ViP: A User-Centric View-Based Annotation Framework for Scientific Data

Qinglan Li, Alexandros Labrinidis, and Panos K. Chrysanthis

Advanced Data Management Technologies Laboratory
Department of Computer Science, University of Pittsburgh
Pittsburgh, PA 15260, USA
{qinglan,labrinid,panos}@cs.pitt.edu

Abstract. Annotations play an increasingly crucial role in scientific exploration and discovery, as the amount of data and the level of collaboration among scientists increase. In this paper, we introduce ViP, a user-centric, view-based annotation framework that promotes annotations as first-class citizens. ViP introduces novel ways of propagating annotations, empowering users to express their preferences over the time and network semantics of annotations. To efficiently support such novel functionality, ViP utilizes database views and introduces new caching techniques. Through an extensive experimental study on a real system, we show that ViP can seamlessly introduce new annotation propagation semantics while significantly improving the performance over the current state of the art.

1 Introduction

Without a doubt, data management is playing a pivotal role in scientific exploration nowadays, constantly fueling the pace of discovery. In addition to efficiently managing the tsunami of experimental data generated, data management also facilitates effective collaboration among scientists, by recording *data provenance* [6] and *data lineage* [1,2,9,11,12], and by supporting *annotations* [3,4,5,15,23]. Data provenance and lineage essentially keep 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).

In this paper, we present *ViP*, a novel annotation framework that introduces new annotation propagation methods, utilizes *views* both as a specification mechanism and as a user-interface mechanism, and employs caching techniques for improved performance compared to the state of the art [1].

Our interest in this research area came from our participation in the Center for Modeling Pulmonary Immunity (CMPI). CMPI is bringing together experimentalists and modelers to study pulmonary immunity in response to three bio-defense pathogens. Our group is responsible for the design and development of the *data sharing platform* (DataXS), where experimental data, analysis, and models will be shared among project participants. In such a diverse setting, the ability to record annotations and propagate

[1] This research was supported in part by NIH-NIAID grant NO1-AI50018.

B. Ludäscher and Nikos Mamoulis (Eds.): SSDBM 2008, LNCS 5069, pp. 295–312, 2008.

them to all related data items and interested parties is crucial to the success of the project.

As part of the design process and during the implementation of our first prototype, we were able to identify two distinct *usage patterns* related to the specification and the propagation of annotations within a Database Management System (DBMS), which were not currently supported by the state of the art.

Support for user-centric time semantics for annotations. The first usage pattern that we observed 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 was entered at completely different times, since more than one labs were 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).

To address this, we propose the concept of *valid time* of annotations, where annotations should be propagated to all data items matching a certain description within the validity time interval specified by users. We refer to this feature as "*user-centric time semantics*".

Support for propagation of annotations in user-defined ways. 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 or their lineage*. Such paths materialize because, for example, tissues from multiple, independent experiments were processed together, in a single assay (for example, on a single plate that needed to be filled up to minimize costs). Annotations associated with this one assay would be propagated to all experiments that shared the plate, e.g., in the case of a contamination.

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-centric network semantics*".

User-friendly Implementation. First of all, we need to identify a way to formally define data items matching a certain criterion, to be used by our algorithms. The answer is somewhat obvious: use *database views*, which describe the results of a query. Similar approaches have been proposed in the past (e.g., [21]); ours significantly extends the use of views with additional semantics. Secondly, we need a realistic way for users to utilize the new features. Clearly, they are not to be expected to provide view definitions in SQL! In DataXS, a user can easily specify filtering conditions to locate certain data items, in other words, to specify views using a point and click interface (Figure 1); these views were initially implemented to provide an easy reference for frequently used queries (e.g., the In Vitro experiments tab in Figure 1), but can also be trivially used to implement all functionality of the ViP framework.

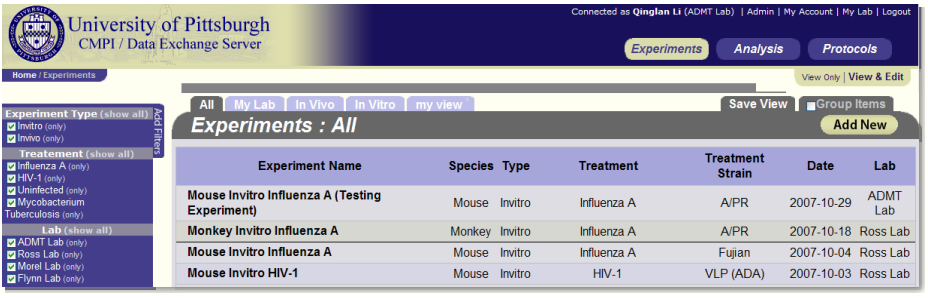


Fig. 1. DataXS User Interface

Contributions: This paper has both theoretical and practical contributions:

- based on our experience from a real system implementation, we introduce new annotation propagation methods, suitable for scientific data,
- we propose user-centric features (u*ser-centric time semantics, network semantics, and access control*) that enable users to personalize annotation propagation,
- we propose to use *views* as the formal mechanism to implement the new annotation propagation features and also as a user-interface paradigm,
- we utilize *caching* to significantly improve the performance over the state of the art,
- we experimentally evaluate the proposed ViP framework using a real system implementation and simulated workloads.

Roadmap: Section 2 presents the details of the proposed annotation framework, along with related work. Section 3 describes our implementation. Section 4 presents the results from our extensive experimental study using a real system.

2 Annotation Propagation Semantics in ViP

In this section, we present the details of our proposed semantics for annotation propagation. In each case, we also present the corresponding statements in ViP-SQL, our proposal for a simple extension to SQL that would handle the new semantics based on the concept of views.

2.1 User-Centric Time Semantics

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

If 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,
- *future*, where an annotation is only propagated to data items that are added in the database in the future,

- *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. This is essentially the combination of the *now* and *future* approaches,
- *future interval*, where an annotation is propagated to data items that are added in the database in the time interval a user specifies.

These four cases represent all the possible alternatives if one considers the concept of future time semantics and also wants to give users the flexibility to specify the validity interval for their annotations.

Related Work. Most of the current annotation management frameworks utilize *now* time semantics, propagating annotations to only existing data items [2,3,4,13,14]. In contrast, our system supports now and future semantics (in all the four "flavors" described above), which we will assume for the rest of this paper. To the best of our knowledge, only the work in [21] functions similar to the *now+future* time semantics presented in this paper.

Motivating Example #1. To properly motivate the need for time semantics, let us assume a setup like that in the DataXS system, where experimental data are stored (in the Experiments table) and shared among project participants. Let us assume that a contamination happened in the ADMT Lab between Oct 1, 2007 and Oct 20, 2007, and we would like to annotate all experimental data accordingly, with *now+future* time semantics. Clearly, if we only attach an annotation to all the files matching the ADMT Lab and happened Oct 1 - 20, 2007 (at the time of annotating), we will miss all the files that are potentially added into the DataXS system at a later time, and still meet these conditions. As we discussed earlier, this is a typical usage pattern, making *now+future* time semantics a necessity. 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 >= "10/01/07"
             and Date <= "10/20/07")
   VALUE "ADMT Lab was contaminated between Oct. 1st
             & Oct. 20, 2007. Please use data with caution."
   VALIDTIME [now,  )
```

General Case. 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 attaching it to 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 *now+future* time semantics.

Given that annotations are associated with views instead of individual data items, the expected behavior in cases of modifications is straightforward (Figure 2). Assuming a view V_i with annotation a, the following actions can be defined:

- 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 (attached to V_i) when it is queried.

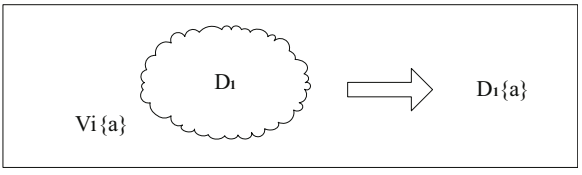


Fig. 2. View-based Annotation Propagation: User-centric Time Semantics. (Annotation a is associated with view V_i. Data item $D_1 \in V_i$ receives annotation a.)

- 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 when it is queried.
- 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.

2.2 User-Centric Network Semantics

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

Through the ViP framework, we propose to empower users to specify *explicit annotation 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, and 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.

Motivating Example #2. Continuing from Motivating Example #1, we have that the ADMT Lab and the Ross Lab are 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 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, )
```

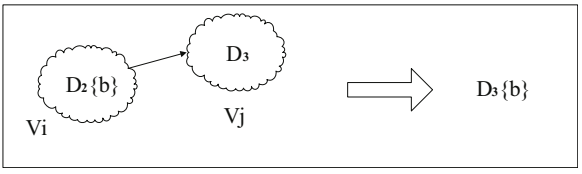


Fig. 3. View-based Annotation Propagation: User-centric Network Semantics (Disjoint source/destination). There exists an annotation propagation path from V_i to V_j. Data item $D_2 \in V_i$ has annotation b. Data item $D_3 \in V_j$ receives annotation b.

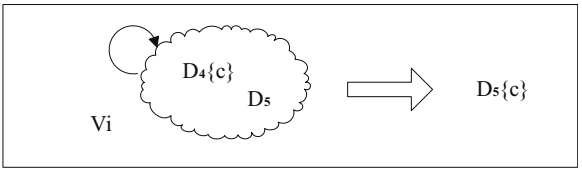


Fig. 4. View-based Annotation Propagation: User-centric Network Semantics (Identical source/destination). There exists an annotation propagation path from V_i to self. Data item $D_4 \in V_i$ has annotation c. Data item $D_5 \in V_i$ receives annotation c.

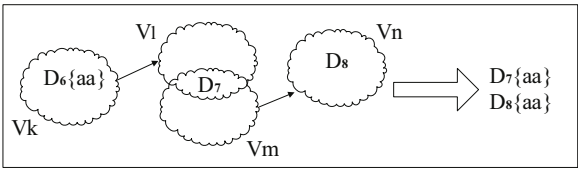


Fig. 5. View-based Annotation Propagation: User-centric Network Semantics (Overlapping source/dest.). There exists an annotation propagation path from V_k to V_l and another path from V_m to V_n. V_l and V_m overlap. Data item $D_6 \in V_k$ has annotation aa. Data item D_7 is a member of both V_l and V_m. Data item D_7 receives annotation aa. Data item D_8 receives annotation aa.

General Case. 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 *network semantics*. With regards to view membership, the behavior is very similar to that in the case of *time semantics*, as presented in the previous section. With regards to the relation between the source and destination views, we consider the following cases:

- source and destination views are disjoint (Figure 3)
- source and destination views are identical (Figure 4)
- source and destination views are overlapping (Figure 5)

Related Work. In the context of metadata management, [21] considered implicit paths from queries to queries but they have not considered the explicitly-defined network

paths as we do in this paper. In the context of schema mapping, there are multiple works that consider paths of "similar" tables [8,19].

2.3 User-Centric Access Control

We advocate that scientific annotation must have a strong access control component. First of all, much of the data is not public, so appropriate access controls 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 with each other (i.e., by specifying explicit paths for annotation propagation) corresponds to private information that should not be made public. Given all these reasons, the ViP framework includes multiple user-centric access control features.

On the annotation level. First of all, we 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).

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

On the path level. One important innovation of the ViP framework is the explicit path functionality (network semantics). We support three different access control features, as they apply to user-centric network semantics:

- **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 a 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.
- **Maximum HAP:** Given explicit annotation paths and the ensuing network semantics, an annotation can theoretically be propagated over an unreasonable number of paths, if left unconstrained. Towards this, the ViP framework includes a system variable, **MAX-HAP**, short for *maximum number of hops allowed to propagate*, which puts a system-wide upper bound over how many hops any annotation is allowed to propagate. The number of hops starts counting after we follow the first "direct" path (i.e., in Figure 5 the number of hops is 2). This was inspired by the TTL value of queries in unstructured peer-to-peer networks.
- **HAP on insert:** The ViP framework enables users to specify a variable, **HAP-i**, or *Hops Allowed to Propagate* at insertion, to indicate how far the newly-inserted annotation can be propagated. HAP-i = 0 means the annotator just wants to limit this annotation to data items specified in the view. HAP-i = 1 means the annotator allows this annotation to be propagated only to neighboring nodes. HAP-i = MAX-HAP means the annotator is not placing any restriction on the propagation of his/her annotation.

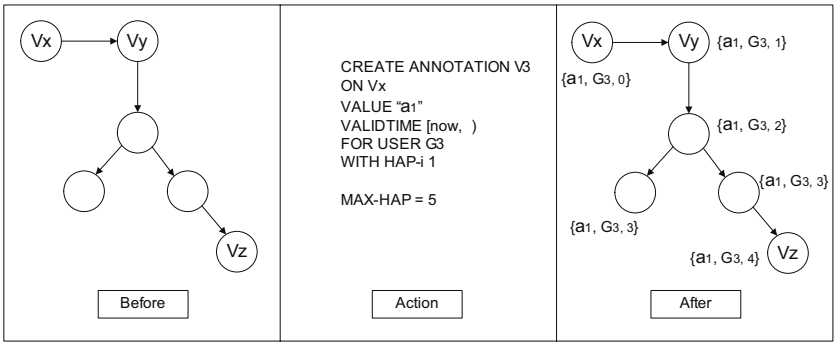


Fig. 6. User-Centric Annotation Propagation Example

- **HAP 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., because of information "decay"). In cases of a network of paths (e.g., as in Figure 5), it may not be prudent to exhaustively follow all paths in the network to propagate annotations. Similarly with the *HAP on insert*, the ViP framework gives the option to specify a *maximum number of hops an annotation is allowed to propagate at query time*, or **HAP-q**. Given these three parameters (some of which are optional), the maximum number of hops followed is MIN(MAX-HAP, HAP-i, HAP-q). By setting HAP-i or MAX-HAP to 0, we effectively disable explicit annotation direct paths; by setting MAX-HAP to 1, we effectively disable cascading annotation propagation.

Motivating Example #3. We illustrate the user-centric semantics of the ViP framework using the example in Figure 6. Figure 6/Before has a network of paths; Figure 6/Action indicates that an annotation is added on node V_x; Figure 6/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 the annotation a_1 is propagated to V_y within HAP 1 as (a_1, G_3, 1), and to V_z within HAP 4 (which is bigger than HAP-i). Clearly, users that neither belong to group G_3 nor specify a HAP-q high enough will not "see" annotation a_1. Besides, if HAP-i of a_1 is set to 0, even if users specify a high HAP-q will still not "see" annotation a_1. The queries and the results are shown in Table 1.

Related work. There is significant related work in personalization, especially in connection with information retrieval [16,18]. There is also additional work in user-centric data management, allowing users to express their preferences on the execution of their

Table 1. Queries and Results for Figure 6

HAP-i	Query	Result	User	HAP-q	Annotation
1	1	V_y	$U_1 \neg \subseteq G_3$	3	No a_1
1	2	V_z	$U_3 \subseteq G_3$	3	No a_1
0	3	V_z	$U_3 \subseteq G_3$	5	No a_1
MAX-HAP	4	V_z	$U_3 \subseteq G_3$	5	a_1

Table 2. Standard Annotation Management Features Comparison

Standard Features	DBNotes[3]	Mondrian[14]	ULDB[2]	bdbms[13]	MMS[21]	ViP
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	Hybrid
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

queries, such as [17,20]. However, to the best of our knowledge, this is the first work to address in a unifying framework all the user-centric features that we proposed as part of ViP.

2.4 Discussion

There are many systems that support in isolation, some of the features that are part of ViP without any one single system incorporating all of them. Additionally, many of the semantics introduced by ViP are not found in other systems.

Most current systems, for example, do not support annotations that are also valid in the future (Table 2). Only MMS [21] supports future time semantics in an implicitly-defined way (i.e., without giving the user options to select as ViP is doing through the valid time concept).

One of ViP's novel ideas is the explicit paths for annotation propagation, which also have privacy controls. Although existing systems support implicit annotation propagation paths, none except for ViP supports explicit, user-defined annotation propagation paths. ViP supports large scale annotation management, thus employs a hybrid propagation scheme while [3,13] use eager propagation, whereas [2,14,21] use an on-demand scheme.

To the best of our knowledge, ViP brings user-centric features in many aspects that are not considered in most related work as shown in Table 3. ViP enables users to specify the propagation method. In DBNotes [3], users can specify *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*.

Some systems consider *access control* on the data level, or even on the update authorization part [13]. Instead, we propose to fully support this feature in a broader domain, on annotations, annotation views, and annotation paths.

Table 3. User Centric Annotation Management Features Comparison

User-centric Features	DBNotes[3]	Mondrian[14]	ULDB[2]	bdbms[13]	MMS[21]	ViP
Time Semantics:						
· Valid Time	No	No	No	No	No	Yes
Network Semantics:						
· Propagation Method	Yes	No	No	Limited	No	Yes
Access Control:						
· Annotations	No	No	No	Limited	No	Yes
· Annotation Views	No	No	No	No	No	Yes
· Annotation Paths	No	No	No	No	No	Yes

3 Implementation

3.1 The ViP Framework

The ViP framework is illustrated in Figure 7. ViP-SQL queries are rewritten automatically into SQL queries evaluated by the annotation query processor, then registered with the annotation register and the path setup manager. They are sent to DBMS and the resulting annotation set is merged with the regular query result by the postprocessor for matching and presentation. Our DataXS application "fits" on top of this framework, providing a point-and-click user interface.

3.2 Annotations Registration

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

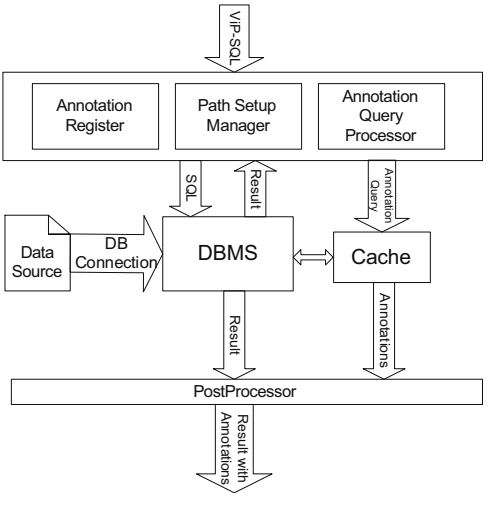


Fig. 7. ViP System Architecture

is responsible for insertion, deletion and updating. If it is an annotation path, the path setup manager will update the auxiliary table to record path source and target, with appropriate path query conditions. Obviously, sorting views or addressing the view containment problem [7,10,22] 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.

3.3 Implementing Auxiliary Tables

It is quite naturally to use auxiliary tables storing the attributes of the annotation views. Like MMS [21], ViP also uses auxiliary tables to store annotation view conditions, which will work as filters to drop 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); as such, for every data change, 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 DataXS project, data updates happen more often than annotation views/paths updates, in which case an index approach would require a lot of Q-value updating. Thus, ViP relies on caching instead of indexing.

3.4 Querying Result with Annotations

We use ViP-SQL to allow users to retrieve regular results with annotations. A query with annotation is rewritten as standard SQL with preprocessing and postprocessing. Preprocessing checks the auxiliary table for possible early annotation filtering. If a query result is satisfied in an annotation view, then the annotation query processor will lookup the annotations associated with the query result. The cache is used to optimize system performance. We present the pseudocode for the corresponding algorithms accordingly.

Caching to Optimize Annotation Search. If a data tuple is not found in the cache, ViP will execute the annotation query and save its annotation result set into the cache. If a data tuple is found in the cache, we need to verify if it is still "fresh." Cache management will take no action if a *data item* is inserted, deleted, or updated in the database. Whenever an *annotation registration* is updated/inserted, our system will reset the cache appropriately. If an annotation registration is removed, our system will remove its related entries from the cache as well. The algorithm is shown in Figure 8.

Search Associated Annotations. To Search the annotations associated with of a data item, we need to search in both directions: its direct annotations (via annotation views) and its inherited annotations (via annotation paths) as shown in Figure 9.

```
hit_caching(Ti)
   Tj = search_in_cache_index(Ti.table, Ti.col, Ti.id)
   if Tj is found,
      compare(Ti.data, Tj.datasnapshot)
      if matches
         hit-counter++
         return Tj.CachedAnnotationQueryResult
   return false

insert_into_caching(Ti)
   if cache is full
      evict as LFU algo
   insert Ti to cache
   save a snapshot of data referred by Ti

after_annotation_delete(Ti)
   delete cached AnnotationQueryResult R
   where R.table = Ti.table and R.id = Ti.id
```

Fig. 8. Algorithm of Annotation Cache Management

```
search_associated_annotation(Ti)
   find_direct_associated_annotation(Ti)
   find_dependent_associated_annotation(Ti)
   return Ti.annotationQueryResult

find_direct_associated_annotation(Ti)
   A = search_in_Annotation_Attribute_table(Ti.table, Ti.col)
   for each annotation Aj in A
      compare_condition_parameter (Aj, Ti)
      if match, add Aj.id to Ti.annotationQueryResult

find_dependent_associated_annotation(Ti)
   H = search_in_Inhertance_Definition_table(Ti.table, Ti.col)
   for each Hj in H
      R = find_records_in_associated_table(Hj.inheritance_rule)
      R_column = Hj.inhertance_through_rule.attribute
      for each record Rm in R
         search_associated_annotation(Rm.R_column)
```

Fig. 9. Algorithm of Searching Associated Annotations

4 Experimental Results

We have implemented the ViP system as a Ruby on Rails application that interfaces to MySQL. We used simulated users, annotations, and query workloads to be able to scale our experiments to desired levels.

To the extent possible, we compare our system with MMS [21], the latest and the most related work. In [21], MMS was compared to other systems, specifically DBNotes [3] and MONDRIAN [14]. MMS showed significant benefits over those systems both in query times and storage space usage. That is because in DBNotes every relational table column is associated to one additional annotation column, and if a value in a tuple has more than one annotation, the tuple is recorded multiple times, one for each annotation. On the other hand, MONDRIAN 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. In [21], the experimental results showed that MMS reduced the redundant space used in DBNotes and MONDRIAN; also, it decreased query

Table 4. Experiment Parameters

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]

time even with the cost of updating the Q-index and querying additional metadata table. Our system works similar to MMS in the way that there are annotation tables instead of additional annotation columns. Thus, it is expected our system will perform similar to MMS when compared to DBNotes and MONDRIAN if the association between the data and the annotation is explicit and static. In this paper, 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 time. For those features that ViP supports and MMS does not (such as the user-centric access control), we performed a sensitivity study of our framework.

Data. We gathered data from what has already been stored in our DataXS prototype. To test the scalability, we enlarged the dataset using uniform and Zipf distributions. The experiment parameters are shown in Table 4.

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

Query Traces. We generated queries with Zipf distribution on both (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 time is measured in milliseconds unless otherwise indicated.

4.1 View-Based Annotation Propagation

We compared the query time of our system, ViP, with MMS. Both systems retrieved the same annotations associated with the same queries. In our first experiment, we varied the total number of annotation registrations (Figure 10). ViP always outperformed MMS due to its caching optimization. With more annotation views registered, ViP gained more benefit. In the case of 50,000 annotation views registered, ViP took about 25% less time, indicating that ViP works better for large numbers of annotation views.

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 time (ms) is $(1468.06 \mp 7.36) = (1460.7,$

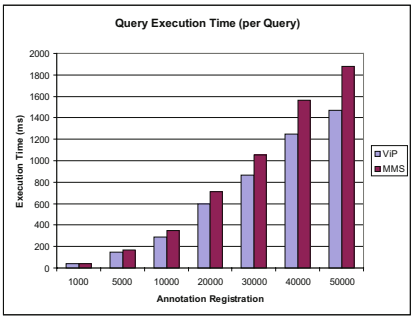


Fig. 10. Query Time

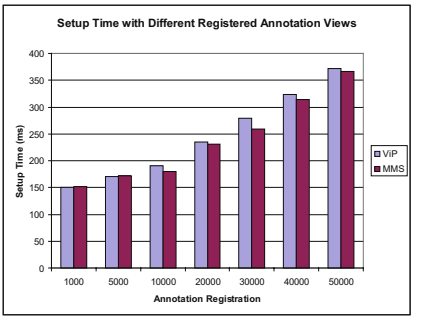


Fig. 11. Setup Time

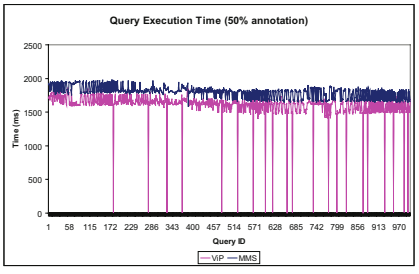
1475.42); the 95% confidence interval for MMS mean query time (ms) is $(1878.91 \mp 4.05) = (1874.86, 1882.96)$. The results presented in the paper were acquired as the average value from 1000 repeated experiments with random parameter settings. Due to the limited space, not every confidence interval is listed in the paper; all results were similar to this experiment.

In all experiments, we started with 80% annotation views and paths insertions. When the query traces were executed, the remaining 20% of the annotation registrations were performed, with their arrival times uniformly distributed over the duration of the experiment. We assume each query or annotation registration operation is atomic. The query time includes (1) data query 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 time per query for both systems is shown in Figure 11. Although ViP took extra time to manage the cache, the overhead is negligible compared to the gain from the query time.

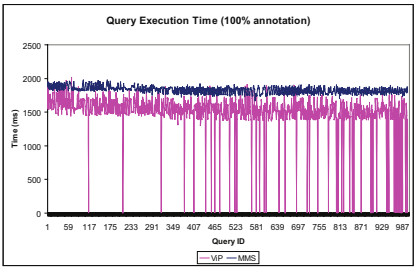
In the next set of experiments, we investigated the effect of various annotation densities, which is the percentage of data associated with annotation views. In Figure 12, 1000 queries were plotted in each subfigure to display the various query times. The density was changed from 50% to 200%, and the query time increased accordingly. In these figures, a vertical line corresponds to a cache hit (near 0 response time) on all annotations the query expects to return. We found in the extremely dense case, which is 200% in Figure 12(d), that ViP had so many cache hits, that the overall query time was reduced significantly. The detailed summary of average query time is presented in Table 5. Again, ViP works better in large scale of annotation views because of its optimized scheme. For fairness, we used only a 10% annotation density, which is the least beneficial setting for ViP, in all other experiments.

Table 5. Query Time with Different Annotation Densities

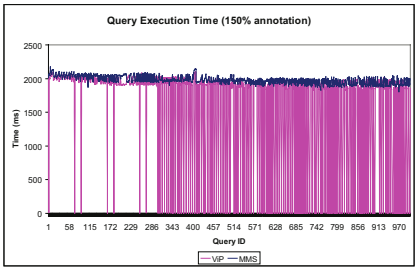
Anno. Density	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**	**150%**	**200%**
MMS Time (ms)	1804.81	1808.66	1812.50	1867.94	1878.02	1895.51	1928.80	1979.92	2178.67
ViP Time (ms)	1471.38	1419.73	1445.81	1499.44	1394.39	1386.27	1484.16	1483.84	1250.99



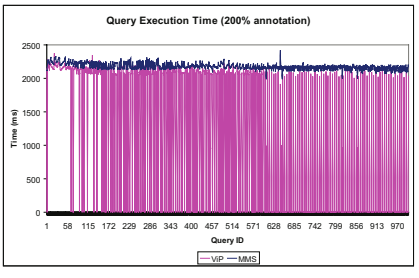
(a) Query Time of 50% Annotation Density



(b) Query Time of 100% Annotation Density



(c) Query Time of 150% Annotation Density



(d) Query Time of 200% Annotation Density

Fig. 12. Query Time with Different Annotation Densities

4.2 Annotation Propagation with Caching

In our optimization scheme, caching plays a major factor to improve system performance. However, the cache management time was insignificant compared to the query time, shown in Figure 13. Even with 50,000 annotation views, the cache management time is just about 3% in query time.

We performed a set of experiments to test the sensitivity of ViP to the cache size (Figure 14). We found that ViP worked best at 10% to 17.5% of the overall size. When

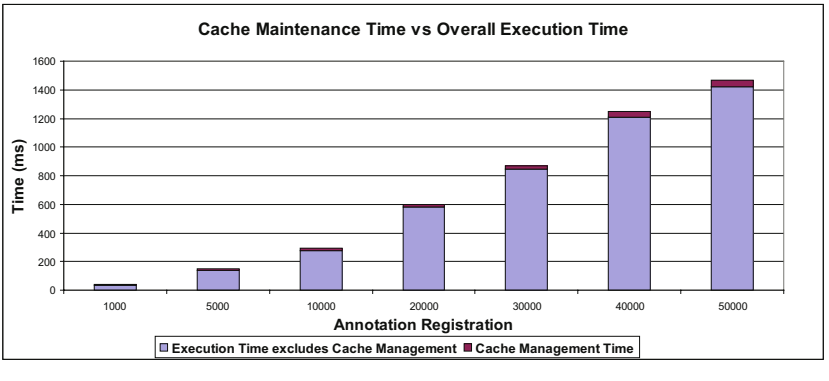
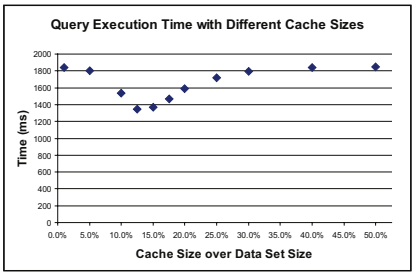
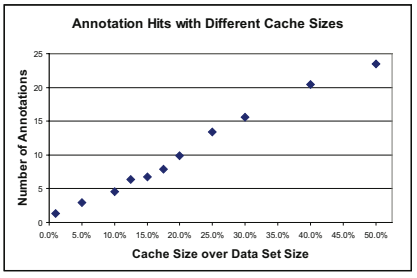


Fig. 13. Cache Management Time



(a) Query Time with Different Cache Sizes



(b) Annotation Hits with Different Cache Sizes

Fig. 14. Various Cache Sizes

the cache size was larger than 30%, not much benefit was gained (i.e., query times did not improve much) from further cache size increases, although the cache hits may be increased. This is clearly because a larger size cache brings extra effort to lookup and manage the cache, so the overall query time will not be reduced.

4.3 View-Based Annotation Path Propagation

We conducted a set of experiments where varied the HAP variable (HAP-q) in annotation path propagation. HAP-i was set as MAX-HAP. We present the results in Table 6. It is obvious that with deeper hops search, more annotations got matched and more time it took to retrieve them. Nonetheless, ViP increased the query time gradually.

4.4 User-Centric Access Control

Another interesting feature of ViP is its user-centric access control features. Not only users may issue queries that include their search preference, but also users can specify public/private annotation views when they register the annotations. The first set of experiments, in Figure 15 and Figure 16 illustrate how the different search coverage affected the query times and the number of annotations found. The most restrictive user-specified condition decreased the query time as well as the associated annotations.

On the other hand, Figure 17 and Figure 18 present the query 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 time almost decreased linearly as the public annotation views decreased; however, it decreased faster when the public annotation paths were decreased. Since

Table 6. Path Propagation in Network Semantics

HAP-q	**1**	**2**	**3**
Time (sec)	10.1445	11.1853	13.5833
Annotations Found	269	278	289

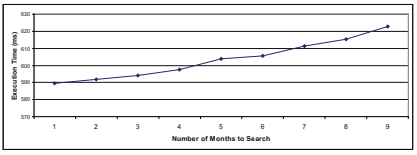


Fig. 15. Query Time for Different User Search Conditions

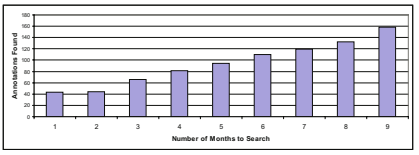


Fig. 16. Annotation Found for for Different User Search Conditions

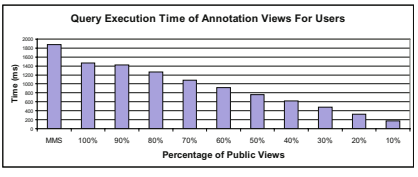


Fig. 17. Query Time with Different Public Annotation View Percentages

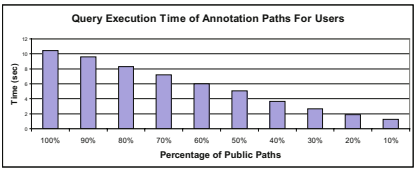


Fig. 18. Query Time with Different Public Annotation Path Percentages

annotation paths have the transitivity property, once the dependent views are not visible, it may speed up the query time exponentially. This essentially works like a first priority "filter" to reduce the query search time. In general, we expect such user-centric features to have a compound effect if used together, dramatically reducing query times.

5 Conclusions

In this paper we presented ViP, a view-based user-centric annotation framework. ViP introduced user-centric time semantics, network semantics, and access control for annotation propagation. Using database views as the underlying mechanism to implement these semantics enabled us to have a well-defined formal framework and also have a natural mapping to the existing user-interface, so that users of ViP do not have to learn SQL in order to specify their annotations. An other major advantage of ViP, compared to existing systems, is its use of caching techniques that significantly improve performance, as verified by our extensive experimental study on a real system.

References

1. Agrawal, P., Benjelloun, O., Sarma, A.D., Hayworth, C., Nabar, S., Sugihara, T., Widom, J.: Trio: A system for data, uncertainty, and lineage. In: Proc. of the VLDB conference (2006)
2. Benjelloun, O., Sarma, A.D., Halevy, A.Y., Widom, J.: ULDBs: Databases with uncertainty and lineage. In: Proc. of the VLDB conference, pp. 953–964 (2006)
3. Bhagwat, D., Chiticariu, L., Tan, W.-C., Vijayvargiya, G.: An annotation management system for relational databases. In: Proc. of the VLDB conference, pp. 900–911 (2004)
4. Buneman, P., Chapman, A., Cheney, J.: Provenance management in curated databases. In: Proc. of the ACM SIGMOD conference, pp. 539–550 (2006)

5. Buneman, P., Khanna, S., Tajima, K., Tan, W.-C.: Archiving scientific data. ACM Transaction Database Systems 29(1), 2–42 (2004)
6. Buneman, P., Khanna, S., Tan, W.-C.: Why and where: A characterization of data provenance. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 316–330. Springer, Heidelberg (2000)
7. Buneman, P., Khanna, S., Tan, W.-C.: On propagation of deletions and annotations through views. In: Proc. of the PODS conference (2002)
8. Chiticariu, L., Tan, W.-C.: Debugging schema mappings with routes. In: Proc. of the VLDB conference, pp. 79–90 (2006)
9. Chiticariu, L., Tan, W.-C., Vijayvargiya, G.: DBNotes: a post-it system for relational databases based on provenance. In: Proc. of the ACM SIGMOD conference (2005)
10. Cong, G., Fan, W., Geerts, F.: Annotation propagation revisited for key preserving views. In: Proc. of the CIKM, pp. 632–641 (2006)
11. Cui, Y., Widom, J.: Practical lineage tracing in data warehouses. In: Proc. of the ICDE, pp. 367–378 (2000)
12. Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. The VLDB Journal, pp. 471–480 (2001)
13. Eltabakh, M.Y., Ouzzani, M., Aref, W.G.: bdbms – a database management system for biological data. In: Proc. of the CIDR (January 2007)
14. Geerts, F., Kementsietsidis, A., Milano, D.: Mondrian: Annotating and querying databases through colors and blocks. In: Proc. of the ICDE, p. 82 (2006)
15. Jagadish, H.V., Olken, F.: Database management for life sciences research. The SIGMOD Record 33(2), 15–20 (2004)
16. Koutrika, G., Ioannidis, Y.: Personalization of queries in database systems. In: Proc. of the ICDE, p. 597 (2004)
17. Labrinidis, A., Qu, H., Xu, J.: Quality contracts for real-time enterprises. In: Bussler, C.J., Castellanos, M., Dayal, U., Navathe, S. (eds.) BIRTE 2006. LNCS, vol. 4365, pp. 143–156. Springer, Heidelberg (2007)
18. Lauzac, S.W., Chrysanthis, P.K.: Personalizing information gathering for mobile database clients. In: Proc. of the ACM SAC, March 2002, pp. 49–56 (2002)
19. Melnik, S., Adya, A., Bernstein, P.A.: Compiling mappings to bridge applications and databases. In: Proc. of the ACM SIGMOD conference, pp. 461–472. ACM, New York (2007)
20. Qu, H., Labrinidis, A., Mosse, D.: Unit: User-centric transaction management in web-database systems. In: Proc. of the ICDE, April 2006, pp. 1–10 (2006)
21. Srivastava, D., Velegrakis, Y.: Intensional associations between data and metadata. In: Proc. of the ACM SIGMOD conference, pp. 401–412 (2007)
22. Tan, W.-C.: Containment of relational queries with annotation propagation. In: Proc. of the DBPL conference (2003)
23. Tan, W.-C.: Provenance in databases: Past, current, and future. Special Issue on Data Provenance, Bulletin of the Technical Commmittee on Data Engineering 32(4) (December 2007)