TARGET-ORIENTED KEYWORD SEARCH OVER TEMPORAL RELATIONAL DATABASES

JIA XIANYAN

NATIONAL UNIVERSITY OF SINGAPORE

2016

TARGET-ORIENTED KEYWORD SEARCH OVER TEMPORAL RELATIONAL DATABASES

Jia Xianyan

(*B.Sc, Sichuan University*)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2016

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Jia Xianyan

7th July 2016

Acknowledgment

I would like to express my sincere gratitude to my supervisors Professor Wynne Hsu and Professor Mong Li Lee, for their guidance and encouragement. The discussion with them is inspiring and fruitful. I benefit a lot in how to think, write and present, which is an invaluable asset to me.

I would like to thank the seniors Furong, Chonggang, Zhao Gang, Shiling, Xiangnan and Shangxuan, for the fun and unforgettable time with them. I would also like to thank my friends Kaiping, Jiakun, Jae, Abha, Zhongle, Zhaojing and Wentian, for their friendship and help.

In the end, I am deeply grateful to my parents and my sister Xianjie, for their unconditional love and support. Dad is such a wise and responsible man that I could always learn from. Mom spends a lot of time with me and is always thoughtful. Xianjie is the one that I can talk to regardless of whether I am happy or sad. Without them, I would never have gained so much today.

Contents

Abstract **vii**

List of Tables **ix**

List of Figures **xi**

1 Introduction **1**

1.1 Contribution 5

1.2 Thesis Organization 6

2 Related Work **7**

2.1 Keyword Search over Relational Database 7

2.1.1 Schema based Keyword Search 7

2.1.2 Data Graph based Keyword Search 9

2.2 Keyword Search over XML Database 11

2.3 Identify Query Search Target 13

3 Preliminaries **15**

3.1 Temporal Keyword Query 15

3.2 Answer to Temporal Keyword Query 18

3.3 Temporal Ranking Model 21

4 Proposed Solution **23**

4.1 Temporal Index for Keywords Associated with Time 24

4.2 Target-oriented Search 26

4.3 Time-aware Pruning 27
4.4 Algorithms 31

5 Performance Study 37

5.1 Experiments on Queries without Time Constraints 41
5.2 Experiments on Queries with Time Constraints 43
5.3 Experiments on Scalability 45
5.4 Experiments on Search Quality 49
5.5 Case Study 51

6 Conclusion and Future Work 57

Bibliography 58

Abstract

Temporal database is prevalent in many applications such as finance, business, bank, and health care. With a series of historical records, people are interested in finding information in a certain time period or that satisfies some temporal relationships. On the other hand, keyword search in relational databases has gained popularity due to its ease of use. Instead of writing complicated *SQL*s, people can issue queries with a few keywords. However, none of the existing works have considered time associated keywords in the query, which is important and useful.

In this thesis, we extend keyword queries to allow temporal information to be associated with keywords, as well as support temporal relationships between two keywords. We design a target-oriented search algorithm to evaluate such queries. We incorporate overlapping interval partitioning into the keyword inverted lists to filter nodes that do not satisfy the time constraints. We also augment selected nodes in the data graph with time boundaries to enable time-aware pruning during the search process. Experiments on 3 datasets demonstrate the efficiency of the proposed approach to answering complex keyword queries over temporal relational databases.

List of Tables

3.1 Syntax of temporal keyword query in BNF 16

3.2 Temporal keyword queries for *Clinic* database 17

4.1 Computation of valid range 31

5.1 Dataset schema and the number of tuples for each relation . 38

5.2 Temporal keywords queries for *Employees* dataset 39

5.3 Temporal keywords queries for *ACMDL* dataset 40

5.4 Datasets generated for scalability experiments 46

5.5 *MAP* with different number of answers 50

List of Figures

1.1 Example *Clinic* database 4

3.1 Data graph for the example database in Fig. 1.1 19

3.2 Candidate answers for query `{Patient: fever, cough}` . 20

3.3 Possible answers for query `{Patient: male, fever, cough}` 20

4.1 Inverted index for keyword `Pastia` in *Clinic* database (Fig. 1.1) 25

4.2 Augment inverted index in Fig. 4.1 by relation 25

4.3 *OIP* index for keyword `Pastia` in *Clinic* database (Fig. 1.1) 26

4.4 Schema graph of the *Clinic* database in Fig. 1.1 27

4.5 Augmented data graph with time boundary 30

4.6 Construction of a partial answer tree for query C_4 32

5.1 Runtime for queries without time constraints 42

5.2 Runtime for queries involving time constraints 44

5.3 Runtime for queries with different dataset sizes 47

5.4 Runtime for queries with different time interval length . . . 48

5.5 First three answers returned to query C_{10} by *ATQ* 52

5.6 First three answers returned to query C_{10} by *Bidirectional* . 53

5.7 First three answers returned to query C_{11} by *ATQ* 54

5.8 First three answers returned to query C_{11} by *Bidirectional* . 55

Chapter 1

Introduction

Temporal data is prevalent in many applications such as finance, business, bank, and health care. This has led to the need to support querying of temporal data. Initial efforts has focused on extending structured query language to temporal databases [25, 18]. However, structured query language is not suitable for non-expert or casual users for several reasons: First, structured query language is difficult for them to learn and use especially when the query is complex. Second, users need to understand the database schema when issuing a query, which is not easy when the schema structure is complicated or if the number of attributes is large.

The success of web search engine such as *Google*[1] and *Baidu*[2] has shown that keyword search is intuitive and highly acceptable by common users. Motivated by this, keyword search over relational databases [1, 15, 10, 7, 13] has been extensively studied to provide a simple and user-friendly interface to access relational databases without having to write complicated *SQL* queries. However, existing relational keyword search techniques assume that keywords are not associated to time constraints and there is no relationship among keywords in the queries.

We illustrate this with an example. Fig. 1.1 shows a relational database

[1] www.google.com
[2] www.baidu.com

with two snapshot relations (`Patient` and `Doctor`) and two temporal relations (`Visit` and `Symptom`). The `Visit` relation records the date at which a patient sees a doctor, while the `Symptom` relation gives the start and end dates where a patient experiences various symptoms. For example, the first two tuples (id s_1 and s_2) in the `Symptom` relation depict that a patient p_1 complained of `cough` and `headache` in the same consultation visit. These two different symptoms occurred over different periods of time. On the other hand, the tuples with id s_{32} and s_{33} show that the same patient p_3 visited the doctor on different occasions for his `cough`.

If a user wants to find *patients who have cough on 1 January 2015* in this database, s/he can issue a keyword query such as `{Patient, cough, 01/01/2015}`. However, this query will return additional answers such as patient p_2 who is born on 1 January 2015 but has `cough` on 10 January 2015. In order to retrieve answers that match the user's intention, we need to associate the time information to the appropriate keywords. Here, we use square brackets to indicate this association. Hence, the query `{Patient, cough[01/01/2015]}` refers to the patients who have cough on 1 January 2015 while the query `{Patient[01/01/2015], cough}` refers to the patients who are born on 1 January 2015 and have cough at some point in time.

We further extend the time information to support queries with intervals. For example, the query `{Patient, fever[01/01/2015-01/31/2015]}` will return patient p_1 who has fever in the month of January 2015. Besides associating a keyword with time information, we also support queries with temporal relationships between keywords. The work in [2] identified 13 temporal relationships between two time intervals including `OVERLAP`, `BEFORE` which form the set of reserved words in our temporal keyword queries. For example, query `{Patient, fever BEFORE cough}` will return patient p_1 who has fever before cough.

We have seen the need for temporal keyword queries where the keywords are associated with time constraints and temporal relationships may exist among the keywords. Next, we need to be able to answer such temporal keyword queries efficiently. A closer look at techniques for answering normal keyword queries over relational databases shows that there are two main approaches: schema graph approach [26, 22, 29, 15, 1, 14], and data graph approach [7, 16, 13, 10, 20, 11]. In the schema graph approach, the database schema is modeled as a directed graph where each node is a relation and edges are key-foreign key reference between two relations. The answer to a query is a minimum total joining network of tuples. In the data graph approach, the database is modeled as a graph where nodes represent tuples and edges represents key-foreign key. Given a data graph G_D and a query consists of a set of keywords, the problem is to find a set of sub-graphs of G_D where each sub-graph contains all keywords in the query. One naive way to answer temporal keyword queries is to apply these existing keyword search techniques to obtain an initial set of answers, and then filter out those answers that do not satisfy the time constraints. However, this approach will lead to the generation of a huge set of candidate answers of which many are wasted as they eventually do not satisfy the time constraints or the temporal relationships.

Patient

pid	YOB	Gender	Name	Ethnicity
p_1	02/03/1982	F	Anna	Indian
p_2	01/01/2015	M	Andy	Chinese
p_3	09/01/1986	M	John	Eurasian

Doctor

did	Name	Gender
d_1	Ben	M
d_2	Anna	F
d_3	Pastia	M

Visit

vid	pid	did	date
v_1	p_1	d_1	05/01/2015
v_2	p_1	d_1	12/01/2015
v_3	p_1	d_1	25/01/2015
v_4	p_1	d_2	02/02/2015
v_5	p_1	d_2	26/04/2015

vid	pid	did	date
v_6	p_1	d_2	20/04/2015
v_7	p_2	d_2	10/01/2015
v_8	p_2	d_3	24/01/2015
v_9	p_3	d_3	26/04/2015
v_{10}	p_3	d_3	16/04/2015

Symptom

sid	vid	Name	start	end
s_1	v_1	cough	01/01/2015	04/01/2015
s_2	v_1	headache	02/01/2015	05/01/2015
s_3	v_1	pastia	01/01/2015	03/01/2015
s_4	v_1	dizzy	02/01/2015	04/01/2015
s_5	v_2	cough	05/01/2015	07/01/2015
s_6	v_2	headache	06/01/2015	12/01/2015
s_7	v_2	pastia	04/01/2015	05/01/2015
s_8	v_3	fever	20/01/2015	21/01/2015
s_9	v_3	headache	20/01/2015	25/01/2015
s_{10}	v_3	pastia	20/01/2015	24/01/2015
s_{11}	v_4	headache	26/01/2015	29/01/2015
s_{12}	v_4	cough	25/01/2015	01/02/2015
s_{13}	v_4	flu	27/01/2015	02/02/2015
s_{14}	v_4	pastia	19/01/2015	25/01/2015
s_{15}	v_5	fever	09/04/2015	16/04/2015
s_{16}	v_5	dizzy	12/04/2015	25/04/2015
s_{17}	v_5	pastia	09/04/2015	15/04/2015

sid	vid	Name	start	end
s_{18}	v_5	headache	09/04/2015	18/04/2015
s_{19}	v_6	fever	08/04/2015	16/04/2015
s_{20}	v_6	dizzy	13/04/2015	18/04/2015
s_{21}	v_6	pastia	12/04/2015	16/04/2015
s_{22}	v_6	cough	17/04/2015	20/04/2015
s_{23}	v_6	headache	13/04/2015	16/04/2015
s_{24}	v_7	fever	03/01/2015	10/01/2015
s_{25}	v_7	headache	02/01/2015	06/01/2015
s_{26}	v_7	pastia	01/01/2015	05/01/2015
s_{27}	v_8	cough	10/01/2015	15/01/2015
s_{28}	v_8	pastia	04/01/2015	24/01/2015
s_{29}	v_9	dizzy	05/04/2015	15/04/2015
s_{30}	v_9	fever	08/04/2015	13/04/2015
s_{31}	v_9	pastia	07/04/2015	14/04/2015
s_{32}	v_9	cough	10/04/2015	26/04/2015
s_{33}	v_{10}	cough	14/04/2015	16/04/2015
s_{34}	v_{10}	pastia	12/04/2015	16/04/2015

Fig. 1.1. Example *Clinic* database

1.1 Contribution

In this thesis, we propose a general framework to support keyword search over temporal relational databases. Specifically, our contribution can be summarized as follows:

1. We address the problem of keyword search in temporal relational databases by providing support for complex queries with temporal relationships between keywords.

2. We introduce time-associated keywords and pre-defined temporal relationships in queries, and design a target-oriented search algorithm to evaluate such queries.

3. We augment selected nodes in the data graph with time boundaries to enable time-aware pruning during the search process. We also incorporate overlapping interval partitioning into the keyword inverted lists to filter nodes that do not satisfy the time constraints.

4. Experiment results on 3 datasets demonstrate that the proposed approach is efficient and effective in pruning invalid answers early.

To the best of our knowledge, this is the first attempt to support keyword search over temporal relational databases.

1.2 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 reviews the related works, including keyword search in relational databases and XML databases, as well as works on query search targets.

- Chapter 3 gives the preliminaries of this thesis, including the definition of temporal keyword query, the answer to the temporal keyword query, and the temporal ranking model.

- Chapter 4 shows our proposed solution to answer temporal keyword queries. We first present a temporal index used to retrieve matching nodes for keyword associated with time. Next we show our target oriented search strategy, and introduce the time aware pruning to fasten the search process. Then we integrate the above methods and propose ATQ algorithm to answer keyword queries over temporal relational databases.

- Chapter 5 presents the results of our experiments. We design queries for three datasets and show the efficiency and effectiveness of our algorithm.

- Chapter 6 shows our conclusion as well as directions for future work.

Chapter 2

Related Work

In this chapter, we review the previous works related to this thesis. First, we survey the existing keyword search technologies over relational databases in Section 2.1, including schema based approach and data graph based approach. Next we include some related works about XML keyword search in Section 2.2. Then we present works for identifying query search target in Section 2.3.

2.1 Keyword Search over Relational Database

Keyword search over relational databases allows users to issue simple keyword queries without having to write complicated SQLs. Existing works on keyword search over relational databases can be classified into schema graph approach and data graph approach [28].

2.1.1 Schema based Keyword Search

In the schema graph approach, the database schema is modeled as a directed graph where each node is a relation and the edges are key-foreign key reference between two relations. The answer to a query is a minimum total joining network of tuples ($MTJNT$).

DBXplorer [1] first uses a *symbol table* to identify the relations, attributes and rows that contain each keyword. Then they enumerate all possible join trees that can cover all the query keywords. For each join tree, they generate a *SQL* statement to retrieve the answers. Each answer is presented as one row (either from one relation, or by joining multiple relations) such that the row contains all the query keywords.

DISCOVER [15] generates a set of candidate networks by performing a breadth-first traversal over the schema graph and limits the number of joins in the query. To improve query efficiency, they propose an optimal execution plan by reusing the shared common components among candidate networks, i.e. common join structures among *SQL* statements.

Works in [14] and *SPARK* [22] focus on finding top-k answers since it is ineffective and inefficient to return large number of answers. [14] proposes algorithms for applicable use in different conditions. The *Sparse* algorithm avoids evaluating candidate networks that can not contribute to top-k answers. *Global-Pipelined* algorithm first get top-k $MTJNT$s for each candidate network, and then combine them together to get the final results. Each time it selects candidate network that will maximize the score. *Sparse* performs best when there are relatively small number of results, while *Global-Pipelined* has best performance with large number of answers. A hybrid algorithm is proposed by first estimating the answer size and then choosing which algorithm to use. The *SPARK* [22] proposes a ranking function by extending existing IR techniques by modeling the joined tree as a virtual document. They takes both AND or OR semantics into consideration. They first finds a set of candidate networks, then SQL statements are generated from the top-k networks.

2.1.2 Data Graph based Keyword Search

In the data graph approach, the database is modeled as a graph where nodes represent tuples and edges represents key-foreign key. Given a data graph G_D and a query consists of a set of keywords, the problem is to find a set of minimal sub-graphs (Steiner tree) of G_D where each sub-graph contains all keywords in the query.

Banks [7] uses backward expansion search algorithm to find Steiner trees that contain all the keywords. It models the database as a directed data graph. For each keyword k_i, a set of matching nodes S_i containing k_i are retrieved by using an inverted list index. Note that the entry of the inverted list is keyword and the posting list is a list of keys that denote nodes. They union matching nodes of each keyword into a big set S, i.e. $S = \bigcup_i S_i$. Then $|S|$ copies of *Dijkstra*'s algorithm runs concurrently in reverse direction to find the shortest path. If the iterator for keyword node u reaches a node v, then the shortest path from v to u has been found. If there exists node that lies on all the shortest paths of keyword nodes in each set S_i, then an answer containing all keyword is returned. However, *Banks* [7] is not efficient if some keywords have a lot of matching nodes or the iterator reaches node with large number of incoming edges.

Bidirectional [16] overcomes the limitations of *Banks* [7] with bidirectional search technique. The main idea is to perform both forward and backward search to improve search efficiency. A spreading activation is proposed to prioritize the search. There are two main iterators namely *incoming iterator* and *outgoing iterator*. The *incoming iterator* is similar to backward search iterator in *Banks* except that it merges iterators for each keyword matching node into one. The *outgoing iterator* starts from the nodes that have been explored by *incoming iterator* and follows the outgoing edges to forward search some keyword nodes. They use a *spreading activation* mechanism to decide the next iterator to be called and the next node to be visited. Matching nodes for each keyword are added to the *incoming iterator*, and the initial activation score a_{u,k_i} for each node u on keyword k_i is computed as follows:

$$a_{u,k_i} = \frac{nodePrestige(u)}{|S_i|}, \forall u \in S_i \tag{2.1}$$

where S_i is the set of matching nodes for keyword k_i, and the nodePrestige(u) is the node score that can be computed by algorithms such as pagerank. An attenuation factor μ is used when spreading the activation score: for each node u with activation score a_u, it spreads a fraction of score $\mu * a_u$ to its neighbors. $\mu * a_u$ is divided equally and distributed to each neighbor node. Suppose the neighbors number is N, then for each neighbor node v, v received $\frac{\mu * a_u}{N}$ from node u. Node u remains the activation score $(1-\mu) * a_u$. Node with the highest activation will be explored first.

Blinks [13] uses a bi-level index to speed up *Bidirectional* [16] search process. They first partition graph into blocks, then build *intra-block index* for each block and *block index* across blocks. The *intra-block index* keeps the shortest distance information from each node to each keyword node within blocks, and the *block index* keeps the information at block level. To answer the query, *Blinks* [13] first retrieve blocks that contain each keyword, then for each matching node, the *intra-block index* is used to check whether this node can reach all the keywords. If the node is a portal node among blocks, then the *block index* is used to expand these blocks to find reachable keyword nodes.

DPBF [10] employs a dynamic programming technique to identify the top-k answers. The primitive state is a single node tree with cost 0 and keyword set p. There are two basic components in the search process: First, *Tree grow*: given a tree $T(v, p)$ rooted at v, and let u be the neighbor node of v, if the growing tree $T(v, p) \oplus (v, u)$ has smaller cost than $T(u, p)$, then $T(u, p)$ is updated to the growing tree. Second, *Tree merge*: If there are two trees $T(v, p_1)$ and $T(v, p_2)$ rooted at the same node v with different keyword sets, trees are merged if the cost of merged tree $T(v, p_1) \oplus T(v, p_2)$ is smaller than the total cost of two trees.

2.2 Keyword Search over XML Database

In this section, we review some related works about keyword search over XML database. We will only discuss a few classical XML keyword search works, since our focus in this thesis is relational database. We also present several works [23, 8] that have explored the keyword search problem over temporal XML.

XML is modeled as rooted and labeled tree, where each internal node is element node and each leaf node is value node. Each element node is assigned a unique *Dewey ID*. *Dewey ID* for node u is concatenated with the IDs in the path from root node to u, separated by dots. There are some differences between tree model of *XML* and data graph model of relational database: First, all nodes in data graph are value nodes. Second, XML tree uses *Dewey ID*s to label the data nodes, while data graph of relational database usually uses primary key to label the data nodes.

Answering keyword queries in XML trees is different from that in relational data graph. For the former, they use *Dewey ID*s to compute the answers because *Dewey ID*s contain nodes position information in the XML trees. For the latter, graph is commonly needed as it contains node connection information.

Xrank [12] proposes a DIL algorithm to answer *XML* keyword queries. A data structure *Dewey Inverted List* is designed to keep the *Dewey ID* lists for each keyword, and the *Dewey ID* lists are sorted by *Dewey ID*s. Given a query, *Xrank* [12] merge the *Dewey ID* lists for each keyword in sorted order. Then it reads each node in order and compute the longest common prefix of of *Dewey ID*s for different query keywords. This process is equivalent to finding the lowest common ancestors (*LCA*s) of keyword matching nodes.

However, finding all *LCA*s is expensive since as the number of keywords and number of keyword matching nodes increases, the number of combinations is huge. *XKSearch* [27] optimizes the search efficiency by only considering part of the matching lists. *XKSearch* [27] starts with the keyword that has the smallest matching list size. For each matching node u, only the left match and right match of u is considered in constructing the answers. The left (right) match v is the nearest node in u's left (right) side and contain some other keywords. In this way, the number of combinations to be computed is largely reduced.

[23] is the first work on temporal XML keyword search. The temporal query is composed of three components, namely, non-temporal operand, temporal operand and temporal operator. E.g., in the query `{president after 2000}`, `president` is non-temporal operand, `after` is temporal operator and `2000` is temporal operand. Time information is stored as XML nodes. An index called *ClosestTemporalNode* is created to determine the closest temporal node given a node. To answer a temporal query, [23] first separate the query into two parts: non-temporal keywords and temporal predicates (temporal operator and temporal operand). The non-temporal keywords are sent to conventional XML keyword search engine to get the candidate answers. Then, for keyword nodes in answers, the closest temporal nodes are got by looking up *ClosestTemporalNode* index. The temporal node with the shortest distance is checked with temporal predicate, and only satisfied answers are returned. There are some limitations in this work: First, separating temporal predicates from non-temporal keywords in the search process may result in wrong interpretation as time is independent from keywords. Second, it is not efficient to use a post processing

to filter invalid answers especially when the time constraint is strict.

In [8], the temporal query is defined as a set of keywords attached with time, e.g., `{Anna, Peter, 2000}`. To answer the temporal query, first a set of candidate answers are got by conventional XML keyword search engine, then the answers are ranked by a time-aware ranking function. The ranking function considers both keyword similarity and temporal similarity. For the temporal similarity, they first compute the temporal similarity between each answer node time o_t and query time constraint q_t with scoring function in Equ 2.2. The overall similarity score is the sum of similarity scores for all nodes in the answer.

$$score_t = \begin{cases} \frac{|q_t \bigcap o_t|}{|q_t| \times |o_t|} & if\ q_t \bigcap o_t \neq \emptyset \\ \epsilon & otherwise \end{cases} \tag{2.2}$$

This temporal similarity function assumes that every node in the answers should have similar time constraint as the query time constraint. This assumption may not be true in general.

2.3 Identify Query Search Target

Query search target is the key part of the query, which indicates users' search intention in mind.

XReal [3] specifies the search target of *XML* keyword queries. They propose three guidelines for inferring a search target node with type T. First, search target node should be relevant to each keyword in the query, i.e., there exists some nodes in its subtree that can cover the query keywords. Second, search target node should contain enough relevant information. In other words, search target node should be at a higher level of XML tree. Third, search target nodes should not be near the root node that contain overwhelming information. However, these rules are limited to the *XML* hierarchical structure and cannot be extended easily to relational database keyword queries.

Expressq [29] specifies the search target of relational database keyword queries. They regard node whose keyword matches relation name or attribute name as search target node and uses it as the output object of the query. Othere works [19, 17, 4] allow users to indicate the query intention interactively. *NaLIR* [19] allows users to issue complex queries using natural language. The query is parsed to query trees and multiple interpretation of query trees are presented to users for verification. Once the query interpretation is verified, *SQL* statements are generated to get the answers. Similarly, *MeanKS* [17] and *ClearMap* [4] also allows the user to specify their interests and search target through a user interface and disambiguate the query interactively.

Our focus in this thesis is to solve temporal keyword query issues, so we allow the users to indicate the query search target at the head of query.

Chapter 3

Preliminaries

In this chapter, we first give the syntax of temporal keyword query and show a variety of example queries based on this query grammar. Then we define answers to the temporal keyword query and present the temporal ranking model.

3.1 Temporal Keyword Query

Existing keyword queries do not include time constraints in keywords, so in this section, we extend keyword queries to allow time associated keywords as well as temporal relationships between two keywords.

Temporal databases are known to support two time dimensions: the transaction time and the valid time [24]. Here, we focus on the *valid time* where the attribute value holds. The temporal attributes such as "date", "start" are predefined and we assume that the system is aware of these attributes.

We represent a temporal keyword query as $\{head : body\}$ where

1. *head* is a set of keywords indicating the search target. The search target is the user's search intention when issuing a query. Here, we give users the option to explicitly indicate his search target in the head of the query. If the user does not specify any search target, we

would use existing methods to identify them [29, 6, 5], and rewrite the query into the above temporal keyword query format where *head* is the search targets identified.

2. *body* is a set of keywords indicating the query condition. Some of these keywords may be constrained by time intervals, and the user may specify temporal relationships among the keywords.

Table 3.1 gives the syntax of temporal keyword query in Backus-Naur Form (BNF). Based on the grammar, we can formulate a variety of temporal keywords queries as shown in Table 3.2. Queries C_1 to C_4 are similar to standard keyword queries, except that the search target is explicitly specified at the head of the query to facilitate the efficient retrieval of relevant answers. Queries C_5 to C_{11} involve time information and temporal relationships between keywords which are not handled by existing keyword queries.

Table 3.1
Syntax of temporal keyword query in BNF

<table>
<tr><td><query></td><td>::=</td><td>{ <head> : <body> }</td></tr>
<tr><td><head></td><td>::=</td><td>ϵ | <search_list></td></tr>
<tr><td><search_list></td><td>::=</td><td><relation> | <value> | <relation>,<search_list> |
<value>,<search_list></td></tr>
<tr><td><body></td><td>::=</td><td><cond> | <cond>, <body></td></tr>
<tr><td><cond></td><td>::=</td><td><term> | <term> <temporal_relation> <term></td></tr>
<tr><td><term></td><td>::=</td><td><keyword> | <time_associated_keyword></td></tr>
<tr><td><keyword></td><td>::=</td><td><relation> | <value></td></tr>
<tr><td><time_associated
_keyword></td><td>::=</td><td>keyword [<time>] | keyword [<time> ,
<time>]</td></tr>
<tr><td><temporal_relation></td><td>::=</td><td>BEFORE | AFTER | EQUAL |
MEET | MET BY | START | STARTED BY |
OVERLAP | OVERLAPED BY | CONTAIN |
DURING| FINISH | FINISHED BY</td></tr>
</table>

Table 3.2
Temporal keyword queries for *Clinic* database

Query		**Meaning**
C_1	`{Patient : fever }`	Find patients who have fever
C_2	`{Patient : fever, cough}`	Find patients who have fever and cough
C_3	`{Patient, male : fever, cough}`	Find male patients who have fever and cough
C_4	`{Doctor, Patient : fever, cough }`	Find doctors and patients pairs with fever and cough
C_5	`{Patient : fever BEFORE cough }`	Find patients who have fever before cough
C_6	`{Patient : fever[1/1/2015, 31/1/2015], cough[1/1/2015, 31/1/2015] }`	Find patients who have fever and cough in January 2015
C_7	`{Patient : fever[1/1/2015, 31/1/2015] BEFORE cough[1/1/2015, 31/1/2015] }`	Find patients who have fever before cough in January 2015
C_8	`{Doctor, Patient : Visit[1/1/2015, 31/1/2015] }`	Find doctors and patients pairs with consultation visits in January 2015
C_9	`{Doctor, Patient : Visit[1/1/2015, 31/1/2015], fever[1/1/2015, 31/1/2015]}`	Find doctors and patients pairs with consultation visits for fever in January 2015
C_{10}	`{Patient : fever[1/1/2010, 1/1/2015] OVERLAP headache}`	Find patients with fever and headache, fever from 2010 to 2015, and fever overlap headache
C_{11}	`{Patient: fever[1/1/2010, 1/1/2015] OVERLAP cough, headache BEFORE fever[1/1/2000, 1/1/2015]}`	Find patients who have fever, headache and cough, with fever from 2010 to 2015, fever overlap cough, headache before fever

3.2 Answer to Temporal Keyword Query

An *answer* to a temporal keyword query Q over a data graph G (Fig. 3.1 shows the undirected data graph G of our example database in Fig. 1.1) is a minimal subgraph which contains nodes that match all the keywords in Q.

Fig. 3.2 shows the possible answers to the query C_2 which finds patients who have fever and cough. Nodes that match the keywords in the query body are highlighted and patients p_1, p_2, and p_3 are retrieved.

Note that the placement of a keyword in the query head or query body may lead to different answers. For example, Fig. 3.3 shows the possible answers to the query {`Patient: male, fever, cough`} which include male patients who have fever and cough (Fig. 3.3(a), Fig. 3.3(b) and Fig. 3.3(c)) as well as female patients who have seen male doctors for fever and cough (Fig. 3.3(d) and Fig. 3.3(e)). However, if the keyword "`male`" is in the head of the query as in query C_3, the answers will consist of only Fig. 3.3(a), Fig. 3.3(b) and Fig. 3.3(c). This allows user to clearly indicate his search intention.

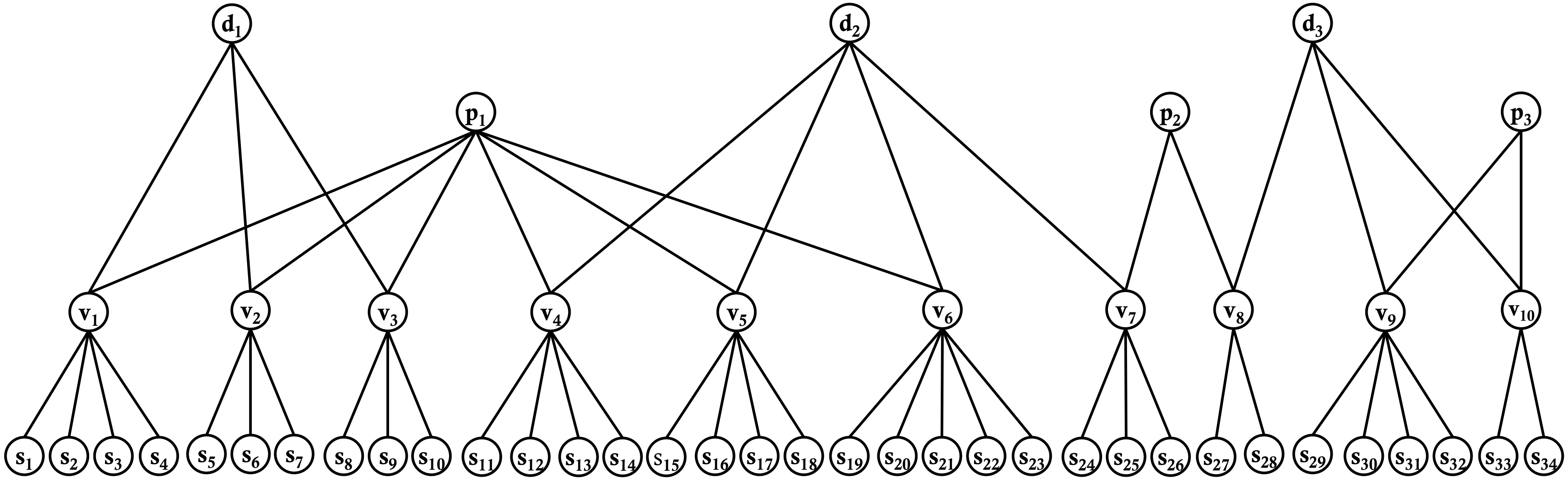


Fig. 3.1. Data graph for the example database in Fig. 1.1

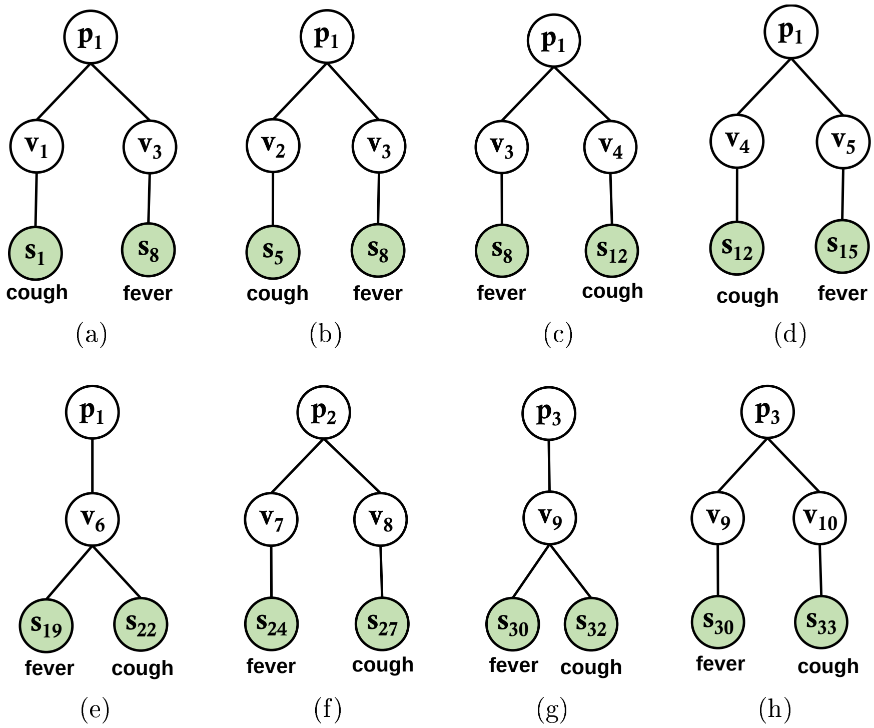


Fig. 3.2. Candidate answers for query {`Patient: fever, cough`}

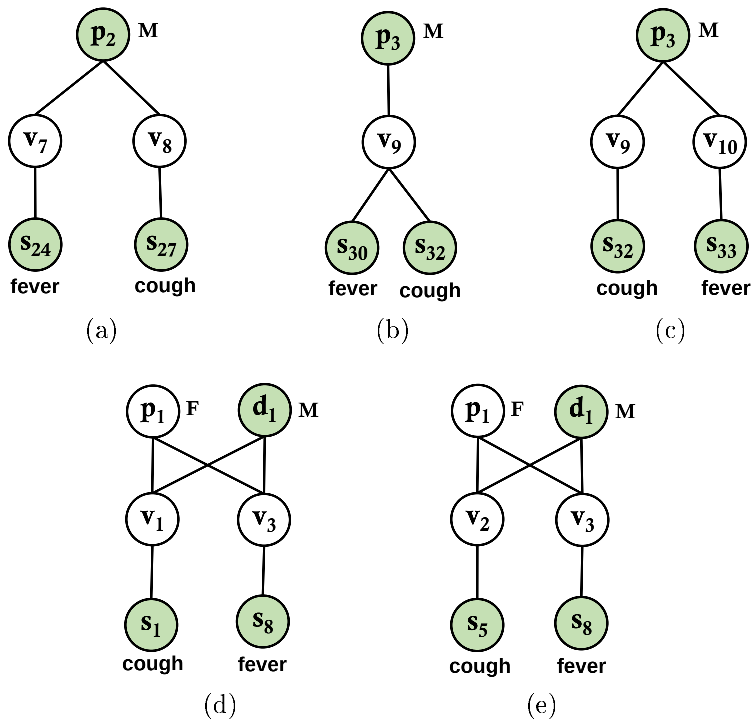


Fig. 3.3. Possible answers for query {`Patient: male, fever, cough`}

3.3 Temporal Ranking Model

In existing works [7, 16, 10] without considering time constraints, answers with smaller size are ranked higher. However, this is not enough if the query is associated with time. As usual, users tend to be more interested in recent answers. So in this thesis, we consider both answer structure and temporal freshness into the ranking function.

We use the structure scoring function defined in *Expressq* [29], as shown in Equ 3.1. The idea behind is that the matching nodes should be closely connected to the search target nodes, and smaller answers that have fewer nodes are preferred. Given an answer a, the structure scoring function $scoreS$ considers two factors: First, the distances from keyword nodes S to search target nodes ST. Note that $dist(st, s)$ is defined as the number of edges between node st and s. Second, the answer size N, i.e. total number of nodes in the answer.

$$scoreS(a) = \frac{|ST| * |S|}{N * \sum_{st \in ST} \sum_{s \in S} dist(st, s)} \tag{3.1}$$

When considering answer freshness, we adopt the exponential decay function $scoreT$(Equ 3.2) introduced in [21].

$$scoreT(a) = e^{-(q.t_e - a.t_e)} \tag{3.2}$$

$q.t_e$ is the latest end time of query time constraints and $a.t_e$ is the latest end time of answer time constraints.

The scoring function that considers both answer structure and recency is obtained by combining $scoreS$ and $scoreT$, as shown in Equ 3.3.

$$score(a) = scoreS(a) \times scoreT(a) \tag{3.3}$$

Chapter 4

Proposed Solution

We design a target-oriented search algorithm to answer keyword queries over a temporal relational database modelled as a data graph. Existing data graph keyword search techniques such as BANKS [7] and Bidirectional [16] regard time constraints as keywords to be matched and will return answers that may not satisfy users' search intention. A naive approach to process temporal keyword queries is to extend these methods by first ignoring the time constraints to retrieve all the possible matches and then using the time constraints to filter out invalid answers. This is computationally inefficient.

The proposed algorithm, called ATQ, utilizes the following two strategies to prune the search space:

1. Target-oriented search. Since our query allows users to specify their search intention, we make use of the schema graph to direct the search to the relevant nodes.

2. Time-aware pruning. Given that our query contains temporal constraints, we augment nodes in the data graph with time boundaries to quickly determine if a subtree can satisfy the time constraints. Subtrees that cannot satisfy the time constraints will not be explored.

Before we elaborate on these two strategies in the following subsections, we first parse a given query $\{head : body\}$ into 3 sets:

a. K_{head} is a set of $<k, t>$ pairs where k is a keyword that occurs in $head$ and t is the time information associated with k.

b. K_{body} is a set of $<k, t>$ pairs where k is a keyword that occurs in $body$ and t is the time information associated with k.

c. TR is a set of (p_1, tr, p_2) where $p_1 \in K_{body}$ and $p_2 \in K_{body}$ and tr is the temporal relationship between p_1 and p_2.

Consider query C_5. We have $K_{head} = \{<\texttt{Patient}, \_ >\}$, $K_{body} = \{<\texttt{fever}, \_ >, <\texttt{cough}, \_ >\}$ and $TR = \{(<\texttt{fever}, \_ >, \texttt{BEFORE}, <\texttt{cough}, \_ >)\}$. For query C_6, we have $K_{head} = \{<\texttt{Patient}, \_ >\}$, $K_{body} = \{<\texttt{fever, [1/1/2015, 31/1/2015]} >, <\texttt{cough, [1/1/2015, 31/1/2015]} >\}$ and $TR = \emptyset$. These information will be utilized in the ATQ algorithm.

The ATQ algorithm begins by finding matching nodes for the keywords in K_{head} and K_{body}. Since our keywords may be associated with time information, it is not efficient to use the standard keyword inverted list to retrieve all the tuples that contain the keyword, and then filter them based on time constraints. Thus, we introduce a time-augmented index to efficiently retrieve matching nodes that overlap query intervals.

4.1 Temporal Index for Keywords Associated with Time

In traditional keyword search techniques where time interval is not considered, the inverted list maps each keyword to the list of nodes containing that keyword (Fig. 4.1 shows the inverted list for keyword `Pastia`[1]).

[1] `Pastia's line` is a clinic symptom named after the Romanian physician Constantin Chessec Pastia. (https://en.wikipedia.org/wiki/Pastia's_lines)

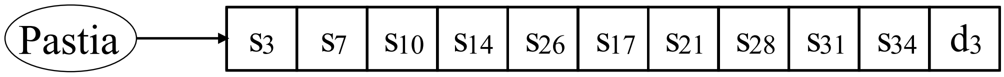


Fig. 4.1. Inverted index for keyword `Pastia` in *Clinic* database (Fig. 1.1)

However, retrieving the whole list of matching nodes for a time associated keyword is wasteful because those nodes whose time intervals do not satisfy the query time constraints will not contribute to the answers. Thus, a better idea is to partition the node list along the timeline and retrieve only the partitions that overlap with the query interval.

Here, we adapt the state-of-the-art interval index technology *OIP* [9] to index the keyword nodes by their corresponding time intervals. *OIP* [9] divides the whole time range (the earliest date time to the latest date time) into m base granules, and each partition is composed of one or more contiguous granules. Given a relation R with time range $U = [U_S, U_E]$, An *OIP* configuration is defined as (m, d, o), where m is the number of base granules, $d = \lceil \frac{|U|}{m} \rceil$ is the granule length, and $o = U_S$ is the earliest time of relation time range.

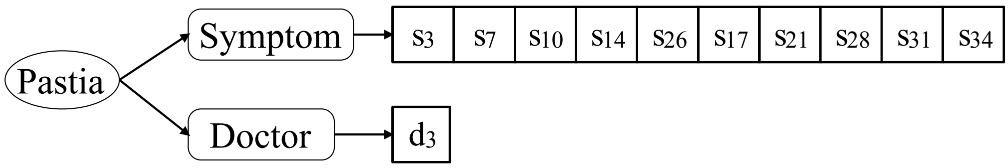


Fig. 4.2. Augment inverted index in Fig. 4.1 by relation

Before partitioning the list according to time intervals, we first group the list according to their relations since tuples from different relations may vary considerably in time unit, e.g., patient birthday and symptom time interval. Fig. 4.2 shows grouped lists for keyword `Pastia`.

Then *OIP* partitions are built for the list of nodes that are associated with time, as shown in Fig. 4.3. Each partition is associated with a time range $[t_s, t_e]$, and nodes are put into this partition if their time intervals are contained by $[t_s, t_e]$. Detailed implementations can be found in [9].

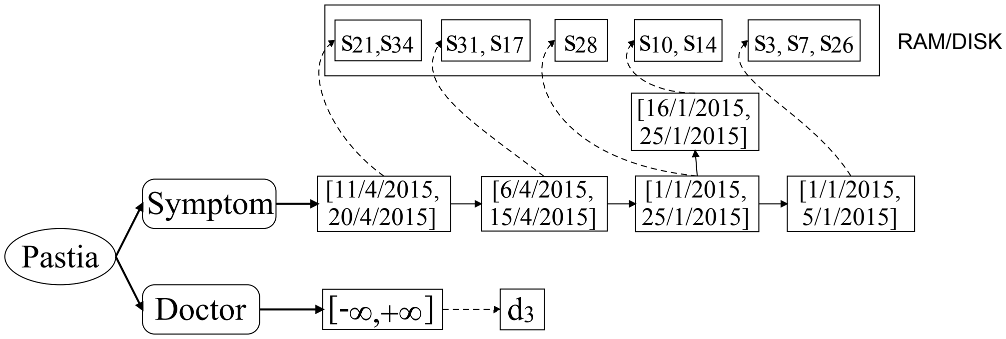


Fig. 4.3. *OIP* index for keyword `Pastia` in *Clinic* database (Fig. 1.1)

4.2 Target-oriented Search

Having found the matching nodes, we construct answers to the query by connecting them. The work in [7] uses Dijkstra's algorithm to find the connecting paths between all pairs of matching nodes. This leads to overwhelming number of answers, many of which are complex and do not satisfy the user's search intention. The Occam's razor principle states that the simplest answer is always favored and this translates to the shortest path that connects the matching nodes. Here, we utilize the schema graph to find the shortest path between the relations corresponding to the matching nodes.

Fig. 4.4 shows the schema graph of the *Clinic* database in Fig. 1.1. Each node is a relation and an edge denotes the key-foreign key constraint between two relations. For example, in query C_5 = {`Patient: fever BEFORE cough`}, the keyword *Patient* in K_{head} corresponds to the `Patient` relation, while keywords `fever` and `cough` in K_{body} correspond to the `Symptom` relation. Based on the schema graph, the shortest path between these relations is via the `Visit` relation. As such, when we traverse the data graph to construct query answers, we do not need to visit nodes that correspond to the `Doctor` relation as they are not part of the shortest path.

With this, our target-oriented search consists of two phases. The first phase aims to construct a partial answer by starting from a node that

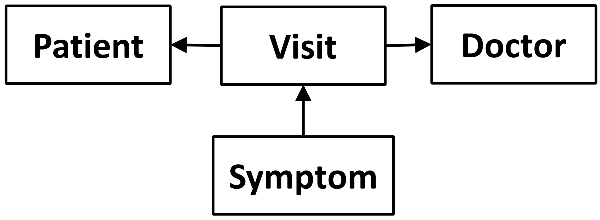


Fig. 4.4. Schema graph of the *Clinic* database in Fig. 1.1

matches a keyword in K_{body} to find a connected component involving nodes that match all the keywords in K_{head}. The second phase completes the search process by finding nodes that match the remaining keywords in K_{body} as well as satisfy the temporal constraints, if any.

Consider the query C_5 in Table 3.2 and the data graph in Fig. 3.1. We start with s_8, a matching node for the keyword `fever`, and visit the node v_3, followed by p_1. Note that we do not need to visit d_1 as it corresponds to the `Doctor` relation which does not lie on the shortest path from `Symptom` to `Patient` (see Fig. 4.4). At this point, we have found a partial answer, that is, patient p_1 with fever. Next, we complete the search by checking if p_1 has a cough which occurs after fever. We traverse the data graph from p_1 to the $Visit$ nodes v_1 ,v_2, v_3, v_4, v_5 and v_6. The nodes v_3 and v_5 do not have any neighbor nodes that match the keyword `cough`, whereas v_1 has the matching node s_1, v_2 has the matching node s_5, v_4 has the matching node s_{12}, and v_6 has the matching node s_{22}. Comparing the time intervals of (s_1, s_8), (s_5, s_8), (s_{12}, s_8) and (s_{22}, s_8), only (s_{12}, s_8) and (s_{22}, s_8) satisfy the temporal relationship `BEFORE`. Thus, we return this subtree $(s_8 - v_3 - p_1 - v_4 - s_{12})$ and $(s_8 - v_3 - p_1 - v_6 - s_{22})$ as two answers to the query.

4.3 Time-aware Pruning

In general, a node may have large number of neighbors. Here, we want to use the temporal constraints in a query to prune subtrees that will not contribute to the query answer. We allow nodes in the data graph to be augmented with time boundaries. In selecting which relations whose nodes need to be augmented with time boundaries, we focus on relations which have a key-foreign key constraint. Given two such relations R_1 and R_2 where R_2 contains the foreign key, we estimate the pruning power obtained by augmenting the nodes of R_2 as $|R_2|/|R_1|$. For our example clinic application, suppose the `Patient` relation has 100 tuples and the `Visit` relation has 5000 tuples, then it will be useful to augment `Visit` nodes with time boundaries to direct the search since each patient will have an average of 50 visits.

Let u be a node in the data graph, S_u be the set of nodes in the subtree rooted at u, and $S_u[R]$ be the set of nodes in S_u that belong to the relation R. Suppose $min(S_u[R])$ and $max(S_u[R])$ are the earliest and latest time of the nodes in $S_u[R]$. Then we associate u with the triplet $<R, min(S_u[R]), max(S_u[R])>$ to indicate the time boundary of a subset of nodes for R. We use this information to eliminate subtrees whose time boundaries are outside the query's time constraints.

Fig. 4.5 shows a data graph where the `Visit` nodes of patient p_1 are augmented with the time boundaries of the `Symptom` nodes. For `Visit` node v_1, it has four `Symptom` nodes s_1, s_2, s_3 and s_4 spanning the periods [01/01/2015, 04/01/2015], [02/01/2015, 05/01/2015], [01/01/2015, 03/01/2015] and [02/01/2015, 04/01/2015] respectively. Thus, the time boundary covered by v_1 is [01/01/2015, 05/01/2015]. A partial answer for the query C_5 = {`Patient: fever BEFORE cough`} over this data graph is $s_8 - v_3 - p_1$, indicating that patient p_1 has fever from 20/01/2015 to 21/01/2015.

Recall that the `BEFORE` relation in Allen's Algebra [2] requires that the start time of the second interval must be greater than the end time of the first interval. Hence, when we try to check if p_1's fever is `BEFORE` cough, we do not need to check all p_1's `Visit` nodes. Instead, only cough that occurs after 21/01/2015 up to the current date (*currentDate*) can contribute to the query answer. Our time-aware pruning strategy determines a valid range [21/01/2015, *currentDate*] and check if this range overlaps with the time boundaries of p_1's `Visit` nodes. In this example, we only need to traverse v_3, v_4, v_5 and v_6 since their time boundaries overlap with the valid range.

On the other hand, suppose cough is associated with a time interval as in query {`Patient` : `fever[1/1/2015, 31/1/2015]` `BEFORE` `cough[1/1/2015, 31/1/2015]`}. Then the valid range for cough should be [21/01/2015, 31/1/2015]. In this case, only the time boundary of v_3 and v_4 overlap with this valid range. When checking the symptom nodes connecting to v_3 and v_4, we find an answer $s_8 - v_3 - p_1 - v_4 - s_{12}$ that contains both keyword `fever` and `cough`, and has satisfied temporal relationship.

Table 4.1 shows the valid ranges corresponding to all possible temporal relationships when we are given the interval of a partial answer $I_1 = [s_1, e_1]$ and the interval $I_2 = [s_2, e_2]$ of a time-associated keyword. A dash entry ($'-'$) indicates that there is no valid range, and the partial answer can be pruned in this case.

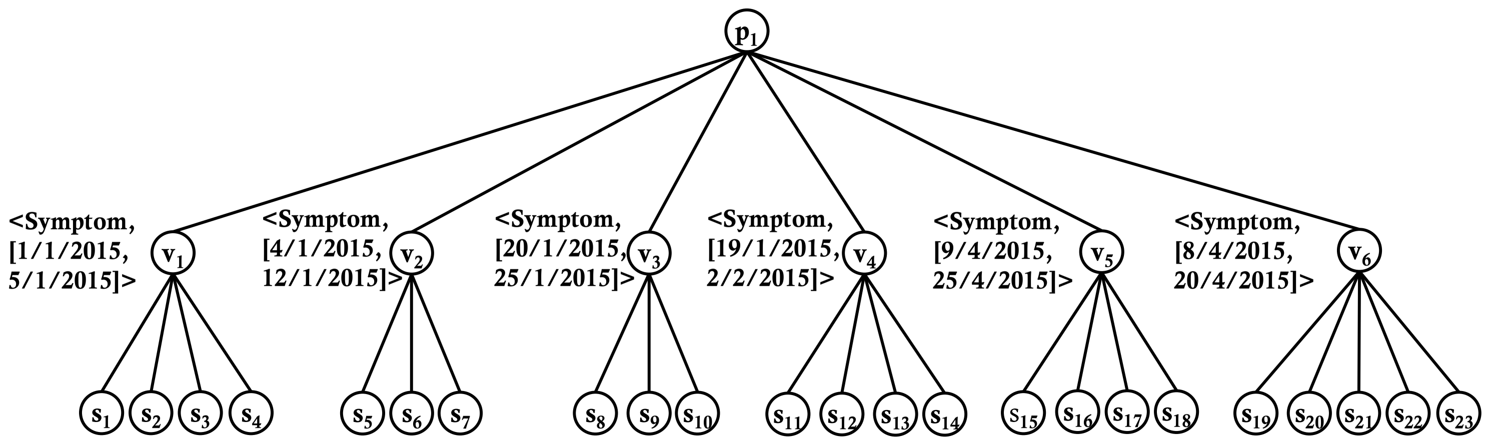
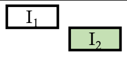
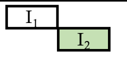
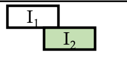
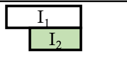
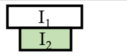
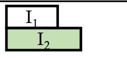
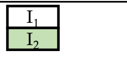
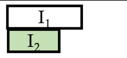
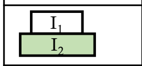
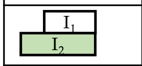
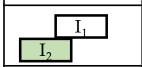
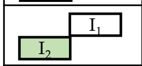
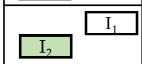


Fig. 4.5. Augmented data graph with time boundary

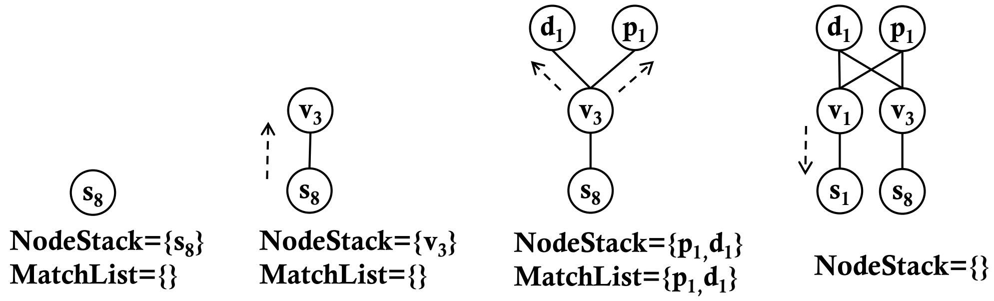
Table 4.1
Computation of valid range

	BEFORE	MEET	OVERLAP	FINISHED BY	CONTAINS	STARTS	EQUALS
	$[s_2,e_2]$	-	-	-	-	-	-
	$[s_2,e_2]$	$[e_1,e_2]$	-	-	-	-	-
	$[e_1,e_2]$	$[e_1,e_2]$	$[s_2,e_1]$	$[s_2,e_1]$	$[s_2,e_1]$	-	-
	-	$[e_1,e_2]$	$[s_2,e_2]$	$[s_2,e_1]$	$[s_2,e_1]$	-	-
	-	-	$[s_2,e_2]$	-	$[s_2,e_2]$	-	-
	$[e_1,e_2]$	$[e_1,e_2]$	$[s_2,e_1]$	$[s_2,e_1]$	$[s_2,e_1]$	$[s_1,e_2]$	$[s_1,e_1]$
	-	$[e_1,e_2]$	$[s_2,e_2]$	$[s_2,e_1]$	$[s_2,e_2]$	$[s_1,e_2]$	$[s_1,e_1]$
	-	-	$[s_2,e_2]$	-	$[s_2,e_2]$	$[s_1,e_2]$	-
	$[e_1,e_2]$	$[e_1,e_2]$	$[s_1,e_1]$	$[s_2,e_1]$	$[s_1,e_1]$	$[s_1,e_2]$	$[s_1,e_1]$
	-	$[e_1,e_2]$	$[s_1,e_2]$	$[s_2,e_1]$	$[s_1,e_1]$	$[s_1,e_2]$	$[s_1,e_1]$
	-	-	$[s_1,e_2]$	-	$[s_1,e_2]$	$[s_1,e_2]$	-
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-

4.4 Algorithms

We incorporate the target oriented search strategy and the time-aware pruning strategy into our *ATQ* (**A**nswering **T**emporal **Q**uery) algorithm (see Algorithm 1). We first parse the input query into three sets: K_{head} and K_{body} keep the keywords and their associated time information for the query's head and query's body respectively, while TR keeps the temporal relationships among these keywords (Line 1).

For each tuple $<k,t>$ in the set K_{head}, we retrieve the set of relations corresponding to the nodes that match k (Lines 2-3). For each tuple $<k,t>$ in the set K_{body}, we retrieve the set of nodes that match k and satisfy its associated time constraint t (Lines 4-5). We select the set $V_{k_{min}}$ that has the least number of matched nodes for a keyword in K_{body} to start the search (Line 6). For example, in query C_4, the nodes that match the keyword `fever` are $\{s_8, s_{15}, s_{19}, s_{24}, s_{30}\}$, and the nodes that match `cough` are $\{s_1, s_5, s_{12}, s_{22}, s_{27}, s_{32}, s_{33}\}$. We start the search with the smaller set as it enables us to narrow the search space quickly.



(a) (b) (c) (d)

Fig. 4.6. Construction of a partial answer tree for query C_4

For each node $v \in V_{k_{min}}$, we search from v along the shortest path based on the schema graph to connect nodes that can match the keywords in K_{head} (Lines 7-25). We maintain two stacks: *NodeStack* keeps the traversed nodes in G, and *Partial* stores the subtrees of partial answers built during the search process. We also maintain a $MatchList$ to keep track of the keywords in K_{head} that we have found so far. In our example, suppose we start with node s_8. We first add it to $NodeStack$, and a partial tree is created with s_8 as shown in Fig. 4.6(a). Since s_8's relation does not match any keyword in K_{head}, we get its relevant neighbor v_3 in the shortest path $\{Symptom - Visit - Patient\}$, add v_3 to $NodeStack$ and connect v_3 to the partial answer tree (see Fig. 4.6(b)).

When a node v matches some keyword in K_{head}, we add v to $MatchList$ (Lines 14-15). If not, we call function $getRelevantNeighbours()$ to find the set of nodes to traverse next (Lines 26-39). From Fig. 4.6(b), we see that v_1 does not match any keyword in K_{head}. Hence, we obtain v_1's relevant neighbor p_1. Since p_1's relation matches *Patient*, we add p_1 to the $MatchList$ and connect p_1's node to the partial answer tree. At this point, $MatchList$ has not satisfied K_{head} as we still need to match *Doctor*. Hence, the al-

Algorithm 1: ATQ Algorithm

input : query Q, data graph G, schema graph H
output: Result set *Results*

Parse query Q to get K_{head}, K_{body}, and TR
foreach *tuple* $\langle k, t\rangle$ *in* K_{head} **do**
    $R_k \leftarrow$ the set of relations corresponding to the nodes that match k
foreach *tuple* $\langle k, t\rangle$ *in* K_{body} **do**
    $V_k \leftarrow$ the set of nodes in G that match k and satisfy the time constraint t
Let k_{min} be the keyword in K_{body} with the least number of matched nodes,
foreach $v \in V_{k_{min}}$ **do**
    Initialize *NodeStack*, *Partial* to empty stacks;
    $tree_v \leftarrow$ create a tree with root v
    push(v, *NodeStack*); push($tree_v$, *Partial*)
    MatchList$\leftarrow \emptyset$
    while *NodeStack is not empty* **do**
        $u \leftarrow$ pop(*NodeStack*); $tree_v \leftarrow$ pop(*Partial*)
        if *u's relation matches some keyword in* K_{head} **then**
            add u to MatchList
            if *MatchList satisfy* K_{head} **then**
                if $tree_v$ *satisfy* K_{body} **then**
                    add $tree_v$ to *Results*
                else
                    $W \leftarrow$ getLCA(MatchList)
                    foreach $w \in W$ **do**
                        let $tree'_v$ be a copy of $tree_v$
                        $tree \leftarrow$ reverseSearch($tree'_v$, w, K_{body} , TR)
                        add $tree$ to *Results*
                MatchList$\leftarrow \emptyset$
        $R = \bigcup_{k \in K_{head}} R_k$
        N = getRelevantNeighbours(u, R, H)
        foreach *node* n *in* N **do**
            let $tree'_v$ be a copy of $tree_v$
            connect n to $tree'_v$
            push(n, *NodeStack*)
            push($tree'_v$, *Partial*)
Function *getRelevantNeighbours(u, R, H)*
    $N \leftarrow \emptyset$
    Let N_u be the set of nodes that are one hop away from u
    foreach v *in* N_u **do**
        if *relation*(v) *is on the shortest path from relation*(u) *to some relation in* R *in the schema graph* H **then**
            $N \leftarrow N \bigcup \{v\}$
    return N

gorithm continues with the next relevant neighbor of v_1. This time, d_1 is found and is added to the $MatchList$. The partial answer tree obtained is shown in Fig. 4.6(c).

When $MatchList$ satisfies K_{head}, we check if the partial answer $tree_v$ satisfies K_{body} (Lines 16-17). If so, $tree_v$ is an answer to the query and we add it into the result set $Results$ (Lines 18). Otherwise, we get the set of lowest common ancestors (LCA) for the nodes in $MatchList$ (Lines 20). In our example, since p_1's relation matches the keyword `Patient` in K_{head} and d_1's relation matches the keyword `Doctor` in K_{head}, we add p_1 and d_1 to $MatchList$. Although $MatchList$ satisfies K_{head}, the partial answer tree does not satisfy K_{body}. As such, we obtain the LCA of the nodes in $MatchList$, that is, $\{v_1, v_2, v_3\}$ in this case.

For each node in the LCA set, we call Algorithm $reverseSearch$ to find nodes that match the remaining keywords in K_{body} (Lines 21-23). This algorithm returns a tree that is an answer to the query and is added to the result set (Line 24). Algorithm reverseSearch (see Algorithm 2) takes as input a partial answer tree and tries to construct the complete answer by finding nodes that match the remaining keywords in K_{body}. It also uses a stack $NodeStack$ to keep track of the nodes to be processed and calls function $getRelevantNeighbours()$ to find the set of nodes to traverse next (Lines 5-6). For each node u to be traversed, if u matches a keyword in K_{body}, we check that u satisfies the time constraints and connect u to the answer tree (Lines 7-10). When $tree$ matches all the keywords in K_{body}, we have an answer (Lines 11-12). If u does not match a keyword in K_{body}, we perform time-aware pruning by calling the function $hasOverlap()$ (Lines 14-17). This function computes the valid range and checks if this range overlaps with the time boundary of node u (Lines 19-26).

Continuing with our example in Fig. 4.6, we try to match the remaining keyword `cough` in K_{body}. The relevant neighbors of v_1 are s_1, s_2, s_3, s_4. Since

s_1 matches the keyword *cough*, s_1 is connected to the partial tree as shown in Fig. 4.6(d). We return this tree as an answer to query C_4 since it contains all the keywords in K_{body}.

Algorithm reverseSearch (see Algorithm 2) takes as input a partial answer tree and tries to construct the complete answer by finding nodes that match the remaining keywords in K_{body}. It also uses a stack *NodeStack* to keep track of the nodes to be processed and calls $getRelevantNeighbours()$ function to find the set of nodes to traverse next (Lines 5-6). For each node u to be traversed, if u matches a keyword in K_{body}, we check that u satisfies the time constraints and connect u to the answer tree (Lines 7-9). When *tree* matches all the keywords in K_{body}, we have an answer (Lines 10-11). If u does not match a keyword in K_{body}, we perform time-aware pruning by calling the function $hasOverlap()$ (Lines 13-16). This function computes the valid range and checks if this range overlaps with the time boundary of node u (Lines 18-25).

Algorithm 2: reverseSearch ($tree$, v, K_{body}, TR)

input : partial answer $tree$, LCA node v, K_{body}, temporal relationship TR
output: result $tree$

Initialize *NodeStack* to an empty stack
push(v, *NodeStack*)
while *NodeStack is not empty* **do**
    $u \leftarrow$ pop(*NodeStack*)
    Let R be the set of relations that correspond to the remaining keywords in K_{body} that has not been matched in $tree$
    N = getRelevantNeighbours(u, R, H)
    foreach *node u in N* **do**
        if *u matches keyword in K_{body}* **then**
            if *u satisfies the time constraints* **then**
                connect u to $tree$
                if *$tree$ matches all the keywords in K_{body}* **then**
                    return $tree$
        else
            Let I be the interval constrained by $tree$
            if *hasOverlap(I, u, K_{body}, TR)* **then**
                connect u to $tree$;
                push(u, *NodeStack*)
return $\emptyset$

Function *hasOverlap(I, u, K_{body}, TR)*
    foreach $\langle k, t\rangle \in K_{body}$ **do**
        Let $TR_k \subset TR$ be the set of temporal relationships involving k
        foreach $tr \in TR_k$ **do**
            $range \leftarrow$ getValidRange(I, tr, t)
            if *range overlap Boundary[u]* **then**
                return true
    return false

Chapter 5

Performance Study

In this section, We evaluate the performance of ATQ and compare it with BANKS [7] and Bidirectional [16]. All the algorithms are implemented in Java and experiments are carried out on a 1.4 GHz Intel Core i5 CPU with 4 GB RAM. Each experiment is repeated 10 times and we report the average results. We use the following three datasets in our experiments.

1. *Clinic dataset* [1]. It contains information about patient consultations with doctors. We use 565 records from the real world dataset as seeds whereby we generate 50 visits per day from 2006 to 2016, and randomly choose a patient and a doctor for each generated visit. For each visit, we randomly assign up to 5 symptoms. The start date of each symptom varies between 1 to 14 days before the visit date. The end date of each symptom is set to be the visit date.

2. *Employees dataset* [2]. This dataset contains the job histories of employees, as well as the departments where the employees have worked in from 1985 to 2003.

3. *ACMDL dataset* [3]. This publication dataset is contains information about authors, proceedings, editors and publishers from 1969 to 2011.

[1] This dataset is not available due to patient confidentiality.
[2] https://dev.mysql.com/doc/employee/en/
[3] http://dl.acm.org/

Table 5.1
Dataset schema and the number of tuples for each relation

Clinic	# of tuples
Doctor(did, dname, gender)	149
Patient(pid, pname, gender, birthday, ethnicity, postalCode)	1,033
Visit(vid, date, pid, did)	182,600
Symptom(sid, sname, startDate, endDate, vid)	430,470
Employees	# of tuples
Department(dept_no, dept_name)	9
Employees(emp_no, fname, lname, gender, hire_date)	300,024
Dept_emp(deid, emp_no, dept_no, from_date, to_date)	331,603
Title(tid, title, emp_no, from_date, to_date)	443,308
ACMDL	# of tuples
Publisher(publisherid, code, name)	40
Proceeding(procid, title, date, area, publisherid)	4,176
Editor(editorid, fname, lname)	20,008
Edit(editorid, procid)	20,712
Paper(paperid, procid, date, ptitle)	248,185
Author(authorid, fname, lname)	257,694
Write(authorid, paperid)	550,000

Table 5.1 shows the schema of these datasets and the number of tuples in each relation. We design two sets of queries for each dataset. The first set does not involve any time constraints, while the second set contains keywords associated with time information and temporal relationships. Queries for the `Clinic` dataset is shown in Table 3.2, while queries for the *Employees* and *ACMDL* are listed in Tables 5.2 and 5.3 respectively.

Table 5.2
Temporal keywords queries for *Employees* dataset

Query		Intended meaning
E_1	`{Employee: Engineer }`	Find employees who are engineers.
E_2	`{Employee: Engineer, Manager }`	Find employees who have been engineer and manager before.
E_3	`{Employee, Female: Engineer, Manager }`	Find female employees who have been engineer and manager before.
E_4	`{Employee, Department: Engineer}`	Find employees who are engineers and their departments
E_5	`{Employee: Engineer BEFORE Manager}`	Find employees who are engineers before coming managers.
E_6	`{Employee: Manager[1/1/1990,1/1/2000], Engineer[1/1/1990,1/1/2000]}`	Find employees who have been engineer and manager from 1990 to 2000
E_7	`{Employee: Manager[1/1/1990, 1/1/2000] BEFORE Engineer[1/1/1990,1/1/2000]}`	Find employees who are engineers before becoming managers from 1990 to 2000
E_8	`{Employee, Department: Engineer[1/1/1990,1/1/2000]}`	Find employees and their departments where these employees are engineers from 1990 to 2000
E_9	`{Employee, Department: Manager[1/1/1990,1/1/2000], Engineer[1/1/1990,1/1/2000] }`	Find employees who have been engineer and manager from 1990 to 2000 and their departments
E_{10}	`{Employee: Engineer[1/1/1990, 1/1/2000] MEET ‘‘Senior Engineer’’[1/1/1990,1/1/2000]}`	Find employees that the end time of title “engineer” is the same as the start time of “senior engineer” from 1999 to 2000
E_{11}	`{Employee: ‘‘Assistant Engineer’’[1/1/1990, 1/1/2000] MEET Engineer, Engineer MEET ‘‘Senior Engineer’’[1/1/1990,1/1/2000]}`	Find the employees that the end time of “assistant engineer” is the same as the start time of “Engineer”, and the end time of “engineer” is the same as the start time of “senior engineer” from 1990 to 2000

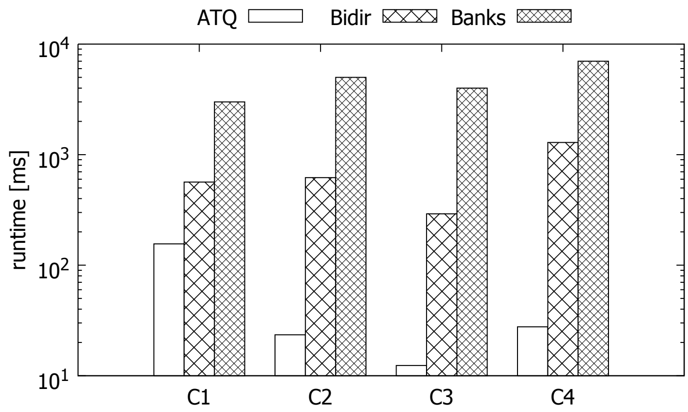
Table 5.3
Temporal keywords queries for *ACMDL* dataset

Query		Intended meaning
A_1	`{Author: Integration }`	Find authors who has published papers on "Integration"
A_2	`{Author: Integration, Cleaning}`	Find authors who has published papers on "Integration" and "Cleaning"
A_3	`{Proceeding, SIGMOD: Integration}`	Find papers published in the "SIGMOD" proceeding that are on "Integration"
A_4	`{Publisher, Proceeding: Data, Integration }`	Find publishers and proceedings pair where the proceedings contain papers on "Data Integration"
A_5	`{Author: Media BEFORE AI }`	Find authors who have published papers related to "Media" prior to publishing papers related to "AI"
A_6	`{Author: Media[01/01/2000,01/01/2008], AI[01/01/2000,01/01/2008] }`	Find authors who have published papers related to both "Media" and "AI" from 2000 to 2008
A_7	`{Author: Media[01/01/2000,01/01/2008] BEFORE AI[01/01/2000, 01/01/2008]}`	Find authors who have published papers related to "Media" before publishing papers related to "AI" from 2000 to 2008
A_8	`{Proceeding, Publisher: Integration[1/1/2000,1/1/2008]}`	Find the publishers and proceedings that have included papers on "Integration"from 2000 to 2008
A_9	`{Proceeding, Publisher: Integration[1/1/2000, 1/1/2008], Data[1/1/2000, 1/1/2008]}`	Find the publishers and proceedings that have included papers on "Data Integration"from 2000 to 2008
A_{10}	`{Author: WWW AFTER CSCW }`	Find authors who published papers in proceeding WWW before proceeding CSCW
A_{11}	`{Author : SIGMOD AFTER KDD, KDD AFTER WWW}`	Find authors who published papers in Proceedings SIGMOD after KDD and KDD after WWW

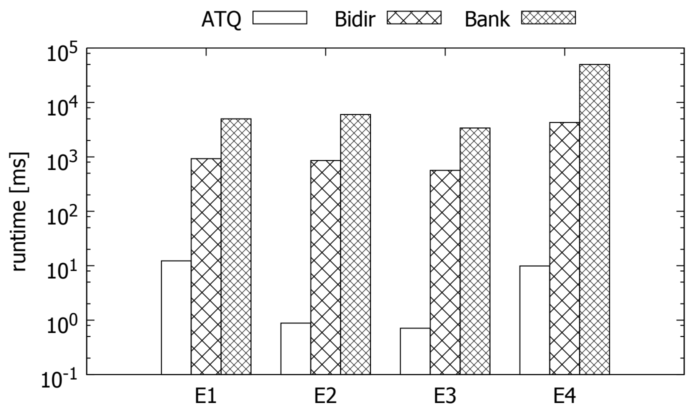
5.1 Experiments on Queries without Time Constraints

We first evaluate the performance of our approach using queries that do not involve time information. These queries correspond to C_1 to C_4 in Table 3.2, E_1 to E_4 in Table 5.2, and A_1 to A_4 in Table 5.3. We compare the runtimes of *ATQ* with *Banks* [7] and *Bidirectional* [16]. Since both *Banks* and *Bidirectional* do not handle keywords that match relation names, we modify these algorithms to consider all the nodes of the queried relation as matching nodes. For fair comparison, we report the time taken by these methods to return the first 20 answers.

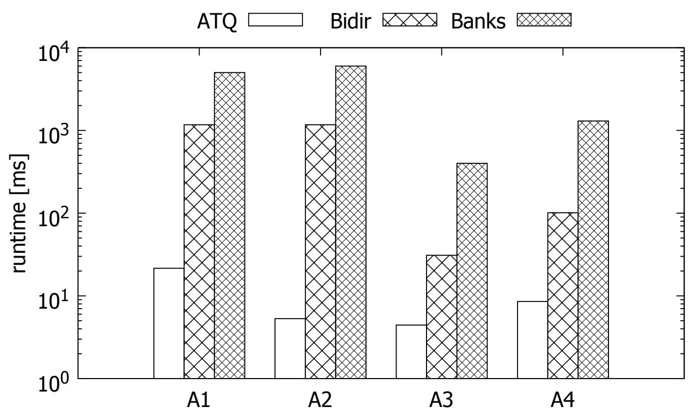
Fig. 5.1 shows the results for the 3 datasets. We observe that *ATQ* outperforms *Bidirectional* and *Banks* for all the queries, with *Banks* being the slowest. This indicates the advantage of our target-oriented search strategy. For the *Clinic* dataset, we see that the runtimes of *ATQ* for queries C_2 and C_3 are lower than C_1 although these queries have more keywords than C_1. This is because *ATQ* will make use of the keyword with the least number of matching nodes to generate a small set of partial answers. This reduces the time needed to check if these partial answers are valid during the *reverseSearch* process to obtain the complete answers. On the other hand, the runtimes of *ATQ* for query C_4 increases. This is because C_4 has an additional search target relation in the head of the query, leading to a larger number of matching nodes, thus the time needed to find the partial answers is longer. We observe similar trends for the queries on the *Employees* and *ACMDL* datasets.



(a) Clinic



(b) Employees



(c) ACMDL

Fig. 5.1. Runtime for queries without time constraints

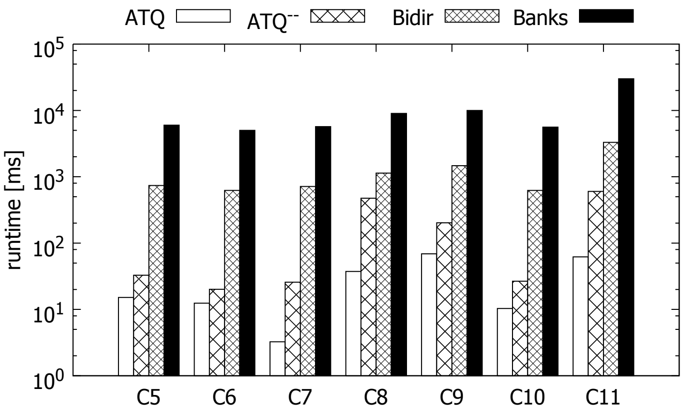
5.2 Experiments on Queries with Time Constraints

Next, we evaluate the performance of our approach to process keyword queries that involve time. These queries correspond to C_5 to C_{11} in Table 3.2, E_5 to E_{11} in Table 5.2, and A_5 to A_{11} in Table 5.3. In particular, we allow different types of temporal relationships in the same query such as C_{11}, and we also allow keywords to be optionally associated with time intervals such as C_{10}, C_{11} and E_{11}.

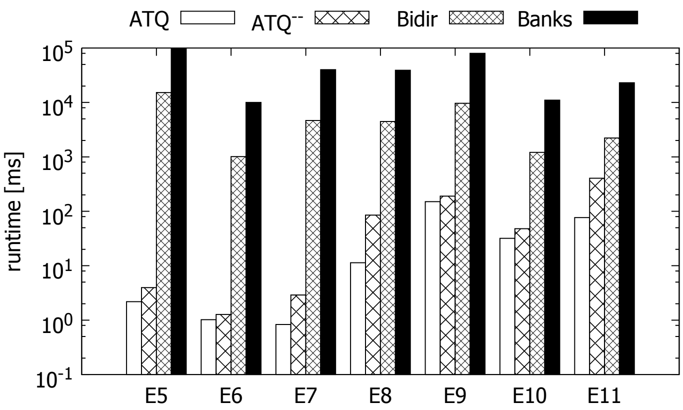
We extend existing methods *Banks* and *Bidirectional* to handle temporal keyword queries by ignoring the time intervals and temporal relationships in these queries and processing the keywords to obtain candidate answers. Answers that do not satisfy the time constraints are filtered by a post-processing step.

At the same time, we implemented ATQ^-, a variant of the ATQ algorithm which does not utilize the augmented data graph (time boundaries in the nodes) and the overlapping time interval in the inverted lists for the keywords. Instead, ATQ^- also has a post-processing step to filter invalid answers.

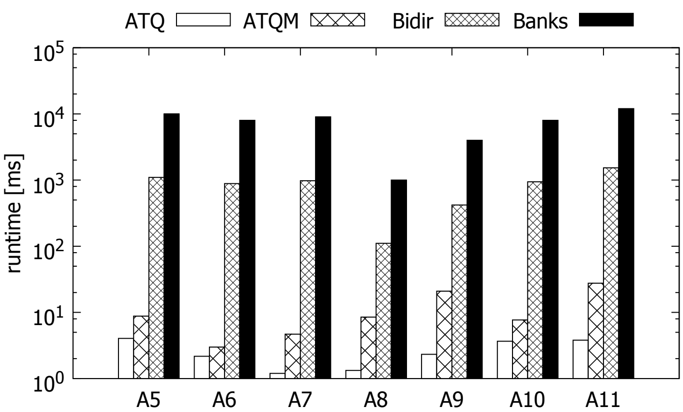
Fig. 5.2 shows the results for the 3 datasets. We observe that both ATQ and ATQ^- outperform *Banks* and *Bidirectional* for all the queries by a large margin. Further, we see that time-aware pruning strategy enables ATQ to be faster than ATQ^-. In particular, for query C_7, A_7, we observe that ATQ is very much faster than ATQ^-. This is because the combination of time interval constraints and temporal relationships leads to a narrow valid range that allows more invalid partial answers can be pruned. For E_7, the pruning effect is not as significant as C_7 and A_7 here, this is due to nature of *Employees* dataset, as for each employee, the number of titles are limited.



(a) Clinic



(b) Employees



(c) ACMDL

Fig. 5.2. Runtime for queries involving time constraints

5.3 Experiments on Scalability

In this section, we evaluate the scalability of the proposed approach from two aspects: dataset size and query time interval length.

For each dataset, we generate different sizes by taking tuples from different years. Table 5.4 shows the various dataset sizes generated.

Fig. 5.3 shows the average runtime of *ATQ*, ATQ^-, *Bidirectional* and *Banks* in returning the first 20 answers for the queries in Table 3.2, Table 5.2 and Table 5.3. We observe that *ATQ* outperforms emphATQ$^-$, *Bidirectional* and *Banks*. Further, as the dataset sizes increases, the runtime of *Bidirectional* and *Banks* increase at a much faster pace compared to *ATQ*. This demonstrates clearly the scalability of *ATQ* in answering temporal keyword queris.

Next, we evaluate the scalability with respect to different query interval lengths. We use the following query templates to generate queries of different interval lengths by replacing t_s and t_e with the start and end periods of the corresponding datasets in Table 5.4.

Clinic	:	{Patient: fever[t_s, t_e] OVERLAP cough[t_s, t_e]}
Employees	:	{Employee:Engineer[t_s, t_e] MEET "Senior Engineer"[t_s, t_e]}
ACMDL	:	{Author:Media[t_s, t_e] BEFORE AI[t_s, t_e]}

Fig. 5.4 shows the runtimes of queries with different time interval lengths. We see that with the increase of time interval lengths, the runtimes of ATQ^-, *Bidirectional* and *Banks* decrease. This is because these algorithms apply the time constraints only after the candidate answers have been generated. When the query time interval becomes larger, more tuples will satisfy the time constraints, hence the time taken to generate the first 20 answers is faster.

Table 5.4
Datasets generated for scalability experiments

(a) Clinic

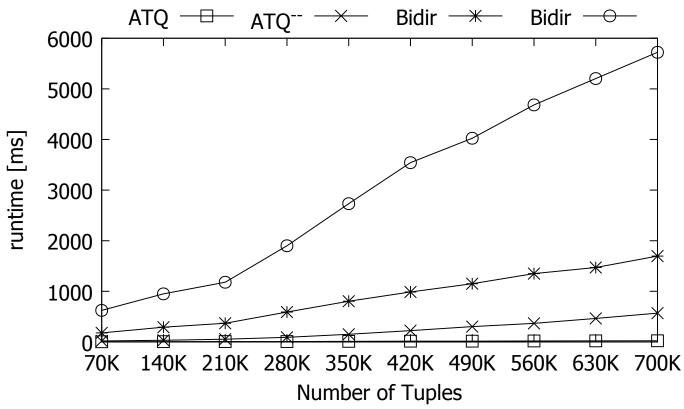
Start Period	End Period	Dataset Size
01/01/2015	31/12/2015	70K
01/01/2014	31/12/2015	140K
01/01/2013	31/12/2015	210K
01/01/2012	31/12/2015	280K
01/01/2011	31/12/2015	350K
01/01/2010	31/12/2015	420K
01/01/2009	31/12/2015	490K
01/01/2008	31/12/2015	560K
01/01/2007	31/12/2015	630K
01/01/2006	31/12/2015	700K

(b) Employees

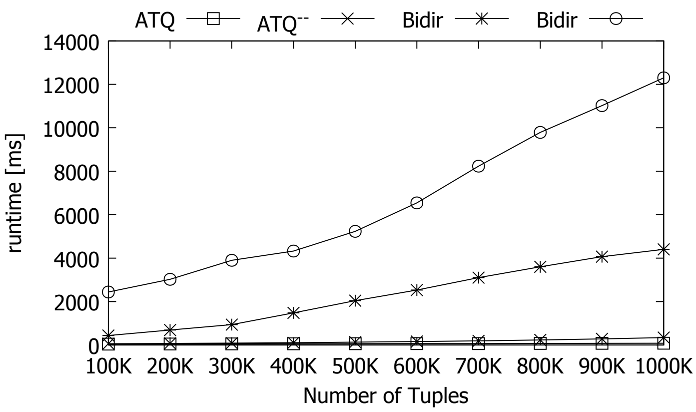
Start Period	End Period	Dataset Size
01/04/1999	31/08/2002	100K
01/11/1997	31/08/2002	200K
01/06/1996	31/08/2002	300K
01/01/1995	31/08/2002	400K
01/08/1993	31/08/2002	500K
01/02/1992	31/08/2002	600K
01/06/1990	31/08/2002	700K
01/09/1988	31/08/2002	800K
01/12/1986	31/08/2002	900K
01/01/1986	31/08/2002	1000K

(c) Employees

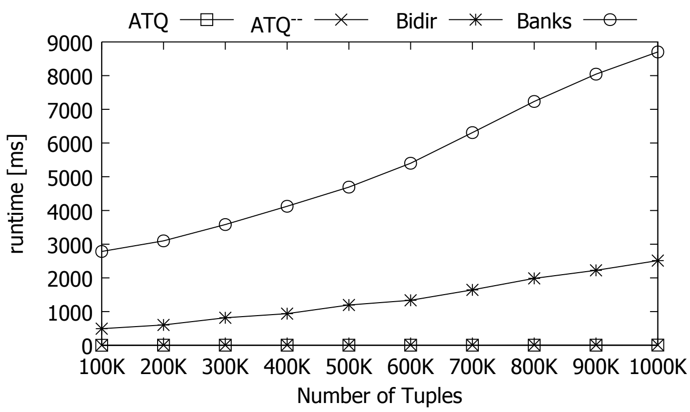
Start Period	End Period	Dataset Size
01/04/2010	30/06/2011	100K
01/03/2009	30/06/2011	200K
01/10/2007	30/06/2011	300K
01/05/2006	30/06/2011	400K
01/06/2004	30/06/2011	500K
01/08/2001	30/06/2011	600K
01/11/1997	30/06/2011	700K
01/10/1992	30/06/2011	800K
01/07/1983	30/06/2011	900K
01/01/1969	30/06/2011	1000K



(a) Clinic

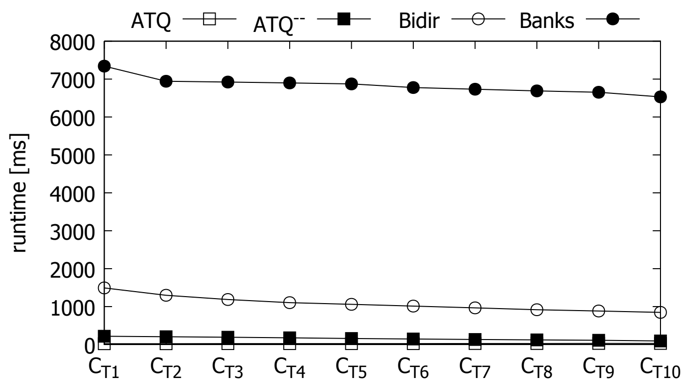


(b) Employees

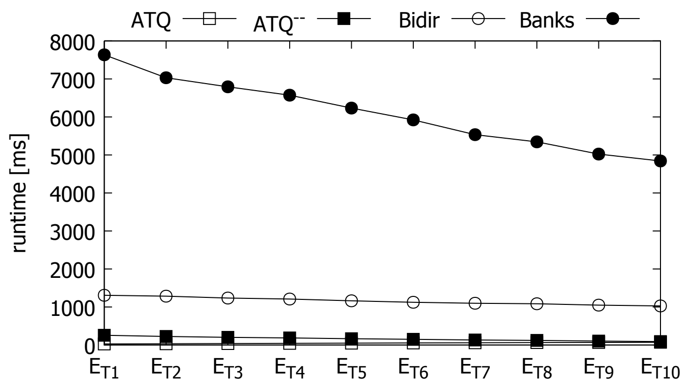


(c) ACMDL

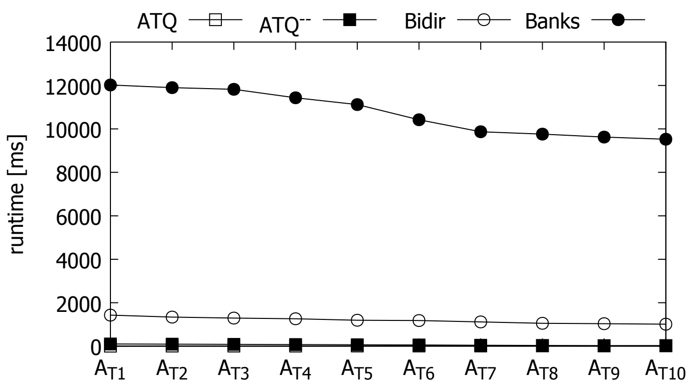
Fig. 5.3. Runtime for queries with different dataset sizes



(a) Clinic



(b) Employees



(c) ACMDL

Fig. 5.4. Runtime for queries with different time interval length

5.4 Experiments on Search Quality

In this section, we want to evaluate the search quality by checking whether the returned answers satisfy search intentions, i.e., all the keywords in the query body must be closely related to search targets in the query head. We use the *mean average precision* (MAP) as the metric, where $MAP@k$ for a set of queries Q is the mean of the average precision scores of k results for each query $q \in Q$, defined as follows:

$$MAP@k = \frac{1}{|Q|} \sum_{q \in Q} \Big(\frac{1}{k} \sum_{i=1}^{k} P(i) \Big) \tag{5.1}$$

where $P(i)$ is the precision at answer position i.

We evaluate the answers returned by ATQ and *Bidirectional*. We separate the queries into two sets: the first set consists of simple queries (C_1 to C_3, A_1 to A_3, E_1 to E_3), while the second set are complex queries with more constraints or multiple search targets (C_4 to C_{11}, A_4 to A_{11}, E_4 to E_{11}). Table 5.5 shows the MAP values for these two sets of queries.

We observe that ATQ can always return relevant answers for both simple and complex queries for all the datasets. This demonstrates the effectiveness of the proposed target oriented search algorithm.

Bidirectional can return highly relevant answers for simple queries, with MAP value 1 for *Clinic* and *Employees*. For *ACMDL* dataset, some irrelevant answers are returned for query A_2 = {`Author: Integration, Cleaning`} with structure *Author-Write-Paper-Proceeding-Paper*, which means keywords `Integration` and `Cleaning` are connected because they are from the same proceeding but not the same author. We expect answers are returned with structure *Paper-Write-Author-Write-Paper*, i.e., two papers are written by the same author.

For complex queries, *Bidirectional* can return all relevant answers for *Employees* dataset. However, the MAP drops for *Clinic* and *ACMDL* dataset. This is because the database schemas of *Clinic* and *ACMDL* are more complex than *Employees*. On the other hand, with the increasing number of answers returned, MAP decreases for *Clinic* and *ACMDL* dataset. More meaningless answers are returned by connecting keyword nodes from different search target nodes. For example, meaningless answers to query C_4 with "fever" and "cough" from different patients are returned. For query A_4, meaningless answers with "data" from paper of one proceeding and "integration" from paper of another proceeding, and they are connected because they are from the same publisher. This is different from the intended answer meaning with "data" and "integration" connecting to the same proceeding and publisher pair.

Table 5.5
MAP with different number of answers

(a) Clinic

Query	Simple		Complex	
Algorithm	ATQ	Bidir	ATQ	Bidir
MAP-10	1.0	1.0	1.0	0.21
MAP-20	1.0	1.0	1.0	0.17
MAP-30	1.0	1.0	1.0	0.16
MAP-40	1.0	1.0	1.0	0.15
MAP-50	1.0	1.0	1.0	0.15

(b) Employees

Query	Simple		Complex	
Algorithm	ATQ	Bidir	ATQ	Bidir
MAP-10	1.0	1.0	1.0	1.0
MAP-20	1.0	1.0	1.0	1.0
MAP-30	1.0	1.0	1.0	1.0
MAP-40	1.0	1.0	1.0	1.0
MAP-50	1.0	1.0	1.0	1.0

(c) ACMDL

Query	Simple		Complex	
Algorithm	ATQ	Bidir	ATQ	Bidir
MAP-10	1.0	1.0	1.0	0.53
MAP-20	1.0	0.87	1.0	0.33
MAP-30	1.0	0.80	1.0	0.22
MAP-40	1.0	0.77	1.0	0.16
MAP-50	1.0	0.75	1.0	0.13

5.5 Case Study

Finally, we present a few case studies to demonstrate the effectiveness of *ATQ* in returning relevant answers to the queries.

Fig. 5.5 and Fig. 5.6 show the first 3 answers returned by *ATQ* and *Bidirectional* for query C_{10} = `{Patient: fever[1/1/2010, 1/1/2015] OVERLAP headache}`. We find all the answers returned by *ATQ* in Fig. 5.5 satisfy the query conditions, while the answer in Fig. 5.6(c) is not meaningful as the symptom `headache` and `fever` are from two different patients with the same doctor d_0.

Fig. 5.7 and Fig. 5.8 show the first 3 answers returned by *ATQ* and *Bidirectional* for query C_{11} = `{Patient: fever[1/1/2010,1/1/2015] OVERLAP cough, headache BEFORE fever[1/1/2000, 1/1/2015]}`. C_{11} is more complex than C_{10} since it contains more keywords and temporal relationships. We find that *ATQ* is still able to return the correct answers whereas *Bidirectional* returns more irrelevant answers. For example, Fig. 5.8(b) shows an incorrect answer as the three symptoms do not belong to the same patient. Similarly, Fig. 5.8(c) shows an answer where the three symptoms belonged to different patients who are treated by the same doctor.

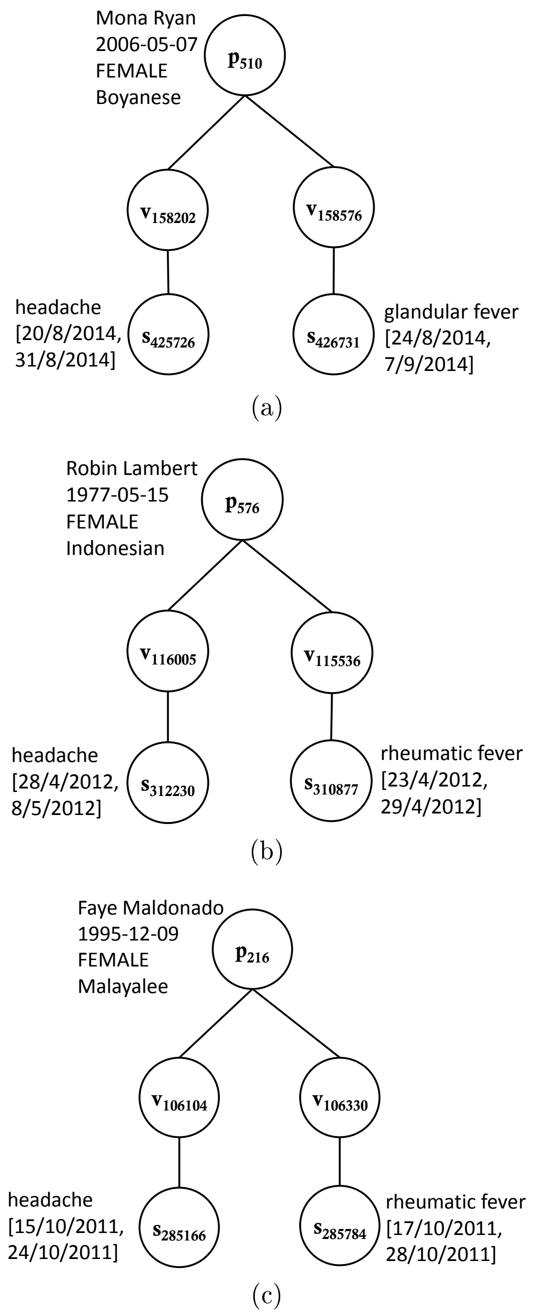


Fig. 5.5. First three answers returned to query C_{10} by ATQ

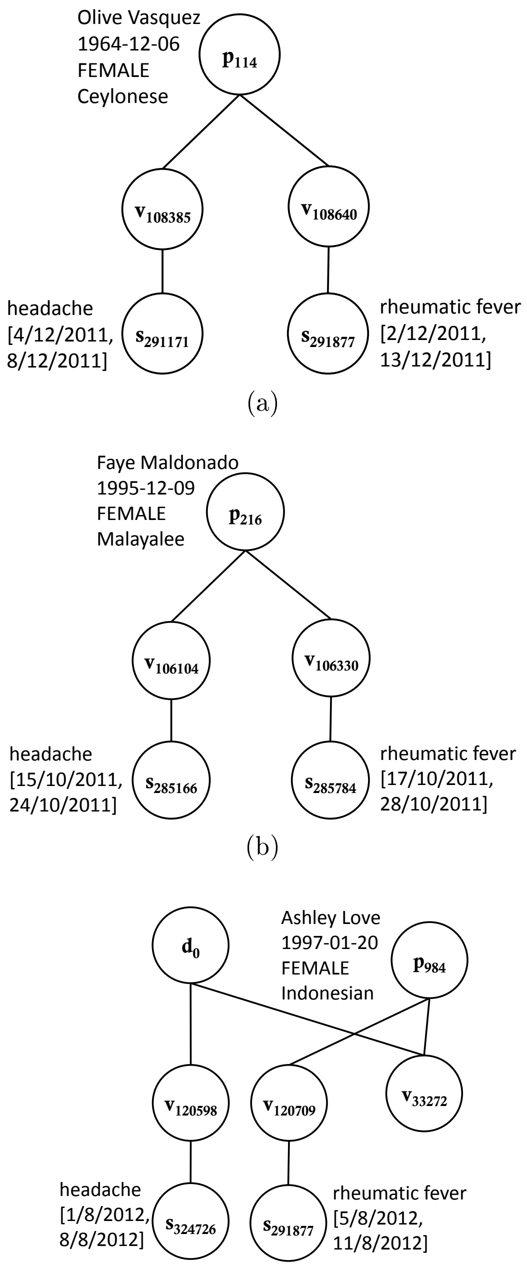


Fig. 5.6. First three answers returned to query C_{10} by *Bidirectional*

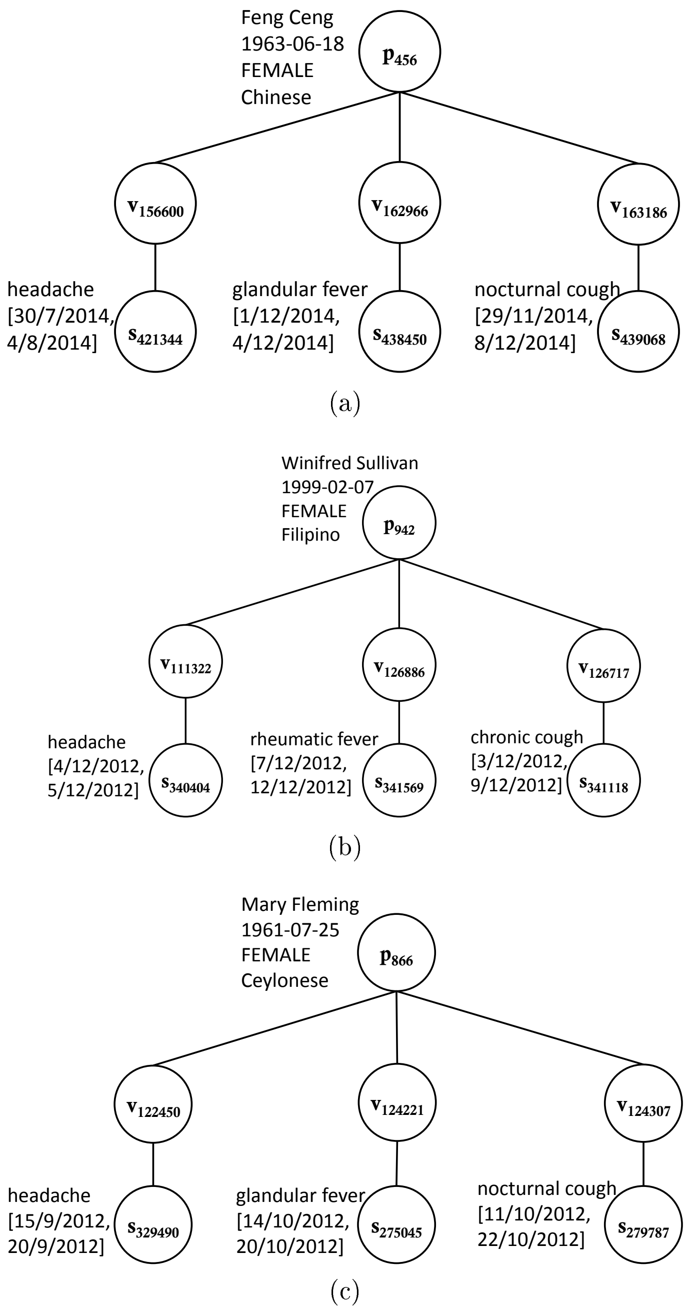


Fig. 5.7. First three answers returned to query C_{11} by ATQ

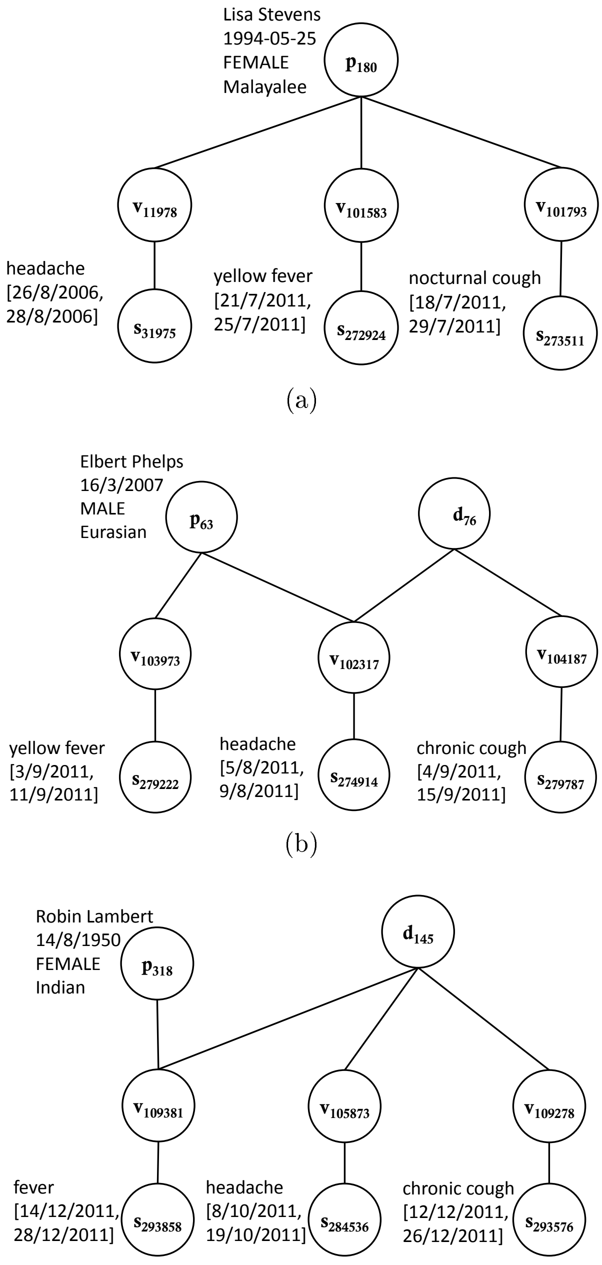


Fig. 5.8. First three answers returned to query C_{11} by *Bidirectional*

Chapter 6

Conclusion and Future Work

In this thesis, we have examined how keyword queries can be expressed and supported over temporal relational databases. We introduced a new representation for users to specify their search target, associate keywords with time constraints and indicate temporal relationships between keywords. This enables flexible querying of complex temporal relationships in the databases. We incorporate overlapping interval partitioning into the keyword inverted lists to filter nodes that do not satisfy the time constraints. We have designed an efficient ATQ algorithm that incorporates a target-oriented search process and time-aware pruning to retrieve answers to these queries. Experimental results on 3 datasets showed that the proposed approach outperforms current state-of-the-art keyword search methods, and the answers returned by ATQ algorithm are more meaningful.

For future work, we plan to extend temporal keyword queries to handle uncertainty. Since many applications contain uncertain data, for example, in *Clinic* database, the start time and the end time of the symptoms are uncertain. Thus, we want to take this uncertainty into consideration when answering the queries.

Bibliography

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *IEEE ICDE*, pages 5–16, 2002.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective xml keyword search with relevance oriented ranking. In *IEEE ICDE*, pages 517–528, 2009.

[4] Z. Bao, Y. Zeng, H. Jagadish, and T. W. Ling. Exploratory keyword search with interactive input. In *ACM SIGMOD*, pages 871–876, 2015.

[5] S. Bergamaschi, E. Domnori, F. Guerra, R. Trillo Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. In *ACM SIGMOD*, pages 565–576, 2011.

[6] S. Bergamaschi, F. Guerra, M. Interlandi, R. Trillo-Lado, and Y. Velegrakis. Quest: A keyword search system for relational data based on semantic and machine learning techniques. *VLDB Journal*, 6(12):1222–1225, 2013.

[7] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *IEEE ICDE*, pages 431–440, 2002.

[8] R. Bin-Thalab, N. El-Tazi, and M. E. El-Sharkawi. Tmix: temporal model for indexing xml documents. In *Computer Systems and Applications, 2013 ACS International Conference on*, pages 1–8. IEEE, 2013.

[9] A. Dignös, M. H. Böhlen, and J. Gamper. Overlap interval partition join. In *ACM SIGMOD*, pages 1459–1470, 2014.

[10] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in database. In *IEEE ICDE*, pages 689–700, 2007.

[11] K. Golenberg and Y. Sagiv. A practically efficient algorithm for generating answers to keyword search over data graphs. In *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*, pages 23:1–23:17, 2016.

[12] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *ACM SIGMOD*, pages 16–27, 2003.

[13] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *ACM SIGMOD*, pages 305–316, 2007.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[15] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[16] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, and R. D. Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[17] M. Kargar, A. An, N. Cercone, P. Godfrey, J. Szlichta, and X. Yu. Meanks: Meaningful keyword search in relational databases with complex schema. In *ACM SIGMOD*, pages 905–908, 2014.

[18] K. Kulkarni and J.-E. Michels. Temporal features in sql:2011. *SIGMOD Record*, 41(3):34–43, Oct. 2012.

[19] F. Li and H. Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[20] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: Efficient and adaptive keyword search on unstructured, semi-structured and structured data. In *ACM SIGMOD*, 2008.

[21] X. Li and W. B. Croft. Time-based language models. In *ACM CIKM*, pages 469–475, 2003.

[22] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *ACM SIGMOD*, pages 115–126, 2007.

[23] E. Manica, C. F. Dorneles, and R. Galante. Supporting temporal queries on xml keyword search engines. *Journal of Information and Data Management*, 1(3):471, 2010.

[24] G. Özsoyoğlu and R. T. Snodgrass. Temporal and real-time databases: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 7(4):513–532, 1995.

[25] R. T. Snodgrass, I. Ahn, G. Ariav, D. S. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. G. Kulkarni, T. Y. C. Leung, N. A. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. TSQL2 language specification. 23(1):65–86, 1994.

[26] S. Tata and G. M. Lohman. SQAK: doing more with keywords. In *ACM SIGMOD*, pages 889–901, 2008.

[27] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *ACM SIGMOD*, pages 527–538, 2005.

[28] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: a survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.

[29] Z. Zeng, Z. Bao, T. N. Le, M. L. Lee, and T. W. Ling. Expressq: Identifying keyword context and search target in relational keyword queries. In *ACM CIKM*, pages 31–40, 2014.