A USER-DRIVEN ANNOTATION FRAMEWORK FOR SCIENTIFIC DATA

by

Qinglan Li

Bachelor of Engineering, Northern Jiaotong University, 1996

Master of Science, Central Michigan University, 2002

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences

in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES, DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Qinglan Li

It was defended on

June 19th 2013

and approved by

Alexandros Labrinidis, University of Pittsburgh

Panos K. Chrysanthis, University of Pittsburgh

G. Elisabeta Marai, University of Pittsburgh

Christos Faloutsos, Carnegie Mellon University

Dissertation Advisors: Alexandros Labrinidis, University of Pittsburgh,

Panos K. Chrysanthis, University of Pittsburgh

A USER-DRIVEN ANNOTATION FRAMEWORK FOR SCIENTIFIC DATA

Qinglan Li, PhD

University of Pittsburgh, 2013

Annotations play an increasingly crucial role in scientific exploration and discovery, as the amount of data and the level of collaboration among scientists increases. There are many systems today focusing on annotation management, querying, and propagation. Although all such systems are implemented to take user input (i.e., the annotations themselves), very few systems are user-driven, taking into account user preferences on how annotations should be propagated and applied over data. In this thesis, we propose to treat annotations as first-class citizens for scientific data by introducing a user-driven, view-based annotation framework. Under this framework, we try to resolve two critical questions: Firstly, how do we support annotations that are scalable both from a system point of view and also from a user point of view? Secondly, how do we support annotation queries both from an annotator point of view and a user point of view, in an efficient and accurate way?

To address these challenges, we propose the *VIew-base annotation Propagation* (ViP) framework to empower users to express their preferences over the time semantics of annotations and over the network semantics of annotations, and define three query types for annotations. To efficiently support such novel functionality, ViP utilizes database views and introduces new annotation caching techniques. The use of views also brings a more compact representation of annotations, making our system easier to scale. Through an extensive experimental study on a real system (with both synthetic and real data), we show that the ViP framework can seamlessly introduce user-driven annotation propagation semantics while at the same time significantly improving the performance (in terms of query execution time) over the current state of the art.

Keywords: User-driven, Annotation, Scientific Data, Scalable Data, Big Data, Annotation Queries, Caching, DBMS.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

PREFACE

I would like to express the deepest appreciation to my Advisors: Dr. Alexandros Labrinidis and Dr. Panos K. Chrysanthis, who have continually supported me throughout all these years. Without their guidance and persistent help this dissertation would not have been possible. I would also like to thank all my committee members for their useful comments, remarks and advice. Parts of this work were joint work with Alex Connor. I appreciate his input and suggestions. Furthermore, I would like to thank the people who have given me assistance and feedback to complete this thesis.

A special thanks goes to my parents for their endless love and support. Last but not least, I would like to thank my beloved husband and son, who have supported me throughout the entire process. I will be grateful forever for your love.

1.0 MOTIVATION

We are witnessing an explosion of the pace of discovery and innovation in science research, accelerated mostly by the use of information technology in all parts of the process. This is true across all sciences, from gene sequencing and drug discovery to weather/climate modeling and the exploration of the Universe.

Without a doubt, data management is playing a pivotal role in scientific exploration, constantly fueling the high pace of discovery through effective collaboration among scientists. WIRED Magazine, in its July 2008 issue[1], goes as far as to pronounce *the end of the scientific method*: instead of formulating hypotheses to be proved later, all scientists have to do today is to simply mine the petabytes of data at their disposal. Although we do not adopt such an extreme viewpoint, we firmly believe that in addition to efficiently managing the tsunami of experimental data generated, data management is extremely useful to scientific discovery, because it facilitates *effective collaboration among scientists*. In particular, data management techniques are used to record *data provenance/lineage* [55] and also to support *annotations* [30, 25]. Data provenance essentially keeps track of where the data is coming from (and what transformations it has been through), whereas annotations enable users to record additional information about the data stored and propagate this information to all "related" data items.

- ***Data annotation***: annotations (i.e., comments or tags) are associated with data items stored in a database or distributed data stores, and are propagated along with the data when this data is used or retrieved as part of queries. Data annotations effectively organize and integrate notes and thoughts, traditionally recorded in individual scientists' notebooks, making them easily sharable among scientists (as well as general users) in a controlled manner.

[1] http://www.wired.com/wired/issue/16-07, *The End of Science*, WIRED, 16.07, July 2008.

- ***Data provenance/lineage***: data lineage is the sequence of transformations that a given data item has been through (i.e., recording its "genealogy"). Tracking data lineage (that includes data sources, transformations and annotations) is crucial for the repeatability and the authentication of scientific discoveries. It also enables the quick and easy identification of derived works, that need for example to be modified if the data/prior work is updated in the future.

Although such technologies exist today, they are not fully capable to address the needs of scientists. As the amount of data and the level of collaboration among scientists increases, there are more needs of capture, curation, storage, search, sharing, analysis, and visualization of large sets of data. In addition to the increased complexity, the sheer volume of available/generated data forces a rethinking of traditional data management tools.

The main motivation for this work came from the **DataXS** project [83, 81, 82, 84, 90, 69], which was part of the Center for Modeling Pulmonary Immunity (CMPI). The Center was a joint effort between the University of Pittsburgh, Carnegie Mellon University, and the University of Michigan, bringing together experimentalists and modelers to study pulmonary immunity in response to three bio-defense pathogens (the influenza A virus, Mycobacterium tuberculosis, which causes TB, and Francisella tularensis, the bacterium responsible for tularemia). The center capitalized on a longstanding tradition of collaboration between immunologists and mathematicians and computer scientists for the purpose of developing mathematical models of the immune system. The developed data exchange server (DataXS) played a critical role in data sharing, storage, and communication.

While building DataXS, we have identified a challenge, drawn from the requirements gathering and from our experience from early prototypes of DataXS. Let us assume (as illustrated in Figure 1) that there are two machines (1 and 2) that belong to the ADMT Lab, and data files are associated with these machines: Files 1120, 1121, 1122 are already added in the database and File 1123 that has not been added yet. A user wants to add annotation N to data files that were created between 8/31/06 and 9/15/06 (saying, for example, there was a problem with the calibration of the machine). Under traditional annotation propagation, only existing files will be associated with annotation N. We want to find a way to define the target of annotation N to include files that are added later but still fit the original target of the annotations (i.e., files that were created between 8/31/06 and 9/15/06).

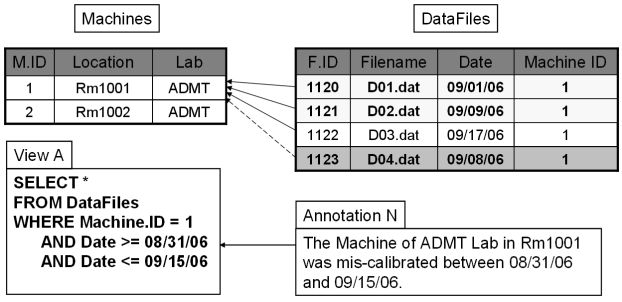


Figure 1: View-based Annotation Propagation

In addition to the need to "capture" annotation targets descriptively, instead of enumerating them, another need we identified has to do with *user-driven annotation management*. Such a focus on the users' needs can streamline privacy protection and also give the ability to scientists and general users to bring previously unrelated data together (through user-driven ways of propagating annotations). Finally, user-driven features can also help users when retrieving data (and associated annotations).

Solving similar needs to the DataXS project, but for a different domain and with many additional unique challenges is the **AstroShelf** project [93, 91, 92, 87, 66], which is targeted to manage the growing onslaught of astronomical data. Astronomy lacks an easy-to-use and scalable way to collect and distribute expert information about objects from data sets of tens of thousands to billions of individual events and objects. Over the next decade, the amount of information available to the typical astronomer will grow by two orders of magnitude both in raw data size and in the number of objects. One important component of this project is an annotation framework to enable linking of observations to specific experiments, models, or other observations. How to manage and represent the annotations of those large scales scientific data is a big challenge.

Biology and Astronomy are just two examples of sciences where machine-generated data and metadata become more common-place and the volume of annotations and data items, as well as annotation views, is becoming enormous. Big data [123] refers to a collection of tools, techniques and technologies for working with data productively, at any scale. In "Data, data everywhere" of the Economist 25 February 2010 issue, it is stated that the amount of digital information increases

tenfold every five years. By 2013 the amount of traffic flowing over the internet annually will reach 667 exabytes. A vast amount of that information is shared, and that only adds to the often complained about "information overload" problem. To solve the problem well, big data must be seen as more than simply a issue of size; it is an opportunity to find insights in new and emerging types of data and content, to answer questions that were previously considered beyond our reach. The work on big data allows correlations to be found to "spot business trends, determine quality of research, prevent diseases, link legal citations, combat crime, and determine real-time roadway traffic conditions" [15, 35, 16] among many interesting use cases.

The challenge undertaken in the thesis is how to deal with large-scale data with annotations, more specifically, how to retrieve and manage data associated with annotations in a quick and effective way. When considering big data, it takes a lot of effort of evaluation, migration, construction and integration. However, it should address the existing deficit of traditional applications and bring more research opportunities.

All in all, through these interesting projects, and by the implementation of a web-based data exchange server prototype, we were able to identify the technological "gaps" in the current state-of-the-art and propose to address these as part of this Ph.D. dissertation.

1.1 PROBLEM STATEMENT

We propose to treat annotations as first-class citizens for scientific data management. Towards this, we propose to address two distinct usage patterns related to the specification and the management of annotations within a Database Management System (DBMS).

1. ***Support for annotations that are scalable both from a system point of view and a user point of view.*** The first usage pattern that we observed is that big data are the current trend of scientific data (such as medical data or astronomical data) and general data (such as social media data or movie data, etc.) Hundreds of thousands of annotations are linked to data, so it is typical for people to be lost in the sea of annotations. How can a system handle such large-scale data is an interesting question, and also has practical applications. On the other hand, users want

to retrieve information via annotations quickly and accurately. The annotation associated with data should be the most related, and helpful in understanding the data it is associated with.

2. ***Support for propagation and querying of annotation in annotator/user-defined ways.*** The second usage pattern is that annotators want to specify who can read their annotations, when to display their annotations, where to propagate their annotations, etc. On the other hand, users, i.e., those performing the queries, want to express their preference on whose annotations they want to read (such as users trusting high-rated reviewers' annotations), how to retrieve annotations (such as how many levels they want to check, like in a social network, check how many hops of friend-friend's comments), or what time period of annotations to search. Such user-driven preferences (from both the annotator and the querier point-of-view) are critical to annotation management and querying.

1.2 USING VIEWS

We propose to use the concept of *database views* [97, 41] as the building block to implement the technologies mentioned above. Database views can be used at a high-level to describe the results of a database query. For example, instead of attaching a comment about mis-calibration to individual files (and miss files that are added in the future), using views enables the system to record this annotation in a single location (the view) and to also associate this annotation with files matching the view definition in the future.

In the example illustrated in Figure 1, instead of simply applying annotation N to the individual files that are initially in the system and meet the description (e.g., 1120, 1121, 1122), we apply it to the *query or view* that describes the *property* files that need to have in order to get annotation N. This allows the system to properly mark File 1123 with annotation N, even if it is added later. This way, the annotation has been properly propagated to all files, even if they are added later. As we have discovered, out of order data entry is typical in scientific data management, so such cases are quite often, and therefore, addressing them would greatly improve the quality of the data stored in the system.

In this work, we introduce the notion of *"view-based annotation propagation"* as an alternative

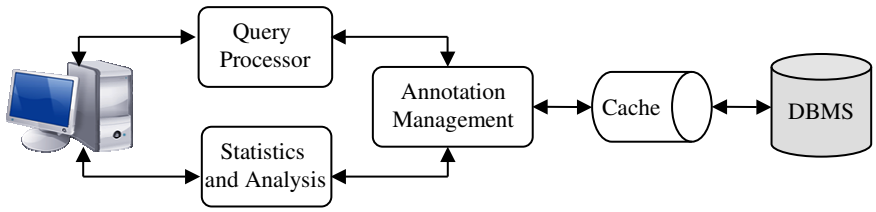


Figure 2: The ViP System Model

way of propagating annotations. The basic idea is that with view-based annotation propagation, users are specifying a view (i.e., a query) over which annotations should be propagated, *now and in the future*. Although the "now" semantics make View-based annotation Propagation (ViP) identical to Traditional Annotation Propagation (TAP), the "future" semantics is what differentiates the two. Under TAP, a query would have to be executed again and again for the annotations to be propagated in the future (while making sure to eliminate duplicates), whereas under ViP the query only needs to be defined once. Essentially, ViP can be considered as supporting *continuous annotation queries*.

Using the views concept, our work builds on over three decades of research [77, 76, 78, 129, 62, 60, 33, 73, 68, 61] on view management while providing a novel and clear framework, ViP, for the management of large-scale annotations and their propagation in an efficient way. Although the concept of database views is a powerful mechanism, we need a realistic way for users to utilize views in the proposed ViP framework. Clearly, they are not to be expected to provide annotation view definitions in SQL, though at the backend we propose to modify standard SQL to support annotation features.

In our data-sharing platform DataXS, a user can easily specify filtering conditions to locate certain data items. These filtering conditions are essentially a query (i.e., a view) and can be used by the ViP framework. As shown in Figure 2, the user interface takes the user input and sends the requests to the Query Processing module, which will forward the operations of annotations and annotation views, such as insertion, deletion and update, to the Annotation Management module. The queries are executed as two parts, one part is to retrieve the data items, and the other part is to

retrieve the associated annotations. There is a cache to enhance the system performance. Finally, users will receive the result presented as data + annotation(s). We will discuss more details about the system model in the next chapter and the user interface in Section 6.6.

1.3 CONTRIBUTIONS AND EVALUATION

We present ViP, a novel annotation framework that introduces new annotation definition and propagation methods based on views, utilizes views both as a specification mechanism and as a user-interface mechanism, and employs caching [20] for improved performance compared to the state of the art. This research project has both theoretical and practical contributions as follows:

1. based on our experience from real system implementations, we introduce new annotation definition and propagation methods, suitable for scientific data,
2. we introduce ViP-SQL, an SQL-based query language supporting annotation propagation,
3. we introduce user-driven features that enable users to personalize annotation propagation,
4. we introduce the use of views as formal mechanism to implement the new annotation definition and propagation features and also as a user-interface abstraction,
5. we propose a benchmarking of three types of annotation queries, and discuss the algorithms needed to manage data and annotations in a scalable way,
6. we utilize techniques such as caching to significantly improve the performance of query execution over the state of the art,
7. we experimentally evaluate the proposed ViP framework using a real system implementation with real and simulated workloads.

1.4 ROADMAP

The rest of the dissertation is organized as follows. The next chapter presents the ViP system model. In Chapter 3 we introduce the feature comparison of ViP with the related work. The details of the proposed annotation framework (User-driven Time semantics and Network Semantics),

along with the formal ViP-SQL definition are presented in Chapter 4. Three types of queries, along with annotation management and query processing are discussed in Chapter 5. In Chapter 6 we describe the details of implementation, including the user interface and a graphical representation of server statistics with regards to annotations. The experimental evaluation is conducted and analyzed in Chapter 7, three individual data sets are evaluated from different dimensions. We conclude our work in Chapter 8 and discuss the future plan beyond the scope of this thesis.

2.0 SYSTEM MODEL

In this thesis, we define *annotation* as anything helpful in understanding data, such as comments, tags, labels, or experiment results etc. An annotation can be a 100MB size video or audio clip, or even a bigger size metadata file. We will use a simple example in the following paragraphs to illustrate how annotations are associated with data.

2.1 ANNOTATION REPRESENTATION METHODS

There are several projects that deal with annotation propagation and management, for example, DBNotes [25], Mondrian [57], ULDB [23], bdbms [53], and MMS [115].

In DBNotes, every relational table column is associated with one additional annotation column. This traditional naive method is illustrated in Figure 3A. Data items $d_1, d_2, ...d_4$ are listed in the column of "Data", while annotations such as "Hail to Pitt", "Pittsburgh" and "Maryland" are listed in the column of "Annotations". If a data item does not come with an annotation, it will be an empty cell in the annotation column. If a data item has multiple annotations, there will be multiple duplicated rows for this data item. One can easily realize that this method will be a huge waste of space when the annotations are sparse.

MONDRIAN uses an improved scheme, of colors and blocks, to present annotations. It associates one extra annotation column to each relation, plus one shadow column for each attribute to indicate whether the annotation refers to the respective attribute or not. By separating the data from the annotation representation, it saves space and keeps tables normalized. Data that has no annotations (e.g., d_4) will not be addressed in the annotation representation. However, if there is more than one tuple associated to one annotation, there should be more than one annotation tuple

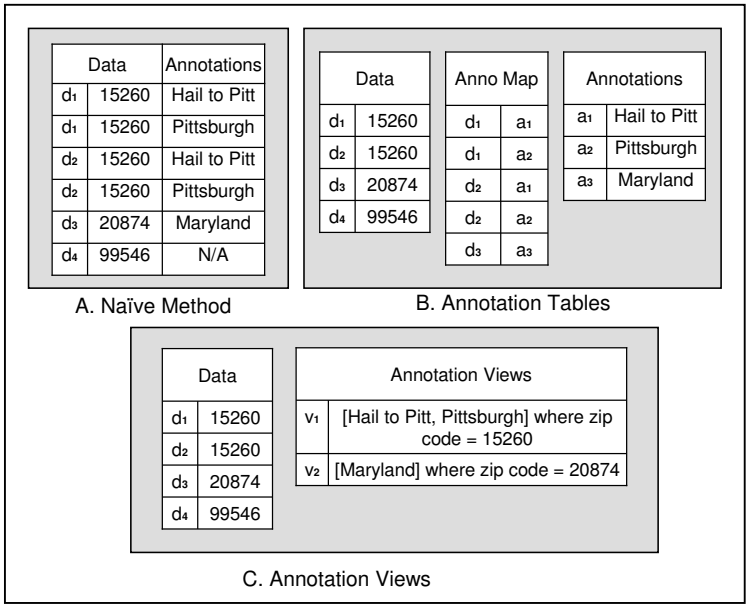


Figure 3: Annotation Representation Methods

in the annotation representation, like the sets (d_1, a_1), (d_1, a_2), (d_2, a_1), and (d_2, a_2). Considering that this method still uses the same structure as the data table to represent annotations, we classify it as the *traditional naive method*.

Other work such as MMS [115] showed significant benefits over the above systems both in query execution times and storage space usage because of its scheme to treat a query as a value. There is an *annotation table* instead of additional annotation columns. This method is illustrated in Figure 3B. Data items are listed in the "Data" table, while annotation association relations are presented in the "Anno Map" table. There is also an annotation metadata table. Clearly, this scheme reduces the redundant space used by the naive method. Also, it lowers query execution time even when we consider the cost of updating the annotation index structure and querying additional tables of metadata. ULDB [23] and bdbms [53] also use annotation tables for annotation storage and management. It is expected that such schemes outperform traditional methods if the association between the data and the annotations is not uniformly distributed and it not static.

Annotation tables make annotation representation flexible and scalable. Our framework, ViP, uses this scheme as well [83]. More than that, in ViP, we treat a view as an annotation registration,

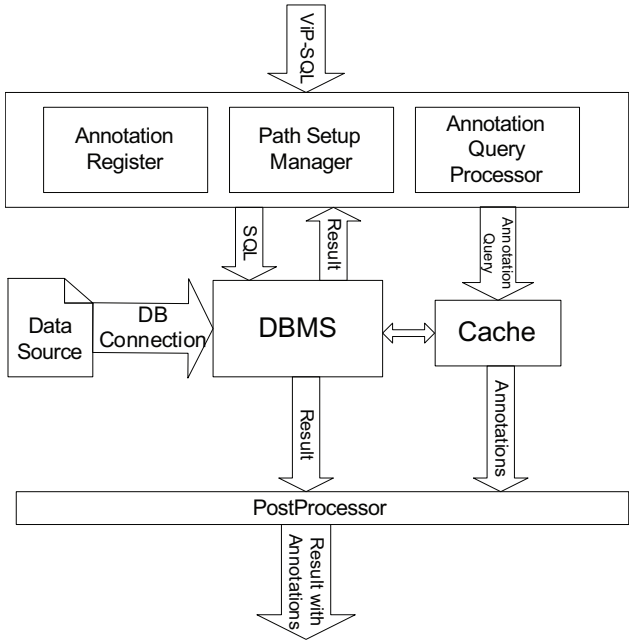


Figure 4: High-level System Architecture

which will be stored in “Annotation Views” tables as shown in Figure 3C. In addition to the “Data” table, the annotations are recorded for certain data items with a defined query value. It is one step further, compared to the fixed structure of annotation table, as this method makes annotations more flexible and comprehensive, thus it can present complex condition definitions. We expect that in a dynamic, constantly evolving environment (both for data and annotations), the cost of maintaining and querying should be less with ViP compared to previous methods.

2.2 SYSTEM ARCHITECTURE

ViP deploys the architecture shown in Figure 4. Queries and Annotation registrations (annotation definitions) issued from clients are rewritten into SQL queries evaluated by the annotation query processor, then handled by the annotation register and the path setup manager. They are pushed forward to a database server with a cache in the middle, which will work to optimize system

performance. Annotation tables include annotation registration tables and auxiliary tables. The resulting annotation set is merged with the regular query results by the postprocessor for matching and presentation. The processing procedures and methods will be discussed in Section 6.3. We will discuss more details of the system architecture in Section 6.4 and introduce a simple workflow for the system. One of our implementation, the DataXS application, "fits" on top of this framework, providing a point-and-click user interface. In Section 6.6, there is a demostration of the functionalities of the ViP framework.

2.3 ANNOTATION-OBJECT DIAGRAM

If we view the proposed the ViP framework from an Entity-relationship (ER) model point of view [109], then we get the ER diagram of Figure 5. Each annotation or object has its own annotation/object ID and content. One annotation may annotate one or multiple objects (data), whereas one object may have more than one annotation. ViP allows annotations to annotate not only objects, but also annotations themselves.

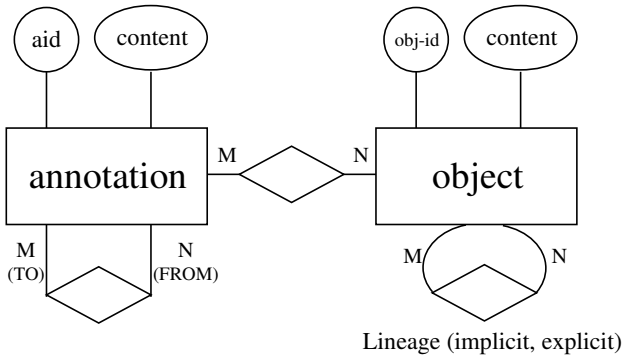


Figure 5: ViP Annotation-Object Diagram

The *data lineage* of objects is treated as implicit annotations associated to object/data, and additionally, it can also be explicitly illustrated by annotation paths. The data provenance is more complicated as the flow of processing may involve intermediate staging and data transformations that come from programs external to the DBMS, thus users' annotations play an important role in

understanding data sources. In this thesis, there is no automatic data provenance tracking, instead, we provide a user-defined way to understand and trace the data lineage and provenance.

2.4 SUMMARY

In this chapter, we briefly reviewed current annotation representation methods, and introduced our annotation views table as an efficient and compact representation solution. We also presented the system architecture. Finally we illustrated the ER model of the ViP framework.

3.0 RELATED WORK

In this chapter, we discuss the related work in annotations and annotation management, user-driven data management, social networks, and big data. In addition to introducing their salient features, we compare them with our ViP framework.

3.1 ANNOTATIONS

In our work, an annotation is being treated as an object, and could be comments, files, or anything helping to understand the data.

In natural language processing (NLP) [114, 124], annotation could be a useful tool to automatically produce semantic representations. Linguistic annotations corpora will have most basic types of semantic information annotated according to high-quality schemes (which often involve human experts or massive crowd-sourcing efforts). Crucially, all individual annotations, although unified, will be kept separately in order to make it easy to produce alternative annotations of a specific type of semantic information (word senses, anaphora, etc.) without modifying annotations at other levels [48].

Compared to NLP, our annotations are more flexible. They could be in a very simple form, such as a line of plain text, or they could be a set of multiple types of files to describe/explain the data. There is no restriction on our annotations, either in size or type.

3.2 DATA PROVENANCE AND ANNOTATIONS

The problem of provenance (also known as lineage) of data has been studied for many years [119, 30, 45, 44]. Provenance, which means "origin" or "source", could be used to trace data flow and in addition, could be used to verify the authenticity of the data source, in which case, it could be used to explain why and where the data coming into their current status.

In [32, 29], the authors described an approach tracking the user's actions while browsing source databases and copying/pasting data into an integrated database, which forms a curated database. An update log (provenance table) is used to record every operation.

In [39], the authors introduced DBNotes, an annotation management system that attaches notes to every value in a relation. When a relation is queried, DBNotes propagates the attached notes to the result of the query in three different schemes: *default*, *default-all*, and *customized* based on provenance. As a consequence, the notes associated with a value in the result of a query show the provenance of the value. This is the *where-provenance* problem [31]. Since it addressed the annotation propagation issue and introduced annotation management for scientific databases, this work inspired our work on exploring how to attach and propagate annotations with data, according to user preferences.

The Trio project [19, 23, 107] is a database management system built to address data uncertainty and lineage. The extended query language, TriQL modifies the semantics of SQL to take uncertainty and lineage into account, and let data, uncertainty, and lineage work together to provide confidence values for data. Previous projects applied data provenance concepts in web information gathering and criminal data tracking. It inspired us to use a user-specified confidence threshold in annotation propagation.

[53, 127] are projects supporting annotations for e-Science data. bdbms is a prototype that supports annotation and provenance management. The authors discussed annotation storage and indexing similar to our structure. However, they are not user-driven, and not designed for continuous data, so no annotation will be propagated to future data (i.e., they lack the time semantics introduced in this thesis). In the survey by Simmhan et al. [112], there are additional related works in this area, but they follow what we termed in the previous chapter as "traditional annotation propagation".

More theoretically, [32] explored the *side effects* of deletion annotation through propagation, and [40] discussed propagation analysis of annotations under the key preservation condition for both the *side-effect* problem and the *annotation placement* problem. Tan [118] discussed the problem of containment in queries with annotation propagation. [40] analyzed the complexity of annotation propagation and suggested a key preserving condition on SPJ views.

In this thesis, we consider data provenance as a special kind of annotation and propagate such annotations accordingly.

3.3 USER-DRIVEN DATA MANAGEMENT

The Advanced Data Management Technologies (ADMT) laboratory at the University of Pittsburgh [1] has focused on projects related to query processing, data sharing, and annotation management. A key idea behind the research is to find ways to satisfy user-specific requirements while processing the data efficiently [106, 80, 126, 101, 74, 86, 103, 125, 88, 110, 117, 59, 75, 102, 100, 99, 101]. From sensor nodes deployment, to energy consumption strategy, and from web crawling, to query scheduling, the goal is to maximize the satisfaction of users over their different quality metrics. In one such project, **User-centric Data Management**, users are empowered to specify their preferences for the different dimensions of quality (Quality of Service, Quality of Data, Quality of Information) through an intuitive, integrated framework and influence resource allocation decisions according to their preferences.

In [72], the authors developed a personalization framework in database systems based on user profiles. The preference model assigned to each atomic query carries a personal degree of interest, so the system can calculate a comprehensive interest degree for any query. It inspired our design of introducing personalization or user-driven views into annotation propagation, with the real life requirements materializing from the design and implementation of our DataXS prototype. In this thesis, being user-driven is one of the top priorities for annotation management.

3.4 ANNOTATION MANAGEMENT FEATURES

3.4.1 Standard Annotation Management Features

There are many systems that support some of the features that are part of the ViP framework, but as far as we know, there is no single system that supports all the functionality. For example, most current systems do not support annotations that are also valid in the future (Table 1). The only exception is MMS [115], which supports future time semantics (i.e., without giving the user the option to define the time) in an implicit way, but is not explicitly defined. In addition, we propose explicit user-defined network paths, which will be discussed in Section 4.2.

Standard Features	**DBNotes [25]**	**Mondrian [57]**	**ULDB [23]**	**bdbms [53]**	**MMS [115]**	**ViP [83, 81]**
Annotation	Yes	Yes	Confidence	Yes	Yes	Yes
Provenance	Yes	Yes	Lineage	Yes	Yes	Yes
Time Semantics:						
· Implicitly defined	No	No	No	No	Yes	Yes
· Explicitly defined	No	No	No	No	No	Yes
Network Semantics:						
· Implicitly defined	Limited	Limited	Limited	Limited	Yes	Yes
· Explicitly defined	No	No	No	No	No	Yes
Propagation Type	Eager	On-demand	On-demand	Eager	On-demand	Both
Annotation Storage	Naive	Naive	x-relations	Anno. table	q-type	A-table
Scalability	Small	Medium	Medium	Medium	Large	Large
Query	pSQL	Color algebra	TriQL	A-SQL	Predicate	ViP-SQL

Table 1: Standard Annotation Management Features Comparison

ViP builds explicit paths for annotation propagation. It also allows users to protect data privacy on these paths, that is, each user can have private paths to propagate annotations. Although existing systems support implicit annotation propagation paths, none except for our proposal supports explicit, user-defined annotation propagation paths (Table 1). In Chapter 4 we will discuss these semantics in detail. [58] provided update exchange with mappings and provenance. It is a

kind of implicit provenance tracking. The Open Provenance Model (OPM) in [89, 122] proposed a provenance model, focusing on provenance tracking and management. In this dissertation, we want to focus on explicit annotation propagation.

ViP aims to support large-scale annotation management. Towards this, ViP employs a hybrid propagation scheme, while [25, 53] uses eager propagation, and [23, 57, 115] uses an on-demand scheme. We will discuss the propagation types in more detail in Section 6.2.1.

3.4.2 User-driven Annotation Management Features

ViP brings user-driven features in many aspects of annotation management that are not considered in most related work as shown in Table 2. The *valid time* in time semantics is a unique feature we propose in Section 4.1. ViP also enables users to specify the propagation method. In DBNotes [25], users can specify a *custom* propagation scheme to bind the source and target tuples while there is a join operation, so that the annotations that are associated to the source tuples will be propagated to the target tuples. ViP provides a stronger and more complex scheme, that is the *annotation path*, which will be discussed in Section 4.2. In the table we use "N/A" to represent a feature which was not included in the system design, so there is no such comparison available.

Some systems consider *access control* at the data level, or even at the update authorization part [53]. Instead, we propose to fully support this feature in a broader domain, on annotations, annotation views, and annotation paths. This is different than traditional access control, since access control on annotation views (given user-driven time semantics) and on annotation paths (given user-driven network semantics) essentially means who can "execute" the annotation propagation mechanism, not the access control on the data itself. We will present this in more detail in Sections 4.3 and 4.4.

In [51, 52], the authors continued their work on the bdbms system [53], where they proposed and modeled a Directed Acyclic Graph (DAG) generated from user-defined dependencies. They mentioned "snapshot, view and join" as annotation types that inherit different human behaviors. Similar to the usage patterns mentioned in Chapter 1, they found that human actions (more broadly real world activities) have the concept of "real-world dependencies". In [50], they evaluated experimentally the performance of HandsOn DB which supported their dependencies theory. Though

User-driven Features	**DBNotes [25]**	**Mondrian [57]**	**ULDB [23]**	**bdbms [53]**	**MMS [115]**	**ViP [83, 81]**
Time Semantics:						
· Valid Time	N/A	N/A	N/A	N/A	No	Yes
Network Semantics:						
· Propagation Method	Yes	N/A	N/A	Limited	No	Yes
Access Control:						
· on Annotations	N/A	N/A	N/A	Limited	No	Yes
· on Annotation Views	N/A	N/A	N/A	No	No	Yes
· on Annotation Paths	N/A	N/A	N/A	No	No	Yes

Table 2: User-driven Annotation Management Features Comparison

they discussed dependencies a lot, they did so mostly from a traditional data provenance and lineage point of view.

3.5 SOCIAL NETWORKS

Social and information networks are a fundamental medium for the spread of information, ideas, viruses, and behavior. Transmission of infectious diseases, propagation of information, and the spreading of ideas and influence through social networks are all examples of diffusion. A cascade graph can be used to represent the "contagion" across the network. As information or actions spread from one node to other nodes through the social network, a cascade is formed [56, 27, 104, 105]. Although this is somehow similar to the annotation propagation we are proposing in this thesis, nonetheless, they are different in many other ways.

Existing and widely deployed social networks allow users to annotate all sorts of information (from photos to web sites) such as Facebook [4], Twitter [11] or MySpace [8]. Annotation works as an assistant tool to help users understand the data items. However, in the scientific community, annotation is not only important in understanding the data, but it is also unique in the ability to trace data provenance, and in addition, propagating annotations to appropriate data, which is useful, but

Features	**ViP Framework** [83, 81]	**Social Networks** [4, 11, 8]
Time Semantics:	Yes	N/A
Network Semantics:	Yes	N/A
· Propagation	Yes	N/A
· Path Strength	Yes	N/A
Access Control:	Yes	Yes
· on Data Item	Yes	Yes
· on Annotation	Yes	Yes
· The Way to Access (Path)	Yes	N/A

Table 3: Comparison of ViP and Social Networks

not necessary in social networks.

In social networks, it is usual to build relations from explicit assertions by users that they have some relation (such as co-authors or friends or follower) with other users or by the implicit evidence of such relation [70, 130]. However, such kind of inference will not suggest to propagate annotations or to propose *explicitly* multiple levels of inferences. Along the same lines, data provenance is expressed in the format of links in social networks; it is not *explicitly* presented to users. In summary, we compare our system with social networks[1] in Table 3.

3.6 BIG DATA

Big data is an important characteristic of scientific data as well as general-purpose data, in our current information-based economy and way of life. There have been many articles, including in the popular press, about the challenges and opportunities of Big Data. There are many solutions that deal with big data challenges, from a data management point-of-view; it is beyond the scope of this thesis to perform a survey, but for completeness we present a few representative works.

[1]Social networks are improving everyday; we have the comparison based on the current development and to the best of our knowledge

3.6.1 NoSQL Systems

NoSQL [9], commonly interpreted as"not only SQL", is a broad class of database management systems identified by non-adherence to the widely used relational database management system model. It is defined as the next generation databases mostly addressing some of the points: *being non-relational, distributed, open-source, and horizontally scalable*.

NoSQL databases are not built primarily on tables, and generally do not use SQL for data manipulation. NoSQL database systems are often highly optimized for retrieval and appending operations and often offer little functionality beyond record storage (e.g., keyvalue stores), making them especially fit for big data. The reduced run-time flexibility compared to full SQL systems is compensated by marked gains in scalability and performance for certain data models [120, 108].

Google has developed BigTable [37], which is extensively used in Google's own operations. At the same time, Amazon designed Dynamo [46], a proprietary, highly available eventually-consistent key-value structured storage system. These two leading web applications represent a trend: the cost-effective management of data behind modern web and mobile applications is gaining more attention than ever.

3.6.2 Big Data Computing Services

Big data solutions often go hand-in-hand with cloud solutions, which has led to the development of big data computing and storage services [28]. We list a few representative examples below.

1. Amazon web services [3], a web service that provides re-sizable computing capacity in the cloud. It fully supports Amazon DynamoDB. Although it makes web-scale computing easier for developers, it is commercially developed and used.
2. Apache Hadoop [111, 26] is an open-source software for reliable, scalable, distributed computing of large data sets. The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. It is free.
3. Google Cloud Platform, provides services such as Big Data Cloud Analytics, Cloud Storage, BigQuery etc. In 2012, Google released Spanner [42], a scalable, multi-version, globally

distributed, and synchronously-replicated database, using atomic clocks and GPS to provide a time API. It is the successor to BigTable.

3.6.3 Big Data Sets

There are many public big data sets available for research or application purpose. We found some of them are suitable for ViP framework, such as

- Amazon Public Data Sets [2]
- Twitter social graph [12]
- IMDB Data Sets [5]
- MovieLens Data Sets [7]

In Section 7.7, we will discuss some of them in detail, choose one of them to apply in our evaluation, and enhance the data attributes to have annotation features. The results show that our framework is working smoothly even with large-scale data sets.

3.7 SUMMARY

In this chapter, we discussed the related projects about annotation, data provenance and user-driven management. In each case, we introduced other systems' features as well as our unique features. We summarized the comparison from two dimensions: Standard Annotation Management Features and User-driven Annotation Management Features. Finally we discussed current related technologies such as social networks and Big Data.

4.0 THE VIP FRAMEWORK - ANNOTATION POINT OF VIEW

In this chapter, we present the details of our ViP framework from an annotation-point of view, i.e., focussing on what happens when annotations are added (or deleted or updated) in our system. We describe user-driven time and network semantics for annotation propagation in Sections 4.1 and 4.2. We propose user-driven access control on annotation views in Section 4.3, and on paths in Section 4.4. In each case, we present the corresponding statements for ViP-SQL, our proposal for a simple extension to SQL that would handle the new semantics. Formal definition of ViP-SQL is given in Section 4.5.

In the following chapter, Chapter 5, we discuss the views and queries both from annotators' and users' point of view. We also present three types of queries we propose in this framework. A detailed discussion of algorithms and technique of implementation is given in Chapter 6.

4.1 USER-DRIVEN TIME SEMANTICS

One of the usage patterns we observed in our DataXS project was that *experimental data was almost always entered in the database in an order different than the one it was generated*. In fact, even data about the same experiment could be entered at completely different times, since more than one lab was involved in generating the data (for example, one lab would generate the luminex data whereas a different lab would produce microarray data for the same tissues). Looking at annotations, this means that if one wanted to annotate data from a particular experiment with an observation about the tissues, it would **not** be enough to do this once, as additional experimental data may be added into the database later (which would not automatically "inherit" the annotation). Since users have different understanding/explanations of why/how certain biological process un-

fold, it is possible that they also want to personalize the time setting of such annotations (i.e., whether they would apply just now, or also in the future), we refer to this feature as "*user-driven time semantics*".

The main idea behind view-based annotation propagation is that we can attach an annotation to a **view**, i.e., a query definition that corresponds to a set of data items, instead of individual data items. If we do not materialize the view, then the annotations will always be properly associated with the corresponding data items, according to the *valid time* of time semantics.

We propose the concept of *valid time*, which is the validity time interval of an *annotation view* or an *annotation path*. It allows users to specify what time period they want to associate the annotations with corresponding data or to propagate the annotations via a certain path.

When we consider the time dimension of annotation propagation, we can easily distinguish four different cases:

- ***now***, where an annotation is only propagated to data items currently in the database, (e.g., mark all the data that have been processed until today) - this is the approach taken by the majority of annotation management systems,
- ***now*** + ***future***, where an annotation is propagated to data items currently in the database, and also to those that are added in the database in the future, (e.g., assume that we have a microarray scanner which was mis-calibrated on a certain day; when this is first discovered, we want to be able to annotate all experimental data in the system accordingly, but also do this for all data that would fall in this category, but are entered in the system later),
- ***future***, where an annotation is only propagated to data items that are added in the database in the future, (e.g., all the files until today have been fixed, but all files submitted in the future should be marked accordingly),
- ***future interval***, where an annotation is propagated to data items that are added in the database in the time interval a user specifies, (e.g., for one week after the Daylight Saving Time show an annotation that reminds scientists to make sure they have accounted for Daylight Saving Time in experiment settings). Since it is unrealistic to annotate past query results, the valid time period starts from now. Future interval will not be marked as a past time period.

The cases above (illustrated in Figure 6) present the four *valid time* uses in user-driven time

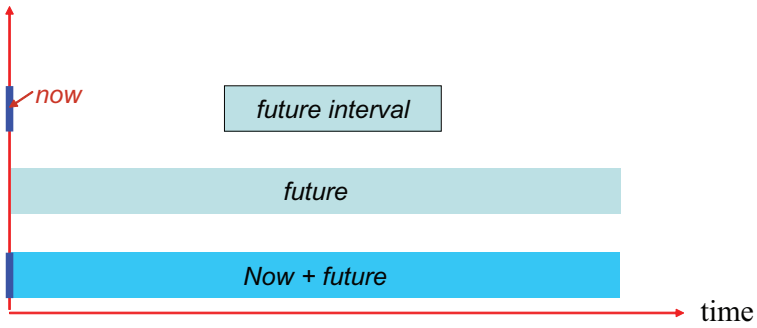


Figure 6: User-driven Time Semantics

semantics of ViP. This is specified by the user for each annotation and works in tandem with using *database views* to describe the annotation targets. Using views allows us to declaratively describe the data to be annotated instead of simply enumerating them. Combined together, we can set, for example, the valid time for an annotation to be $[now, \infty)$, or in a simplified form $[now, \ )$, which means that the annotation will be applied to matching items now and also in the future.

Most of the current annotation management frameworks utilize *now* time semantics, propagating annotations to only existing data items [25, 53, 23, 29, 57]. In contrast, our system supports *valid time* in Time Semantics, which we will assume for the rest of this thesis. Only the work in [115] considers time semantics similar to those presented in this thesis; both approaches use the concept of database views in their frameworks. However, the approach in [115] does not consider time semantic in such a user-driven and explicit way.

Motivating Example #1: To properly motivate the need for time semantics, let us assume a setup like that in our DataXS system, where experimental data are stored in table Experiments and shared among project participants. Let us assume that a contamination happened in the ADMT Lab between Oct 1, 2012 and Oct 20, 2012, and we would like to annotate all experimental data accordingly, with *now* + *future* time semantics. Clearly, if we only attach an annotation to the files matching the ADMT Lab AND happened Oct 1 - 20, 2012, we will miss all the files that are potentially added into the DataXS system at a later time, but still meet these conditions. As we discussed earlier, this is a typical usage pattern, making valid time in time semantics a necessity.

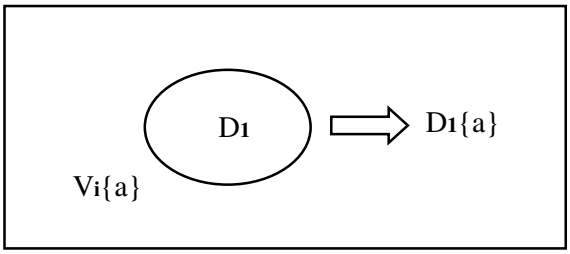


Figure 7: View-based Annotation Propagation: Time Semantics. (Annotation a is associated with view V_i. Data item $D_1 \in V_i$ receives annotation a.)

We can describe such an annotation in *ViP-SQL* as follows:

```
CREATE ANNOTATION V1 ON Experiments
    AS (SELECT ExpID FROM Experiments
        WHERE Lab = "ADMT" and Date >= "10/01/12"
              and Date <= "10/20/12")
    VALUE "ADMT Lab was contaminated between Oct. 1st
              & Oct. 20, 2012. Please use data with caution."
    VALIDTIME [now,  )
```

Given that annotations are associated with views instead of individual data items, the expected behavior in cases of modifications is straightforward (Figure 7):

- INSERT(data) into VIEW:
  if D_1 becomes a member of view V_i (either through insertion or an update or a creation of an annotation view), then it will also be associated with annotation a when it is queried.
- DELETE(data) from VIEW:
  if D_1 is no longer a member of view V_i (either through deletion or an update), then it will not be associated with annotation a.
- DELETE(view):
  if V_i is deleted, then all the data items that were members of V_i and were associated with annotation a will no longer be associated with it.

4.2 USER-DRIVEN NETWORK SEMANTICS

The second usage pattern that we observed was that *there exist many relationships, or paths, between data items that cannot be inferred by the existing database schema*. Such paths materialize because, for example, tissues from multiple, unrelated experiments are processed together, in a single assay (for example, on a single plate that needs to be filled up to minimize costs). To address this, we propose to enable users to specify explicit *annotation paths*, thus allowing for more "interconnections" among data and knowledge. Annotations should be propagated along these paths, reaching "related" data items, as specified by users. Since these paths are essentially forming a network, we refer to this feature as "user-driven network semantics".

Most annotation-enabled systems propagate annotations along data provenance paths. In other words, annotations are propagated over existing implicit annotation propagation paths between source data and derived data (i.e., driven by the database schema and data transformations). Although this can happen over multiple derivation levels, it fails to capture relationships between data items that do not share a common "ancestry" in the database. As we have witnessed from our involvement in the CMPI/DataXS project, this happens often in scientific databases.

The ViP framework empowers users to specify *explicit paths* between data items, thus establishing additional annotation propagation paths. Such explicit paths are defined using views as follows:

- given a source view, V_s
- given a destination view, V_d
- an explicit annotation propagation path $V_s \rightarrow V_d$ is defined, such that any annotation that is added in a member of V_s must be propagated to all members of V_d.

Note that there are no constraints on views V_s and V_d, which means that they can be disjoint, overlap, or even be identical.

Continuing from Motivating Example #1, let us assume another example where we have that the ADMT Lab and the Ross Lab be next to each other, and the ADMT Lab provides the Ross Lab with tissues for model analysis. As such, there is a need to propagate all annotations regarding ADMT Lab experiments to the Ross Lab (to properly record, for example, if there has been any

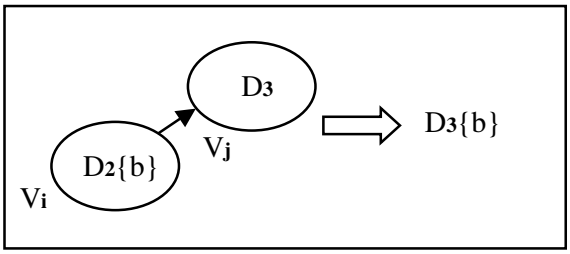


Figure 8: View-based Annotation Propagation: Network Semantics (Disjoint source/destination). There exists an annotation propagation path from V_i to V_j. Data item $D_2 \in V_i$ has annotation b. Data item $D_3 \in V_j$ receives annotation b.

contamination).

We can describe such an annotation in *ViP-SQL* as follows:

```
CREATE ANNOTATION V2
ON Experiments
AS (select Date from Experiments
    where  Lab = "ADMT"
           and Treatment = "Influenza A")
TO Experiments
AS (select Date from Experiments
    where  Lab = "Ross"
           and Treatment = "Influenza A")
VALIDTIME [now, )
```

Considering the general case of using source/destination views to describe explicit paths for annotation propagation, we can see that such paths essentially form a **network**, hence the need for *user-driven network semantics*.

With regards to view membership, we have behavior that is very similar to that in the case of *user-driven time semantics*, as presented in the previous section.

With regards to the relation between the source and destination views, we consider the follow-

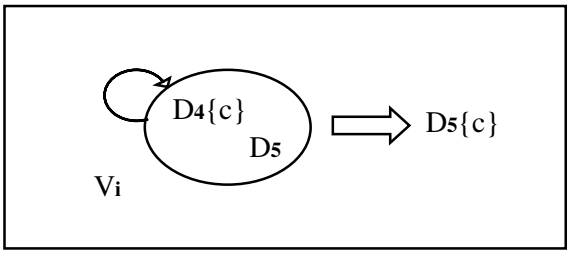


Figure 9: View-based Annotation Propagation: Network Semantics (Identical source/destination). There exists an annotation propagation path from V_i to self. Data item $D_4 \in V_i$ has annotation c. Data item $D_5 \in V_i$ receives annotation c.

ing cases:

- source and destination views are disjoint (Figure 8)
- source and destination views are identical (Figure 9)
- source and destination views are overlapping (Figure 10)

In the context of metadata management, [115] considered implicit paths from queries to queries, but they have not considered the full user-driven network semantics as we do in this thesis. In the context of schema mapping, there are multiple works that consider links of "similar" tables [85, 38].

We believe it is our unique feature to mark the explicit links between annotation views.

4.3 USER-DRIVEN ACCESS CONTROL ON ANNOTATION VIEWS

We advocate that scientific annotation must have a strong user-driven component. First of all, much of the data is not public, so appropriate access controls [67] need to be in place for the raw data, and the annotations on them. Secondly, even for public data, the annotations are often private, since they reflect additional analysis that is not ready to be made available to all. Thirdly, in many cases, even the way that raw data are associated (i.e., by specifying explicit paths for annotation

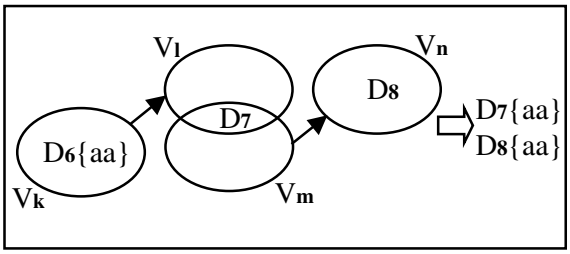


Figure 10: View-based Annotation Propagation: Network Semantics (Overlapping source/destination). There exists an annotation propagation path from V_k to V_l and another path from V_m to V_n. V_l and V_m overlap. Data item $D_6 \in V_k$ has annotation aa. Data item D_7 is a member of both V_l and V_m. Data item D_7 receives annotation aa. Data item D_8 receives annotation aa.

propagation) corresponds to private information that should not be made public. Given all these reasons, the ViP framework includes multiple user-driven features, as they relate to access control. We enumerate them next.

1. For *data items*, the data owner decides who can view the data, the user who sends the query can select whose data to view.
2. For *annotations*, the annotator decides who can view the annotation, the user who sends the query can select whose annotation to view based on the annotator's reputation/confidence.
3. For *annotation views*, the annotator decides who can view the annotation view, the user who sends the query can select whose annotation to view based on the annotator's reputation/confidence. This can also be used in conjunction with groups of users.
4. For *annotation paths*, the annotator decides who can view the annotation propagated by the path, also if the annotation is allowed to be propagated or not. The user who sends the query can select how many levels down the network to retrieve annotations based on the annotator's reputation/confidence.

   We will explain these in detail in the following paragraphs.

First of all, we expect there to be access control at the individual data item level. This is mostly a solved problem, and is handled nicely by relational database management systems. Secondly, we easily implement access control at the level of individual annotations. In other words, when an individual data item receives an annotation from a user, the user can specify who can access the annotation. We support arbitrary user hierarchies (i.e., specific users, groups of users, groups of groups of users, etc.)

We expect the majority of annotations to happen through views, to take advantage of user-driven time semantics. In this case, user access controls are also implemented, with the expected behavior.

4.4 USER-DRIVEN ACCESS CONTROL ON ANNOTATION PATHS

4.4.1 User-driven Access Control on the Paths

One important contribution of the ViP framework is the explicit path functionality (user-driven network semantics). We support three different user-driven features on annotation paths:

- **access control**: users would want to control who can take advantage of the explicit annotation propagation paths that they introduce. This is necessary for two reasons: (a) confidentiality of paths, i.e., not willing to make relationships between data public; and (b) scalability of paths from an information absorption point of view, i.e., not everybody is interested in everybody else's beliefs on which data is related. This of course means that certain paths will not be visible to some users.
- **HAF on insert**: user would like to control the sharing of annotations. When an annotation is inserted, the ViP framework enables users to specify a Boolean variable, A-HAF, Annotation's *Hops Allowed to Follow (HAF)*, to indicate either to propagate the annotation to neighbors or not. A-HAF = 0 (or false) means the annotator just wants to limit this annotation to data items specified in the view. A-HAF = 1 (or true) means the annotator allows this annotation to be propagated.

- **HAF on query**: although if $A \rightarrow B$ and $B \rightarrow C$ implies that $A \rightarrow C$, this may not be applicable for all cases (i.e., there is information "decay"). In cases of a network of paths (e.g., as in Figure 10), it may not be prudent to exhaustively follow all paths in the network to propagate annotations. Similarly with the *HAF on insert*, but in a more specific way, the ViP framework gives the option to specify (at query time) *a maximum number of hops an annotation can follow*. The number of hops starts counting after we follow the first "direct" path (i.e., in Figure 10 the number of hops is 2). We refer to this value as U-HAF (or user HAF). There is also a system-wide maximum HAF, which we refer to as MAX-HAF. Given these three parameters (some of which are optional), if A-HAF is true, t*he maximum number of hops followed* will be the minimum of the (U-HAF and MAX-HAF), otherwise it is 0:

$$max\_number\_of\_hops\_followed = MIN(U-HAF, MAX-HAF) \times A-HAF. \quad (4.1)$$

By setting A-HAF to false (A-HAF = 0) or MAX-HAF to 0, we effectively disable annotation paths; by setting MAX-HAF to 1, we effectively disable cascading annotation propagation.

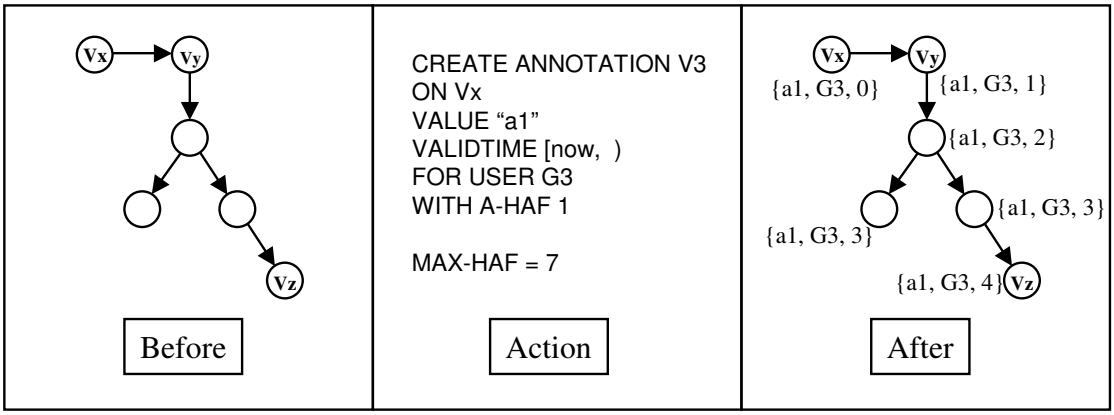


Figure 11: User-driven Annotation Propagation Example

We illustrate the user-driven semantics of the ViP framework using the example in Figure 11. Figure 11/Before has a network of paths; Figure 11/Action indicates that an annotation is added on node V_x; Figure 11/After shows how annotations would be propagated (the third number in the set corresponds to the number of hops required to reach each node). We see that annotation a_1 is

propagated to V_y within HAF 1 as (a_1, G_3, 1), and to V_z within HAF 4. Clearly, users that neither belong to group G_3 nor specify a U-HAF high enough will not "see" annotation a_1. Besides, if the A-HAF of a_1 is set to 0, even if users specify a high U-HAF, there will still not "see" annotation a_1. The queries and the results are shown in Table 4.

Query	**Result**	**User**	**A-HAF**	**U-HAF**	**Annotation**
1	V_y	$U_1 \neg \subseteq G_3$	1	3	No a_1
2	V_z	$U_3 \subseteq G_3$	1	3	No a_1
3	V_z	$U_3 \subseteq G_3$	0	5	No a_1
4	V_z	$U_3 \subseteq G_3$	1	5	a_1

Table 4: Queries and Results for Figure 11

There is a significant amount of related work in personalization, especially in connection with information retrieval [79, 72]. There is also additional work in user-driven data management, allowing users to express their preferences on the execution of their queries, such as [75, 126, 101]. However, to the best of our knowledge, this is the first work to address in a unifying framework all the user-driven features that we proposed as part of ViP, on the specific domain of annotation management.

4.4.2 Path Strength/weight Definition and Management

We propose to define strong, regular, and weak paths that work with network semantics, e.g., path $\longrightarrow$ [2, 1, 0] can be customized to consider user preferences. Path strength is used to represent the quality of annotation and path relations. Path strength can be considered as the confidence level of the annotator about that particular path, i.e., his/her belief of how closely the source and target views are related. The strength includes two aspects: (1) the annotator's confidence level of the relation, (2) the expert quality of the annotator.

We view the path strength idea as analogous to bridges and weight limitations. In Pittsburgh, there are hundreds of bridges across the area's rivers. Some of the bridges allow trucks to go

through, some bridges may only allow cars and trucks less than four metric tons to go through, and some small bridges may only allow pedestrians and bikes to go through. Obviously not all trucks can cross all bridges. Furthermore, if a truck wanted to go over a sequence of bridges, it should have to adhere to the weight limitations of all bridges in the sequence. In this analogy, a path is the bridge from a source view to the target view, which is shown in Figure 12. Similarly to vehicles and weight limitations, not all annotations in ViP can be propagated via all paths, but have to adhere to strength limits.

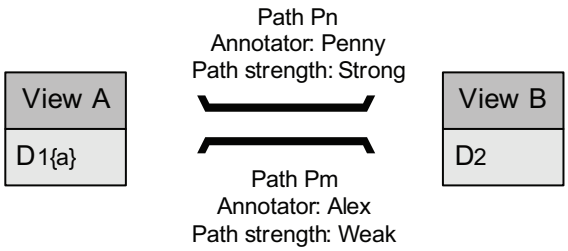


Figure 12: Path Strength

In Figure 12, D_1 belongs to V_A, and D_1 has annotation *a* associated with it. In the case that *a* is 0 hops away from V_A (*a* is the annotation directly associated to V_A), it is considered as a strong annotation. *a* will be propagated to V_B via P_n and P_m, if Alex and Penny are the users who are allowed to view this annotation. If *a*'s directly associated view is at least 1 hop away from V_A and it is propagated to V_A, *a* is a weak annotation. It will still be propagated via P_n but not via P_m.

Path Strength Pattern:

As we discussed above, we use a multi-hop pattern to present the path strength, in the format of (x, y, z), where x presents strong paths, y presents regular paths, and z presents weak paths. It is shown in Figure 13. In the multiple n hops, we use 0 for nonexistent, n for all paths. (n, n, n) means all strong paths, $(0, n, n)$ means all regular paths, $(0, 0, n)$ means all weak paths, and $(1, n, n)$ means the paths between the source and the destination, that have one strong hop, and all other hops are regular.

The total sum of (x, y, z) should be equal to the number of hops. The prior ordered strength

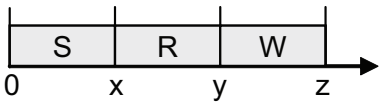


Figure 13: Path Strength Presentation

overwrites the other strength indications. For example, there are 5 hops of the paths, we have $(1, 4, 2)$ as the path strength, then there is one hop through a strong path, 4 hops through regular paths, and none weak hop, though the weak path strength has a number of 2.

In the path matrix, we may use individual numbers to present path strengths. 2 presents the strong path, 1 presents the regular path, and 0 presents the weak path. For example, in the previous case, we have a path map as $1 \rightarrow 2 \rightarrow 1 \rightarrow 1 \rightarrow 1$ to present the path strength and order. This presentation method could speed up the strength computation and work as a pre-matching condition.

In our prior work [106, 80], we described a multi-criteria based routing policy that exploits both the semantics of queries and the state of sensor nodes to improve network service longevity. We look at routing in sensor networks from this perspective and propose an adaptive multi-criteria routing protocol. Our algorithm offers automated reconfiguration of the routing tree as demanded by variations in the network state to meet application service requirements.

In the online annotation path weight define/update environment, how to improve the retrieval speed while keeping the accuracy (return related annotations associated to data) and completeness (return full set of related annotations) will be a big challenge. The path strength will play an important role in route decision, and as a "filter" to improve the query execution. As our strategy with dynamic routing trees in our prior work, we can consider a multi-criteria, dynamic recomputing, and adaptive annotation search algorithm, which will be our future work.

4.5 VIP-SQL DEFINITION - ANNOTATION

We formally define the ViP-SQL language by extending the SQL query language as follows.

Definition 1: ANNOTATION

```
CREATE ANNOTATION Annotation-Name
    ON (table-name [column-name(s)])
    AS (Query)
VALUE object1 [, object2, .. objectn]
[VALIDTIME [start, end]]
[FOR USER user-name(s)]
[WITH A-HAF value]
```

- VALIDTIME provides a specification of a particular valid time period from start time to end time. As for particular time semantics, *now* = [now, now], *future* = (now,), *now+future* = [now,). As for *future interval*, users need to specify a future time period,
- FOR USER, if not specified, the default value is for *all* users,
- WITH A-HAF, the value is 0 or 1. If not specified, the default value is 1,
- As in general SQL language, a table can be a view.

Definition 2: ANNOTATION-PATH

```
CREATE ANNOTATION-PATH Annotation-Name
    ON (table-name [column-name(s)])
    AS (Query1)
    TO (table-name [column-name(s)])
    AS (Query2)
[VALIDTIME [start, end]]
[FOR USER user-name(s)]
```

It defines how to setup paths for annotation propagation. The system is transitive, which means each path will be sorted in topological order.

4.6 SUMMARY

In this chapter, we proposed user-driven time semantics and user-driven network semantics for annotation propagation. We also proposed user-driven access control on annotation views and annotation paths. Finally, we presented the registration of annotation views and paths in the form of ViP-SQL.

5.0 THE VIP FRAMEWORK - QUERY POINT OF VIEW

After having presented how the ViP framework behaves at annotation time, in this chapter, we present how the ViP framework handles queries.

5.1 VIEWS AND QUERIES

5.1.1 What Views Do We Have?

There are two type of views from different points of view: annotators and users (i.e., queriers), as we illustrate in Figure 14. First of all, an annotator creates an annotation view as follows:

```
CREATE ANNOTATION V1 ON Experiments
   AS (SELECT ExpID FROM Experiments
       WHERE Lab = "ADMT" and
             Date >= "10/01/12" and
             Date <= "10/20/12")
   VALUE "ADMT Lab was contaminated
         between Oct. 1st & Oct. 20, 2012.
         Please use data with caution."
   VALIDTIME [now,  )
```

Secondly, users issue queries to retrieve such information, which is explained in the next section. A similar type of query language has been studied in the Meta-SQL system [47].

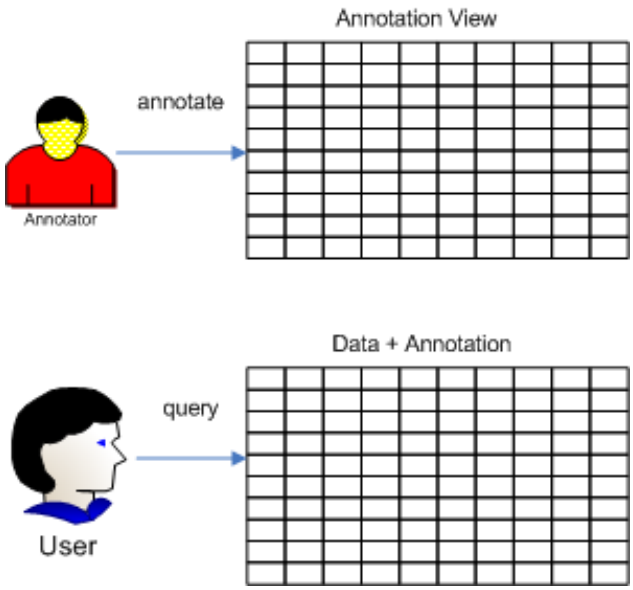


Figure 14: Views from Annotators and Users

5.1.2 What Results Do Users Want?

We classify queries with annotations into the following three types:

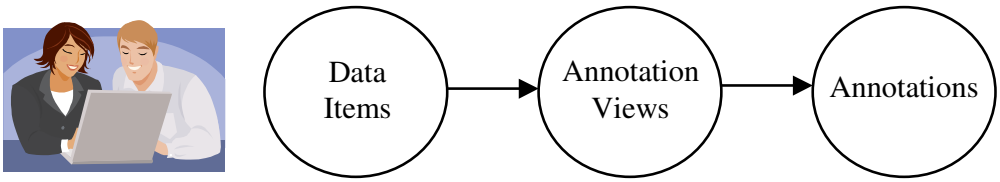


Figure 15: Query Type I

- **Type I**: *Query Data + Browse associated Annotations*.

  Under this type, a user would primarily issue a query to retrieve data items meeting certain criteria, and the job of the annotation management system would be to identify all the annotations that are related to the data that was returned to the user. If we assume the order of Data Items → Annotation Views → Annotations, then the type I queries could be viewed as a "forward" query (Figure 15).

```
SELECT [DATA] Exp FROM Experiments
WHERE Lab = "ADMT" and Date >= "09/01/12"
                   and Date <= "09/20/12"
```

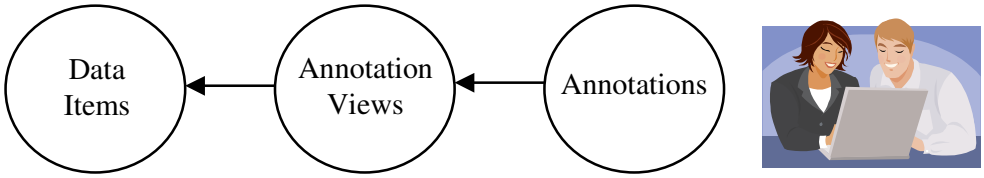


Figure 16: Query Type II

- **Type II**: *Query Annotations + Browse associated Data*.

  Under this query type, a user would issue a query over the stored annotations (e.g., all annotations that contain the substring "TB") and the annotation management system would need to retrieve those, plus all the related data items (i.e., that were "connected" with the annotations returned). Using the same Data Items → Annotation Views → Annotations order as before, query type II would be viewed as a "backwards" query (Figure 16).

```
SELECT ANNOTATION Anno FROM Annotations
WHERE Anno LIKE '%TB%'
```

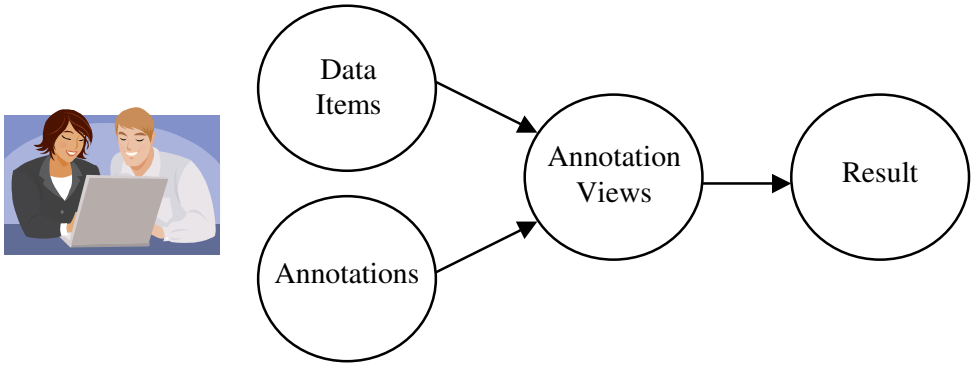


Figure 17: Query Type III

- **Type III**: *Query Data + Query Annotations*.

  This query type essentially combines Query Type I and Query Type II, by allowing a search

on both the data and on the annotations, and returning the data and annotations that meet the query criteria, plus all the related annotations (for the data results) and all the related data (for the annotation results), as illustrated in Figure 17. Note here that there are two possible interpretations of the combined query clause:

- *AND-semantics:* In this case, we adopt a strict AND semantics, which means that for a data item or an annotation to be returned, the criteria specified in the original query need to be met (i.e., Lab=ADMT and Anno contains substring TB). This should essentially return the same results as if we broke the search query into two parts (one Type I and one Type II), executed one of the two, and then applied the search conditions for the second.
- *OR-semantics:* In this case, we adopt a looser set of semantics, which means that for a data item or an annotation to be returned, the criteria specified in the original query need to be met OR the result would need to be related to a data item or annotation that meets the original query criteria. Essentially this would return the same as if the two constituent Type I and Type II queries were run one after the other and a union of the two results sets was returned to the user.

It is worth noting that although both flavors of the Type III queries can be decomposed into multiple Type I and Type II queries, the performance of doing these queries separately will be dramatically different than the combined, Type III query.

```
SELECT Exp FROM Experiments,
       ANNOTATION Anno FROM Annotations
WHERE Lab = "ADMT" and Anno LIKE '%TB%'
```

5.1.3 ViP-SQL Definition - Query

Continuing with the formal definitions of ViP-SQL, we define an annotation query as following:

Definition 3: QUERY

```
SELECT attribute
FROM (table-name)
SELECT ANNOTATION anno-attribute
FROM (anno-table-name)
```

```
WHERE (condition)
[WITH U-HAF]
[FOR USER user-name(s)]
```

where:

- *U-HAF* (or user HAF) is defined when a query is issued. It means a maximum number of hops an annotation can *follow*.
- *A-HAF*, a threshold, is defined when the annotation is created, compared to U-HAF, it will decide either pass the annotation or not.
- *MAX-HAF* means how many hops the system can propagate annotations, if there exist some paths. If we set it as infinite, it will be an NP-hard problem, but with a limitation, it should be traversed within polynomial time complexity.

  As for the overall HAF value it is given by the following formula:

  $HAF = MIN(system\ MAX-HAF, query-user-specified\ U-HAF) \times annotator-specified\ A-HAF$.

  We also claim there is no cycle in the propagation, once an item has been visited before, we will stop propagating the current annotation on that path.
- *FOR USER* is used in the user-driven views: When the annotation is defined, FOR USER means who will have access to a particular annotation. When a query is issued, FOR USER means whose annotation(s) this user wants to view.
- User information is retrieved from user profiles automatically. We assume there is a user hierarchy, that is, users are managed in multiple levels, *user group* and *user* etc. Each user has own unique ID and belongs to one or more particular groups.

5.2 QUERY PROCESSING

In this section, we present the algorithms to process query type I, II and III. In subsection 5.2.1, we present a complete set of algorithms for Query type I, including annotation management (insertion, deletion, and update) and cache replacement algorithms (deterministic and probabilistic

approaches). We also discuss the operations for two other query types in subsections 5.2.2 and 5.2.3.

5.2.1 Query Type I: Query Data with associated Annotations

5.2.1.1 Query Processing The ViP framework relies heavily upon the concept of *database views* to declaratively describe annotations and annotation paths, thus in query processing, annotation view management is very important. Even if an annotation is associated to a data item directly, in our implementation, we still register it in the annotation view table, except that it has a direct data item link in the condition attribute.

We use ViP-SQL to allow users to retrieve regular results with annotations. A query with annotations is rewritten as standard SQL with preprocessing and postprocessing. Preprocessing checks the auxiliary table for possible early annotation filtering. If a query result (a data item) is satisfied by an annotation view's condition, the annotation query processor will return this annotation, and lookup the annotations associated with the query result, if there are any annotation paths associated.

The cache is used to optimize system performance. Once there is a query result, the system will lookup at the cache first, if there is no match (cache miss), then it will lookup in the actual annotation tables. If there is a cache hit, the system will return the data item with the cached annotations. We present the pseudo code for the corresponding algorithms in the following.

Searching Associated Annotations To Search the annotations associated with a data item, we need to search in both directions: its direct annotations (via annotation views) and its inherited annotations (via annotation paths). The pseudo code is presented in "Algorithm 1: Direct and Inherited Annotation Search", in the next page.

5.2.1.2 Caching to Optimize Annotation Search If a data tuple is not found in the cache, the ViP query processor will execute the annotation query and save its annotation query results set into the cache. If a data tuple is found in the cache, we need to verify if it is still "fresh". There is a cross-check of the data with its snapshot, which is taken when the annotation query set is stored into the cache. It is like a "signature", including its query condition and value.

Algorithm 1 Direct and Inherited Annotation Search

procedure search_associated_annotation T_i:

find_direct_associated_annotation T_i

find_dependent_associated_annotation T_i

return T_i.annotationQueryResult

procedure find_direct_associated_annotation T_i:

Let A ← search_in_Annotation_Attribute_table(T_i.table, T_i.col)

for each annotation A_j in A **do**

compare_condition_parameter (A_j, T_i)

if match **then**

add A_j.id to T_i.annotationQueryResult

end if

end for

procedure find_dependent_associated_annotation T_i:

Let H ← search_in_Inhertance_Definition_table(T_i.table, T_i.col)

for each H_j in H **do**

Let R ← find_records_in_associated_table(H_j.inheritance_rule)

Let R_column ← H_j.inhertance_through_rule.attribute

for each record R_m in R **do**

search_associated_annotation(R_m.R_column)

end for

end for

The two strategies we considered are ***immediate delete (eager)*** and ***incremental insert/update (lazy)***. The cache hit algorithm is presented in "Algorithm 2: Annotation Search in Cache", and the cache management algorithm is presented in "Algorithm 3: Cache Management". While in the Notification mode (an eager strategy), whenever a record is updated in the database, the system will update the cache. It is illustrated as a part of "Algorithm 3: Cache Management".

Algorithm 2 Annotation Search in Cache

```
procedure hit_caching T_i:
Let T_j ← search_in_cache_index(T_i.table, T_i.col, T_i.id)
if T_j is found then
    compare(T_i.data, T_j.datasnapshot)
    if matches then
        hit-counter++
        return T_j.CachedAnnotationQueryResult
    end if
    return false
end if
```

There are four alternatives in our solution:

- lazy strategy with cache size = 0
- lazy strategy with cache size = n, n is a number between 0 to full memory size
- lazy strategy with cache size = infinite
- eager strategy with cache size = infinite

We will evaluate these alternatives in Chapter 7 and report the experimental results.

Cache Management will take no action if a *data item* is inserted, deleted, or updated in the database. However, whenever an *annotation registration* is updated/inserted, our system will update the cache appropriately. If an annotation registration is removed, our system will remove its related entries from the cache as well.

The cache operations are illustrated below with the internal cache structure organized as is the table shown in Figure 18.

Algorithm 3 Cache Management

procedure insert_into_cache T_i:

if cache is full **then**

evict as ViP-LFU algo (or other cache replacement algorithms)

end if

insert T_i to cache

save a snapshot of data referred by T_i

procedure after_annotation_delete T_i:

Let R $\leftarrow$ cached AnnotationQueryResult

while R.table = T_i.table and R.id = T_i.id **do**

delete R

end while

procedure after_record_update T_i:

if working_in_Notification_mode **then**

Let R $\leftarrow$ cached AnnotationQueryResult

while R.table = T_i.table and R.id = T_i.id **do**

delete R

end while

end if

Data

DataID
D_1
D_2

Annotation View

ViewID	TimeStamp
V_1	t_1
V_2	t_2
V_3	t_3

Annotation

AnnoID	Content
a_1	keyword1
a_2	keyword2
a_3	keyword3

Cache

TimeStamp	DataID	ViewID	AnnoID
tc_1	D_1	V_1	a_1
tc_2	D_1	V_2	a_1
tc_3	D_1	V_3	a_3
tc_4	D_2	V_1	a_1

Figure 18: Relational Tables of Cache Operations

Cache Operation When Data Changes in The Database:

- *Insert a data item:* NO action
- *Delete a data item:*
  - Regular: NO action
  - Eager: Invalidate the data entries in the cache
- *Update a data item:* NO action

Cache Operation When Data Is Retrieved:

- *No match in the cache:* Retrieve the associated annotations, then save the annotation set into the cache
- *Match is found in the cache:* Compare the timestamp t_{cache} of the annotation view which the data item associated with, with the timestamp of the annotation view t_{view}. If $t_{cache} >= t_{view}$, the content is still "fresh". Otherwise, we need to retrieve annotations in the annotation view table. For those views added after t_{cache}, check the condition of views, if the data item satisfies the condition, retrieve the related annotations.

Cache Operation When Annotation View Changes:

- *Insert an annotation view:* Insert the timestamp in the annotation view table
- *Update an annotation view:* Update the timestamp of the annotation view

- *Delete an annotation view:* Remove the related entries in the cache

There is another option, namely to delay cache storing if there are frequent updates happening to the annotations. It is similar to the *Forced Delay* re-computation algorithm in [18].

5.2.1.3 Cache Replacement Algorithms As is typically the case, users' experience highly depends on the query response time, which in turn mainly depends on the efficiency and precision of the materialized view cache. The ideal situation is that the cache has exactly the data items and the annotations associated with them. However, this is difficult to achieve since (1) the cache capacity is limited (2) data and annotation views are frequently updated.

In general, cache algorithms (also called cache replacement algorithms or replacement policies) are optimizing instructions/algorithms that the operating system, a computer program, or a hardware-maintained structure can follow to manage a cache of information stored on the computer [113]. When the cache is full, the algorithm must choose which items to discard to make room for the new ones. LRU (Least Recently Used) and LFU (Least-Frequently Used) are two classic and popular algorithms. We identify representative cache replacement algorithms from previous works and map them into equivalent solutions to fit our system. Next we propose our own algorithm as an alternative in order to achieve better performance and, as a result, higher user satisfaction.

Assume the cache consists of N data objects and their annotation sets. The entire data set is denoted as $D = \{d_k | 0 \leq k \leq N\}$. We assume that data objects are independent of each other and they are independently updated.

We partition the space of cache replacement algorithms into two categories, *Deterministic Approaches* and *Probabilistic Approaches*, which we present in more detail next.

Deterministic Approaches to Cache Replacement:

In deterministic approaches, we assign a *priority* for the replacement of each data item in a deterministic way, and order the data items in a priority queue. When we schedule the cache replacement, we pick the data item with the highest score from the priority queue to replace. We consider eight deterministic approaches to cache replacement, as follows.

1. **Least Recently Used (LRU):** discards the least recently requested items first. This algorithm

requires keeping track of the timestamp of each data item when it is accessed.

$$Pri(R_i) = \frac{1}{Ren(i)} \tag{5.1}$$

where Ren(i) is the last accessed time stamp of the data item *i*.

2. **Least Frequently Used (LFU-ORIG):** the traditional LFU algorithm (which we refer to as LFU-ORIG) counts how often an item is needed. Those that are used least often are discarded first. Deterministically, the more accesses that have been accumulated, the higher priority that this data needs to be maintained in the cache.

3. **ViP-LFU:** in this thesis, we propose a hybrid solution: we keep two counters: one is for the access frequency, the other is for the last access timestamp. If there is a tie situation, we prefer to keep the data item which was accessed most recently. In addition, we utilize a tumbling time window for taking advantage of locality. We call this algorithm "**ViP-LFU**" or "**LFU**" for short, which is simple, but efficient, as we will show in the experimental evaluation and compare it with other algorithms.

$$Pri(R_D) = \frac{1}{ACC(D_i)} \tag{5.2}$$

where ACC(D_i) is the access of the data item *D* during the given time window *i*

$$ACC(D_i) = \alpha \times AccNum(D_i) + (1 - \alpha) \times ACC(D_{i-1}) \tag{5.3}$$

where $AccNum(D_i)$ is the number of accesses of the data item *D* during the given time window *i*. We use aging for the previous accesses of the data item and use an aging factor α equal to 0.8.

4. **Update-Count-based Deterministic (UCD):** correspondingly, the more updates on the data item, the higher priority for it to be replaced.

$$Pri(R_i) = Upa(i) \tag{5.4}$$

where Upa(i) is the number of updates applied on the data item i.

5. **Annotation Cost (AC):** in [34], the GreedyDual-Size algorithm associates a value H with each cached page p, by setting H to *cost/size* where *cost* is the cost of bringing the document, and *size* is the size of the document in bytes. In this thesis, we assume the size of each data entry in the cache is uniform, rather we focus on the cost of bringing the data item and associated annotations into the cache.

$$Pri(R_i) = \frac{1}{AC(i)} \tag{5.5}$$

where AC(i) is the time to retrieve all annotations associated to the data item *i*.

6. **Annotation Size Cost (ASC):** in addition, we consider the annotation size associated with the data item. If there are two data items needing the same amount of time to retrieve, we will pick the one with fewer annotations to kick off.

$$Pri(R_i) = \frac{Size(i)}{AC(i)} \tag{5.6}$$

where $Size(i)$ is the number of annotations associate to the data item *i*, and AC(i) is the time to retrieve all annotations.

7. **Popularity with Cost (PC):** in addition to the cost, we include the popularity of the data item into consideration.

$$Pri(R_i) = \frac{1}{AC(i) * ACC(i)} \tag{5.7}$$

where AC(i) is the time to retrieve all annotations associated to the data item *i* and ACC(i) is the number of accesses of the data item.

8. **Update Popularity Cost (UPC):** we extend the criteria to also include the number of updates.

$$Pri(R_i) = \frac{Upa(i)}{AC(i) * ACC(i)} \tag{5.8}$$

where Upa(i) is the number of updates applied on the data item *i*, AC(i) is the time to retrieve all annotations associated to the data item *i*, and ACC(i) is the number of accesses.

Probabilistic Approaches to Cache Replacement:

In probabilistic approaches, we assign a *probability* on each data item. When we schedule the replacement, we randomly select one data item by following the assigned probabilities.

1. **Popularity-based Probabilistic (AFP):** intuitively, the more accesses on one data object, the more chances there are that it should be maintained in the cache. Correspondingly, we calculate access frequencies on each data item. The probability is then determined by the access frequencies of the data items.

$$P(R_i) = \frac{\sum_{i \in \mathbf{D}} ACC_i}{ACC_i} \tag{5.9}$$

   where ACC_i is the number of accesses of the data item i.

2. **Update-Frequency-based Probabilistic (UFP):** one hypothesis is that frequently updated data items should not be kept in the cache. Under this belief, we calculate update frequencies for each data item, and assign the priority score from these update frequencies accordingly. We will pick the data item with the highest probability score to replace.

$$P(R_i) = \frac{UPD_i}{\sum_{i \in \mathbf{D}} UPD_i} \tag{5.10}$$

   where UPD_i is the number of updates happened on the data item i, and **D** is the data set.

Many of the above algorithms were adapted from traditional data caching; we implemented all of them in a way more appropriate for the context of this thesis and experimentally compare the performance of these algorithms in Chapter 7.

5.2.2 Query Type II: Query Annotations with associated data

Another useful type of queries is one that includes conditions on the annotations and retrieves the annotations that match, along with all associated data items. We refer to this as a Type II query.

One trivial solution is to handle this in a way that is essentially symmetric to what we did with Type I queries. First of all, let the users specify the keywords or criteria about the annotations.

Once the annotations have been found, match the annotation views registration, thus retrieve the data items as described in the annotation view table. However, to implement this solution we would need to have the annotation views be *materialized*, which brings a lot of challenges. In that case, each annotation view may keep the keyword and data items it associates to, but the maintenance cost maybe high, although the retrieval time will be reduced significantly.

There are multiple ways that annotations can be associated with data. We discuss (1) Direct Annotations on Data, (2) Annotations on views, and (3) Annotation Paths, in the following.

1. **Direct Annotations on Data**
    - What happens when an annotation is added:
        • [simple solution:] insert an annotation into the annotation table; no update on the annotation table when a data item is inserted
        • [advanced solution:] update index on annotations, add the annotation into the cache if needed
    - What happens when searching annotations:
        • [simple solution:] query the annotation, get results from keyword matching; and then look up in the data table, match the annotation with the data item
        • [advanced solution:] add an index on annotations, cache annotation associated with data items. It is similar as the solution in query type I, but in a reverse way.
2. **Annotations on Views**
    - What happens when an annotation is added:
        • [simple solution:] insert annotation into annotation view table; no update on annotation table when a data item is inserted
        • [advanced solution:] update caching table when data or annotation is inserted (invalidate certain contents in the cache with the expired timestamp, in the eager strategy)
    - What happens when searching annotations:
        • [simple solution:] query the annotation, get results from the SQL execution (result 1); retrieve the data table to find data item(s) satisfying the annotation view condition; return result 1 with related data item(s)
        • [advanced solution:] 1. add indexing on certain attributes,

2. cache annotations,

3. Extension: find the overlapping ranges, e.g., having a partial match with a keyword

(1) find which view definitions are applicable (need to be fast)

(2) find which data items are part of specific view definitions (can be a regular query - materialized views)

3. **Annotation Paths**

- What happens when an annotation is added:
  - [simple solution:] insert annotation into annotation view table; insert annotation path in to the annotation path table; no update on annotation table when a data item is inserted
  - [advanced solution:] update the index and the cache table (invalidate certain contents in the cache with the expired timestamp, in the eager strategy)
- What happens when searching annotations:
  - [simple solution:] it is an option for users to view additional annotations since the query is an exact match of annotation keywords, or more complicated, the exact match of the annotation view definition. There is also the inheritance of annotation views, if applicable. Users have the option to view additional related annotations, thus find the associated data items.

5.2.3 Query Type III: Query Data and Annotations

As we did in the previous section, we summarize the solutions for query type III next, according to the three different ways of annotation association.

1. **Direct Annotations on Data**

- What happens when an annotation is added:
  - [simple solution:] insert an annotation into the annotation table; no update on annotation table when a data item is inserted
  - [advanced solution:] update the index both on data items and annotations
- What happens when searching annotations:
  - [simple solution:] it is a join or union of the data result set with the annotation result set, according to AND-semantics or OR-semantics, which we discussed in Section 5.1.2

• [advanced solution:] add indexing on both data attributes and annotation attributes, make a hybrid cache (data and annotations)

2. **Annotations on Views**

- What happens when an annotation is added:

  • [simple solution:] insert annotation into annotation view table; no update on annotation table when a data item is inserted

  • [advanced solution:] update the caching when data or annotation is inserted (invalidate certain contents in the cache with the expired timestamp, in the eager strategy)

- What happens when searching annotations:

  • [simple solution:] query the data item, get results from SQL execution (result 1); query the annotations, get result 2; match result 1 with result 2

  • [advanced solution:] add indexing on both data attributes and annotation view attributes, make a hybrid cache (data, annotations, and queries), keyword search

3. **Annotation Paths**

- What happens when an annotation is added:

  • [simple solution:] insert annotation into annotation view table; insert annotation path into the annotation path table; no update on annotation table when a data item is inserted

  • [advanced solution:] update index and caching table (invalidate certain contents in the cache with the expired timestamp, in the eager strategy)

- What happens when searching annotations:

  • [simple solution:] query annotation, join with the results from query type I.

  Step 1: match query on annotation with the annotation view definitions,

  Step 2: run query on data and get results,

  Step 3: compare the query results with a subset of the view definitions, that is, Type 3 = Type 2 + Type 1

  • [advanced solution:] indexing on multiple attributes, caching (data, annotation views and queries), rewriting annotation view queries, group local queries together, pre-matching, keyword search etc.

5.2.3.1 Keyword Search Keyword search has been studied extensively in the literature, including keyword search in databases. We propose to utilize well-established techniques and use indexing which takes advantage of the structure of annotation views in our system.

Let us assume that we have a Type III query on both data and annotations, and that if executed independently, the query on data will generate a set of related annotations (Set A) and the query on annotations (which is often a keyword-based search) will generate a different set of annotations (Set B). The proper result of the original type III query should return the intersection of sets A and B, which will refer to as C.

One (naive) approach of computing C involves finding the set of data items that answer the query, then finding their related annotations (Set A) and then finding the annotations out of Set A that contain the desired keywords. Although this approach avoids computing all of Set B, it still could be a costly proposition. Another (still naive) approach of computing C, takes the opposite route: first identify the set of all annotations that match the query on annotations (i.e., compute Set B), and then only keep the annotations whose related data items satisfy the conditions for the query on data.

In either case, the desired set is C; but in both cases the smaller of A and B must be searched in its entirety. To eliminate the need to search extraneous annotations, and only search the subset C, our proposed solution will index each annotation view in the system with the keywords contained in annotations to it and that propagate to it. As one must search these views as a graph to find annotations on the data items in a query's result set, one would ideally be able to prune sub-graphs where no annotation to any view in that sub-graph matches the keyword search.

This is still sub-optimal, though, as a particular view may produce many annotations. Although the pool of keywords contained in these annotations may contain all of the search keywords, there may not be any annotation that contains all of the desired keywords. Because of this, every annotation on such a view must be examined. In the worst case, every annotation in the set A would be searched while the set C is empty. Suppose now that one knew that for any view, the first half of the annotations on that view and propagated to that view contained a set of keywords and that the other half of those annotations contained a different set of keywords. One would then potentially be able to prune half of the annotations on and propagated to such a view.

In this way one can construct a tree-like keyword index for each view. This index will be called

the *annotation term provenance graph*, or ATP graph:

Definition 1. Let W be a set

$$V \times A \subseteq W$$

where each element in W represents a node on the *annotation term provenance graph* $G(W, E)$. Any child of any $w \in W$ will by definition have more terms in common with any of its siblings than with any of its cousins. Let $\rho : V \rightarrow W$ be a function that maps views to nodes in the ATP graph. Each view will be mapped to the node that represents all terms used in annotations on that view and preceding views. Let $\alpha : W \rightarrow A$ be a function that maps nodes in the ATP graph to annotations. Let T be a set of terms used in annotations, defined as

$$T = \bigcup_{a \in A} T_a$$

Let H be a hash function

$$H : V \times T \times \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$$

Let $h(v, t, \lambda) \geq 1$ for all v and t where t is a term of an annotation on a view exactly λ hops before v.

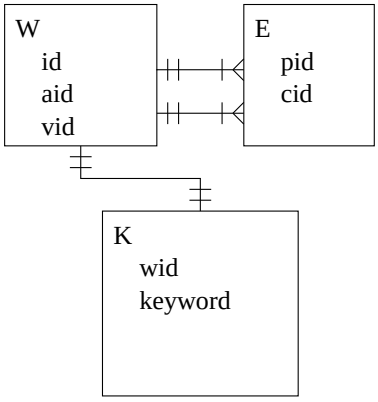


Figure 19: Keyword Index Structure

W maps internal nodes to annotations and views in Figure 19. K maps nodes to keywords, E defines the keyword inheritance edges.

When **adding an annotation**, we start at $\rho(v)$ and greedily explore the child with the most terms in common. When we encounter a node with more terms in common than any of its children, we pair the new node with that node. We then explore up and add the new terms to each ancestor node within HAF views. Pseudo-code for the process is given as algorithm 4.

Theorem 1. *Algorithm 4 will run in expected time equal to* $O(|S|b^{HAF_{max}} \lg^2(K))$*, where* S *is the set of views to annotate,* b *is the maximal branching factor of* $G(V, E)$*,* HAF_{max} *is the maximal number of edges to propagate over and* K *is the maximal number of annotations on any view.*

Proof. One can observe that the first, outermost for-loop iterates over every member of S, thus requiring $O(|S|)$ time. Within this loop, the ATP graph starting at $\rho(s)$ is explored top-down for each member $s \in S$. Because the terminal nodes of the sub-graph to be explored map to annotations on s, and because there can be at most K annotations on any view or data item, and because the sub-graph has a bounded out-degree, this takes time equal to $O(\lg(|K|))$, and the outer for-loop takes time equal to $O(|S| \lg(|K|))$.

The while loop will operate on nodes in the ATP graph starting with those added to the boundary by the for loop. As we know over how many views an annotation can propagate (HAF_{max}) and the maximal number of internal nodes under any view ($O(K)$), if we know the maximal out-degree of all the views (b) then we can determine an upper bound for the while loop's runtime. Because there are $|S|$ starting nodes, there will be up to $|S|b^{\text{HAF}_{\text{max}}}$ views that need to be explored.

□

Corollary 1. *In the worst case, algorithm 4 will run in time equal to* $O(lg(K)|V|)$ *where* K *is the maximal number of annotations on any view and* V *is the complete set of views.*

When **deleting an annotation** from a set of views and data items, we must determine which views and items have been annotated and for each of those, which node in the ATP maps to the annotation. We then delete these nodes and follow the edges of the ATP graph while removing the annotation's keywords from each node in the ATP graph that is HAF views or fewer away from the original item (Algorithm 5).

Algorithm 4 Adding an annotation

Require: views V, annotations A, items to annotate S, annotation a, adjacency list $G(W, E)$, inverted adjacency list $F(W, E^{-1})$, hash function H

procedure add an annotation:

$A \leftarrow A \cup a$

Let explored $\leftarrow \emptyset$

for each $s \in S$ **do**

Let $w_0 \leftarrow \rho(s)$, $l \leftarrow -1$, $w_p \leftarrow w_0$, $w_c \leftarrow \emptyset$

for $w \leftarrow F_{w_0}$, $w \neq \emptyset$, $w \leftarrow w_{\text{next}}$ **do**

if LCS$(T_a, T_w) > l$ **then**

$l \leftarrow$ LCS(T_a, T_w), $w_c \leftarrow w$

end if

end for

while LCS$(T_a, T_{w_c}) \geq$ LCS(T_a, T_{w_p}) **do**

temp $\leftarrow w_c$

for $w \leftarrow F_{w_p}$, $w \neq \emptyset$, $\forall v[\rho(v) = w \rightarrow v = s]$, $w \leftarrow w_{\text{next}}$ **do**

if LCS$(T_a, T_w) > l$ **then**

$l \leftarrow$ LCS(T_a, T_w), $w_c \leftarrow w$

end if

end for

$w_p \leftarrow$ temp

end while

$F \leftarrow F \cup \{w_m, w_n\}$, $G \leftarrow G \cup \{w_m, w_n\}$

$F_{w_n} \leftarrow \emptyset$, $G_{w_n} \leftarrow \{w_m\}$

$F_{w_m} \leftarrow \{w_c, w_n\}$, $G_{w_m} \leftarrow \{w_p\}$

$F_{w_p} \leftarrow (F_{w_p} - w_c) \cup w_m$, $G_{w_c} \leftarrow (G_{w_c} - w_p) \cup w_m$

Let $\alpha(w_n) \leftarrow s$, boundary $\leftarrow w_n$, explored$_{w_n} \leftarrow 1$, hops$_{w_n} \leftarrow 0$

end for

Algorithm 4 (Continued)

while boundary $\neq \emptyset$ **do**

  Let $u \leftarrow \text{boundary}_0$

  boundary $\leftarrow$ boundary $-$ boundary_0

  Let $\lambda \leftarrow \text{hops}_u$

  if $\lambda < \text{HAF}_{\text{max}}$ **then**

    for Each $t \in T_a$ **do**

      $h(u, t, \lambda) \leftarrow h(u, t, \lambda) + 1$

    end for

    for $w \leftarrow G_u$, $w \neq \emptyset$, $w \leftarrow w_{\text{next}}$ **do**

      if $\neg\text{explored}_w$ **then**

        boundary $\leftarrow$ boundary $\cup\, w$, $\text{explored}_w \leftarrow 1$

        if $\exists v[\rho(v) = u]$, $\lambda + 1 < \text{HAF}_{\text{max}}$ **then**

          $\text{hops}_w \leftarrow \lambda + 1$

        else if $\forall v[\rho(v) \neq u]$ **then**

          $\text{hops}_w \leftarrow \lambda$

        end if

      end if

    end for

  end if

end while

Algorithm 5 Deleting an annotation

Require: views V, annotations A, items to remove annotation from S, annotation a, adjacency list $G(W, E)$, inverted adjacency list $F(W, E^{-1})$, hash function H

procedure delete an annotation: Let explored $\leftarrow \emptyset$

for each $s \in S$ **do**

$W_c \leftarrow \alpha^{-1}(s)$

for Each $w_c \in W_c$ such that $\rho^{-1}(s)$ is an ancestor of w_c in G **do**

for Each $w_x \in F_{w_c}$ **do**

$G_{w_x} \leftarrow (G_{w_x} - w_c) \cup G_{w_c}$

end for

$F \leftarrow F - \{w_c, w_p\}$, $G \leftarrow G - \{w_c, w_p\}$

end for

end for

while boundary $\neq \emptyset$ **do**

Let $u \leftarrow \text{boundary}_0$, boundary $\leftarrow$ boundary $-$ boundary_0, $\lambda \leftarrow \text{hops}_u$

if $\lambda < \text{HAF}_{\text{max}}$ **then**

for Each $t \in T_a$ **do**

$h(u, t, \lambda) \leftarrow 1$

end for

for $w \leftarrow G_u$, $w \neq \emptyset$, $w \leftarrow w_{\text{next}}$ **do**

if $\neg\text{explored}_w$ **then**

boundary $\leftarrow$ boundary $\cup\, w$, $\text{explored}_w \leftarrow 1$

if $\exists v[\rho(v) = u]$, $\lambda + 1 < \text{HAF}_{\text{max}}$ **then**

$\text{hops}_w \leftarrow \lambda + 1$

else if $\forall v[\rho(v) \neq u]$ **then**

$\text{hops}_w \leftarrow \lambda$

end if

end if

end for

end if

end while

Theorem 2. *In the worst case, algorithm 5 will run in expected time equal to* $O(|S|b^{HAF_{max}} \lg(K))$*, where* S *is the set of views to remove annotations from,* b *is the maximal branching factor of* $G(V, E)$*,* HAF_{max} *is the maximal number of edges to propagate over and* K *is the maximal number of annotations on any view.*

Corollary 2. *In the worst case, algorithm 5 will run in time equal to* $O(lg(K)|V|)$ *where* K *is the maximal number of annotations on any view and* V *is the complete set of views*

The **annotation search** is proposed in Algorithm 6.

Theorem 3. *Algorithm 6 will run in expected time equal to* $O(L)$ *where* L *is the number of annotations those match the search criteria.*

5.3 SUMMARY

In this chapter, we introduced three query types, and discussed the solutions and algorithms for each query type. We proposed to use caching to optimize the annotation search, in addition, we proposed ten cache replacement algorithms adapted from our experience and related work.

Algorithm 6 Annotation Search

Require: views V, annotations A, keywords K, search views V', inverted adjacency list $F(W, E^{-1})$, hash function H

procedure search:

Let explored $\leftarrow \emptyset$, boundary $\leftarrow \emptyset$, hops $\leftarrow \emptyset$, r $\leftarrow \emptyset$

for Each $v \in V'$, $\forall \omega \in K[h(v, \omega, 0) \geq 1]$ **do**

Let explored$_v \leftarrow 1$, hops$_v \leftarrow 0$, boundary $\leftarrow$ boundary $\cup\, v$

end for

while boundary $\neq \emptyset$ **do**

Let $u \leftarrow$ boundary$_0$

boundary $\leftarrow$ boundary $-$ boundary$_0$

Let $\lambda \leftarrow$ hops$_u$

if $\forall \omega \in K[\omega \in T_u]$, $\alpha(u) \in A$, $\lambda <$ HAF **then**

r $\leftarrow$ r $\cup\, \alpha(u)$

end if

for $w \leftarrow F_u$, $w \neq \emptyset$, $w \leftarrow w_{\text{next}}$ **do**

if $\neg$explored$_w$, $\forall \omega \in K[h(u, \omega, \lambda)] \geq 1]$ **then**

boundary $\leftarrow$ boundary $\cup\, w$

hops$_w \leftarrow \lambda + 1$

if $\exists v[\rho(v) = w]$, $\lambda + 1 <$ HAF **then**

explored$_w \leftarrow 1$

else if $\forall v[\rho(v) \neq w]$ **then**

hops$_w \leftarrow \lambda$

end if

end if

end for

end while

6.0 THE VIP FRAMEWORK - PROOF OF CONCEPT IMPLEMENTATION

To fully address the two problems introduced in our motivation examples in Chapter 1 and Chapter 4, we have proposed the ViP framework whose details were presented in Chapter 2, Chapter 4, and Chapter 5. In this chapter, we present some of the implementation details as they relate to the prototypes/proof-of-concept demos that we have built, which realized the different features of the ViP framework.

6.1 USER-DRIVEN TIME SEMANTICS

We proposed the concept of *valid time* in Section 4.1; this was one step beyond the original implementation in the prototype, which at the time only supported *now, future, now+future* semantics.

Our solution: We extend the current implementation to fully support valid time, that is, provide the function of *future interval* time semantics. We performed a large-scale experiment to evaluate the new function, verified that it is correct and complete in the entire spectrum of workloads and environmental settings.

6.2 USER-DRIVEN NETWORK SEMANTICS

We focused on providing as many features of the ViP framework as possible, while providing good overall system performance. There are several issues we wanted to explore. We discuss the issues about performance in Sections 6.2.1, 6.2.2, and 6.2.3; while the issues about providing all of ViP's features are discussed in Section 6.2.4.

6.2.1 Lazy/eager Annotation Propagation Algorithms

ViP employs a variant of *lazy* or *on-demand* propagation of annotations. In the case of annotation views and paths, this scheme has the advantage to retrieve annotations in one batch instead of having to eagerly propagate annotations to individual data items. This is also a very compact representation method. However, the system may perform better in eager propagation in some cases, depending on the actual annotation and data distribution and also on query patterns.

Related Work: *Eager* and *Lazy (on-demand)* annotation propagations have been studied in many papers such as [25] [128]. In the *lazy* approach, a query is generated and executed to retrieve associated annotations only when needed. It is very popular in data lineage tracing [44]. In the *eager* approach, the provenance and annotations of data are carried along with data when they are being transformed. Thus, annotations are immediately available in the output, by the cost of space and time spent in the process. In most available systems, DBNotes [25] follows the eager propagation method, while MMS [115] is following the on-demand scheme.

Our Solution: We enhance the existing annotation propagation algorithms by allowing the system to select between lazy and eager annotation propagation schemes, and finding the switching point. The adaptive adjustment to get a hybrid propagation scheme is part of our future work. We consider this problem as "when" to switch.

6.2.2 Indexing on Annotation Views and Annotation Paths

Related Work: There are a lot of research projects that use indexing on time series, biological sequences, and databases. The traditional mechanisms are B+ tress and hash tables. Beyond that, space-partitioning trees such as kd-tree [24] or SBC-Tree [49] have been applied for biological data.

Our Solution: We want to take advantage of current indexing algorithms, and mainly focus on *annotation* indexing more than *data* indexing. Clearly, we cannot only put indexes on data attributes, we want to use indexing techniques on user-driven features such as users/user groups or public/private views and paths, i.e., have these features as a first-priority "filter" to speed up annotation retrieval.

6.2.3 Cache Management

Our Solution: Another algorithm to explore is dynamic cache adjustment. Currently we use a modified LRU (Least Recently Used) + LFU (Least Frequently Used) caching algorithm (ViP-LFU), and use two levels of counters to keep track of cached annotation sets. That is, we put the first order on cache item's used frequency, and the second order on visited time (as was described in Algorithm 5.2).

An annotation query has the unique feature that it is closely associated to data, so any annotation update, and even data changes will affect annotation result sets. As such, simple time window or aging scheme does not work well in this situation. We want to design an effective cache management for annotations. We consider such problem as "when" to cache. One step further, we want to analyze the most promising caching annotation set from historical data, thus adjust the "hot" cache content. We consider this problem as "what" to cache.

6.2.4 Private/public Views and Paths Management

Users want to set annotation views or annotation paths as public or private in their preferences, in other words, an annotation view or an annotation path could be visible to different users in multiple granularities.

Related Work: Privacy preserving methods are a hot topic in networks, including sensor/wireless network [36], social networks [71], or network commerce [94] etc. It is not until recently that annotation and provenance management have gained attention into this area, but most projects are limited to access authorization such as [71].

Our Solution: we propose to add user preference setting so that each user can setup his/her annotation views and annotation paths as public or private. For private paths, users can specify the access user or user groups, all paths will be public as the system's default setting. That is, the owner or authorized users of a view or a path will have the privilege to make such data public to certain users or user groups. Note that this differs from traditional access control on the data objects because we are talking about *annotation views* and *annotation paths*, i.e., the methods of how the data should be propagated, not the *data*, being set as pubic/private.

In summary, we propose to grant users more access control privileges on data, annotations,

annotation views, and annotation paths. In Section 4.4.2, we also proposed a set of methods for path strength/weight definition and management. In this way, we provide a solution to specify the numeric indicator of how strong annotators' confidence on the path is, and at the same time, users can specify the strength level they want to retrieve via annotation paths. It is another interesting research point, we consider this problem as "how" to retrieve.

6.3 ANNOTATIONS MANAGEMENT

6.3.1 Inserting Annotations

The ViP framework is illustrated in Figure 4. ViP-SQL queries are rewritten automatically into SQL queries evaluated by the annotation query processor, then recorded by the annotation register and the path setup manager. They are sent to the DBMS and the resulting annotation set is merged with the regular query results by the postprocessor for matching and presentation.

We start with registering annotations. Explicit annotations could be a string or a file; while implicit annotations include annotation views and annotation paths. If it is an annotation view, the annotation register is responsible for insertion, deletion and updating. If it is an annotation path, the path setup manager will update the auxiliary table to record the path source and target, with path query conditions. Obviously, sorting views or detecting the view containment problem [118] [40] [32] may bring significant computation and time complexity. To simplify the problem setting, we assume that the network formed by the annotation paths forms a *directed acyclic graph*, when ordered. All views are sorted by topological order to build a hierarchy/dependency tree, thus guarantee the correctness and completeness of the annotation propagation.

6.3.2 Deleting Annotations

When deleting an annotation from a set of views and data items, in the lazy annotation propagation method, removing annotations or annotation views from the annotation registration table finishes the process. If there is an annotation path affected, and either the source or the target is removed, the path will be removed. In the eager annotation propagation method, we must determine which

views and items have been annotated and remove the annotation from each one of those. For example, in the keyword search index, which node in the ATP that maps to an annotation, we need to delete the mapped nodes and follow the edges of the ATP graph while remove the annotation's keywords from each node in the ATP graph that is HAF views or fewer away from the original item.

There is one more point to be mentioned: we do not optimize annotation paths automatically, that is, if there is a path from annotation A to B, and there is another path from annotation B to C, we do not merge them into A to C automatically. If we want to optimize to build a multiple-hop path, we must record the intermediate annotation views and hops, if the middle node(s) is removed, the whole path will be removed from the register.

6.3.3 Implementing Auxiliary Tables

It is quite natural to use auxiliary tables storing the attributes of the annotation views. Like MMS [115], ViP also uses auxiliary tables to store annotation view conditions, which will work as filters to prune unrelated annotation lookups. However, MMS uses Q-indexes (index on queries, which is similar to views in ViP) to maintain indexes on the Q-values (query values), and as such for each data changes all related index tables need to be updated. Unlike MMS, we use caching to improve the performance of computing annotations. The reason is that for the index to be useful, it would need to be efficiently updateable when data and annotations are inserted, deleted and updated; therefore, such index maintenance may require a high cost in space and time. In addition, from the usage pattern we observed in our DataXS project, data updates happen more often than annotation views/paths updates, in which case MMS would require a lot of Q-value updating. Thus ViP relies on caching instead of indexing.

6.4 WORKING WITH ASTROSHELF

ViP [83], together with AstroShelf [93] and CONFLuEnCE [91], deploys a user-driven framework, for large amounts of scientific data. In ViP, we propose to associate annotations with object/data, a

set of objects, or annotations. We treat a view as an annotation registration (annotation definitions), which will be specially handled in the system. Once queries and annotation registrations defined by users via AstroShelf are forwarded to ViP, they are rewritten into SQL queries evaluated by the annotation processor, then sent to the annotation management module, and finally pushed to the DBMS. There is a cache between the annotation management module and the DBMS to expedite the processing time. To maximize the system processing speed, we are considering using an In-memory database (IMDB).

To work with CONFLuEnCE, the ViP framework processes continuous annotations and queries as events. There is an event reporting module pushing annotation changes (insert, delete, or update) to the system monitoring module for pub/sub notification. Our extended system overview is shown in Figure 20.

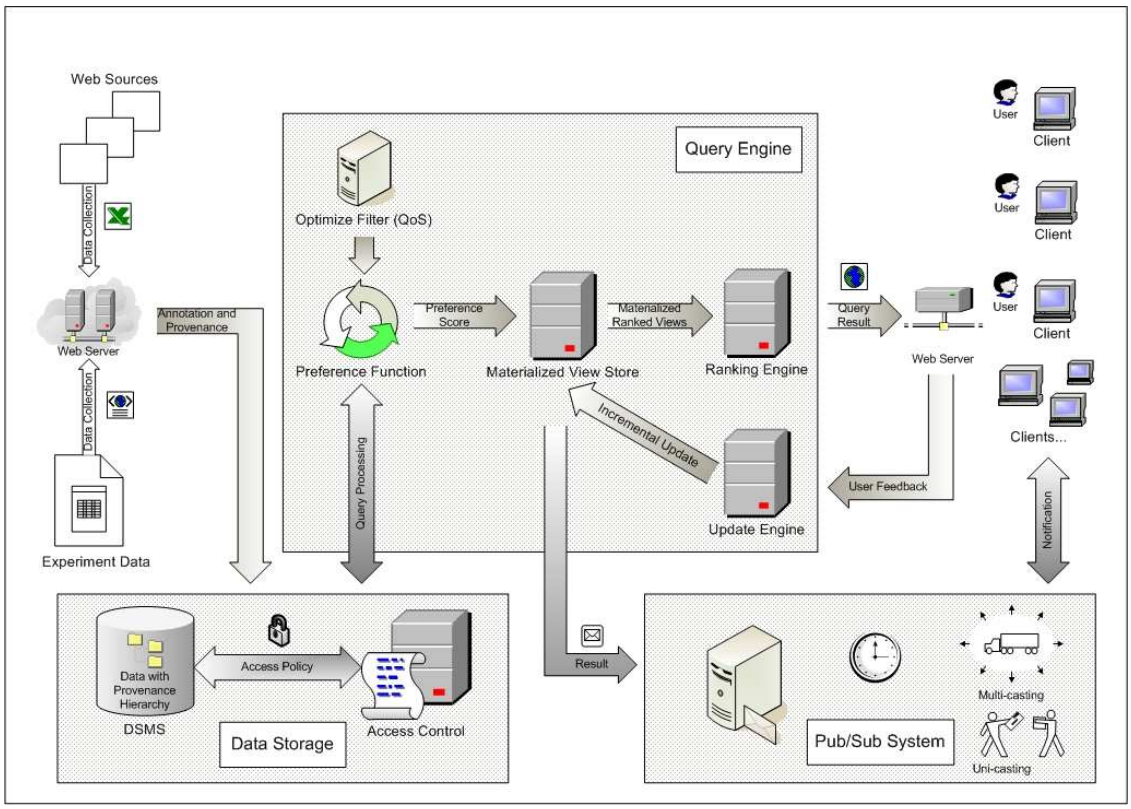


Figure 20: Extended System Architecture

A simple workflow in the system is illustrated as following:

1. Using the SkyView of AstroShelf User Interface, users can define annotations, annotation views, and annotation paths associated with areas of interests in the sky. The annotated object may be arbitrary points, areas, group of areas etc. The ViP framework enables users to specify explicit paths between two or multiple annotations views, thus establishing additional annotation propagation paths. All these annotations are registered into the annotation register in the Data Storage of the ViP framework.

   To present revisions on annotation values or discussions between users on annotation values, annotations on annotation can be recorded. The original annotations will be treated as special data objects in the annotation table.

2. Using SkyView, users can issue queries on certain data in the sky map. AstroShelf will send a request to ViP. The annotation query processor will look up the annotation table and search all related annotation views effective in the valid time period. If there are any annotation paths enabled, all related data items would be retrieved according to the path depth limits specified by annotators and users. The query results will be sent back to SkyView, users can view it by sorting or filtering different conditions such as time or distance.

3. If users register for event notification in the SkyView, all annotation insertion/deletion/updating associated with their interested annotation object (views) will be notified to their browsers or emails, SMS etc. The system may recommend annotations updates to those users who may be interested according to their profiles.

In the Astroshelf collaboration platform, the ViP framework manages adding annotations and queries on annotations according to client requests. It works in the same way as CONFLuEnCE interacts with Astroshelf. To adjust the ViP framework to work with the Astroshelf collaboration platform, there are a couple of challenges:

1. Merge the stand-alone ViP framework with the whole system.

2. CONFLuEnCE manages continuous workflows, so the annotation service will work with continuous annotations, which is in the future work of the ViP framework.

The combined platform will be evaluated with experiments in two parts: the stand-alone version and the one integrated into the Astroshelf collaboration platform. The stand-alone ViP can handle big data sets, so that it could be used for stress testing the system performance, while the integrated annotation engine may implement all functions as proposed, and thus demonstrate the benefit of a hybrid system.

6.5 BIG DATA

Scalability Problem: The advances in computing technology (Moore's Law, cheap and vast storage, increases in network bandwidth, etc.) are creating a greater need for efficient processing of large-scale data, especially in scientific research. The volumes of annotations and data items should be handled in a scalable way, as well as annotation views/annotation paths from users' definitions and queries. We are focusing on how to deal with large-scale data and annotations, how to retrieve and manage data associated with annotations in a quick and effective way. It is challenging to think big, it is also beneficial to bring big data into the system.

6.5.1 Our Solution

We have summarized related work about big data in Section 3.6. To extend our work to address the high levels of scalability required for Big Data, there will be, but not limited, several areas to take care of, as listed below.

- Store the data according to relation/similarity/local feature. We keep track of user preferences and data/query scheme, and identify the data and query pattern. It may also include partial matching patterns. We group similar items together, store the data locally (i.e., "close" to each other when they are "similar"). In other words, we want to "engineer" locality in the system.

- Build up indexes (such as BTree, B+Tree, RTree etc.) on data items and annotations, keep related items close, that is, when we search, find the closet answer first, then expand the search on global domain. In other words, we want to take advantage of the locality.

- Caching. The distributed memory caching system, such as Memcached [6, 54], is intended for use in speeding up dynamic web applications by alleviating database load. The application objects include YouTube [13] and Facebook [4]. It is a good choice in terms of scalable performance. Besides, for the data set and views, we can find the overlapping components and cache them in memory to improve the query processing.

In addition, we also propose in the extended ViP framework to deploy distributed data stores, such as BigTable, and configure indexed data mapping for large-scale data.

Our choices: we explore three choices: partition cache, map cache to a big-data/cloud-ready data structure, and integrated caching, which we elaborate on next.

- **Partition cache**

  If we treat each cache entry as a bucket, there could be several related data or annotation items stored in one bucket. A cache can be distributed in terms of both storage and retrieval, in favor of processing speed.

- **Map cache to a big-data/cloud-ready data structure**

  We map the cache to a distributed structure, with a unique key using which, we can retrieve any cache item in a global environment in a fast way. With existing big data computing services, we do not need to worry about management of such a cache, even in a big scale.

  The problem is how to **generate a unique key** in such large-scale, distributed environment. When Twitter moved away from MySQL and towards Cassandra, the engineers needed a new way to generate id numbers. They designed Snowflake [14, 10], a network service for generating unique ID numbers at high scale with some simple guarantees. There are minimum 10k IDs per second per process and the response rate is 2ms (plus network latency). For high availability within and across data centers, machines generating IDs should not have to coordinate

with each other. As a result of a large number of asynchronous operations, in-order delivery cannot be guaranteed. However, the id numbers will be k-sorted [21, 65] within a reasonable bound (10's of ms), thus meeting our need perfectly.

System Clock Dependency: NTP is used to keep users' system clock accurate. Non-monotonic clocks, i.e., clocks that run backwards, are not supported. If the user clock is running fast and NTP tells it to repeat a few milliseconds, the ID generator will wait until a time that is after the last time generate an ID.

- **Integrated caching**

  Keeping more things in main memory, possibly using a system such as Memcached (Key-value cache in RAM) is another alternative. We may utilize this in our caching scheme, to work with our existing algorithms. There is a comparison of in-memory vs in-disk tables, a reverse-index of data sort vs regular index structures, or big index table. All these comparisons are interesting.

The answers may come from a combined effort taking ideas from several existing solutions. There could be different solutions (databases and caching scheme) for data items and annotation views. We could like to explore this in our future work.

6.6 USER INTERFACE

To answer the challenges we introduced in Chapter 1, in addition to the work we proposed above, we want to provide a user-friendly interface and a graphical representation of server statistics with regards to annotations, to help users understand the ViP framework [95, 96].

In our data-sharing platform DataXS, a user can easily specify filtering conditions to locate certain data items. These filtering conditions are essentially a query (i.e., a view) and can be used by ViP. This functionality enables users to specify views using a point and click interface (Figure 21); these views can then be trivially used to implement all the functionality of the ViP framework.

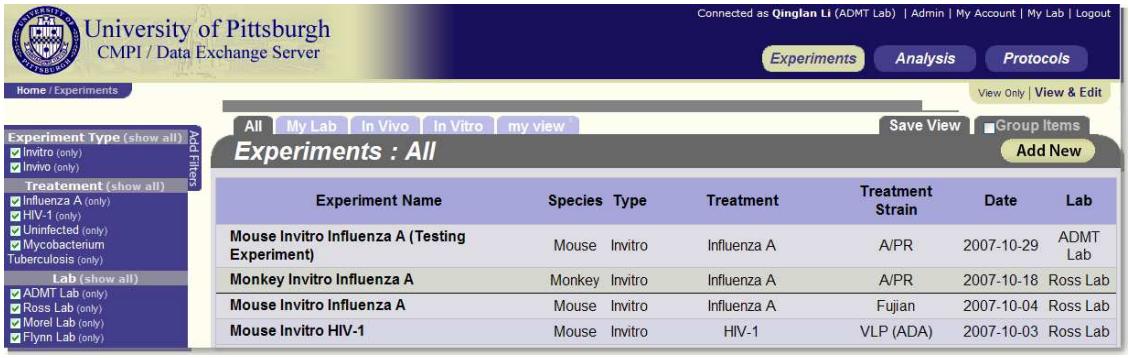


Figure 21: DataXS User Interface

We want to highlight the following three aspects of the ViP framework, as was implemented in conjunction with the DataXS prototype:

- the user interface as well as functionality of the ViP framework, on top of the DataXS platform.
- the performance of the system (through carefully selected statistics at different aggregation granularities), and
- the *"behind-the-scenes"* aspect of the system, through a graphical representation. This tool will enhance the users' understanding of how the ViP framework works, and would help them understand the existing "connections" among (annotated) data.

There are three different aspects of the ViP framework's user interface that we want to present:

- **Define** The ability to add annotations and annotation paths in an intuitive way through a point-and-click interface, as shown in Figure 21 and Figure 22.
- **Search** The ability to browse and search annotations, as shown in Figure 21 and Figure 23.
- **View** The ability to view appropriate statistics for all data items and annotations (including annotation views and annotation paths) in our system via a graphical representation, as shown in Figure 24.

We briefly describe each of these tasks in the next paragraphs. It should be noted that all of these are inherently interactive (everything is done via a user-interface), dynamic (users will immediately see the results of their actions), and can provide additional insight.

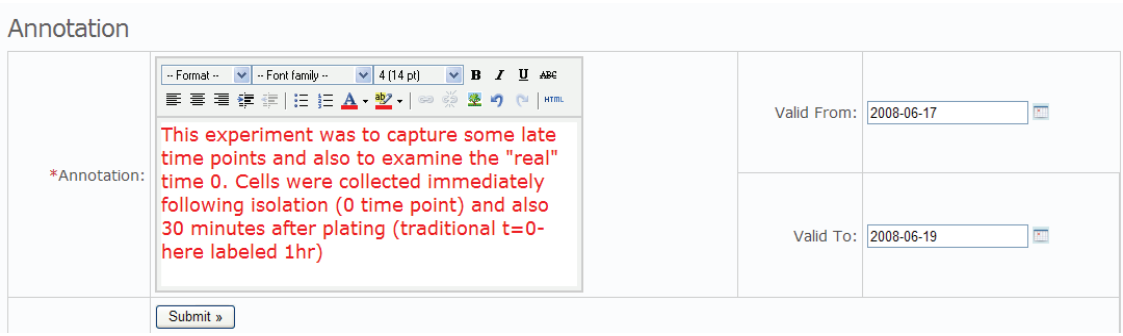


Figure 22: Registering An Annotation View

6.6.1 Define Annotations and Annotation Paths

The ViP framework relies heavily upon the concept of *database views* to declaratively describe annotations and annotation paths[1]. Clearly, users are not expected to provide view definitions in SQL. In our data sharing platform (DataXS), a user can easily specify filtering conditions to locate certain data items. Additionally, users have the option of *saving* a particular query (i.e., set of filtering conditions) and storing it as a *tab* in the system, thus enabling them to rerun the query again

[1]The MMS system [115] advocated the use of views for metadata management; our system is targeting annotations (i.e., a special case of metadata), but on the other hand is significantly extending their proposal with additional semantics (i.e., valid time, network semantics, etc.)

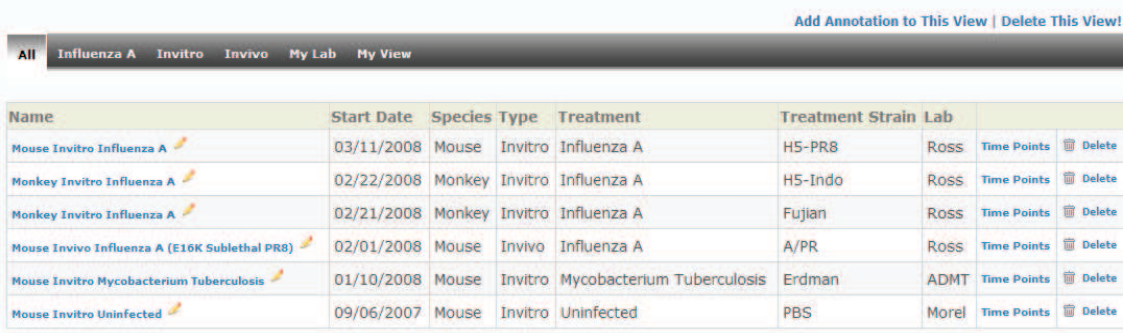


Figure 23: ViP User Interface

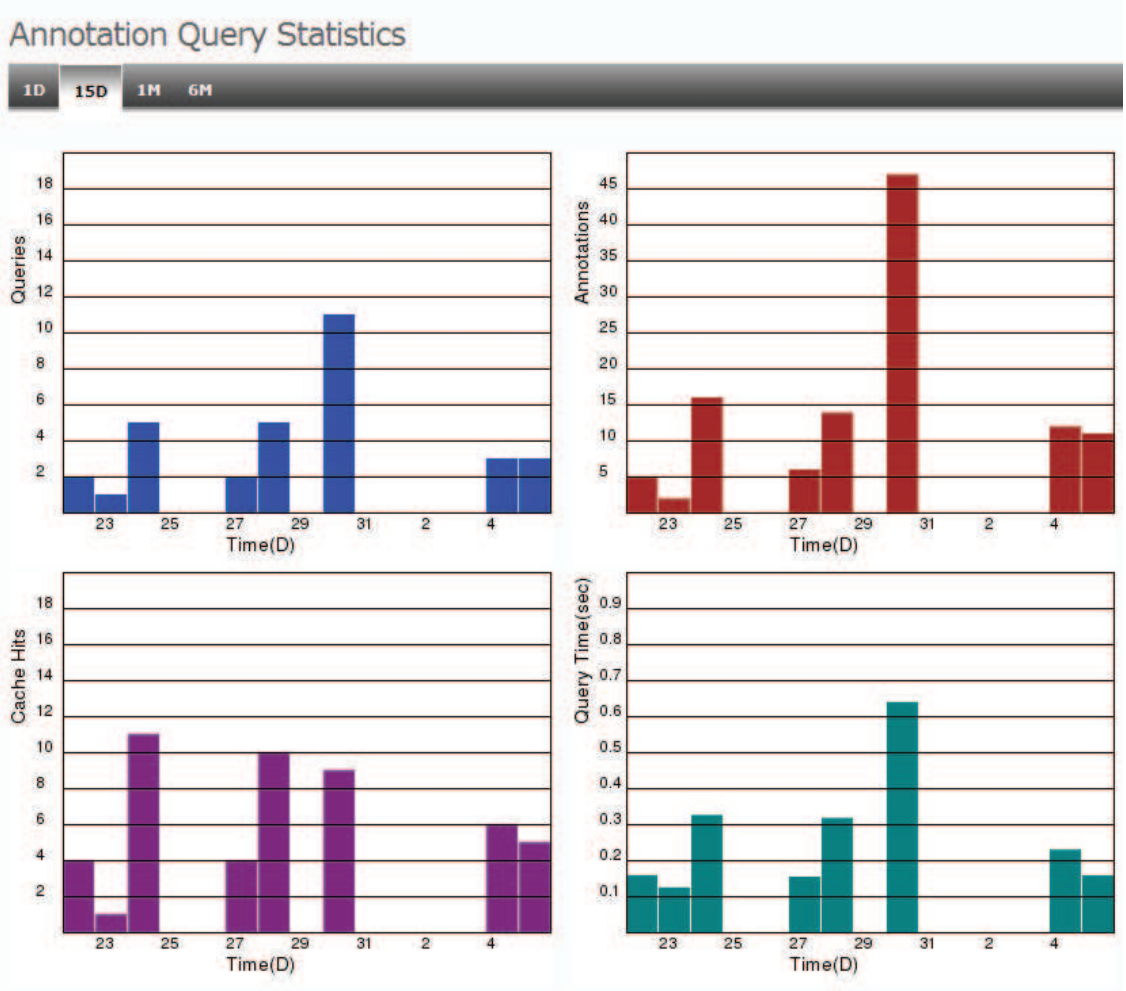


Figure 24: System Monitoring: number of queries (top left), number of annotations (top right), number of cache hits (bottom left) to illustrate the cache behavior, query time (bottom right) to illustrate the system performance

in the future with just a single click instead of having to specify all the filtering conditions from scratch. In other words, this functionality enables users to specify *views* using a point and click interface (Figure 21); we refer to this as the "Save View" function in DataXS. This functionality can be trivially used to support the user-driven time and network semantics for the ViP framework.

We develop a feasible method for users to specify their requirements for the ViP framework. The following ways of adding annotations (i.e., defining *annotation views*) are provided:

- A set of filtering conditions is used in its entirety (exact match); this is similar to the "Save View" tab functionality that exists in our system already.
- If the set of conditions are not enough to adequately describe the properties of the data to be annotated, then we will allow the user to provide additional constraining predicates (typically a date range).
- To also support a simplified interface, we will also enable the user to just specify the list of data items to annotate (i.e., enumerate).

We also showcase the definition of explicit annotation paths, by providing the above *view definition* abilities as a two-step process (for defining the *from* and the *to* "nodes" of the annotation path).

Finally, in addition to defining annotation views, we should be able to add annotations directly to data items (which could, in turn, trigger annotation propagation across pre-established annotation views or annotation paths).

6.6.2 Annotation Browsing and Searching

The default behavior of the ViP framework (and that required by the participants in our CMPI project) is to show the annotations along the associated data items, when the data items are shown as part of a result of a query (or other browsing function of the system). We call this the *browse annotations* mode, as the annotations are retrieved through their association to data items.

Another possible retrieval mode for annotations is through direct search, which we call the *search annotations* mode. In this mode, users specify search parameters against the set of annotations recorded in our database, for example, *get all annotations that a specific user made (and are public)* or *get all annotations that have the word "miscalibration" in them*. The ViP frame-

owrk would then produce the appropriate annotations, but also generate the data items that are associated with the annotations returned. We believe this feature really promotes annotations as first-class citizens in our system.

6.6.3 System Performance Statistics

One important aspect of the ViP framework is the ability to show, with appropriate statistic and graphical representations, a "behind-the-scenes" view of the relationships between data and annotations. In order to do so, we will produce a graphical representation that will display appropriate statistics for all data items in our system (e.g., number of view definitions each data item belongs to, which is called the "interconnections of data"). This would allow us to illustrate what is happening upon insertion/deletion of an annotation view or an annotation path or a data item. Additionally, if there is a data item entered into the system, the user should be informed about any annotation attached according to current valid views.

In addition to illustrating the semantics of the ViP framework, one important aspect that we plan to illustrate is the performance of the system. Since caching was a major part to the efficiency of the ViP framework, illustrating the inner actions of how caching works (i.e., when there was a cache hit and when there was a miss) could prove very interesting. It enhances the user's understanding of what is happening "behind the scenes" in the ViP framework and at the server. One such example is shown in Figure 24.

6.7 SUMMARY

Section	Features	Algorithms	Implementation	Evaluations	Future Work
6.1	Time Semantics	Valid Time	Future Interval	Table 15	
6.2.1	Network Semantics	Lazy/eager Annotation Propagation	Lazy/eager	Table 13	"when" to switch
6.2.2	Network Semantics	Annotation Indexing	Indexes	Figure 32	"filter" for retrieval
6.2.3	Network Semantics	Cache Management	ViP-LFU	Table 14	"what" to cache
6.2.4	Network Semantics	Private/public Views Path Strength	User Preference	Table 17 Table 16	"how" to retrieve
6.3	Anno. mgmt.	Insert/delete Anno.	Anno. Tables	Table 11, 12	optimization
6.4	With AstroShelf	Integrated System	Stand-alone ViP	Future work	Integration
6.5	Big Data (scalability)	Local Storage Indexing Cache Partition Cache Mapping Integrated Caching	Future work	Future work	
6.6	User Interface	Using Views	Figures 23, 22 Figure 24	N/A	Improvement

Table 5: List of Implementations

7.0 EVALUATION OF THE VIP FRAMEWORK

We have implemented the ViP framework as a Ruby on Rails application that interfaces to a MySQL database in Linux OS. To demonstrate that the proposed framework can support all aspects of the technologies described in previous chapters, we evaluated the ViP framework with three different data sets, which were gathered from real data with simulated annotations, users, or query workloads to be able to scale our experiments to desired levels and to fully evaluate the proposed annotation features. In the following sections, we will discuss the experimental setting, the data sets, and the experimental results separately.

7.1 EXPERIMENTAL SETUP

In this section, we introduce the experimental setting of each data set, including system parameters, data set attributes, annotation traces, and query traces.

7.1.1 Description of Data Set 1 (Figure 25, 26, and Table 6, 7)

System: We used ruby 1.9.1, with RubyGems 1.3.6 and Rails 2.3.5. The experimental setting is stated in Table 6.

Data Sets: We gathered data from our DataXS prototype. To test the scalability, we enlarged the data set using Zipf distributions. The experimental parameters are shown in Table 7.

Annotation Traces: There are two types of annotations registered: annotation view and annotation path. *Annotation view* is a query with static annotation associated to it; *annotation path* is the establishment of an annotation(s) propagation link from one annotation view to another annotation

Software	Version
CentOS Linux	Release 6.0 (Final)
Kernel	Red Hat 4.4.4-13
Ruby	1.9.1
RubyGems	1.3.6
Rails	2.3.5
MySQL	6.0

Table 6: Experimental Environment Setting of Data Set 1

view. We generated annotation registrations using two different Zipf distributions: one to identify how many annotation views a data item should participate in, and another one to determine how many data items a particular annotation view should contain, which we also call it "data footprint".

In order to control the data update effect on cache performance, we simulated insertion, deletion, and update both on data items and annotation views. There is a pre-defined annotation registration phase before each experiment starts. We varied the percentages of different operations such as data insertion or annotation view update compared to its pre-defined data range.

Query Traces: We generated queries with Zipf distribution on both (1) data tuples the query associates with (see Figure 25 and 26), and (2) data tuples the annotation associates to. Query conditions vary from 1 to 4 joins. All queries are read-only. Query time is measured in milliseconds unless otherwise indicated. Cost in the AC (algorithm 5.5) is measured by the actual system processing time.

Measure Criteria: The metric we use to compare the algorithm is the *query execution time*. Each experimental result is measured as the average query time of 1,000 runs unless otherwise indicated, such as "total query execution time".

Parameter	Value
Data Tuples	[500 50,000]
Queries	[1,000 10,000]
Annotation Views	[1,000 50,000]
Data Update Frequency	[0%, 60%]
Annotation View Update Frequency	[0%, 30%]
Annotation Insertion Frequency	[0%, 30%]
Annotation Deletion Frequency	[0%, 20%]
Annotation Propagation	(eager, lazy)
Data Verification by	(timestamp, value)

Table 7: Experimental Parameters of Data Set 1

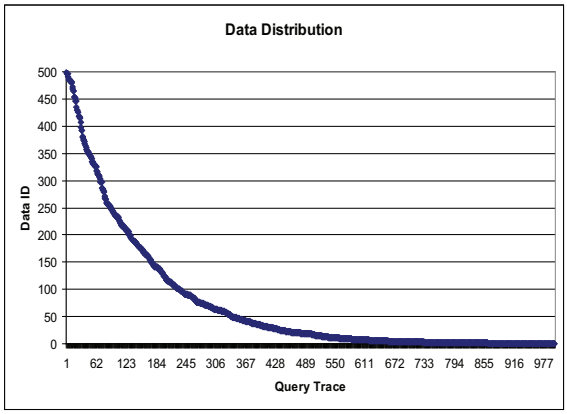


Figure 25: Data Distribution (500 data items)

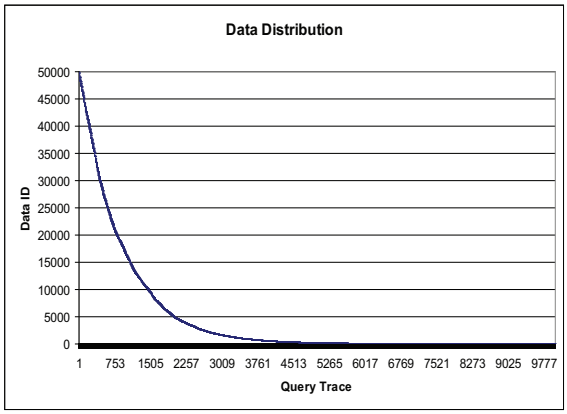


Figure 26: Data Distribution (50,000 data items)

7.1.2 Description of Data Set 2 (Table 8 and 9)

System: The experimental setting is stated in Table 8.

Software	Version
CentOS Linux	Release 6.0 (Final)
Kernel	Red Hat 4.4.4-13
Ruby	1.8
RubyGems	0.9.4
Rails	2.0.2
MySQL	6.0

Table 8: Experimental Environment Setting for Data Set 2

Comparison Systems: For comparison purposes, we chose the latest and the most related work: MMS [115]. In their work, they already discussed the comparison among MMS, DBNotes [25] and MONDRIAN [57]. MMS showed significant benefits over those systems both in query execution times and storage space usage. In [115], the experimental results showed that MMS reduced the redundant space used in DBNotes and MONDRIAN, also, it decreased the query execution time even with the cost of updating the Q-index structure and querying additional tables with metadata. Our system works similarly to MMS in the way that there is an annotation table instead of additional annotation columns. Thus, it is expected our system will perform similarly to MMS compared to DBNotes and MONDRIAN if the association between the data and the annotations is explicit and static.

Features Evaluated: In this work, we focused on implicit annotations, i.e., annotation propagation through annotation views and paths. Since both ViP and MMS can accommodate future tuples and use views to specify annotation registration, we compared our system with MMS mainly in terms of query execution time. Of course, there are additional features that ViP supports and MMS does not (such as the user-driven network semantics). In these cases, we performed a sensitivity study with only our framework.

Data Sets: We gathered the data from the original data set of DataXS [83]. To test the scalability,

we enlarged the data set with uniform random and Zipf distributions. The experimental parameters are set as shown in Table 9.

Parameter	Value	Parameter	Value
Data tuples	300,000	Queries	1,000
Annotation views	[1, 50,000]	Users	[1, 100]
Annotation paths	[1, 2,500]	Path Depth	[1, 10]

Table 9: Experimental Parameters of Data Set 2

Annotation Traces There are two types of annotations registered: *annotation view* and *annotation path*. Annotation view is a query with static annotation(s) associated to it; annotation path is the establishment of annotation(s) propagation from one annotation view to another annotation view. We generated annotation registrations with Zipf distribution on (1) data the annotation is associated to (2) data footprint which is the number of the data tuples an annotation is associated to. Annotation traces include annotation insertion and update.

Query Traces We generated queries with Zipf distribution on (1) data tuples the query is associated with (2) query arrival sequence. Query conditions vary from 1 to 4 joins. All queries are read-only. Query execution time is measured in milliseconds unless otherwise indicated.

7.1.3 Description of Data Set 3 (Figure 27, 28 and Table 10)

System: The following experiments were conducted locally in the ADMT lab. The server was equipped with 4 Intel(R) Xeon(R) 3.00GHz CPU processors, and total memory of 16GB. The operating system is CentOS Linux release 6.0 (Final distribution) and Linux version 2.6.32-71.el6.x86_64 (Red Hat 4.4.4-13, kernel version). The ruby version is ruby 1.9.3, with RubyGems 2.0.3 and Rails 3.2.13. The experimental setting is summarized in Table 10. For more information, please refer to Chapter 9.

Data Sets: We reviewed two popular data sets: the IMDB data sets [5] and the MovieLens data sets [64, 7].

Software	Version
CentOS Linux	Release 6.0 (Final)
Kernel	Red Hat 4.4.4-13
Ruby	1.9.3
RubyGems	2.0.3
Rails	3.2.13
MySQL	Ver 14.14 Distrib 5.1.67

Table 10: Experimental Environment Setting of Data Set 3

The IMDB data sets are available via alternative interfaces, plain text data files, Unix command line search programs, or an IMDB API (a lightweight web service (REST interface) which provides an easy way to access the IMDB data).

IMDB has a huge collection of data. In a single movies.list file (updated till March 1, 2013), it includes 2,446,056 movie entries. The movie list is formatted as follows:

```
movie title                        | year
-------------------------------------
Argo (2012)                          2012
```

It takes a lot of effort to prepare the data, including the following steps:

- Data Integration: integrate movie titles, actors/actresses, year, genre, rating from different source files.
- Data Cleaning: the data we have collected are not clean and may contain errors, missing values, noisy or inconsistent data. For example, there are foreign characters on movie titles, as well as unformatted characters that are hard to process in the query. We need to get rid of such anomalies.
- Data Transformation: we need to aggregate data, if there are duplicate entries. We also need to associate annotations with movie titles, actors/actresses etc.

The MovieLens data sets were provided by the GroupLens Research group. It includes the

movie rating information. The data sets were collected over various periods of time, depending on the size of the set. There are three sizes of data sets.

- MovieLens 100k - Consists of 100,000 ratings (1-5) from 943 users on 1682 movies.
- MovieLens 1M - Consists of 1 million (1,000,209) anonymous ratings from 6,040 users on 3,883 movies.
- MovieLens 10M - Consists of 10,000,054 ratings and 95580 tags from 71,567 users on 10,681 movies.

The user rating table has the following schema:

```
user id | movie id | rating | timestamp
-----------------------------------------
196       242       3        881250949
```

The movie information table has following schema:

```
movie id | movie title | release date | video release date
|IMDb URL | unknown | Action | Adventure | Animation
|Children’s | Comedy | Crime | Documentary | Drama
| Fantasy |Film-Noir | Horror | Musical | Mystery
| Romance | Sci-Fi |Thriller | War | Western
```

There are two ways used in the data set to present the movie information:

```
1::Toy Story (1995)::Adventure|Animation|Children|Comedy|Fantasy
```

-OR-

```
1|Toy Story (1995)|22-Nov-1995|29-Oct-1996|
http://us.imdb.com/M/t...|0|0|0|1|1|1|0|0|0|
0|0|0|0|0|0|0|0|0
```

The user information table has following schema:

```
user id | age | gender | occupation | zip code
-----------------------------------------------
1       |24   |M       |technician  |85711
```

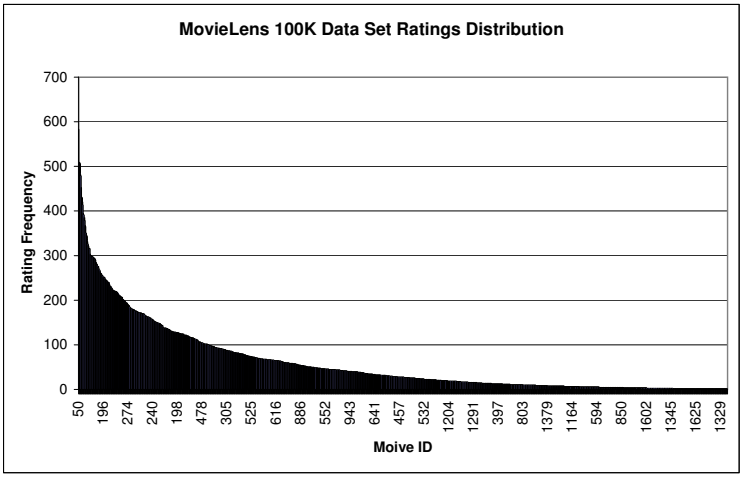


Figure 27: Ratings Distribution on Movies in the 100k Data Set

The frequency of ratings on a movie item has been illustrated in Figure 27. As expected, it follows a Zipf distribution.

Because of the data availability and data integration reasons, we decided to choose the MovieLens data set as our experimental data. Based on the user, movie, and rating information, our system synthetically generated annotation, annotation views, and annotation paths on this data set to complete the evaluation. The data set in this experiment is static, that is, no update or insertion or deletion during the query process. Similarly, the annotation views in this experiment are static too, and are predefined before the query starts processing.

Annotation views: our datagen program generates annotations from certain age/gender/occupation/area of users on certain movie genres at certain time periods or on a specific movie. The attributes are *movie genre, user age, user occupation, user zip code, or rating time period*. Below are some annotation examples:

1. Rating with ID 3 has an input error.
2. Users who are between 25 to 40 are likely to overrate Sci-Fi movies.
3. Users who lived in the northeast and rated movies at the end of October 2012 (October 22, 2012 - October 31, 2012) may not have had enough time to watch movies because of hurricane Sandy.

4. Movie “The Mummy (1999)” had a gross box office income of $155,247,825 in the USA market.
5. Movie “The Mummy Returns (2001)” had a gross box office income of $202,007,640 in the USA market.
6. Movie “The Scorpion King (2002)” is a spinoff of “The Mummy Returns (2001)”. You could also call it a prequel, since it takes place before the events of that film.

Annotation 1 is a simple form of annotation, and annotation 2 and 3 are examples of annotation views. Since Movie “The Mummy (1999)”, “The Mummy Returns (2001)”, and “The Scorpion King (2002)” were made by same production crew and cast, in storyline and details they are related to each other, we may setup annotation path from annotation 4 to 5, and from 5 to 6, thus all annotation associated with movie “The Mummy (1999)” could be propagated to “The Mummy Returns (2001)” and “The Scorpion King (2002)”, if within appropriate access control limits. Our assumption is viewers who are interested in the movie “The Mummy Returns (2001)” may also want to know information about its prequel. In fact, that would be something that the production crew of the three movies would establish, when they add information about the movies. Also worth noting is that this annotation path ability is possible without relying on to the database schema (i.e., even if there is no attribute to mark “prequels”).

An example of the frequency of annotations on ratings has been plotted in Figure 28. In this case, we simulated 100 critics annotating ratings with 90% annotation view on the movie release date condition and 10% on the movie genre attribute. It was randomly generated following Zipf’s law [132, 131, 17]. The simplest case of Zipf’s law is a “1/f function”. Given a set of Zipfian distributed frequencies, sorted from most common to least common, the second most common frequency will occur 1/2 as often as the first. The third most common frequency will occur 1/3 as often as the first. The n^{th} most common frequency will occur 1/n as often as the first. Over fairly wide ranges, and to a fairly good approximation to discrete numbers, many natural phenomena obey Zipf’s law.

Pareto law [22, 63] is given in terms of the cumulative distribution function (CDF), i.e., the number of events larger than x is an inverse power of $x : P[X > x] = x^{-k}$, where k is the Pareto distribution shape parameter.

Given a random variate U drawn from the uniform distribution on the unit interval (0, 1], the

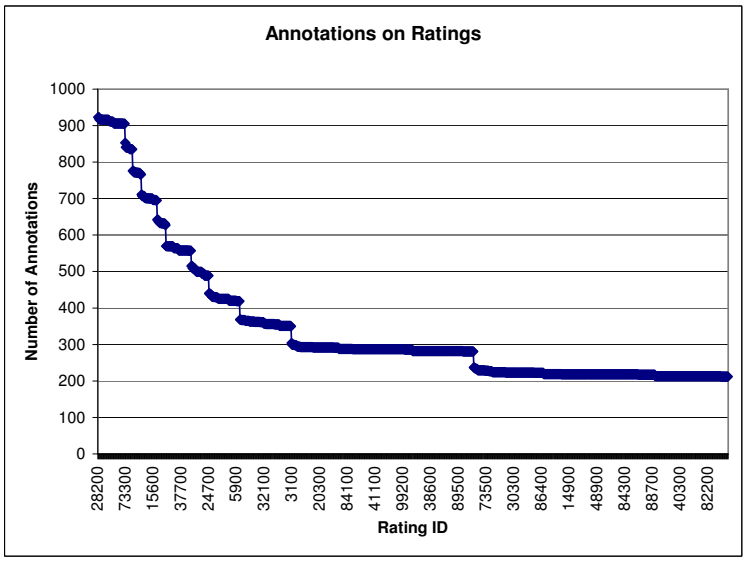


Figure 28: Annotations Distribution on Ratings in the 100k Data Set

variable T given by

$$T = \frac{x_m}{U^{1/\alpha}} \tag{7.1}$$

is Pareto-distributed. x_m is the (necessarily positive) minimum possible value of the random variable. In the case of simulated query trace with Pareto distribution, we set Pareto index α = 1.161 to approximate the "80-20 law" [98].

We support annotations on annotations, as it has been implicitly implemented in the system. From rating's point-of-view, an annotation on its rating value is a direct annotation; from a movie's point-of-view, its ratings can be seen as its annotations, thus annotations on its ratings are *annotation on annotation*. Of course, we also support explicit annotations on annotations, for example, one user puts an annotation on one piece of rating, another user may put his/her annotation on that annotation respectively.

7.2 WORKLOAD SUMMARY

In this section, we summarize the workloads by the different experiments that were part of our extensive evaluation of the ViP framework.

In Section 7.1, we summarized our experiment setup in Table 6 and Table 7 for the first data set, Table 8 and Table 9 for the second data set, and Table 10 for the third data set. The detailed software packages of the running environment is listed in Chapter 9.

We present data, annotations and annotation views/paths parameters and query processing results, including figures index, algorithms, measured attributes(Y axis), varies attributes(X axis), and current status, in Table 11 and Table 12 for all three data sets. We compare Eager vs Lazy annotation propagation methods in Table 13, all of them are evaluated on Dataset-1 with testing case2. Table 14 presents results of cache hits, in some test cases we include the query trace over time to highlight more interesting behaviors. On the other hand, Table 16 presents path strength results. Public vs private views and paths are illustrated in Table 17, while Table 15 presents user-driven features.

All of the workload summary can be found in experimental results on data set 1, 2, and 3. It shows that our proposed framework is realistic, functional, and improving the system performance. Finally, we also listed some test cases as part of our future work.

Fig	Workload	Algorithms	What Measured(Y)	What Varies(X)
53	300000 Data 50000 AViews 1000 Queries 100 Users 2500 APaths **[=Dataset-2]**	ViP vs MMS **[=Case1]**	Query Time	AViews= [1000,50000]
54	Dataset-2	Case1	Setup Time	AViews= [1000,50000]
55	Dataset-2	Case1	Query Time	Query Trace over Time
56	Dataset-2	ViP-LFU	Query Time	AViews= [1000,50000]
Table 18	Dataset-2	Case1	Query Time	Anno Density= [40%, 200%]
31	50000 AViews 50000 Data 5% Dupdate 2% Ainsert 10000 Queries **[=Dataset-1]**	ViP - (LRU, LFU, UCD, AC, PC, UPC, noCache) Lazy **[=Case2]**	Query Time	Zipf AnV Dis
29 30	Dataset-1 60% Dupdate 30% Ainsert	Case2	Query Time	Unif AnV Dis vs Zipf AnV Dis

Table 11: List of Experiments - Query Processing on Data Set 1 and 2

Fig	Workload	Algorithms	What Measured(Y)	What Varies(X)
63	100,000 Data 1000 AViews 1000 Queries 100 Users **[=Dataset-3]**	ViP-LFU	Query Time Anno Found	Cache [on,off]
64	Dataset-3	ViP-LFU	Setup Time	Data = [100k, 1M]
65	Dataset-3	ViP-LFU	Query Time	Cache Capacities= [0%, 100%] **[=Case3]**
67	Dataset-3	ViP-LFU	Query Time	Query Trace over Time
70	Dataset-3 1,000,000 Data	ViP-LFU	Query Time	Case3
72	Dataset-3 10,000,000 Data	ViP-LFU	Setup Time	Data Imported vs Anno Registration vs Query Generation
74	Dataset-3 10,000,000 Data	ViP-LFU	Query Time	Case3 Query Pattern

Table 12: List of Experiments - Query Processing on Data Set 3

Fig	Workload	Algorithms	What Measured(Y)	What Varies(X)
34	Dataset-1	Case2	Query Time	D-Eager vs D-No Action
33	Dataset-1 No Dupdate No Ainsert	Case2	Query Time	Eager vs Lazy
35	Dataset-1	Case2	Query Time	Eager vs Lazy
36	Dataset-1 60% Dupdate 30% Ainsert	Case2	Query Time	Eager vs Lazy
37	Dataset-1	Case2, Eager	Query Time	Dupdate %
38	Dataset-1	Case2, Eager	Cache Mana. Time	No Update
39	Dataset-1	Case2, Eager	Cache Mana. Time	5% Dupdate
40	Dataset-1	Case2, Lazy	Cache Mana. Time	5% Dupdate
41	Dataset-1	Case2 Eager	Query Time	Management Time vs Query Time vs Extra Cost
42	Dataset-1	Case2 Lazy	Query Time	Management Time vs Query Time
45	Dataset-1	LFU Eager	Query Time	AViews = (100, 1000, 5000, 10000)
TBD	x% Dupdate y% Ainsert	Case2	Query Time crossover point	Eager vs Lazy

Table 13: List of Experiments - Eager vs Lazy

Fig	Workload	Algorithms	What Measured(Y)	What Varies(X)
43	Dataset-1 1000 Queries	Case2	Cache Hits	Lazy vs Eager
44	Dataset-1	Case2	Cache Hits	Lazy vs Eager
46	Dataset-1	LFU Eager	Cache Hits Anno Found	AViews = (100, 1000, 5000, 10000)
47	Dataset-1	Case2 Eager	Query Time	Query Trace over Time
48	Dataset-1	LFU Eager	Query Time	Cache Size = (0, 20, 100, 200, Inf)
49	Dataset-1	ViP-LFU Eager	Query Time	Query Trace over Time
50	Dataset-1	ViP-LRU Eager	Query Time	Query Trace over Time
51	Dataset-1	ViP-PC Eager	Query Time	Query Trace over Time
52	Dataset-1	ViP-AC Eager	Query Time	Query Trace over Time
66	Dataset-3	ViP-LFU	Cache Hits	Case3
71	Dataset-3	ViP-LFU	Cache Hits	Case3
75	Dataset-3	ViP-LFU	Cache Hits	Case3 Query Pattern
TBD	x% Dupdate y% Ainsert	Case2	Query Time crossover point	Cache Sizes

Table 14: List of Experiments - Cache Hits

Fig	Workload	Algorithms	What Measured(Y)	What Varies(X)
57	Dataset-2	ViP-LFU	Query Time	User Criteria
58	Dataset-2	ViP-LFU	Anno Found	User Criteria
Table 19	Dataset-2	ViP-LFU	Query Time Anno Found	Path Hops

Table 15: List of Experiments - User-driven Features

Fig	Workload	Algorithms	What Measured(Y)	What Varies(X)
61	Dataset-1	ViP-LFU	Query Time	10%(W)-(R)-(S)
TBD	x% (W)eak Path y% (R)egular Path z% (S)trong Path	Case2	Query Time crossover point	Path Strength Percentages

Table 16: List of Experiments - Path Strength

Fig	Workload	Algorithms	What Measured(Y)	What Varies(X)
59	Dataset-2	ViP-LFU	Query Time	Percentage of Public Views
60	Dataset-2	ViP-LFU	Query Time	Public Paths
TBD	x% View	ViP-LFU	Query Time crossover point	Public Views Percentages
TBD	y% Path	ViP-LFU	Query Time crossover point	Public Paths Percentages

Table 17: List of Experiments - Public vs Private Views and Paths

7.3 EVALUATION OF CACHING ALGORITHMS - DATA SET 1

In this section we present our evaluation of the different caching algorithms, using data set 1 which was described in Section 7.1.1.

7.3.1 Query Distribution (Figure 29, 30, and 31)

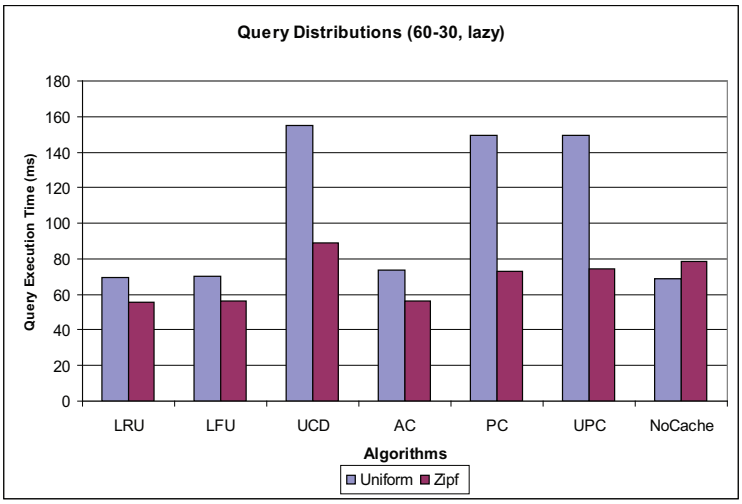
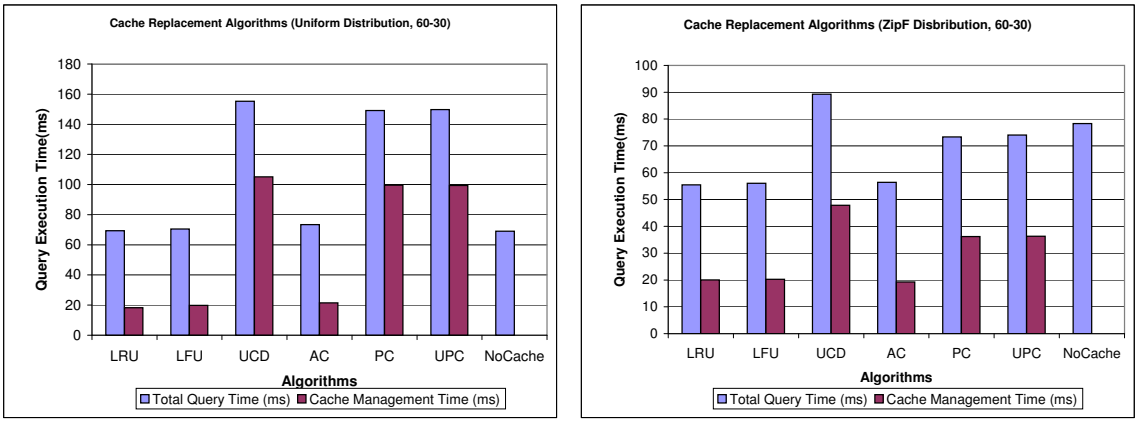


Figure 29: Comparison of Different Caching Schemes

In our optimization scheme, caching plays a major factor to improve system performance. We described it in Section 5.2.1.2. So in the first experiment, we compared the query times of uniform and Zipf query distribution under different cache replacement algorithms. The results are shown in Figure 29. The results presented in this thesis were acquired as the average value from multiple repeated experiments with random synthetic data generated as appropriate. We present results with 7 different cache maintenance algorithms:

- **LRU** (Algorithm 5.1)
- **ViP-LFU** (Algorithm 5.2)
- **UCD** (Algorithm 5.4)
- **AC** (Algorithm 5.5)
- **PC** (Algorithm 5.7)



(a) Uniform Distribution - 60/30

(b) Zipf Distribution - 60/30

Figure 30: Uniform and Zipf Distribution (60% data updates and 30% annotation inserts)

- **UPC** (Algorithm 5.8)
- **No Cache**

In both distributions, LRU and ViP-LFU outperform all other algorithms, We report the breakdown of query time and cache management time (i.e., overhead) in Figure 30 and Figure 31 respectively. Figure 30 has 60% data update frequency and 30% annotation insertion frequency, under uniform and Zipf query trace distribution. It is clear that it takes longer time to retrieve the data and annotations under uniform query trace distribution (scale of (0, 180ms) in Figure 30(a) vs scale of (0, 100ms) in Figure 30(b)). Since the Zipf distribution can highlight the difference between algorithms more precisely, for the rest of the evaluation we used a Zipf query distribution unless otherwise indicated. In Figure 31, we present more results of the query times at 5% data update frequency and 2% annotation insertion frequency. The cache management time is reduced dramatically since the data update and annotation insertion operations are much less in this test case than the operations in previous test case (Figure 30). Regardless of the test case, the no-cache scheme always takes the longest retrieval time.

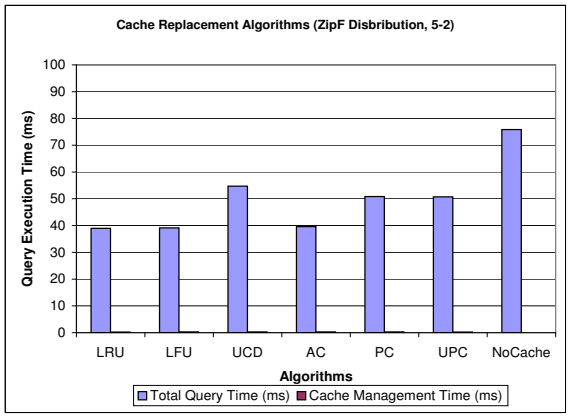


Figure 31: Zipf Data Distribution (5% data updates and 2% annotation inserts)

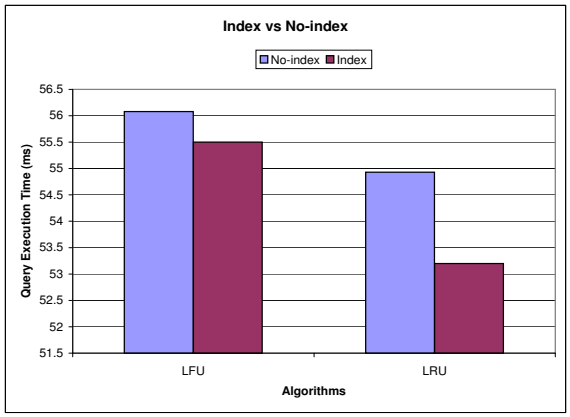


Figure 32: With or Without Indexing

7.3.2 Indexing (Figure 32)

Adding an index on an important attribute will improve the overall query time. For example, in the 5% data update situation, we compared the query time with and without index (B-trees) on the timestamp attribute in the annotation registration table. The cache was maintained with the lazy annotation propagation method. The result is shown in Figure 32 and we can see a clear advantage. For the rest of the evaluation we used indexed attributes unless otherwise indicated.

7.3.3 Caching Algorithms (Figure 33 - Figure 52)

First of all, we tested the effect of different caching schemes as evaluated by the total query execution time. The result is shown in Figure 33.

There are two ways to implement *eager annotation propagation*, one is to apply data and annotation changes in the actual data and annotation view tables. For each data or annotation change, all related data and index tables will be updated. It is obvious that it takes a lot of effort and may not be necessary for the queries. The other method is to apply changes in the cache that stores the existing query results. It is an on-demand method to store the query result, plus an "eager" approach to apply annotation changes in the cache, if there are any. In this thesis, we call

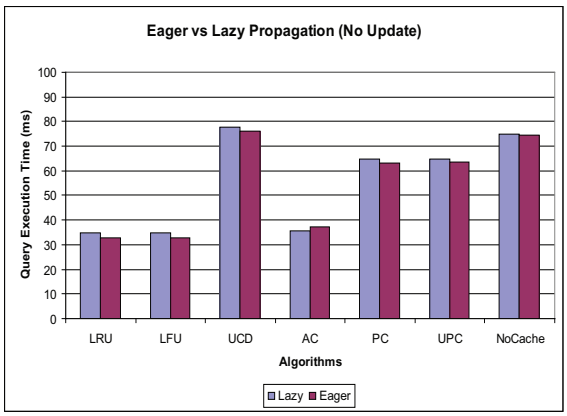


Figure 33: The Query Execution Time with Different Caching Algorithms

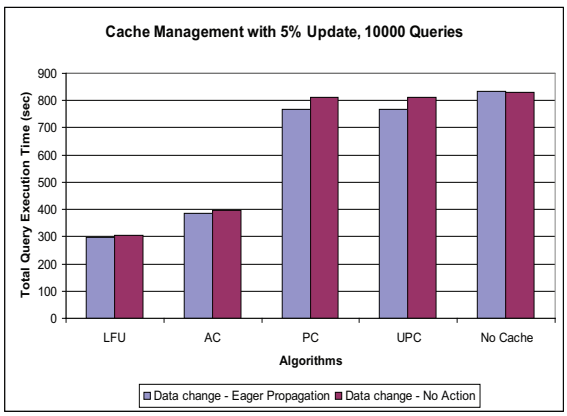


Figure 34: Different Cache Operations When Data Changes

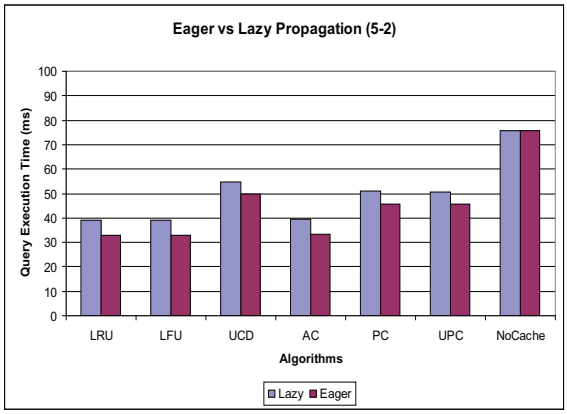


Figure 35: The “Eager” vs “Lazy” Annotation Propagation Case I

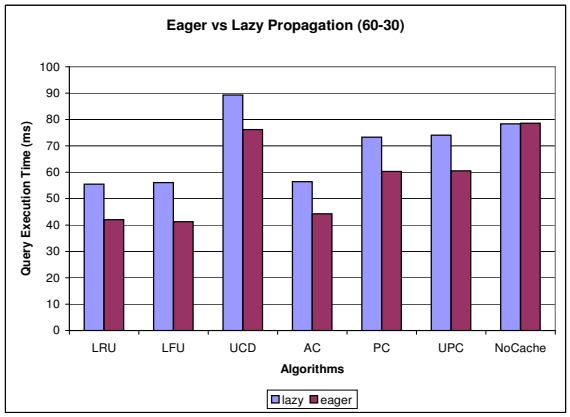


Figure 36: The “Eager” vs “Lazy” Annotation Propagation Case II

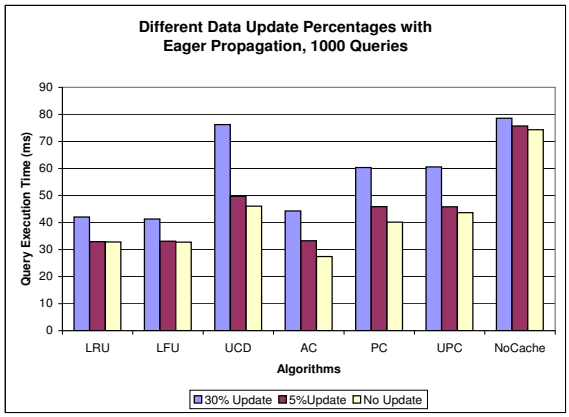


Figure 37: Different Data Updates Percentages with Eager Propagation

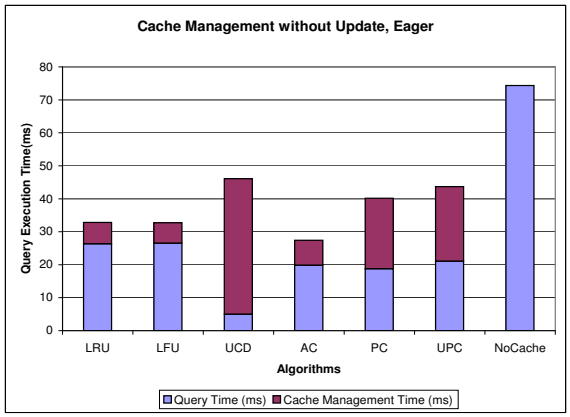


Figure 38: Cache Management without Any Updates

this method “eager annotation propagation”, and use it in all experimental evaluations.

When the annotation views have been inserted into the database, there are two strategies to update the cache, one is “eager”: to propagate the changes into the cache as soon as possible, so when the query comes, there will be less query time. Of course, the cache maintenance time will be increased. The second scheme is to handle the update in a “lazy” fashion. When the annotation views have been inserted, the system will have assigned the appropriate timestamp. When there is a query coming and a cache hit generated, the system needs to verify the correctness of the cache content by examining the recently inserted annotation views with the newer timestamp. This trade-off between lazy and eager is shown in Figure 35. At a dynamic situation with more updates and insertions, the query time is shown in Figure 36.

We tested two cache operation schemes when data changes: *No action* or *Eager propagation*, as discussed in Section 5.2.1.2. The result is shown in Figure 34. We tested in different configurations, eager data propagation always performs better than the “lazy” (No action) scheme since the eager method removes nonexistent data from the cache to save cache space, thus avoiding unnecessary or inaccurate evictions once the cache is full. For the rest of the evaluation we used eager data propagation scheme unless otherwise indicated.

We want to analyze how the eager and lazy annotation propagation schemes work in the cache,

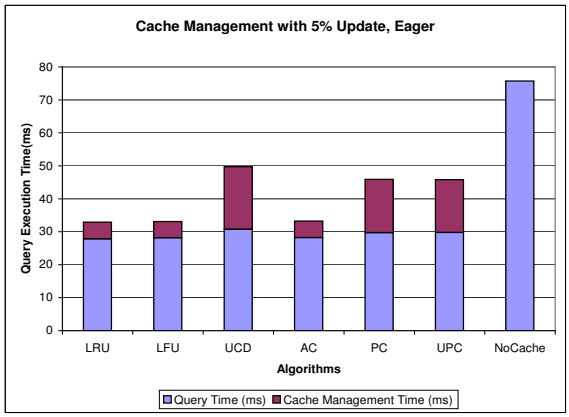


Figure 39: Cache Management with 5% data Updates - Eager

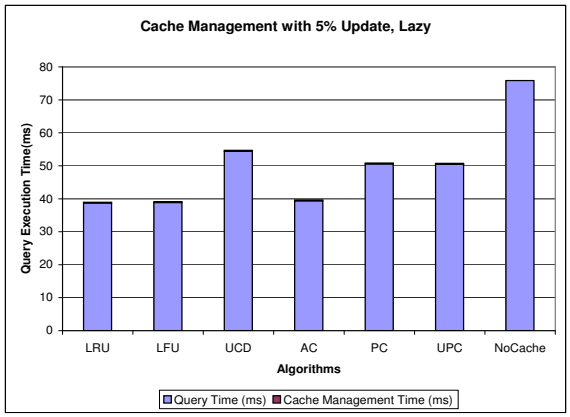


Figure 40: Cache Management with 5% data Updates - lazy

so we gathered the information of query time at different data update situations in Figure 37, under the eager scheme. It is reasonable that when less data is updated, the query times are reduced, though in different cache replacement algorithms, the performance varies. We also measured the cache management time vs query time at 5% data update case, with the eager scheme in Figure 39, with the lazy scheme in Figure 40, and without any annotation updates with the eager scheme in Figure 38. UCD, UPC, and PC algorithms take considerable time to manage the cache, so their performance in terms of query time is worse than that of the other algorithms. The lazy scheme does not update the cache until the query time, so its cache management time is ignorable.

When we introduce updates both on annotation views and data, there is the tradeoff between query time and cache management time. The cache processing times were measured under both strategies, and are shown in Figure 41 and Figure 42. The cache management time is the time to verify if there is a cache hit and retrieve related annotations to the data item. The extra time spent on cache is the time purely spent on cache maintenance such as cache update or eviction computation. In this set of experiment, if we put all kinds of operation time on one bar, it means we count the individual time separately, the total processing time should be the sum of all kinds of operation. For example, in Figure 41, the total query processing time = query time + cache management time + extra cache management time.

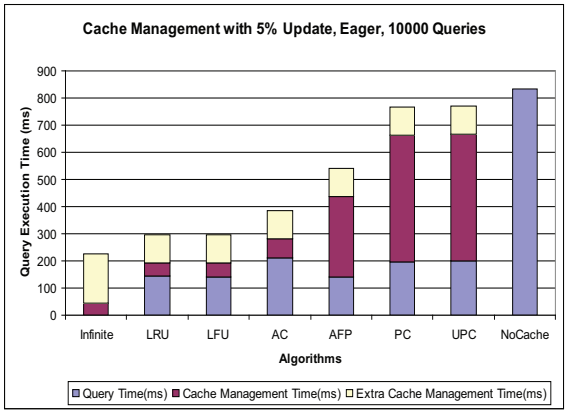


Figure 41: The Query Time vs Cache Management Time - Eager

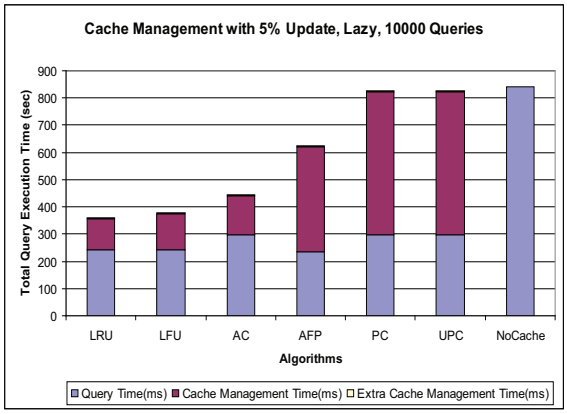


Figure 42: The Query Time vs Cache Management Time - Lazy

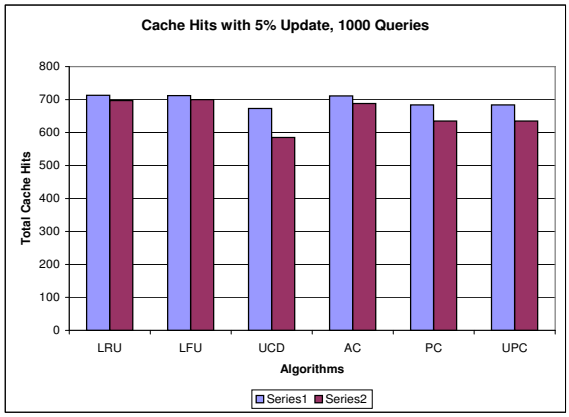


Figure 43: The Total Cache Hits of 1,000 Queries

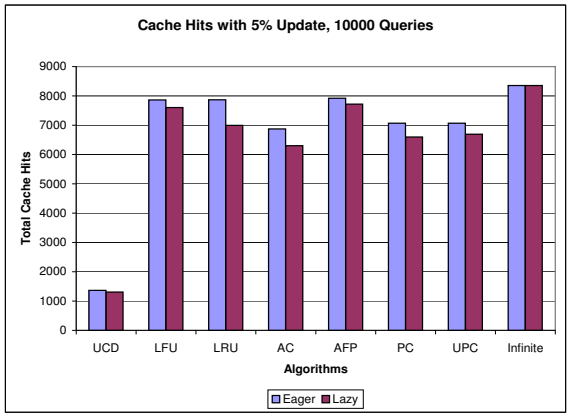


Figure 44: The Total Cache Hits of 10,000 Queries

The cache hits vary according to the different algorithms, data distribution, and query patterns. We illustrate two results from 1000 queries and 10,000 queries at 5% data update and 2% annotation insertion accordingly, in Figure 43 and Figure 44. The eager scheme always has more cache hits since it takes extra time to update cache items before the queries.

In the case of different number of annotation views, we compared the query times and cache management times with 100, 1000, 5000, and 10000 annotation views in Figure 45. It is the total query time and total cache management time under 10,000 queries in seconds, to make it look more clearly, in the total node found and the number of cache hits in Figure 46, we used a log scale. Even with exponential annotation view increase, our system handles it gradually.

We compared the query times of different algorithms on 10,000 queries in Figure 47. To make the figure simple and clear, we gathered query points at a step of 100 to represent the whole query trace. The optimal case is the infinite cache, other than that, LFU and LRU has fast query times, while AC and PC take longer times. To explore it in a more detailed study, the query time series for the LFU algorithm with different cache sizes is illustrated in Figure 48. It is obvious that infinite cache or bigger cache sizes speed up the query processing while the system with smaller cache sizes takes a longer query time since it requires extra time to manage the cache (such as cache replacement). In next Figure 49, Figure 50, Figure 51, and Figure 52, we illustrated the query time series of each individual algorithm over 1000 queries. PC (Figure 51) and AC(Figure 52) algorithms have less cache hits, thus the overall query time is longer than LFU (Figure 49) and LRU (Figure 50) algorithms, which supports and explains the previous result.

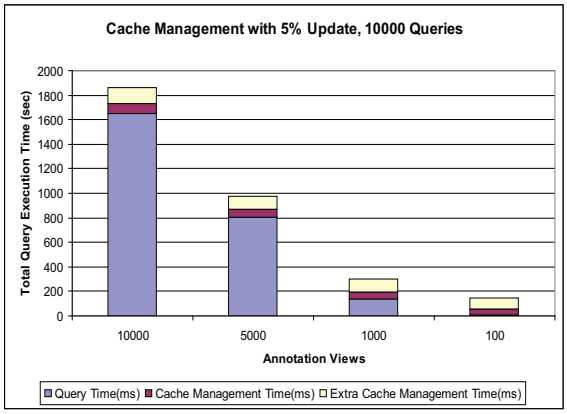


Figure 45: Query Time with Different Annotation Views

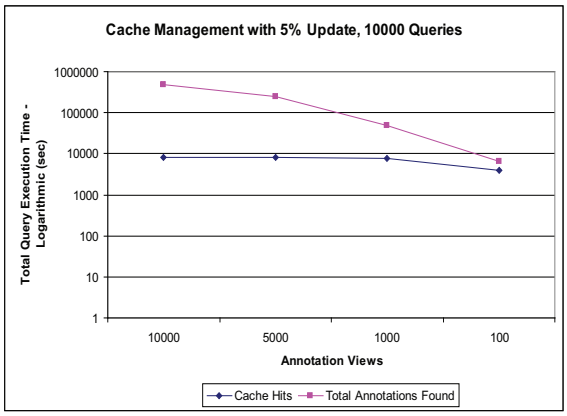


Figure 46: Cache Hits and Annotations Found with Different Annotation Views

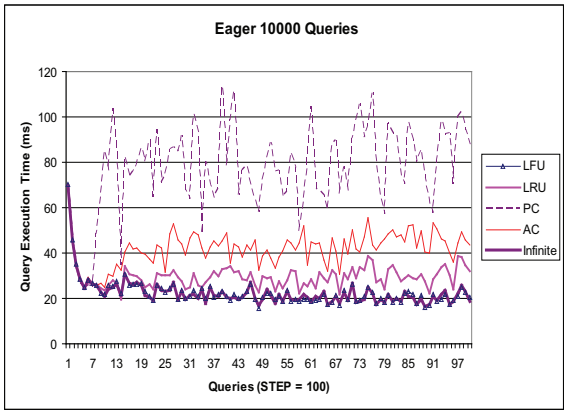


Figure 47: Query Processing Time Over Time

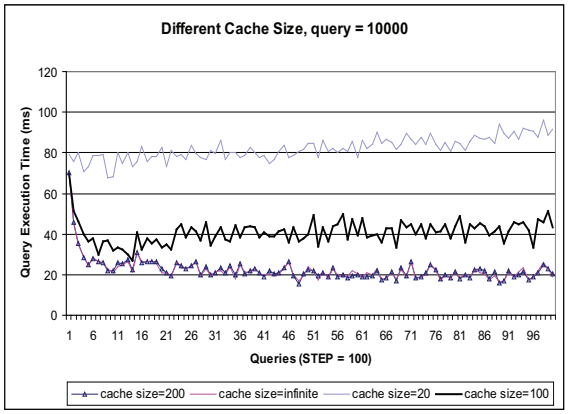


Figure 48: Query Processing Time Over Different Cache Sizes

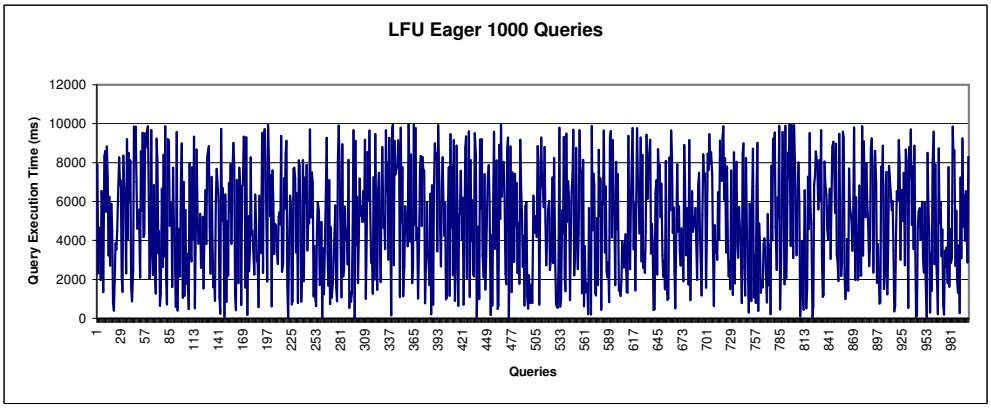


Figure 49: Query Processing Time of LFU Algorithm

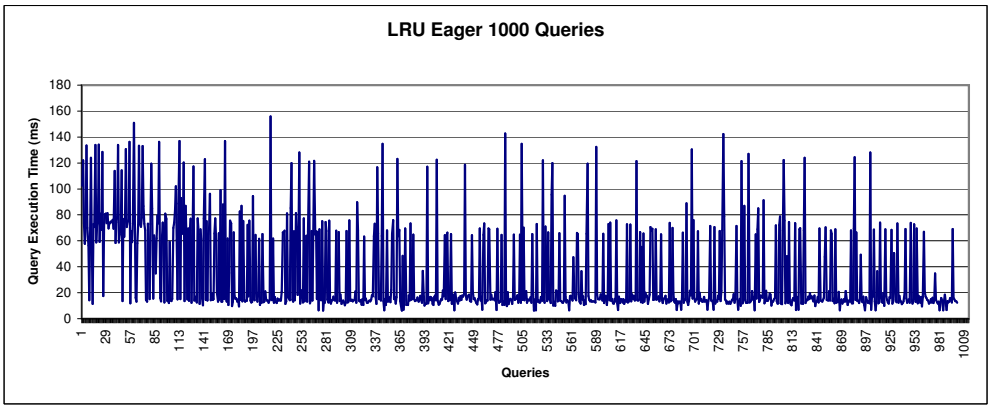


Figure 50: Query Processing Time of LRU Algorithm

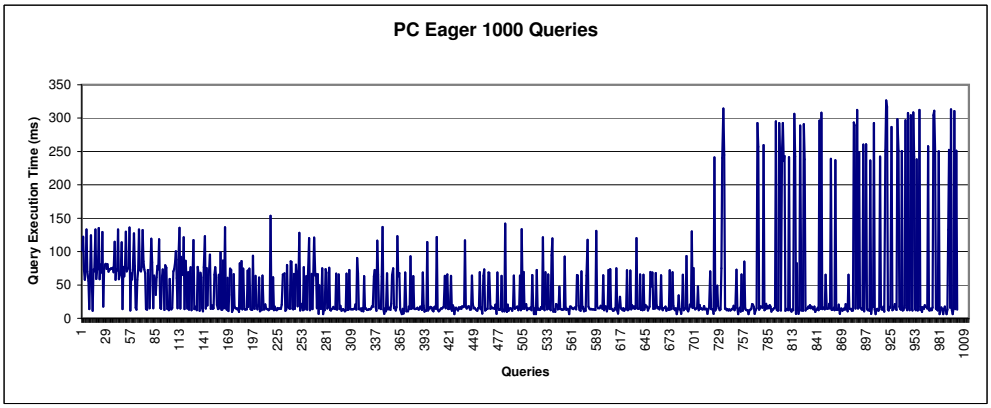


Figure 51: Query Processing Time of PC Algorithm

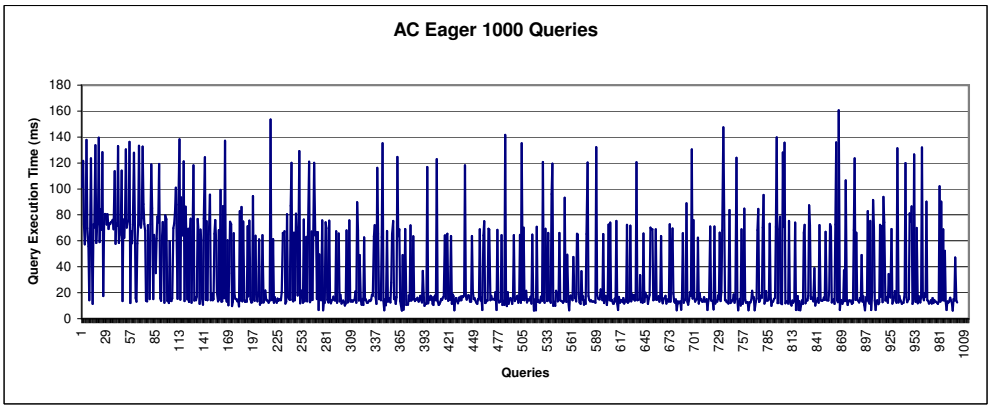


Figure 52: Query Processing Time of AC Algorithm

7.4 COMPARISON OF VIP TO MMS - DATA SET 2

In this section, we compare our proposed ViP framework to the state-of-the-art, MMS, using data set 2 which was described in Section 7.1.2.

7.4.1 View-based Annotation Propagation (Figure 53, 54, 55 and Table 18)

To test our proposed system with other systems, we compared the query execution time of our system, ViP, with MMS. It is the latest and the most related work (introduced in Section 7.1.2). Both systems retrieved the same annotations associated with the same queries. In our first experiment, we varied the total number of annotation registrations (Figure 53). ViP always outperformed MMS due to its caching. With more annotation views registered, ViP gained more benefit. In the case of 50,000 annotation views registered, ViP took about 25% less time, proving that ViP works better for large numbers of annotation views.

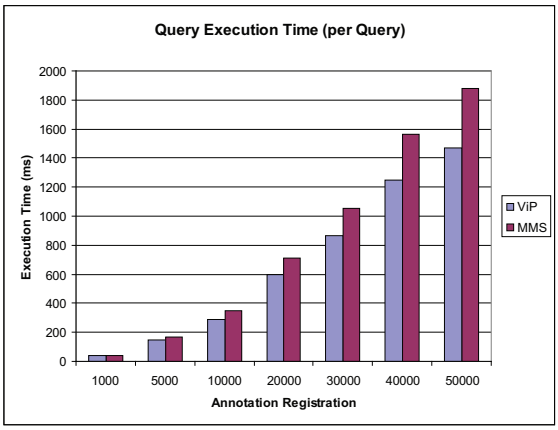


Figure 53: Query Execution Time

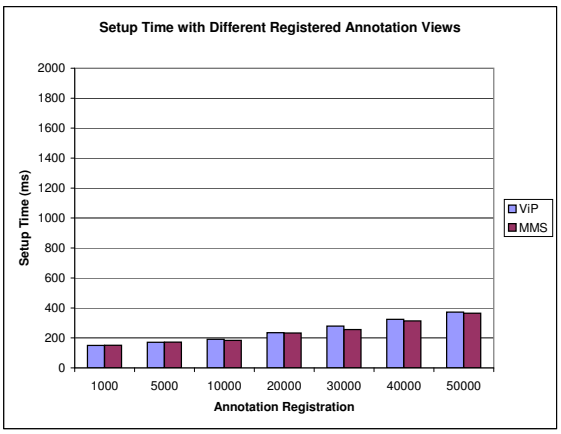


Figure 54: Setup Time

We also measured the confidence interval of the result to make sure they are statistically significant. In the case of 1,000 queries with 50,000 annotation views, the 95% confidence interval for ViP mean query execution time (ms) is $(1468.06 \mp 7.36) = (1460.7, 1475.42)$; the 95% confidence interval for MMS mean query execution time (ms) is $(1878.91 \mp 4.05) = (1874.86, 1882.96)$. The

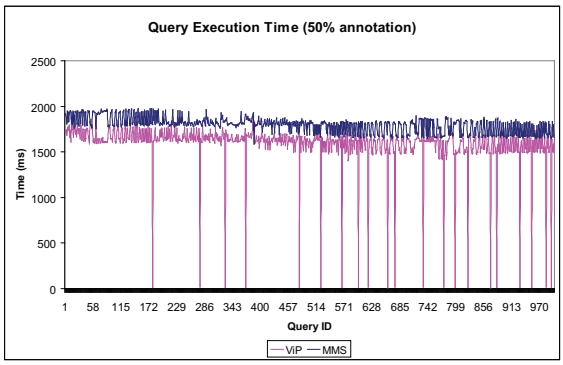
results presented in the thesis were acquired as the average value from 1000 repeated experiments with random parameter settings. Due to the limited space, not every confidence interval is listed here; all results were similar to this experiment.

In all experiments, we started with 80% annotation view and path insertions. While the query traces were being executed, the remaining 20% of the annotation registrations were uniformly distributed over the experiment time period. We assume each query or annotation registration operation is atomic. The query execution time includes (1) data query execution time, (2) annotation lookup time, (3) cache lookup time if cache is used, and (4) cache management time. The setup time includes (1) data insertion time, (2) annotation registration time, and (3) cache setup time. The setup times per query for both systems are shown in Figure 54. Although ViP took extra time to manage the cache, the overhead is negligible compared to the gain from the query execution time.

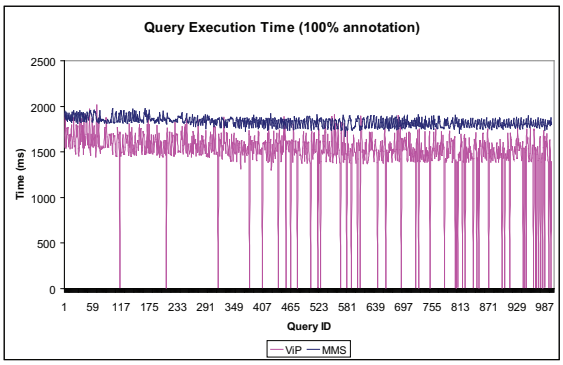
In the next set of experiments, we investigated the effect of various annotation densities, which is the percentage of data associated with annotation views. In Figure 55, 1000 queries were plotted in each subfigure to display the various query execution times. The density was changed from 50% to 200%, and the query execution time increased accordingly. In these figures, a vertical line corresponds to a cache hit on all annotations the query expects to return. We found in the extremely dense case, which is 200% in Figure 55(d), that ViP had so many cache hits, that the overall query execution time was reduced significantly. The detailed summary of average query execution time is presented in Table 18. Again, ViP works better in large scale of annotation views because of its optimized scheme. For fairness, we used 10% annotation density, which is the least beneficial setting for ViP, in all other experiments.

7.4.2 Annotation Propagation with Caching (Figure 56)

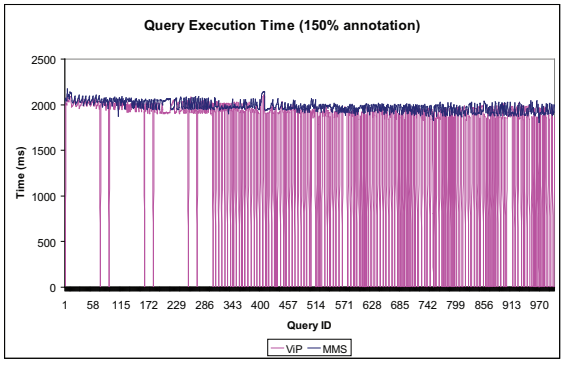
The cache management time was insignificant compared to the query execution time, shown in Figure 56. The number of annotation views varied from 1,000 to 50,000. Even with 50,000 annotation views, the cache management time is just about 3% in query execution time. As the previous experiment results, we used ViP-LFU cache replacement algorithm in the test cases.



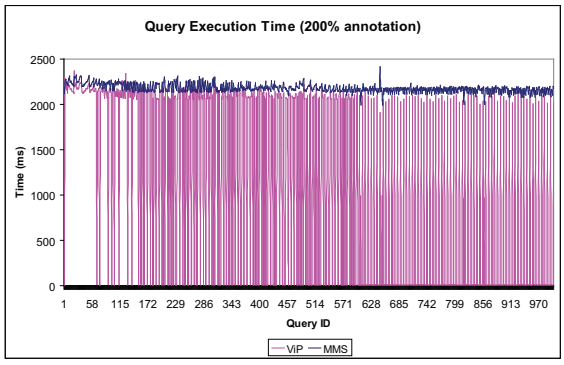
(a) Query Execution Time of 50% Annotation Density



(b) Query Execution Time of 100% Annotation Density



(c) Query Time Execution of 150% Annotation Density



(d) Query Execution Time of 200% Annotation Density

Figure 55: Query Execution Time with Different Annotation Densities

Percentage	40%	50%	60%	70%	80%	90%	100%	150%	200%
MMS Time (ms)	1804.81	1808.66	1812.50	1867.94	1878.02	1895.5	1928.8	1979.9	2178.7
ViP Time (ms)	1471.38	1419.73	1445.81	1499.44	1394.39	1386.3	1484.2	1483.8	1250.9

Table 18: Query Execution Time with Different Annotation Densities

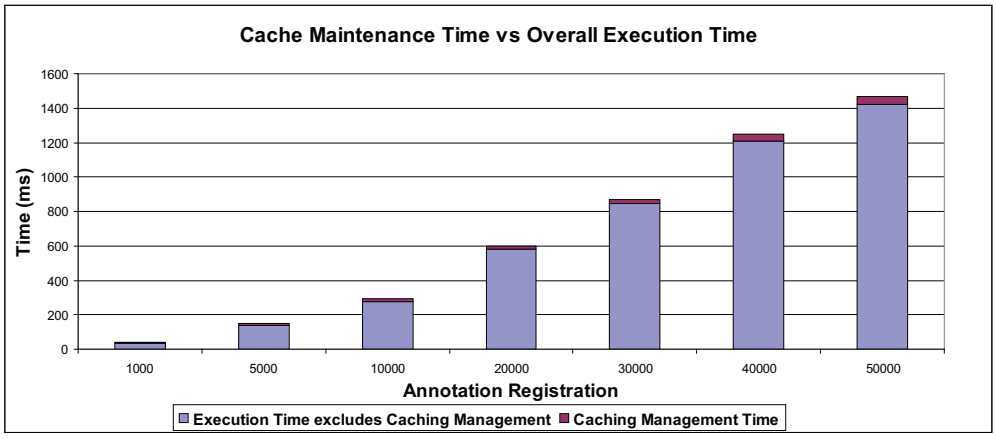


Figure 56: Caching Time

7.5 EVALUATION OF NETWORK SEMANTICS - DATA SET 2

In this section, we present experimental results from our evaluation of network semantics in the ViP framework, using data set 2 which was described in Section 7.1.2.

7.5.1 User-driven Annotation Paths Propagation (Table 19)

One of the unique features of the ViP framework is the user-driven network semantics, which was presented in Section 4.2. We conducted a set of experiments where we varied the U-HAF limits (hops allowed to follow). The results is presented in Table 19. It is obvious that with deeper hops search, more annotations got matched and more time was needed to retrieve them. Nonetheless, ViP increased the query execution time only gradually.

Path Hop(s)	**1**	**2**	**3**
Time (sec)	10.1445	11.1853	13.5833
Annotations Found	269	278	289

Table 19: Path Propagation for User-driven Network Semantics

7.6 EVALUATION OF ACCESS CONTROL ON ANNOTATION VIEWS/PATHS

In this section, we present experimental results of our experimental evaluation of the user-driven access control features of the ViP framework.

7.6.1 User-driven Access Control on Annotation Views and Paths (Figure 57 - Figure 60) - Data Set 2

Access control on annotation views and paths was proposed in Sections 4.3 and 4.4. Not only users may issue queries to express their search preferences, but users can also specify public/private

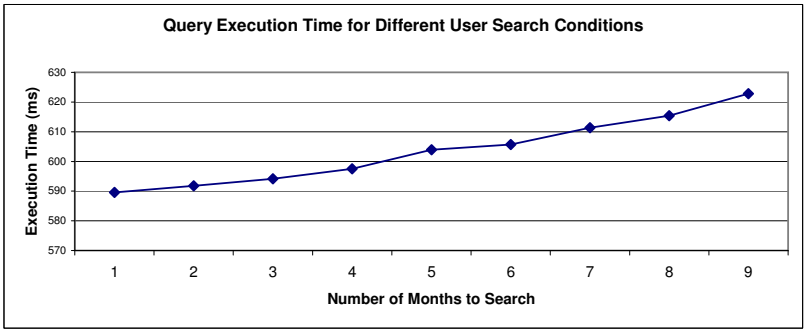


Figure 57: Query Execution Time for Different User Search Conditions

annotation views when they register the annotations. The first set of experiments on **data set 2**, in Figure 57 and Figure 58 illustrates how the different search coverage affected the query execution times and the number of annotations found. The most restrictive user-specified condition dropped down the query execution time as well as the annotations associated.

On the other hand, Figure 59 and Figure 60 present the query execution times with different percentages of public annotation views and annotation paths. In these cases, the remaining "private" annotation views and paths were uniformly distributed among all users. The query execution time almost dropped linearly as the public annotation views decreased; however, it dropped faster when the public annotation paths were decreased. Since annotation paths have the transitive

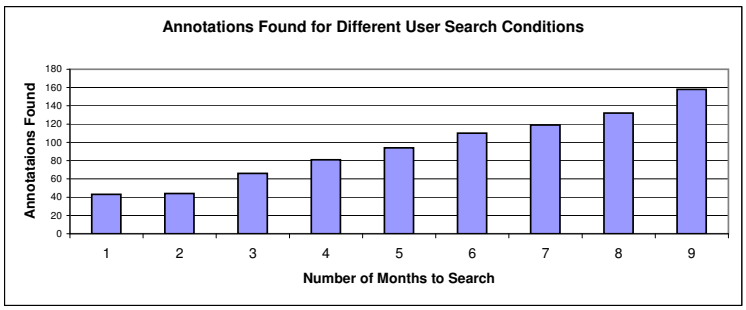


Figure 58: Annotations Found for Different User Search Conditions

property, once the dependent views are not visible, it may speed up the query execution time exponentially. This essentially works like a first priority “filter” to reduce the query search time. In general, we expect such user-driven features to have a compound effect if used together, dramatically reducing query execution times, if taken advantage of, as ViP does.

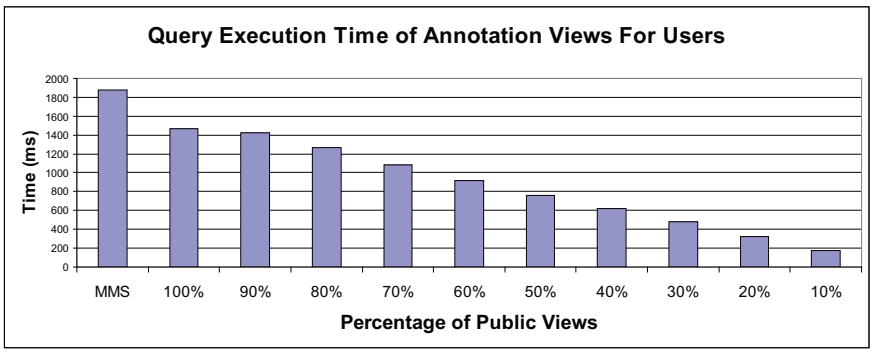


Figure 59: Query Execution Time with Different Public Annotation View Percentages

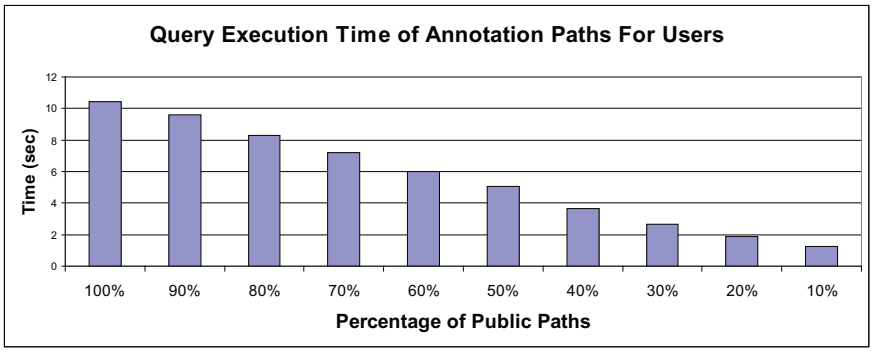


Figure 60: Query Execution Time with Different Public Annotation Path Percentages

7.6.2 Path Strength (Figure 61) - Data Set 1

In Section 4.4.2, we proposed an approach to consider path strengths. In Figure 61, we compared the processing time with different path strengths. We made a slight adjustment, the strength format is given as W(eak)-R(egular)-S(trong), a reversed order compared to Figure 12. When we have more strong paths, more annotations will be propagated via the paths, thus we have more annotations found, and therefore the query processing time is increased also.

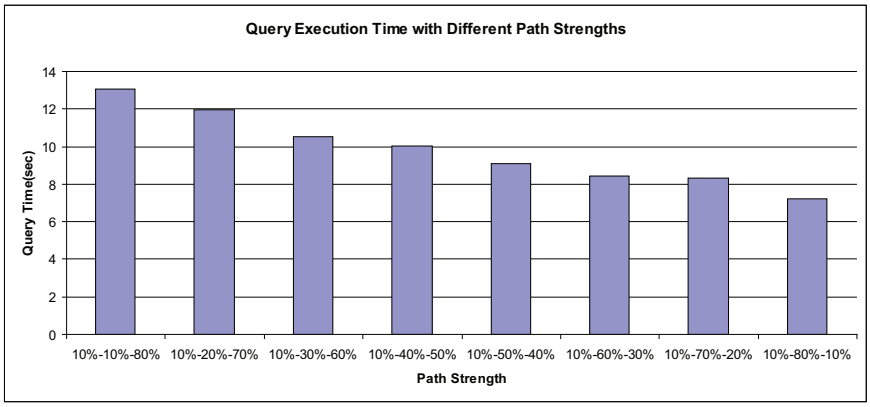


Figure 61: Query Processing Time Over Different Path Strengths

7.7 EVALUATION OF SCALABILITY - DATA SET 3

In this section, we present experimental results with non-scientific data sets, namely movie data, which are typically used in personalization and recommendation system studies. These datasets, allow us to evaluate the scalability of our proposed ViP framework. We used data set 3 which was described in Section 7.1.3.

7.7.1 MovieLens 100k Data Set (Figure 63, 62, 64, 65, 66, and 67)

We measured the query execution time (in seconds), the number of annotations found, the annotation and data setup time, and the cache hits, if the cache is on. The results presented were acquired

as the average value from multiple repeated experiments.

In all experiments, we simulated 100 critics annotating ratings on the movie release date and the genre attributes. These critics have various confidence values. The annotations on ratings were generated with the Zipf distribution. Approximated to “80-20 law”, most (such as 80%) annotations are on certain (such as 20%) ratings of movies.

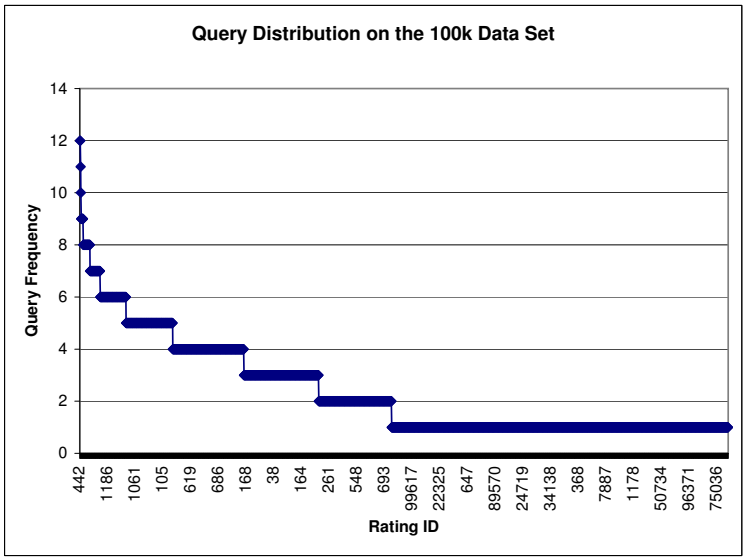
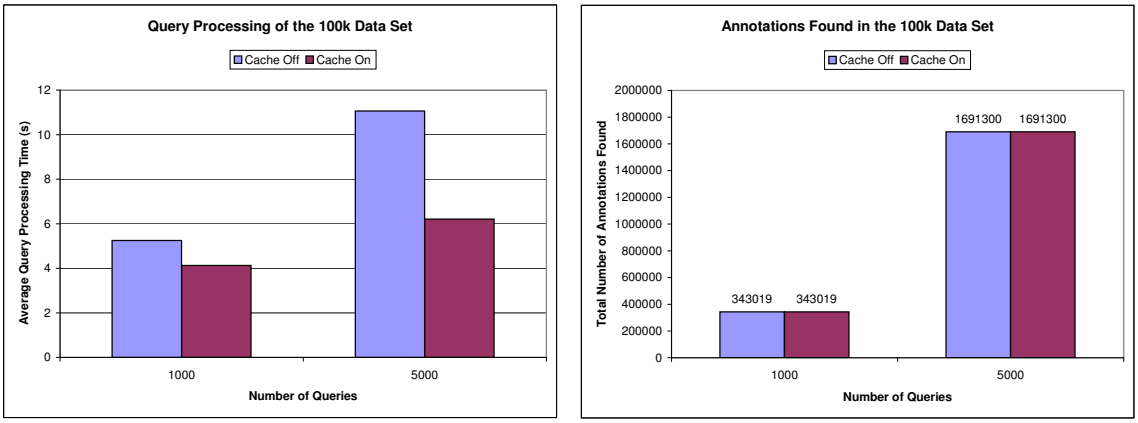


Figure 62: Query Distribution on the 100k Data Set

Query Trace:

There are two scenarios for generating query traces. The first one is that queries are randomly picked and are uniformly distributed over all ratings and movies. A more realistic workload is one that we have the “hot spot” scenario, which means most (such as 80%) queries are on certain (such as 20%) topics, that is, viewers want to read comments or annotations of latest movies, or some “hot” movies. This approaches human behavior more naturally, and makes sense to users. In the later experiments we used such distribution scenario to generate query traces. Unless claimed otherwise, the query trace has 1,000 individual queries. The frequency of queries on ratings is illustrated in Figure 62. In the case of 1,000 and 5,000 queries which are “hot spot” distributed, the query processing time is presented in Figure 63(a) and the total number of annotations found is presented in Figure 63(b). It is expected that the query processing with a cache works faster



(a) Query Processing in the 100k Data Set

(b) Annotations Found in the 100k Data Set

Figure 63: Query Execution Time and Annotations Found in the 100k Data Set

than the query processing without a cache. In the case of 5,000 queries, there are more repeated queries, the gap between cache on and off is enlarged. For comparison purposes, in all experiments of this data set, we fixed the query trace, that is, every test case with the same query trace returned the same amount of total annotations found, no matter if it is with cache on and off (such as Figure 63(b)), and no matter what the cache capacity is.

Query Processing Time:

We assume each query or annotation registration operation is atomic. The query execution time includes (1) data query execution time, (2) annotation lookup time, (3) cache lookup time, if the cache is used, and (4) cache management time. The setup time includes (1) data insertion time, (2) annotation registration time, and (3) cache setup time. A summary of the setup time is presented in Figure 64.

In Figure 63(a) and 63(b), when the cache was on, the cache size was set to 30% of the query requirements. In this thesis, *cache capacity* means the percentage of cache size vs the query trace size. We varied the cache capacity from 10% to 100% (infinite cache), the average query times are presented in Figure 65. The query process time decreases dramatically from cache off to 10% cache capacity, then decreases gradually, once it reaches 70% cache capacity, the benefit increase

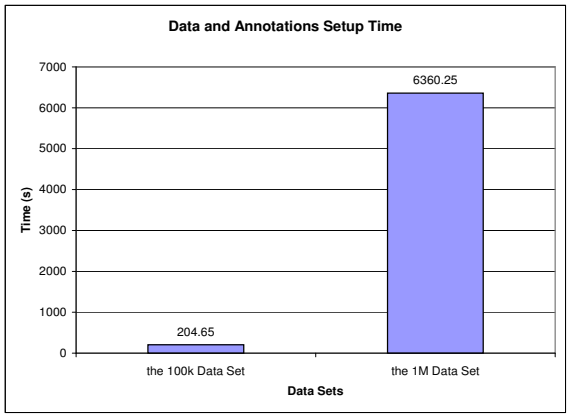


Figure 64: Setup Time of the 100k and the 1M Data Set

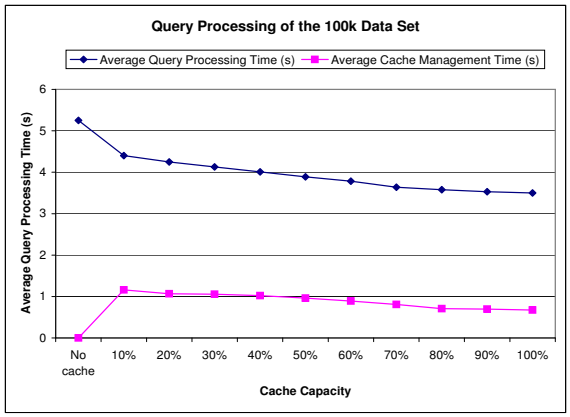


Figure 65: Query Processing of the 100k Data Set with Different Cache Capacities

is less and less. In the case of a cache hit, the processing will be faster compared to a regular query. As such, we explored the cache hits under difference cache capacities. The results are shown in Figure 66. We used *Cache Hits Percentage* to present the ratio of cache hits over the total number of queries. In general, the bigger the cache capacity, the faster the query processing will be. However, when it reaches a certain point, the cache hit rate will not be increased. In this test case, after 70% cache capacity, the cache hits percentages are almost the same. It means the system meets the need of the maximum capacity, after that, no gain we can get from increasing the cache size.

The query execution also depends on the cache management time. In some cases, the overhead of cache management may reduce the query efficiency, and the data distribution and query pattern affect the cache efficiency greatly. In summary, the query processing time is affected by the data distribution, the annotation distribution, and the query trace distribution.

To explore the behavior of every query, we plotted the query time over time. In Figure 67, there are two series of query processing, one with cache capacity of 30% while the other one at 75%. In the bigger cache capacity case, there are more cache hits, and the overall query processing time is decreased.

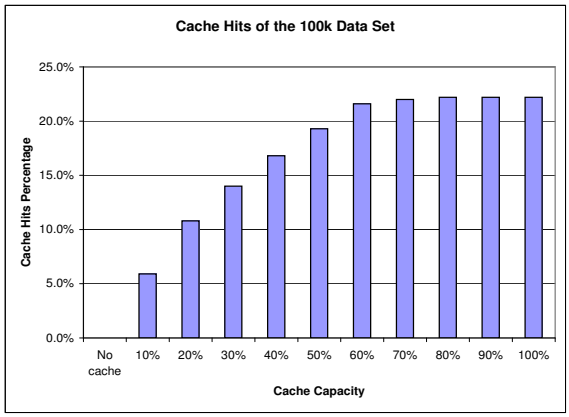


Figure 66: Cache Hits in Queries of the 100k Data Set

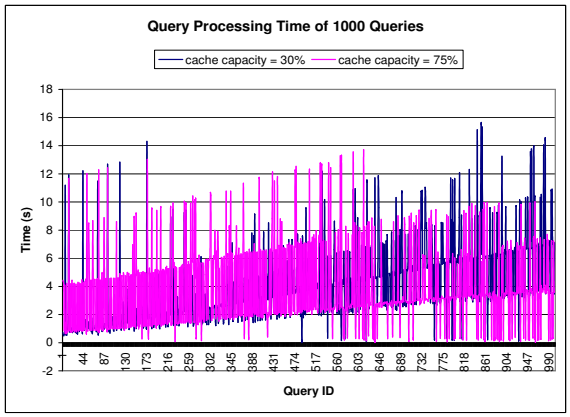


Figure 67: Processing Time Sequence of Query Trace on the 100k Data Set

7.7.2 MovieLens 1M Data Set (Figure 68, 69, 70, and 71)

In this set of data, we loaded "heavier" annotations, that is, a data item is associated with more annotations according to annotation views. Compared to the 100k data set, each rating comes with annotations between [212, 923], in current testing case, each rating comes with 500 $\sim$ 800 annotations. The annotation distribution is illustrated in Figure 68. We also gathered the information of annotation frequency on movies, which is presented in Figure 69. The annotations associated with each movie vary between [0, 68,793]. In this set of experiments, each query comes with 2 to 4 unions on attributes such as movie id, creation date, or viewer id etc, while in the 100k test case we tested queries on the movie id attribute. The total number of annotations found is 811,020, the average time of retrieving each annotation varies from 0.01s to 0.017s. In the case without cache, the time is 0.023s. In the 100K data sets, the total number of annotations found is 343,019, the average time of retrieving each annotation varies from 0.01s to 0.013s. In the case without cache, the time is 0.015s. Since the average annotation retrieval time includes the time to process the query and combine result sets for the query, it is understandable that this test case, which has more operations, takes slightly more time than the 100K test case.

The query processing time is presented in Figure 70. In the 100% cache capacity case, it

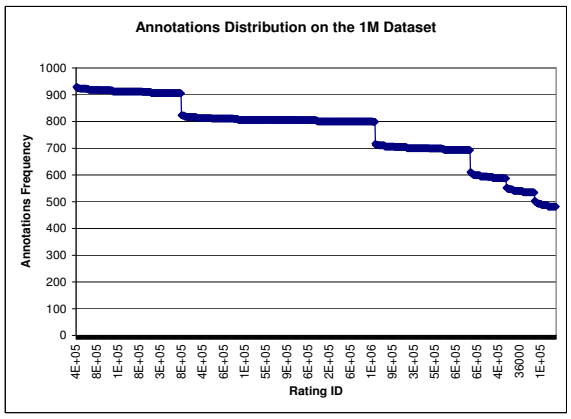


Figure 68: Annotations Distribution on Ratings in the 1M Data Set

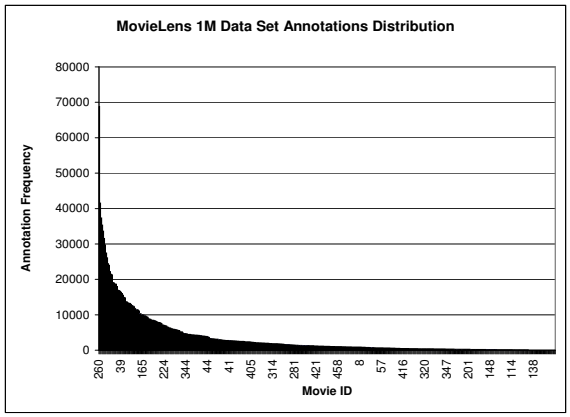


Figure 69: Annotations Distribution on Movies in the 1M Data Set

has the shortest query time, while in the case of no cache, it has the longest query time. On the other hand, the cache management time is decreased as the cache capacity increases, though the difference is insignificant, but the gap between no cache and cache is relatively significant. The cache hits increase as the cache capacity increases (Figure 71), after 60% - 70% cache capacity, the performance is the same, which means there is a saturation point.

7.7.3 MovieLens 10M Data Set (Figure 72, 73, 74, and 75)

In this data set, the ratings table has 10,000,054 records, which took 73,097.9s,(i.e., about 20.3hours) to load into the database. The setup times are presented in Figure 72. Compared to data load time, the annotation view registration takes insignificant time, which is one of the advantages of our scheme. In this experiment we used the fixed query trace, where each query comes with 2 to 4 unions. In multiple runs, the total number of annotations found is 128,900. The number of annotations associated with a data item is moderate compared to previous test cases. Each rating comes with about 50 $\sim$ 350 annotations. The average time of retrieving each annotation varies from 0.01s to 0.012s. In the case without cache, the time is 0.013s.

We evaluated our proposed framework using two different query traces. The first query trace

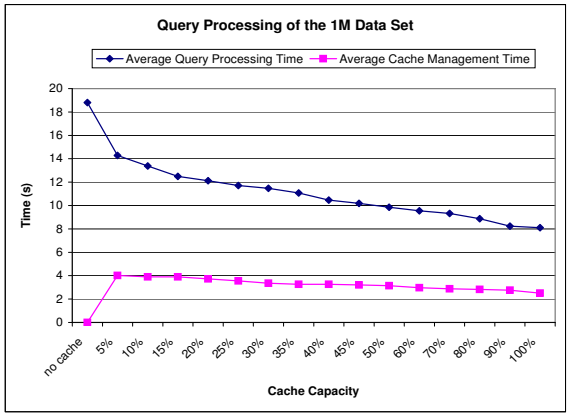


Figure 70: Query Processing with Different Cache Capacities of the 1M Data Set

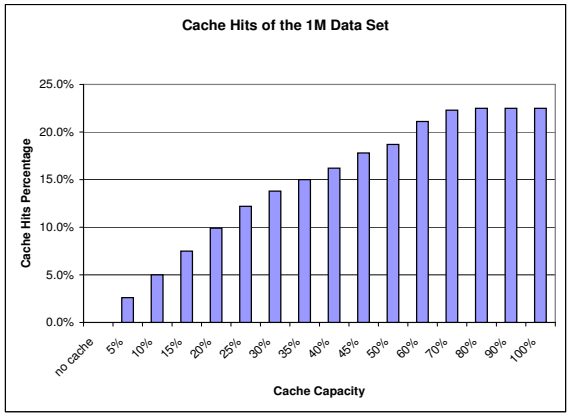


Figure 71: Cache Hits in Queries of the 1M Data Set

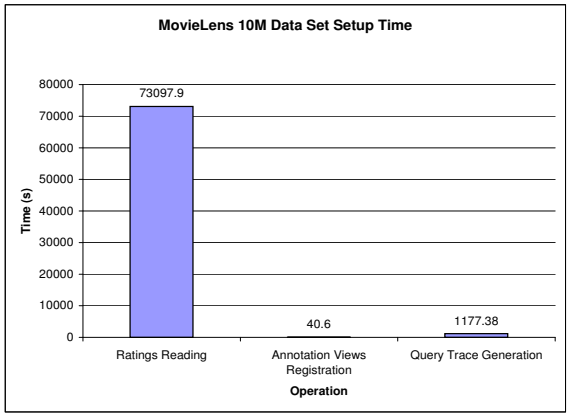


Figure 72: Setup Times of the 10M Data Set

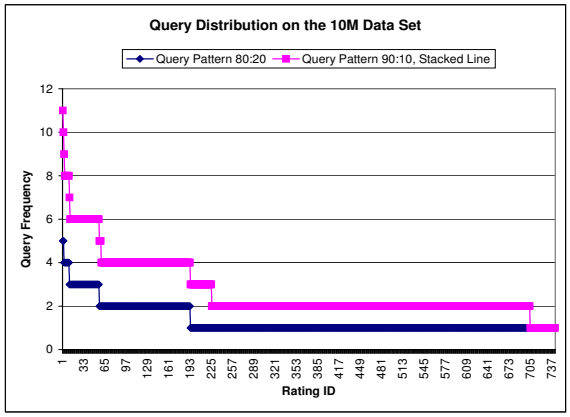


Figure 73: Query Traces of the 10M Data Set

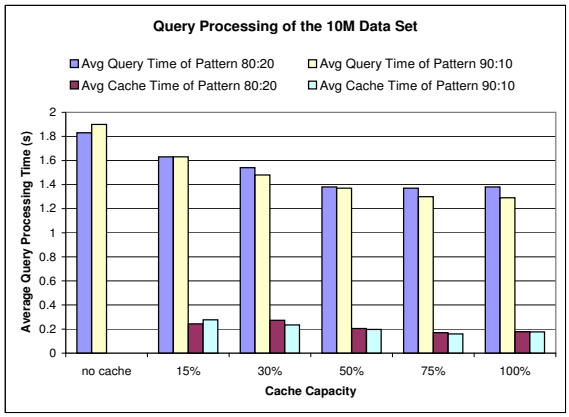


Figure 74: Query Processing with Different Cache Capacities of the 10M Data Set

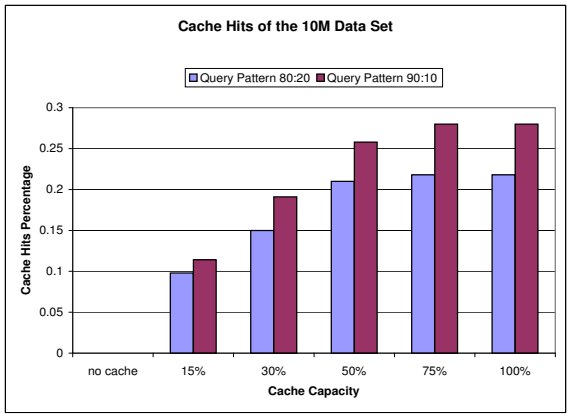


Figure 75: Cache Hits in Queries of the 10M Data Set

randomly picked 80% of queries from the "hot spot" pool (it is called "Query Pattern 80:20") while the second one chose 90% of queries from the pool, which is called "Query Pattern 90:10". The two query traces are presented in Figure 73. The figure uses the stacked line chart type, which means the query pattern 90:10 has query frequency varied from 6 to 1, while the query pattern 80:20 has a query frequency [1, 5].

In Figure 74, the average query processing times are presented with different cache capacities. In Figure 75, the cache hits are illustrated. It is reasonable that if there are more queries from the "hot spot" pool, there is a bigger possibility that the query result is cached, thus the overall query processing time is reduced and the cache hit rate is increased. In the case of no cache, the query processing time is the slowest, although the cache management cost is zero.

There is one more observation, in this experiment, "Query Pattern 90:10"without cache takes 1.9s average query processing time while "Query Pattern 80:20"without cache takes 1.83s. The possible reason is that there are 128,900 annotations found in the test case of "Query Pattern 80:20", while there are 130,550 annotations found in the test case of "Query Pattern 90:10". More annotations in the results means there are more retrieval operations. Nonetheless, in the test case with cache, the average query processing of "Query Pattern 90:10" takes less time than "Query Pattern 80:20". There is a bigger chance to take advantage of the caching scheme, thus the query

time is reduced, and the gap between the query time of two patterns increases as the cache capacity increases.

8.0 CONCLUSIONS

In this chapter, we conclude the thesis and present some initial ideas for extensions of the work, as part of future work.

8.1 CONTRIBUTIONS

ViP works as an annotation framework for large-scale data, which effectively addresses the questions generated from the usage patterns we observed and which we discussed in Chapter 1, namely,

1. ***support for annotations that are scalable both from a system point of view and a user point of view,***
2. ***support for propagation and querying of annotations in annotator/user-defined ways.***

We reached our goals as follows:

1. we introduced new annotation definition and propagation methods,
2. we introduced ViP-SQL, an SQL-based query language supporting annotation propagation,
3. we introduced user-driven features, such as time semantics and network semantics, to enable users to personalize annotation propagation,
4. we introduced the use of views as a formal mechanism to implement the new annotation definition and propagation features,
5. we proposed three types of annotation queries, and discussed the corresponding algorithms,
6. we utilized techniques such as caching and proposed cache replacement algorithms to significantly improve the performance of query execution,

7. we experimentally evaluated the ViP framework using a real system implementation with real and synthetic data sets to cover the entire spectrum of workloads and environmental settings.

We believe this work would further lead to new techniques that process annotations associated with large-scale data in a accurate and efficient way, as well as enable further the managing data-driven scientific discoveries.

8.2 FUTURE WORK

8.2.1 Graphical Representation of Annotation Views/Paths

This section discusses some ideas for possible extensions of the ViP framework, as part of future work.

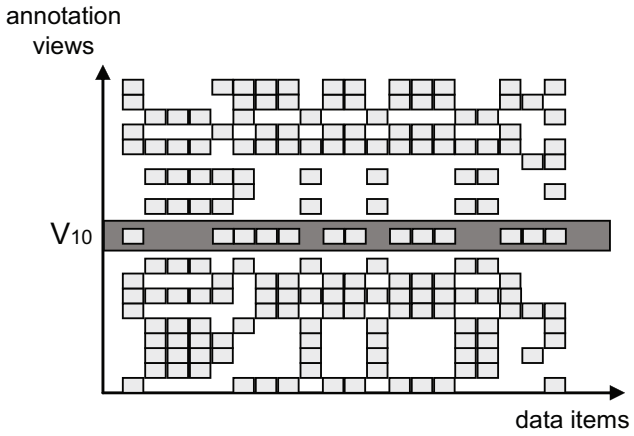


Figure 76: The Annotation View-Data Association Matrix - Initial Idea

As discussed in Section 6.6, our approach is to provide alternate *views* of the same information, allowing the users to explore the entire "space" of annotations. One such view is illustrated in Figure 76 as an initial idea. The x axis enumerates all data objects, whereas the y axis enumerates all annotation view definitions. Position (x, y) has an entry if data object x "matches" view definition y. Clearly, not all positions will be filled, although we do expect to see some patterns. A nearly-complete vertical line indicates that the particular data object participates in all annotation

view definitions. Similarly, a nearly-complete horizontal line would indicate that a particular annotation view definition would encompass all data objects (for example, such as the one for V_{10}). Seeing patterns like that will enable users to better understand the implications of the annotation views/paths definitions.

In general, we believe this graphical annotation views/paths monitor can help users to understand the annotations (space) for a particular database. We expect such a monitor to be *highly interactive* and to allow the users to see changes over time.

8.2.2 Beyond the scope of this thesis

In Chapters 6 and 7 we proposed a benchmark for evaluating an annotation model and system consisting of three annotation query types: Type I: Query Data + Browse associated Annotations (Section 5.2.1), Type II: Query Annotations + Browse associated Data (Section 5.2.2), and Type III: Query Data + Query Annotations (Section 5.2.3). We have extensively discussed and evaluated the query type I. The evaluation of the other two query types is beyond the scope of this thesis, nonetheless, we plan to continue work on the implementation and evaluation of these additional types of queries.

In Section 6.2, we discussed the problem of "when" to switch between eager and lazy annotation propagation methods, "when" to cache annotation result sets, "what" to cache, and "how" to retrieve annotations, annotation views and paths. In Section 4.4, we brought the idea of a multi-criteria, dynamic recomputing, and adaptive annotation search algorithm, considering annotation path strengths. We want to explore each of our original questions with a more in-depth analysis.

There are some issues we have not covered in this thesis, such as *concurrency control* in annotation management, and *synchronization* in semantics. In the user-driven network semantics, how to avoid repeated path propagation is an open issue, it can be converted into the classic algorithm problem, Euler path [43]. In addition, supporting continuous annotations was discussed in Section 6.4, and that would be an additional extension. We want to explore such problems as part of our future work, as well as applying this work into different domains such as social networks, information propagation or business workflow monitoring systems.

Big data and *NoSQL databases* are topics receiving a lot of attention lately, and were briefly

mentioned in this thesis. Current relational databases may not work with large-scale data as fast as NoSQL databases. We have evaluated our ViP framework on MySQL, the most popular open source relational database. In the future, we want to extend our work to more types of databases, such as *graph databases* [121, 116]. One such example is Neo4j, an open-source NoSQL graph database. Its embedded, disk-based, fully transactional Java persistence engine stores data structured in graphs rather than in tables. Due to its graph data model, Neo4j is claimed to be highly agile and "blazing" fast. For connected data operations, Neo4j runs "a thousand times faster than relational databases". We would like to see how our work merges with such new technology.

8.3 BROAD IMPACT

In this thesis, we aimed to solve two problems: (1) support annotations that are scalable both from a system point of view and also from a user point of view, and (2) support annotation queries both from an annotator point of view and a queries/user point of view. With tens of thousands of annotations appearing in scientific data, these problem are critical not only to manage the annotations efficiently but also to take care of users' preferences as the first priority.

Our main goal in this thesis was to bring a high performance solution to the questions and challenges we observed and discovered. In addition, we wanted to propose a new way to process and manage annotations and the querying with associated data, both for scientific data and general-purpose data. We believe our contributions have achieved those goals and will promote annotation management to a higher level, making it applicable more broadly and exhibiting better performance than the previous state of the art.

9.0 APPENDIX

Ruby Gems included by the bundle:

- actionmailer (3.2.13)
- actionpack (3.2.13)
- activemodel (3.2.13)
- activerecord (3.2.13)
- activeresource (3.2.13)
- activesupport (3.2.13)
- arel (3.0.2)
- builder (3.0.4)
- bundler (1.3.5)
- coffee-rails (3.2.2)
- coffee-script (2.2.0)
- coffee-script-source (1.6.2)
- erubis (2.7.0)
- execjs (1.4.0)
- hike (1.2.2)
- i18n (0.6.1)
- journey (1.0.4)
- jquery-rails (2.2.1)
- json (1.7.7)

- libv8 (3.11.8.17)
- mail (2.5.3)
- mime-types (1.22)
- multi-json (1.7.2)
- polyglot (0.3.3)
- rack (1.4.5)
- rack-cache (1.2)
- rack-ssl (1.3.3)
- rack-test (0.6.2)
- rails (3.2.13)
- railties (3.2.13)
- rake (10.0.4)
- rdoc (3.12.2)
- ref (1.0.4)
- sass (3.2.7)
- sass-rails (3.2.6)
- sprockets (2.2.2)
- therubyracer (0.11.4)
- thor (0.18.1)
- tilt (1.3.7)
- treetop (1.4.12)
- tzinfo (0.3.37)
- uglifier (2.0.1)

BIBLIOGRAPHY

[1] The Advanced Data Management Technologies laboratory at the university of pittsburgh. http://db.cs.pitt.edu/group/home.

[2] Amazon Public Data Sets. http://aws.amazon.com/datasetss.

[3] Amazon Web Services. http://aws.amazon.com/.

[4] Facebook. http://www.facebook.com.

[5] IMDB Data Sets. http://www.imdb.com/interfaces.

[6] Memcached. http://memcached.org/.

[7] MovieLens Data Sets. http://www.grouplens.org/node/73.

[8] MySpace. http://www.myspace.com.

[9] NoSQL databases. http://nosql-database.org/.

[10] Snowflake. https://github.com/twitter/snowflake.

[11] Twitter. https://twitter.com/.

[12] Twitter Social Graph. http://an.kaist.ac.kr/~haewoon/release/twitter_social_graph.

[13] YouTube. http://www.youtube.com/.

[14] Announcing snowflake. http://engineering.twitter.com/2010/06/announcing-snowflake.html, Jun 2010.

[15] *Data, data everywhere*. The Economist, February 2010.

[16] *What Technology-Assisted Electronic Discovery Teaches Us About The Role Of Humans In Technology Re-Humanizing Technology-Assisted Review*. Forbes, 2012.

[17] Lada Adamic. Zipf, power-laws, and pareto - a ranking tutorial. http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html.

[18] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. Database support for efficiently maintaining derived data. In *Proc. of the 5th International Conference on Extending Database Technology (EDBT)*, pages 223–240, 1996.

[19] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB)*, 2006.

[20] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems (TODS)*, 15(3):359–384, 1990.

[21] T. Altman and Y. Igarashi. Roughly sorting: sequential and parallel approach. *Journal of Information Processing*, 12(2):154–158, Jan 1989.

[22] Barry C. Arnold. *Pareto Distributions*. International Co-operative Publishing House, 1983.

[23] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 953–964, 2006.

[24] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communication ACM*, 18(9):509–517, 1975.

[25] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 900–911, 2004.

[26] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. In *The Apache Software Foundation*, pages 1–14, 2007.

[27] D. Boyd, S. Golder, and G. Lotan. Tweet, tweet, retweet: Conversational aspects of retweeting on twitter. In *Proc. of the 43rd Hawaii International Conference (HICSS)*, pages 1–10, 2010.

[28] Randal E. Bryant, Randy H. Katz, and Edward D. Lazowska. Big-data computing: Creating revolutionary breakthroughs in commerce, science, and society. pages 1–7, 2008.

[29] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proc. of the 2006 ACM SIGMOD international conference on Management of data*, pages 539–550, 2006.

[30] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving scientific data. *ACM Transaction Database System*, 29(1):2–42, 2004.

[31] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. pages 316–330, 2001.

[32] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On propagation of deletions and annotations through views. In *Proc. of the Symposium on Principles of database systems (PODS)*, 2002.

[33] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[34] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proc. of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, 1997.

[35] Cat Casey and Alejandra Perez. *E-Discovery Special Report: The Rising Tide of Nonlinear Review*. Hudson Global, July 2012.

[36] Haowen Chan and A. Perrig. Security and privacy in sensor networks. *Computer*, 36(10):103–105, Oct 2003.

[37] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of the Seventh Symposium on Operating System Design and Implementation (OSDI'06)*, pages 1–14, November 2006.

[38] Laura Chiticariu and Wang Chiew Tan. Debugging schema mappings with routes. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB)*, 2006.

[39] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *Proc. of the 2005 ACM SIGMOD international conference on Management of data*, pages 942–944, 2005.

[40] Gao Cong, Wenfei Fan, and Floris Geerts. Annotation propagation revisited for key preserving views. In *Proc. of the 15th ACM international conference on Information and knowledge management (CIKM)*, pages 632–641, 2006.

[41] Thomas M. Connolly and Carolyn E. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Addison-Wesley, 4 edition, 2005.

[42] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, , and Dale Woodford. Spanner: Google's globally-distributed database. In *Proc. of the Tenth Symposium on Operating System Design and Implementation (OSDI'12)*, pages 1–14, October 2012.

[43] Thomas H. Cormen. *Introduction to Algorithms*. MIT Press, 2001.

[44] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *Proc. of the international conference on Data Engineering (ICDE)*, pages 367–378, 2000.

[45] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. In *VLDB Journal*, pages 471–480, 2001.

[46] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazons highly available key-value store. In *Proc. of the Symposium on Operating Systems Principles (SOSP'07)*, pages 205–220, 2007.

[47] Jan Van den Bussche, Stijn Vansummeren, and Gottfried Vossen. Meta-SQL: Towards practical meta-querying. *Information System*, 30(4):317–332, 2005.

[48] Julien Deriviere, Thierry Hamon, and Adeline Nazarenko. A scalable and distributed nlp architecture for web document annotation. *Lecture Notes in Computer Science)*, 4139:56–67, 2006.

[49] M. Y. Eltabakh, W.-K. Hon, R. Shah, W. Aref, and J. S. Vitter. The sbc-tree: An index for run-length compressed sequences. In *Proc. of the 11th International Conference on Extending Database Technology (EDBT)*, March 2008.

[50] Mohamed Eltabakh, Walid Aref, Ahmed Elmagarmid, and Mourad Ouzzani. Handson db: Managing data dependencies involving human actions. Technical report, WORCESTER POLYTECHNIC INSTITUTE, Computer Science Department, 2012.

[51] Mohamed Y. Eltabakh, Walid G. Aref, and Ahmed K. Elmagarmid. A database server for next-generation scientific data management. In *Proc. of the Twelfth International Conference on Data Engineering (ICDE) Workshops*, pages 313–316, 2010.

[52] Mohamed Y. Eltabakh, Walid G. Aref, Ahmed K. Elmagarmid, Yasin N. Silva, and Mourad Ouzzani. Supporting real-world activities in database management systems. In *Proc. of the Twelfth International Conference on Data Engineering (ICDE)*, pages 808–811, 2010.

[53] Mohamed Y. Eltabakh, Mourad Ouzzani, and Walid G. Aref. bdbms – a database management system for biological data. In *Proc. of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2007.

[54] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):1 – 5, August 2004.

[55] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: Queries and provenance. In *Proc. of the Symposium on Principles of database systems (PODS)*, 2008.

[56] Maksym Gabielkov and Arnaud Legout. The complete picture of the twitter social graph. In *Proc. of the CoNEXT Student'12*, pages 19–20, December 2012.

[57] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. Mondrian: Annotating and querying databases through colors and blocks. In *Proc. of the 22nd International Conference on Data Engineering (ICDE)*, page 82, 2006.

[58] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *Proc. of the 33rd international conference on Very large data bases (VLDB)*, pages 675–686, 2007.

[59] Shenoda Guirguis, Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Adaptive scheduling of web transactions. In *Proc. of the 25th IEEE International Conference on Data Engineering (ICDE'09)*, pages 357–368, April 2009.

[60] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18:3–18, 1995.

[61] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10:2001, 2001.

[62] Eric N. Hanson. A performance analysis of view materialization strategies. In *Proc. of the 1987 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 440–453, 1987.

[63] Michiel Hazewinkel. Pareto distribution. *Encyclopedia of Mathematics*, 2001.

[64] J. Herlocker, J. Konstan, A. Borchers, and J. Riedl. An algorithmic framework for performing collaborative filtering. In *Proc. of the 1999 Conference on Research and Development in Information Retrieval*, August 1999.

[65] Yoshihide Igarashi and Derick Wood. Roughly sorting: a generalization of sorting. *Journal of Information Processing*, 14(1):36–42, Jan 1991.

[66] Marian K. Iskander, Dave W. Wilkinson, Adam J. Lee, and Panos K. Chrysanthis. Enforcing policy and data consistency of cloud transactions. In *Proc. of the 2nd ICDCS International Workshop on Security and Privacy on the Cloud (ICDCS-SPCC'11)*, pages 253–262, June 2011.

[67] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems (TODS)*, 26(2):214–260, 2001.

[68] Matthew C. Jones and Elke A. Rundensteiner. View materialization techniques for complex hierarchical objects. In *Proc. of the sixth international conference on Information and knowledge management (CIKM)*, pages 222–229, 1997.

[69] Paul E. Carlson Jr., Joseph Horzempa, Dawn M. O'Dee, Cory M. Robinson, Panayiotis Neophytou, Alexandros Labrinidis, and Gerard J. Nau. Global transcriptional response to spermine, a component of the intra-macrophage environment, reveals regulation of francisella gene expression through insertion sequence elements. pages 1–10, September 2009.

[70] Jason J. Jung and Jérôme Euzenat. Towards semantic social networks. In *Proc. of the 4th European conference on The Semantic Web*, pages 267–280. Springer-Verlag, 2007.

[71] Jon M. Kleinberg. Challenges in mining social network data: processes, privacy, and paradoxes. In *Proc. of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 4–5, 2007.

[72] Georgia Koutrika and Yannis Ioannidis. Personalization of queries in database systems. In *Proc. of the ICDE*, page 597, 2004.

[73] Harumi A. Kuno and Elke A. Rundensteiner. Using object-oriented principles to optimize update propagation to materialized views. In *Proc. of the Twelfth International Conference on Data Engineering (ICDE)*, pages 310–317. IEEE Computer Society, 1996.

[74] Alexandros Labrinidis, Qiong Luo, Jie Xu, and Wenwei Xue. Caching and materialization in web databases. pages 3(2):169–266, December 2009.

[75] Alexandros Labrinidis, Huiming Qu, and Jie Xu. Quality contracts for real-time enterprises. pages 143–156, September 2007.

[76] Alexandros Labrinidis and Nick Roussopoulos. On the materialization of web views. In *Proc. of the 2nd International Workshop on the Web and Databases (WebDB)*, pages 79–84, June 1999.

[77] Alexandros Labrinidis and Nick Roussopoulos. Webview materialization. In *Proc. of the 19th ACM International Conference on Management of Data (SIGMOD)*, pages 367–378, May 2000.

[78] Alexandros Labrinidis and Nick Roussopoulos. Adaptive webview materialization. In *Proc. of the 4th International Workshop on the Web and Databases (WebDB)*, pages 85–90, May 2001.

[79] Susan Weissman Lauzac and Panos K. Chrysanthis. Personalizing information gathering for mobile database clients. In *Proc. of the 17th Annual ACM Symposium on Applied Computing (SAC)*, pages 49–56, March 2002.

[80] Qinglan Li, Jonathan Beaver, Ahmed Amer, Panos K. Chrysanthis, Alexandros Labrinidis, and Ganesh Santhanakrishnan. Multi-criteria routing in wireless sensor-based pervasive environments. *Journal of Pervasive Computing and Communications (JPCC'05)*, 1(4):313–326, December 2005.

[81] Qinglan Li, Alexandros Labrinidis, and Panos K. Chrysanthis. User-centric annotation management for biological data. In *Proc. of the Second International Provenance and Annotation Workshop (IPAW)*, pages 54–61, June 2008.

[82] Qinglan Li, Alexandros Labrinidis, and Panos K. Chrysanthis. User-centric annotation management for biological data (demo). In *Proc. of the Second International Provenance and Annotation Workshop (IPAW)*, pages 1–4, June 2008.

[83] Qinglan Li, Alexandros Labrinidis, and Panos K. Chrysanthis. ViP: a user-centric view-based annotation framework for scientific data. In *Proc. of the 20th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 295–312, July 2008.

[84] Qinglan Li, Alexandros Labrinidis, and Panos K. Chrysanthis. ViP: a user-centric view-based annotation framework for scientific data. In *the 7th Hellenic Data Management Symposium (HDMS)*, pages 1–12, July 2008.

[85] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *Proc. of the 2007 ACM SIGMOD international conference on Management of data*, pages 461–472, 2007.

[86] Lory Al Moakar, Panos K. Chrysanthis, Christine Chung, Shenoda Guirguis, Alexandros Labrinidis, Panayiotis Neophytou, and Kirk Pruhs. Admission control mechanisms for continuous queries in the cloud. In *Proc. of the 26th IEEE International Conference on Data Engineering (ICDE)*, pages 1–4, March 2010.

[87] Lory Al Moakar, Alexandros Labrinidis, and Panos K. Chrysanthis. Adaptive class-based scheduling of continuous queries. In *Proc. of the Seventh International Workshop on Self-Managing Database Systems (SMDB'12)*, pages 1–6, April 2012.

[88] Lory Al Moakar, Thao N. Pham, Panayiotis Neophytou, Panos K. Chrysanthis, Alexandros Labrinidis, and Mohamed A. Sharaf. Class-based continuous query scheduling for data streams. In *Proc. of the 6th International Workshop on Data Management for Sensor Networks (DMSN'09)*, pages 1–6, August 2009.

[89] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. McGrath, Jim Myers, and Patrick Paulson. The open provenance model. In *http://wiki.esi.ac.uk/w/files/e/e8/Opm.pdf*.

[90] Panayiotis Neophytou, Panos K. Chrysanthis, and Alexandros Labrinidis. Towards continuous workflow enactment systems. pages 162–178, 2009.

[91] Panayiotis Neophytou, Panos K. Chrysanthis, and Alexandros Labrinidis. Confluence: Continuous workflow execution engine. In *Proc. of the 30th ACM International Conference on Management of Data (SIGMOD'11)*, pages 1311–1314, June 2011.

[92] Panayiotis Neophytou, Panos K. Chrysanthis, and Alexandros Labrinidis. Confluence: Implementation and application design. In *Proc. of the 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'11)*, pages 181–190, October 2011.

[93] Panayiotis Neophytou, Roxana Gheorghiu, Rebecca Hachey, Timothy Luciani, Di Bao, Alexandros Labrinidis, G. Elisabeta Marai, and Panos K. Chrysanthis. Astroshelf: Understanding the universe through scalable navigation of a galaxy of annotations. In *Proc. of the 31st ACM International Conference on Management of Data (SIGMOD'12)*, pages 1–4, May 2012.

[94] B.C. Neuman. Security, payment, and privacy for network commerce. *IEEE Journal on Selected Areas in Communications*, 13(8):1523–1531, Oct 1995.

[95] Jakob Nielsen. *Designing Web Usability*. New Riders Publishing.

[96] Jakob Nielsen and Hoa Loranger. *Prioritizing Web Usability*. New Riders Press.

[97] Patrick O'Neil and Elizabeth O'Neil. *Database: Principles, Programming, and Performance*. Morgan Kaufmann, second edition, 2001.

[98] Vilfredo Pareto. Cours dconomie politique: Nouvelle dition par g.-h. bousquet et g. busino, 1964.

[99] Huiming Qu and Alexandros Labrinidis. Scheduling update and query transactions under quality contracts in web-databases. In *Proc. of the 5th Hellenic Data Management Symposium (HDMS'06)*, pages 1–12, September 2006.

[100] Huiming Qu and Alexandros Labrinidis. Preference-aware query and update scheduling in web-databases. In *Proc. of the 23rd IEEE International Conference on Data Engineering (ICDE'07)*, pages 1–10, April 2007.

[101] Huiming Qu, Alexandros Labrinidis, and Daniel Mosse. Unit: User-centric transaction management in web-database systems. In *Proc. of the 22nd International Conference on Data Engineering (ICDE)*, pages 1–10, April 2006.

[102] Huiming Qu, Jie Xu, and Alexandros Labrinidis. Quality is in the eye of the beholder: Towards user-centric web-databases (demo). In *Proc. of the 26th ACM International Conference on Management of Data (SIGMOD'07)*, pages 1106–1108, June 2007.

[103] Huming Qu, Jie Xu, and Alexandros Labrinidis. Guiding personal choices in a quality contracts driven query economy. In *Proc of Third International Workshop on Personalized Access, Profile Management, and Context Awareness in Databases (PersDB'09)*, pages 1–6, August 2009.

[104] Geerajit Rattanaritnont, Masashi Toyoda, and Masaru Kitsuregawa. Analyzing patterns of information cascades based on users' influence and posting behaviors. In *Proceedings of the 2nd Temporal Web Analytics Workshop (TempWeb'12)*, pages 1–8, New York, NY, USA, 2012. ACM.

[105] Eldar Sadikov, Montserrat Medina, Jure Leskovec, and Hector Garcia-Molina. Correcting for missing data in information cascades. In *Proc. of the 4th International Conference on Web Search and Data Mining (WSDM'11)*, page 912, February 2011.

[106] Ganesh Santhanakrishnan, Qinglan Li, Jonathan Beaver, Panos K. Chrysanthis, Ahmed Amer, and Alexandros Labrinidis. Multi-criteria routing in pervasive environment with sensors. In *Proc. of the IEEE International Conference on Pervasive Services (ICPS'05)*, pages 7–16, July 2005.

[107] A. D Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *Proc. of the international conference on Data Engineering (ICDE)*, 2006.

[108] Marc Seeger. Key-value stores: a practical overview. pages 1–21, 2009.

[109] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[110] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Cristiana Amza. Optimizing I/O-Intensive transactions in highly interactive applications. In *Proc. of the 28th ACM International Conference on Management of Data (SIGMOD'09)*, pages 785–798, June 2009.

[111] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proc. of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–10, 2010.

[112] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, pages 31–36, 2005.

[113] Alan Jay Smith. Design of cpu cache memories. 1987.

[114] Swapna Somasundaran, Josef Ruppenhofer, and Janyce Wiebe. Discourse level opinion relations: An annotation study. In *SIGdial Workshop on Discourse and Dialogue*, June 2008.

[115] Divesh Srivastava and Yannis Velegrakis. Intensional associations between data and metadata. In *Proc. of the 2007 ACM SIGMOD international conference on Management of data*, pages 401–412, 2007.

[116] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proc. of VLDB Endowment*, 5(9):788–799, May 2012.

[117] Jesse Szwedko, Callen Shaw, Alexander G. Connor, Alexandros Labrinidis, and Panos K. Chrysanthis. Demonstrating an evacuation algorithm with mobile devices using an e-scavenger hunt game. In *Proc. of Eighth ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE'09)*, pages 49–52, June 2009.

[118] Wang-Chiew Tan. Containment of relational queries with annotation propagation. In *Proc. of the 19th international conference on Data Base Programmming Languages (DBPL)*, 2003.

[119] Wang-Chiew Tan. Research problems in data provenance. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2004.

[120] Rob Tweed and George James. A universal nosql engine, using a tried and tested technology. pages 1–25, 2010.

[121] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.

[122] Fusheng Wang and Carlo Zaniolo. Xbit: An xml-based bitemporal data model. In *Proc. of the 23rd International Conference on Conceptual Modeling*, 2004.

[123] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012.

[124] Janyce Wiebe, Theresa Wilson, and Claire Cardie. Annotating expressions of opinions and emotions in language. In *Language Resources and Evaluation (formerly Computers and the Humanities)*, 2005.

[125] Hejun Wu, Qiong Luo, Jianjun Li, and Alexandros Labrinidis. Quality aware query scheduling in wireless sensor networks. In *Proc. of 6th International Workshop on Data Management for Sensor Networks (DMSN'09)*, pages 1–13, August 2009.

[126] Jie Xu, Qinglan Li, Huiming Qu, and Alexandros Labrinidis. Towards a content-provider-friendly web page crawler. In *Proc. of the Tenth International ACM Workshop on the Web and Databases (WebDB)*, pages 1–10, June 2007.

[127] Jun Zhao, Carole Goble, Mark Greenwood, Chris Wroe, and Robert Stevens. Annotating, linking and browsing provenance logs for e-science. In *Proc. of the 2nd International Semantic Web Conference Workshop on Retrieval of Scientific Data (ISWC)*, October 2003.

[128] Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. Lazy maintenance of materialized views. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 231–242, 2007.

[129] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proc. of the 1995 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 316–327, 1995.

[130] Cai-Nicolas Ziegler and Georg Lausen. Propagation models for trust and distrust in social networks. *Information Systems Frontiers*, 7(4-5):337–358, 2005.

[131] George K. Zipf. *The Psychobiology of Language*. Houghton-Mifflin, 1935.

[132] George K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.