), pp. 308– 322.

[6] C. Gutierrez, C. A. Hurtado, et al. Introducing time into rdf. TKDE, 19(2):207–218, 2007.

[7]MediaWiki API (https://www.mediawiki.org/wiki/API:Main_page).

[8]Wikipedia History Dump (http://dumps.wikimedia.org/enwiki/latest/).

[9]Interval Tree (https://en.wikipedia.org/wiki/Interval_tree)

[10] Tappolet, Jonas, and Abraham Bernstein. "Applied temporal rdf: Efficient temporal querying of rdf data with sparql." *The Semantic Web: Research and Applications*. Springer Berlin Heidelberg, 2009. 308-322.

[11] Gao and Zaniolo. “SPARQLT : Querying Temporal Knowledge Base with Ease”

[12] B. Becker, S. Gschwind, et al. An asymptotically optimal multiversion b-tree. VLDB, 5(4):264–275, Dec. 1996.

[13] RDF (https://en.wikipedia.org/wiki/Resource_Description_Framework)

[14] M. Atzori and C. Zaniolo. Swipe: Searching wikipedia by example. In WWW, pages 309–312, 2012.

[15] Wikipedia: Piped link (https://en.wikipedia.org/wiki/Wikipedia:Piped_link)

[16] Wikipedia:Template (https://en.wikipedia.org/wiki/Help:Template)

Appendix

Wikiparser

Wikiparser parse the wikihistory dump and extract the Infoboxes. It then parse each line in the Infobox into (key,value,time interval) and construct triple for that entity.

* Below are very high level pseudocodes

```
Function Main()

Input: Wikidump file
Output: parsed file

Initiate entitycontent as String
While not reaching end of file
      Readline l
      If l contains entity end tag
            Append l to entitycontent
            Create new thread for entitycontent
            Empty entitycontent
      Else:
            Append l to entitycontent
```

```
Function thread.run()

Input: entitycontent, sbjset (hashset that store triples and time values for
that entity)

Initiate enterInfobox=false
For each line in entitycontent
      /*Infoboxes in bad formats are removed in this process*/
      If line contains <title>
            title= extract entity title from line
      If line contains <timestamp>
            timestamp= extract timestamp from line
      If line contains <Infobox>
            enterInfobox=true
      If enterInfobox is true
            Call attrparser(sbjset, line, timestamp)
Write sbjset to outputfile
```

```
Function attrparser()

Input: sbjset (hashset that store triples and time values), line, timestamp
```

```
Parse line into key and value:
      key= Content between “|” and “=”
      value= Content after “=” is
If sbjset does not contain key
      starttime=timestamp
      endtime=’9999-12-31 12:59:59’
      Insert triple (key, value, starttime, endtime)
Else
      Retrieve the last triple t inserted for that key
      If t.value is not the same as value
            starttime=timestamp
            endtime=’9999-12-31 12:59:59’
            Update t.endtime=starttime
            Insert triple (key, value, starttime, endtime)
```

Wikicleaner

Wikicleaner send requests to MediaWiki API to parse the templates and pipelines in the triples.

```
Function main()

Input: parsed file
Output: final file

Open HTTP connection
While not reaching end of file
      Readline l
      Send “parse text” request to MediaWiki API along with l.
      /*This parses the templates and pipelink in the triples*/
      Retrieve response from MediaWiki API in JSON format
      Write cleaned line to final file.
```