2014-04-24

Management of Big Annotations in Relational Database Management Systems

Karim Ibrahim
Worcester Polytechnic Institute

Management of Big Annotations in Relational Database Management Systems

by Karim Ibrahim

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in Computer Science

by

May 2014

APPROVED:

_____

Professor Mohamed Eltabakh, Thesis Advisor

_____

Professor Elke A. Rundensteiner, Thesis Reader

_____

Professor Craig Wills, Head of Department

Acknowledgements

I would like to express my gratitude to my advisor professor Mohamed Eltabakh. Thank for his continuous support for my research and thesis work. I really appreciate his time, patient, guide, encouragement, and sharing his knowledge, which help me to continuously grow and improve during my master study.

My thanks also go to my thesis reader professor Elke Rundensteiner, for her valuable advises on my thesis work, which helps me to improve the quality of this thesis.

I thank all my DSRG members in WPI for all the feedbacks and discussions on my work.

I want to thank my whole family: my parents and my brother for being there, supporting and encouraging through the tough times. Thank for giving me tremendous support in my life and study.

Last, I thank GOD for this opportunity, for his guide and for giving me the ability and strength to finish this work.

Table of Content

Contents

List of Figures

Abstract

Annotations play a key role in understanding and describing the data, and annotation management has become an integral component in most emerging applications such as scientific databases. Scientists need to exchange not only data but also their thoughts, comments and annotations on the data as well. Annotations represent comments, Lineage of data, description and much more. Therefore, several annotation management techniques have been proposed to efficiently and abstractly handle the annotations. However, with the increasing scale of collaboration and the extensive use of annotations among users and scientists, the number and size of the annotations may far exceed the size of the original data itself. However, current annotation management techniques don't address large scale annotation management. In this work, we propose three chapters to that tackle the Big annotations from three different perspectives (1) User-Centric Annotation Propagation, (2) Proactive Annotation Management and (3) InsightNotes Summary-Based Querying. We capture users' preferences in profiles and personalizes the annotation propagation at query time by reporting the most relevant annotations (per tuple) for each user based on time plan. We provide three Time-Based plans, support static and dynamic profiles for each user. We support a proactive annotation management which suggests data tuples to be annotated in case new annotation has a reference to a data value and user doesn't annotate the data precisely. Moreover, we provide an extension on the InsightNotes: Summary-Based Annotation Management in Relational Databases by adding query language that enable the user to query the annotation summaries and add predicates on the annotation summaries themselves. Our system is implemented inside PostgreSQL.

Chapter 1: Introduction

1.1 Motivation

Annotations are used by many users and scientists to describe, comment or criticize the data values, and hence many applications use the annotation mechanism to capture users input [8] as in incomplete databases, probabilistic databases and data warehousing. In scientific databases applications, scientists from different fields such as bioinformatics, chemistry and physics labs, rely nowadays on scientific database to manage their data and experiments as it provides a set of features that traditional relational database management systems don't offer as annotation management. Scientific database [7] offers a way to maintain the data provenance that defines the how data was generated [12]. Also, scientific database provides a way to annotate data by attaching comments to the data [1][3][8][9]. Annotated databases are like a social media where scientists can communicate over certain data, but it is hard to analyze the annotations as of the free text unstructured nature. Scientists may annotate table, column, tuple or a cell. Meanwhile, the annotation propagation [11] phase will show all the annotations from different levels for each tuple. In the following example, Alice and Bob are scientists who query table Gene with its annotation. The problem is Alice and Bob get the same result set with same annotation. However, Alice and Bob have different background and different interests so each of them spend a good amount of time in order to find annotations that might be helpful. When users add annotation to set of data, usually it is a small set of data that they focus on. On the other side, when users add like article annotations that refer to different data, users might skip data to be annotated. Therefore, proactive annotation management analyses the newly annotation and verify if annotation can refer to other data. In the current annotation summary system InsightNotes, users can not write queries based on the annotation summaries. So, we introduce new language to enable the exploration and querying the data using summaries as predicates.

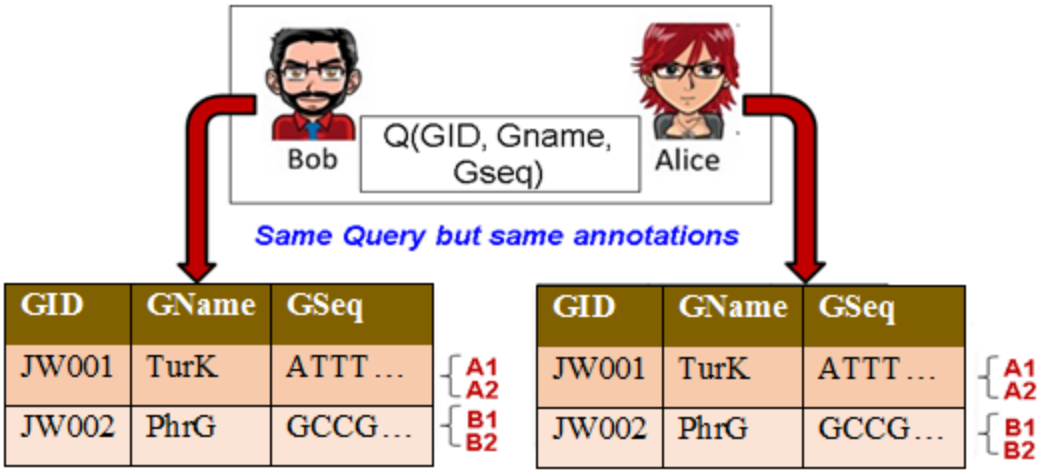


Figure 1: Current annotation propagation limitation

1.2 State-of-art Limitations

As Scientists use the annotation feature more and more[4], the amount of annotations propagated with each tuple increases dramatically as shown in figure 2 (comparing the number of annotation to the data size). With such increase it becomes harder for end-users to go over all reported annotations to find the most relevant ones to the user's interest. In the current systems, this problem is not addressed and the end-user gets all the annotations propagated with each tuple. Scientists find it an overwhelming task to read all the annotations and extract useful information. For example, users may be interested in reading annotations related to specific topic of keywords, or may be interested to check new annotations from trusted curators [2]. On the other hand, time of annotation creation can be of high importance to scientist as new discoveries change their perspective for data. Also, scientists can be interested in annotations that were recently created or within a certain time window. Therefore, there is a need for more efficient propagation techniques that selects among the so many available annotations, the most relevant ones that match each user's interest. In other research, the annotation is used as Provenance or belief but the annotation is structured and has well-defined domain so it can be easily analyzed. However, we consider the unstructured free text annotations.

For proactive annotation, it was not addressed before to analyze the annotation and search for hidden reference. There is a lot of work introduced concerning the keyword search in the relational database management systems and we use some techniques to detect annotation-data reference. InsightNotes, was an important work to summarize the data and propagate the summaries in efficient way with data. The next step in this research was to add the option of exploring the data through the summaries and still it was not addressed.

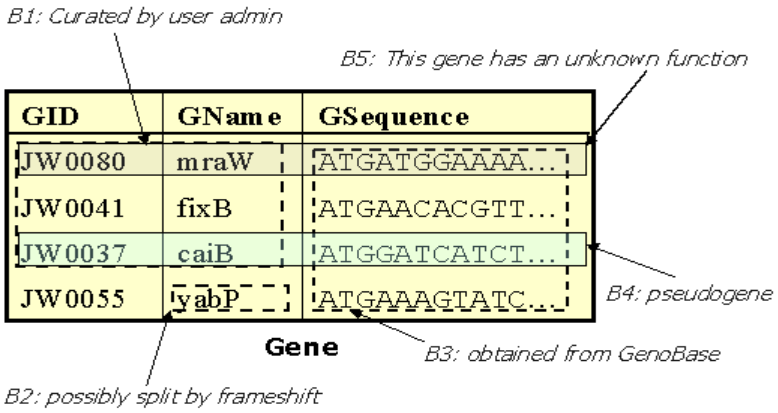


Figure 2: Example of annotated table

Species, Database	Gene Products Annotated	Annotations
Protein Data Bank [multispecies] GO Annotations @ EBI	176214	2538489 (687033 non-IEA)
Reactome [multispecies] CSHL & EBI	12934	97907 (97907 non-IEA)
UniProt [multispecies] GO Annotations @ EBI	17408324	110329676 (1145210 non-IEA)

# of annotations is order of magnitude higher than data itself

Figure 3: Annotation magnitude compared to data

1.3 Challenges & Proposed Solution

Personalizing the output results has been addressed in different contexts because of its effectiveness and its ability to offer user-centric processing. Examples of these systems include Google search engine, Amazon commercial system[10], Netflex, Context-aware databases [6], and many others. However, personalizing the annotation propagation has not been addressed yet and it involves several new challenges over existing systems that we address in this thesis. For example, Amazon and Netflex depend on collaborative filtering techniques [5] that require matching profiles of many users in order to provide recommendations. In our system (chapter 2) , the personalization depends on ranking the annotations based on each user's profile independent of the other users. In systems like Google, they capture users' previous operations, e.g., previous keyword search, in order to customize future results. In our system, we capture and track more complex operations such as the relational queries inserted by users to evolve their profiles. Additionally, we address novel ways to efficiently identify and propagate the annotations of interest, which is different from existing techniques. For proactive annotation management (chapter 3), it basically searching the annotation content and trying to discover matches to data values. The naïve way is to search every token in the annotation and compare it with the data values in the tables. However, this would be time consuming and inefficient. So, we try to find hints in the content of the annotation in order to minimize the search operation and find a match to the data values. After a match is found, users need to verify the annotation – data link if it should exist or it should be broken. In the InsightNotes, users don't know what kind of summaries attached to data, so a new language is introduced to enable exploring of summaries (chapter 4). Then summaries is stored and indexed in an efficient way so that users can add predicates on the summaries and get the data that satisfy this kind of predicates.

1.4 Contributions

In chapter 2, we propose a system that captures users' preferences and profiles over time, and personalizes the annotation propagation at query time by reporting the most relevant annotations (per tuple) for each user. Thus different users with different interests may issue the same query, receive the same data results, but with different set (filtering) of annotations attached to the results. In our system, each user's profile consists of two components: static features and dynamic features. The static features are entered by the user and contain information such as list of annotation curators, time window and list of keywords. The dynamic features are automatically captured and maintained by the system including the users' previous queries. Using the user profiles, we propose Time-Based filtering that helps user to find annotations that map to his/her preferences. The Time-Based filtering consists of three categories: ALL, NEW and TIMELINE. 'ALL' propagates all annotations attached per tuple order by the ranking of the annotations in descending fashion. 'NEW' propagates annotations that were added after the last propagation query on the data tuple. The last category is TIMELINE where the all the annotations are divided in four time segments relative to the last executed propagation query. One of the segments is 'NEW' like previously explained. The other Three of time segments are time intervals starting from date of the last query to number of days specified by the user. We propose another two layers of filtering as the number of annotations still can be overwhelming to the user, we introduce the clustering filter (based on the content of annotations) and Top K filtering (based on the ranking of annotations)

In chapter 3, with Big annotations and large scale datasets, users may not annotate all the data that should be annotated. For instance, user may add an article related to certain Gene sequence, meanwhile a protein name was mentioned in the article that user did not attached the article with. Therefore, proactive annotation management provides an efficient way to detect such hidden links in newly inserted annotation and build temporary annotation-data link until the link is verified by the user.

In chapter 4, in InsightNotes, different types of summaries are built on top of the raw annotation annotations. Users are not able to explore the summaries and add predicated on the summaries, they can only propagate summaries with the data. Therefore, we introduce an extension to InsightNotes to efficiently query the data based on summaries and indexing the summaries for faster query response time.
The key contributions of this thesis can be summarized as follows:

1. Proposing system that enables personalizing the annotation propagation in relational database systems. The main purpose of the system is to minimize the number of annotation to be propagated with data and return the most relevant annotations.

2. Providing different techniques to capture users' profiles including both static and dynamic features. The system is flexible that enable the users to introduce new features to be evaluated for each annotation.

3. Introducing Time-Based filtering with option of Top K operator or clustering that reflects user's preferences. Users get the data with the annotations in descendant order by their scores which reflect most relevant annotations first for each time filter. In order to return fewer annotations, clustering over the annotations content or top K overall score can be used.

4. Introducing proactive annotation, which detects cross referencing in text annotation to data values and enable users to verify if a new annotation should be applied to refer data.

5. Extending the annotation summary, to include new language that enable the user to query the data based on the summaries and efficiently querying the data based on summaries predicates.

6. Implementing the proposed system inside the PostgreSQL database system.

Chapter 2: User-Centric Annotation Propagation

User-centric Annotation system consists of three main components: (1) Profile Manager that manages and maintains users profiles (static and dynamic features), (2) Extended query engine which integrates the filtering operations with the query plans , and (3) PrefNotes language and interface layer for creating the profiles and for personalizing the annotation propagation.

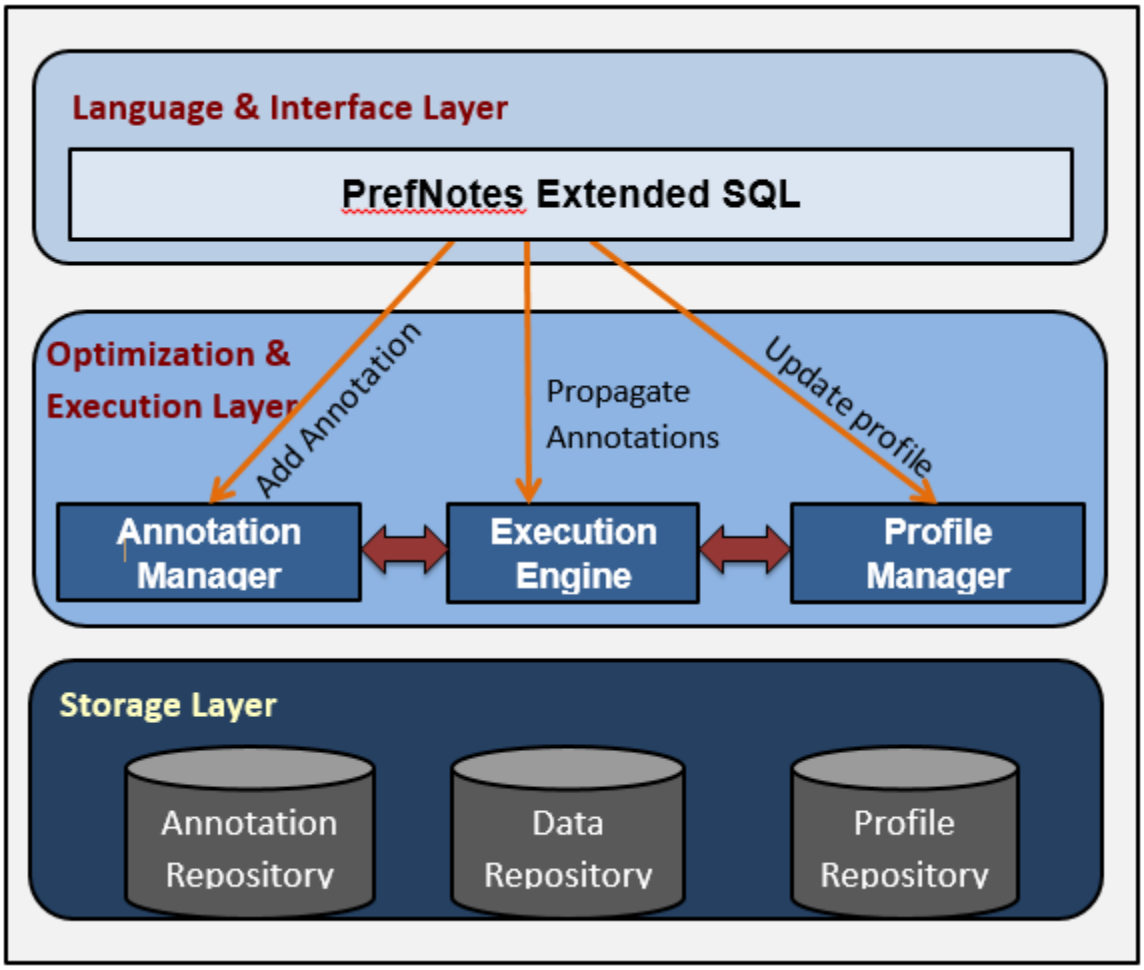


Figure 4: Annotation user-centric system Architecture

2.1 User Profile

The profile manager is responsible for creating and maintaining users' profiles and storing them in the *Profile Repository*.

2.1.1 Static Profile

For the static profile, the system keeps track of three main dimensions: the curator dimension, the time dimension, and the keyword dimensions. The curator dimension, user may trust annotation's curator more than other for example scientists in the same lab would prefer to know their colleagues' annotations first. Each user provides an ordered list of curators and based on that order, the ranking of annotations is computed for that dimension. For the time dimension, scientists may be interested in different time windows. Users can easily specify an annotation's time window that reflects uses' preferences. For the keyword dimension, users provide a list of keywords or topics of interest such as "Gene" or "Protein" or gene sequence "ACAAGA".

2.1.2 Dynamic Profile

For the dynamic feature, User-centric Annotation system keeps a query log that is used to store all the queries with predicates that was successfully executed by each user in a given time. A stored procedure is executed to extract per user the frequent predicates values used in the where clause and the frequency visited data rows (used in version 1 only). The database administrator defines a time threshold for the queries to be expired and removed from the query logs. Also, each user defines a frequency threshold to define the frequency of the predicates and the rows visited. In the user's profile, for each user the dynamic keyword feature is updated with the frequent predicted values, also the frequent visited rows dynamic feature is updated by the Object identifiers (OID) which is unique for each row in the database. The database administrator has the option to execute this stored procedure is the most relevant time according to their application environment. Besides that, PrefNotes also extract keywords of interest from the annotations written by the users (add annotation queries in the query log). We extract the

keywords that refer to a value of the annotated data. Matched words are added to the annotation dynamic keyword feature for each user.

2.1.2.1 Keyword Extraction

User-centric Annotation system executes three different stages in order to extract the dynamic features. First stage is storing the users' queries. For every processed query, the system extract the table name from the from-clause as well as the predicates in the where-clause. Then the output (table name, predicates) are stored in log table along with the user id and timestamp. Each user can control the query to be analyzed in the log by specifying the point of time away from the current date. Second stage is analyzing the queries in the log. For the keyword extraction, the system focuses (constant) strings used in the predicates of the queries. Tokenization of the strings takes place and a word frequency count is performed. The system maintains a keyword frequency table that represents the extracted keywords and their count per user. User-centric Annotation system does an incremental update on this table every day to update the words frequencies by deduction of words in the outdated queries and addition of words of new queries. It is believed that users do not use the string predicates quite often so keyword frequency table will not be huge and incremental update is efficient.

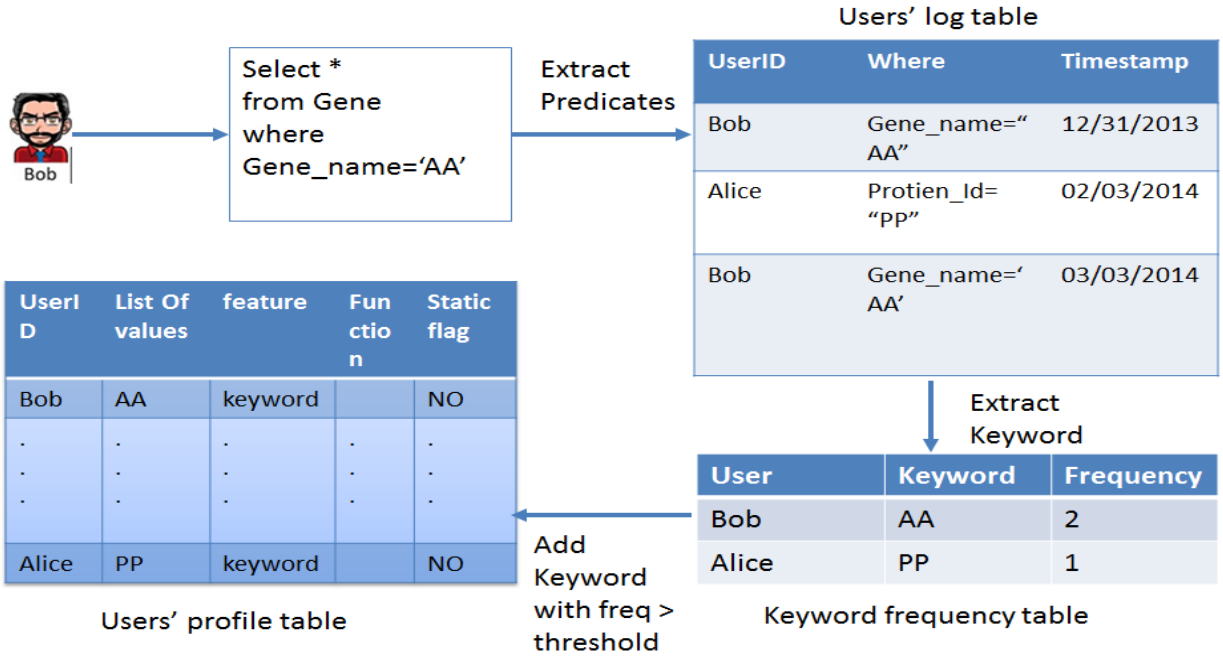


Figure 5: Dynamic keyword extraction

2.1.2.2 Visited Tuple Frequency

For the row frequency, the system creates a function per table (from-clause) that list the predicates to be tested for each row. The system runs the created function against each row in the table to test its frequency .The row frequency is measured as the number of predicates that data row satisfies in the testing. An auxiliary table row frequency is maintained to save the row OID and frequency per user. Incremental update will not be efficient for this table as the number of row OIDs are typically huge so the table will be big and hard to maintain. In the third stage, the system offers the user the option of specifying the thresholds for both the keyword frequency and row frequency. If a keyword frequency is defined by the user, User-centric Annotation system extracts the keywords that satisfy the threshold and add list of dynamic keywords to the user's profile. If the row frequency is defined, the system deletes the OIDs that do not satisfy the threshold. The row frequency table is used version 1 in the proportional Top K to measure the row weight. For each user, the row weight is calculated as the row frequency divided by the maximum row frequency. Finally, the system removes the outdated queries from the log table.

2.2 Ranking Functions

Each feature in User-centric Annotation system has an associated ranking function that scores each annotation with respect to that function. The profile manager will manage a pool of ranking functions that are registered inside the system. The ranking for each feature is normalized so that the overall score is normalized as well. Each feature may have a default ranking function, but users may override these defaults and have their own functions. To get the most relevant annotation, an overall score that represent all different dimensions is calculated to evaluate an annotation among its peer annotations per tuple. The scoring function is pluggable component so user has the control to introduce new dimensions other than the defined ones and to change an existing one as long as all the functions give normalize score. According to the overall score the annotations with dominating score are selected to be propagated with the tuple.

2.2.1 Keyword scoring

The keyword scoring function, the tokens in each annotation is compared to the list of keywords given by the users or generated from the user's query history and if there is a match the score increment by one and the end comparisons stage the score is normalized by the number of tokens in this annotation. An aggregated score is calculated from the scores of the given functions and normalized to give an overall score for each annotation.

2.2.2 Curator scoring

For curator ranking function, we have two lists static list and dynamic list. The static list is the order list of scientist's curator preferences which is provided previously. The dynamic list of curators is an order list generated based on the curator frequencies used in scientist's query history. The system assigns a uniform weight for each curator=(n-i)/n such that the n is the number of curator in the list and i is the position of the curator in the list. For each annotation a curator weight is assign to it.

2.2.3 Time scoring

The time ranking function, given the users' static time window preference and users generated dynamic time preference, for each tuple the time of the annotation creation is compared and if the annotation with creation date closer to the end date in the window gets higher score. Here is the formula for computing the time score: 1-(E-T)/(E-S) where E is the end date , S is the start date and T is the creation date of the tested annotation.

2.3 User-Centric Annotation Propagation: Explored Directions

2.3.1 Top K operators

In this section, we present the Top K ranking operators in PrefNotes. We propose two variants of the Top K operators that differ in their costs and accuracy. Fixed Top K and Proportional Top K. In figure 1, the execution engine gets the answer set from the data repository and propagates the annotations from the annotations repository and then evaluates each annotation based on the profile retrieved from the profile repository.

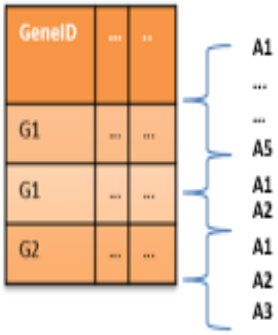
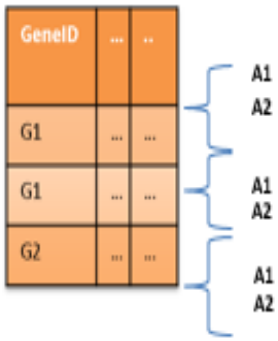
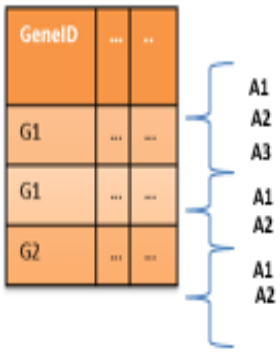


Table Gene with annotation



Fixed top K=2



Proportional top K=2

2.3.1.1 Fixed Top K

Fixed Top K operator in figure 2b returns exactly K annotations for each tuple based on the ranking of annotation. Meanwhile, Fixed Top K operator doesn't take into account the popularity of the tuple so tuples with more annotations are treated equally as tuples with fewer annotations. However, Fixed Top K operator is good as it is non-blocking operator, as for each tuple it calculates annotations scores and selects the Top K.

2.3.1.2 Proportional Top K

The Proportional Top K operator in figure 2c overcomes this shortcoming by assigning proportional K for each tuple according to their popularity. The proportional

equation is :K=(number of annotation per tuple/ number of all annotations) . The Proportional Top K is semantically better as it takes into consideration the popularity of the tuple, however it is an expensive operator because of its blocking nature. The Proportional Top K is a blocking operator because all the annotations of the answer set should be retrieved and calculate for each tuple the ratio of number of annotation over all these annotations (if we have the total number of annotations of the answer set it will be non-blocking). the equation of the K is

K = (# of Annotations Per Tuple * Expected # of Annotations) / Total # of Annotations In Answer Set

For the Top K the following query is used to get the results in the commands:
Select * From gene Propagate Annotation {Top K };

2.3.2 Skyline filtering

Skyline is used in different applications to extract the dominating points in graph. One of the famous problems is the hotel problem where customer compares the distance and price dimensions of different hotels. In our system for every tuple, we build skyline model to filter the annotations attached to that tuple. In the annotations evaluations, the first step is to build the graph with points formulated by the three dimensions scores (keyword, curator and time) calculated by the ranking functions. After scattering the annotations scores in the graph skyline, the system selects the annotations with highest overall score that dominate the other annotations with less overall score. The skyline output annotations represent the most relevant annotations (highest score) to user with different dimension scores. We assume that users don't have certain preference a dimension over another dimension.

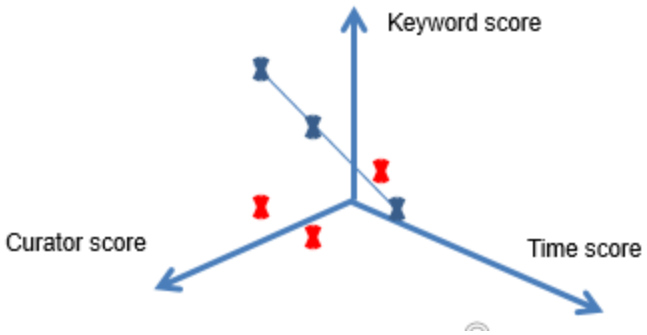


Figure 6: Skyline in User-centric annotation system

2.3.2.1 Score Threshold

However, a small tweak can enable the user to define the importance of a dimension by adding weights to dimension scores in the overall score equation. The skyline will not be efficient if none of the user's preferences are met over the annotations, which leads to propagating all non-relevant annotations. Therefore, users have the option to dene a threshold on the overall score so that no low overall score annotations are propagated with data tuples. This option helps the user to focus more on what they are searching for.

2.3.3 Annotation Clustering

In the previous techniques, the system compares the annotations of based on the three dimensions defined before. While the skyline and Top K filter the annotations, yet the annotations can be similar and having the same overall score. Our system offers the option of diversity of annotations with is based on the content of the annotations. The system enables the user to focus on the difference of annotations based on the similarities of the annotations text. To measure the similarity between two annotations, the system uses cosine similarity function that checks the common words over the overall words. The output of the cosine functions is number range from zero to one, zero represents totally different texts and one represents exactly the same text. The system constructs connectivity based clusters using the cosine similarity as the distance between annotations. A threshold is defined by the system administrator to specify the minimum

distance to construct a cluster. There are two cases are handled when new annotation is introduced to the cluster model:

1) The annotation's distance to any of the annotations in a cluster exceed the threshold, therefore the new annotation belong to this cluster and add to the cluster for further comparisons else compare to the rest of the clusters
2) The annotation's distance to all of the annotations in the given clusters do not exceed the threshold, then a new cluster is created with the new annotations. The clustering technique help the user show various annotations content over the data so that it reflect different information attached to the data.

2.4 User-Centric Annotation Propagation: Final Design

The previous version has a lot of technical concepts that scientists – our targeted users –find so hard to understand and how to use these features. Also, Skyline filter can eliminate annotations that might have an importance to the user though the annotations have low ranking. Therefore, we propose another version of User-centric Annotation that can represent and personalize the propagation of the annotations, yet ease to understand and use as well it sorts the annotation based on their ranks.

Phase I:

The main idea is to represent the annotations of more importance first. Therefore, annotations per tuple are ordered in descending order based on the overall ranking score. Though, User-centric Annotation uses ordering of annotation, yet the order operator used is not blocking like the regular operator used in Postgresql. The ordering takes place only on the data tuple level. Blocking operators like ‘order by ‘or ‘group by’ are operators that will not return any tuple until all the returning data is fetched and processed.

User-centric Annotation provides three features to organize the annotations to the user in a convenient way.

1. ‘ALL’ which returns the all the annotations per tuple ordered by the overall score.
2. ‘New’ which returns all the annotations that were added after last propagate query executed by the user.
3. ‘Timeline’ which arrange the annotation to four segments of time (‘New’, ’Recent’, ’Old’ and ‘Archive’).

2.4.1 Propagate ‘ALL’:

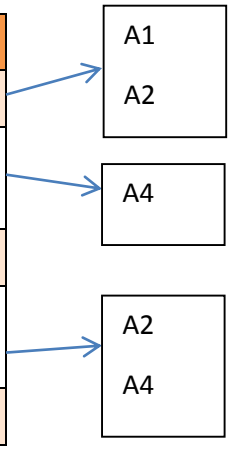
In this type of propagation, each annotation is evaluated against the user preference and an overall score is assigned to the annotation based on the ranking functions illustrated before. Due to the indices build on top the OID and the annotation id that join the annotation table with the data tables, it is granted that the annotations are propagated per tuple first and not interleave with other data tuples. After all the annotations per tuple are evaluated and before propagated to the user, the annotations are sorted in descending order based on their scores.

Annotation table

Annotation ID	Date	Curator	Annotation
A1	01/02/2014	Scientist_1	A b c d
A2	01/02/2014	Scientist_2	A b c d f
A3	11/11/2013	Scientist_2	A b c
A4	01/01/2013	Scientist_1	f

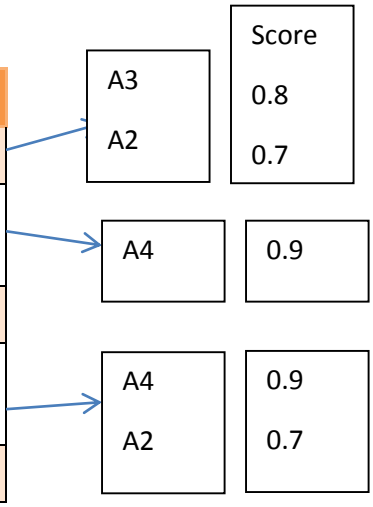
Gene table

OID	ID	Name
19666	AARS	Alanyl-tRNA synthetase
19667	ABCC9	ATP-binding cassette, sub-family C (member 9)
19668	ABHD5	Abhydrolase domain containing 5
19669	ACADVL	Acyl-Coenzyme A dehydrogenase, very long chain
19670	ACTA1	Alpha actin, skeletal muscle



Propagate 'ALL'

OID	ID	Name
19666	AARS	Alanyl-tRNA synthetase
19667	ABCC9	ATP-binding cassette, sub-family C (member 9)
19668	ABHD5	Abhydrolase domain containing 5
19669	ACADVL	Acyl-Coenzyme A dehydrogenase, very long chain
19670	ACTA1	Alpha actin, skeletal muscle



2.4.2 Propagate 'New':

Scientists basically use annotation to communicate and collaborate over the data and their experiments. Hence, time is of great importance as scientists want to explore the new findings. User-centric Annotation offers the option to users to propagate the annotation that was recently added (creation date) after last propagate query executed by the user. Therefore, an auxiliary table annotation_propagation_log is used to keep track of the annotated tuples that were viewed by the users and the time of the query execution.

Tuple_OID	User	Date
19666	Scientist_1	01/01/2014
19666	Scientist_2	01/02/2014
19667	Scientist_3	01/02/2014
19669	Scientist_3	01/02/2014
…	…	…

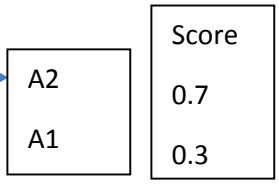
Same tuples different users

Different tuples same user

When Scientist_1 propagate gene table , 'A3' annotation will be eliminated from the returned annotation as the creation date is before the last propagation query date, therefore the annotation is not 'New' any more.

Propagate 'New 'for Scientist_1

OID	ID	Name
19666	AARS	Alanyl-tRNA synthetase
…	…	…



2.4.3 Propagate 'Timeline':

As it was mentioned before, time is of a great importance to scientists to explore new findings and new annotations added by other scientists. The 'timeline' propagation is another type of propagation that helps the user to find what he wants by organizing the in four different segments. A configuration variable of number of days is defined by the database administrator to set the interval between each time segment. Also, the annotations in the same segments are ordered by the overall score.

The 'timeline' divides the annotations propagation in four ordered segments:

1. 'Segment_1': the annotations those haven't seen by the user from the last propagation query
2. 'Segment_2': the annotations those are recently by the user from the last propagation date minus the variable number of days till the last propagation date.

3. ‘Segment_3’: the annotations those are old relative to the last propagation which is from the last propagation date minus twice the variable number of days till the last propagation date minus the variable number of days.
4. ‘Segment_4’: the annotation those are older than last propagation date minus twice the variable number of days.

Example

Gene:

OID	ID	Name
19669	AARS	Alanyl-tRNA synthetase
…	…	…

	Creation Date
A7	01/05/2014
A8	12/01/2013
A9	11/01/2013
A10	01/01/2012
A11	01/05/2014

‘Timeline’ Propagation:

OID	ID	Name
19669	AARS	Alanyl-tRNA
…	…	…

Anno Content	Type	Score
A7	Segment_1	0.7
A11	Segment_1	0.3
A8	Segment_2	0.2
A9	Segment_3	0.6
A10	Segment_4	0.9

2.5 Filtering options:

For the last three types of propagation, scientists still can find the numbers of annotations are overwhelming. Therefore, User-centric Annotation offers two types of filters that return fewer annotations:

- Clustering filtering
- Top K filtering

2.5.1 Clustering filtering:

As in the version 1 of User-centric Annotation, annotations are compered to detect the similarity between them. It is a way to show diverse annotation to the user and eliminated redundancy. The system uses the cosine similarity that compares annotations content to get the ratio of the common words to the total number of words and gives a normalized value that represents the closeness. A threshold is configured to set if two annotations are similar if the value of the similarity function is greater than the threshold (the default is 0.7). The main difference between this version of clustering and the previous one is choosing the representative of the cluster. User-centric Annotation groups the annotations with similar content as shown in the figure below and select a representative:

- In version 1, the representative is selected randomly
- In version 2, the representative is selected with the highest overall score

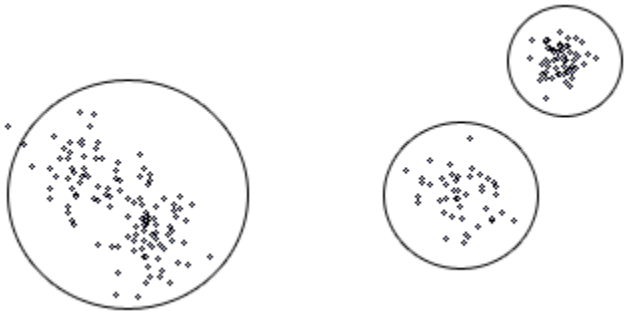


Figure 7:Clustering representation

For each propagation type the clustering filtering can be applied:

Gene:

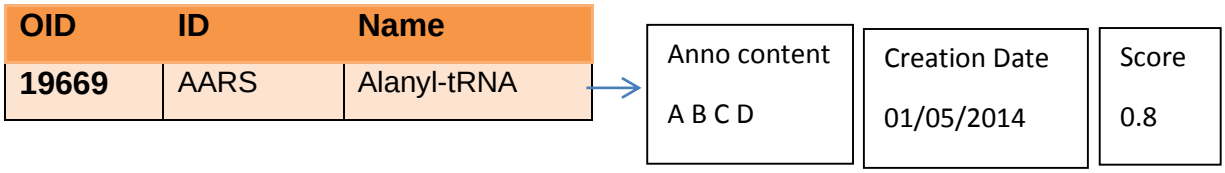
OID	ID	Name
19669	AARS	Alanyl-tRNA

→

Anno content	Creation Date	Score
A B C D	01/05/2014	0.8
A B C D F	12/01/2013	0.9
F C D	12/05/2013	0.6

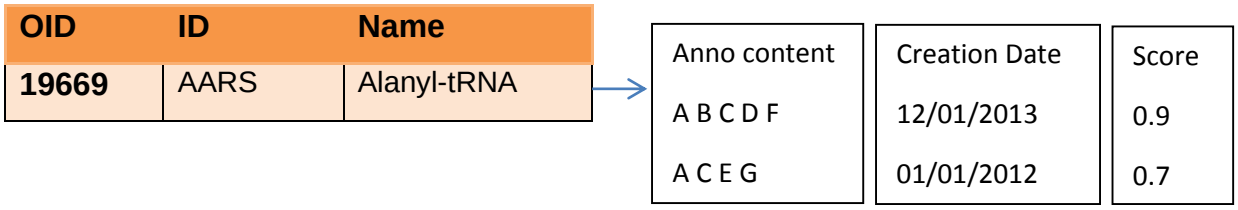
Propagate ‘All’:

The annotation that are similar (A B C D, A B C D F) with value 0.89 while the (A C E G, A D E G) is 0.75. The total number of words can make a difference for example if we remove the G from the last two annotations the similarity value of (A C E, A D E) is 0.66 which is less than the threshold and the two annotation will be considered different. Also, (F C D, A B C D F) is 0.77.



Propagate ‘New’:

The propagate ‘New’ filters the annotation to get the new annotation which user ’scientist_1’ whose last propagate query was in ‘01/01/2014’ will be (A B C D, A D E G) and their similarity is 0.5. Since, the similarity value less than the threshold both of the annotation will be propagated as following:



Propagate ‘Timeline’:

In propagate ‘Timeline’, the annotations are ordered and organized by the four segments of time that represent different intervals. The clustering works on each segment separately form the other segments. In the previous example, the annotations (F C D, A B C D F) are in the same segment and the similarity value is 0.77.

OID	ID	Name
19669	AARS	Alanyl-tRNA

Anno content	Type	Score
A B C D	Segment_1	0.8
A D E G	Segment_1	0.2
F C D	Segment_2	0.7
A C E G	Segment_4	0.9

2.5.2 Top K filtering:

User-centric Annotation offers Top K filtering to the users in order to minimize the propagated annotations returned. In version 1 in User-centric Annotation, two types of top K were introduced fixed top K and proportional top K. The problem with the proportional top K is that is technically complex and hard to understand how to use by the scientists (our targeted users). Therefore, fixed top K was only introduced in version 2 of User-centric Annotation that return only K annotation

Example:
Gene:

OID	ID	Name
19669	AARS	Alanyl-tRNA

Anno content	Creation Date	Score
A1	01/05/2014	0.8
A2	12/01/2013	0.9
A3	12/05/2013	0.6

Propagate 'ALL':

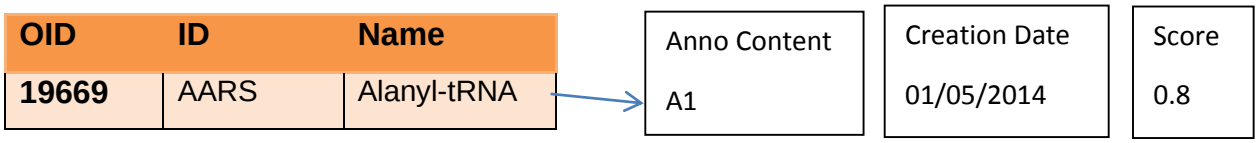
Given K=2, the system will propagate two annotations the top scores.

OID	ID	Name
19669	AARS	Alanyl-tRNA

Anno content	Creation Date	Score
A2	12/01/2013	0.9
A1	01/05/2014	0.8

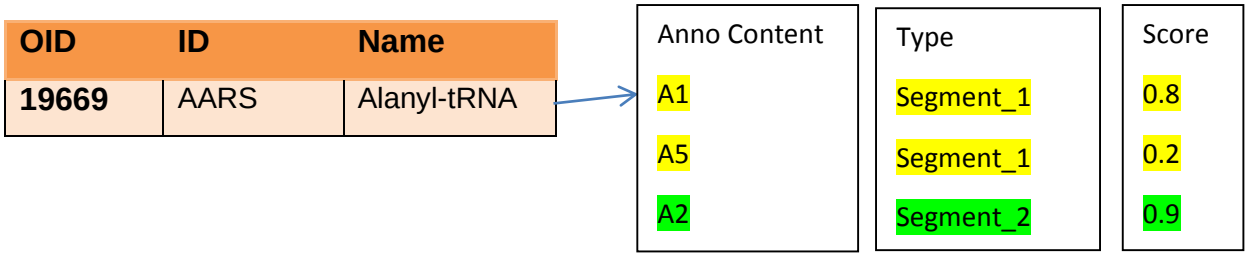
Propagate ‘New’:

If ’scientist_1’ whose last propagate query was in ‘01/01/2014’ choose K=1 to propagate ‘new’ , just one annotation with the top score will be propagated.



Propagate ‘Timeline’:

If ’scientist_1’ whose last propagate query was in ‘01/01/2014’ choose K=3 to propagate ‘Timeline’, just three annotations with the top score will be propagated starting from annotations of type ‘New’.



Chapter 3: Proactive Annotation Management

In this chapter, we introduce proactive annotation management which is a plugin to the scientific database management system. The goal of proactive annotation management is to detect any hidden links in the annotation that may refer to data value, which if found the annotation should attached to this data value. In most of the cases, scientists add annotation that are large in size like scientific article which usually contains many references to the data. Scientists might skip to annotate all the referred data, therefore proactive annotation management is necessary to automate any missing referencing.

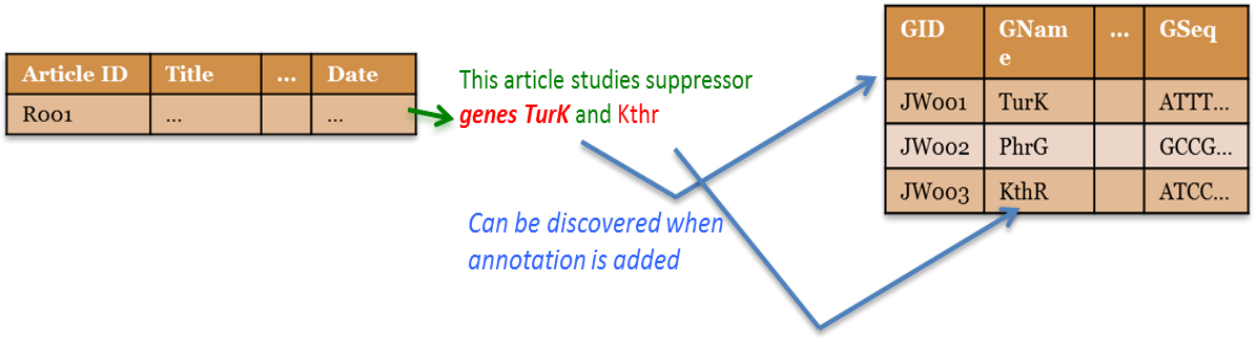


Figure 8: motivation

A naïve way of implementing the proactive annotation management is to search each word in the annotation and trying to find a match in the domains of the whole database schema. Though all the possible hidden references will be discovered, but with big annotations and huge data this approach is time consuming and inefficient.

The first step in the proactive annotation, the database administrator highlights which column in the data schema to search for a match. In most of the cases, when scientists refer to an object they use their scientific identifier or scientific name. Therefore, we assumed that the column to be searched in the proactive annotation management is a singled valued columns. Highlighting columns and tables is not critical step instead the system can search whole schema, but for optimized performance and practical scenarios it is better to select some columns and tables. Indices are built automatically in top of the

highlighted columns in order to speed up the searching process to find a match in the annotation.

The next step is to parse and analysis the annotation. We depend on postgresql text search function “to_tsvector” that transform text to tsvector. tsvector value is a sorted list of distinct lexemes, which are words that have been normalized to make different variants of the same word look alike. Sorting and duplicate-elimination are done automatically, and stop words as (the, a) are eliminated as well. To_tsvector defines the positions of output token in the original text. Tsvector is special kind of vector that it can’t be searchable. Therefore, we transform the vector to array of objects that contain the word and list of positions.

We try to minimize the number of search words as much as we can. So, we create two lists concerning the data schema. The first list is the columns that was mentioned by the database administrator to be searched, the other list is the columns that are not indicated to be searched. For each table name highlighted to explore, we get the columns name and divide them to the list to be ignored and the list to be searched.

The next phase is to search for hints in the annotation. Based on research papers about keyword search in relational database, users tend to specify the type of object they are searching for. For instance, according to users’ behavior queries like ’Department CS’ are used to make it clear that CS is of object department which is equivalent to either a table name or a column name. Therefore, we used the same concept to search for hints in the annotations. The system tries to find if the words that was listed by the database administrator as table names and columns are found in the content. If any of them is found they are marked as hints.

After the hints are highlighted, the system gets the position(s) of each hint marked from the previous step. Then, the context of each hint is extracted from the annotation using these positions. The number of words surround the hints define the context, we show at the experiments section how that affect the performance typically we extract three words after and before the hint words.

Afterwards, the system analyzes the context. If the context contains any of the schema from the list to be ignored, then the system will not consider this context to be searched. Otherwise, if the hint is a table name then the context words are searched in all the column names that are related to that table name and in the list of schema to be searched. If the hint is a column name then the context words are searched only in that specific column. If match found, then the link is established between the new added annotation and the tuple that contain the referred data. The link is added to the join table between the annotation and the data with a flag that indicate that the link is system generated.

The final stage is to verify the links that was detected by the system from the hints found in the annotations. The user who added the annotation has the right to view the created links. Then, the user decide to confirm the link which change the system generated flag to false or to reject the link so the system removes this record form the join table that connects the annotation with data.

3.1 Workflow of Proactive Annotation Management:

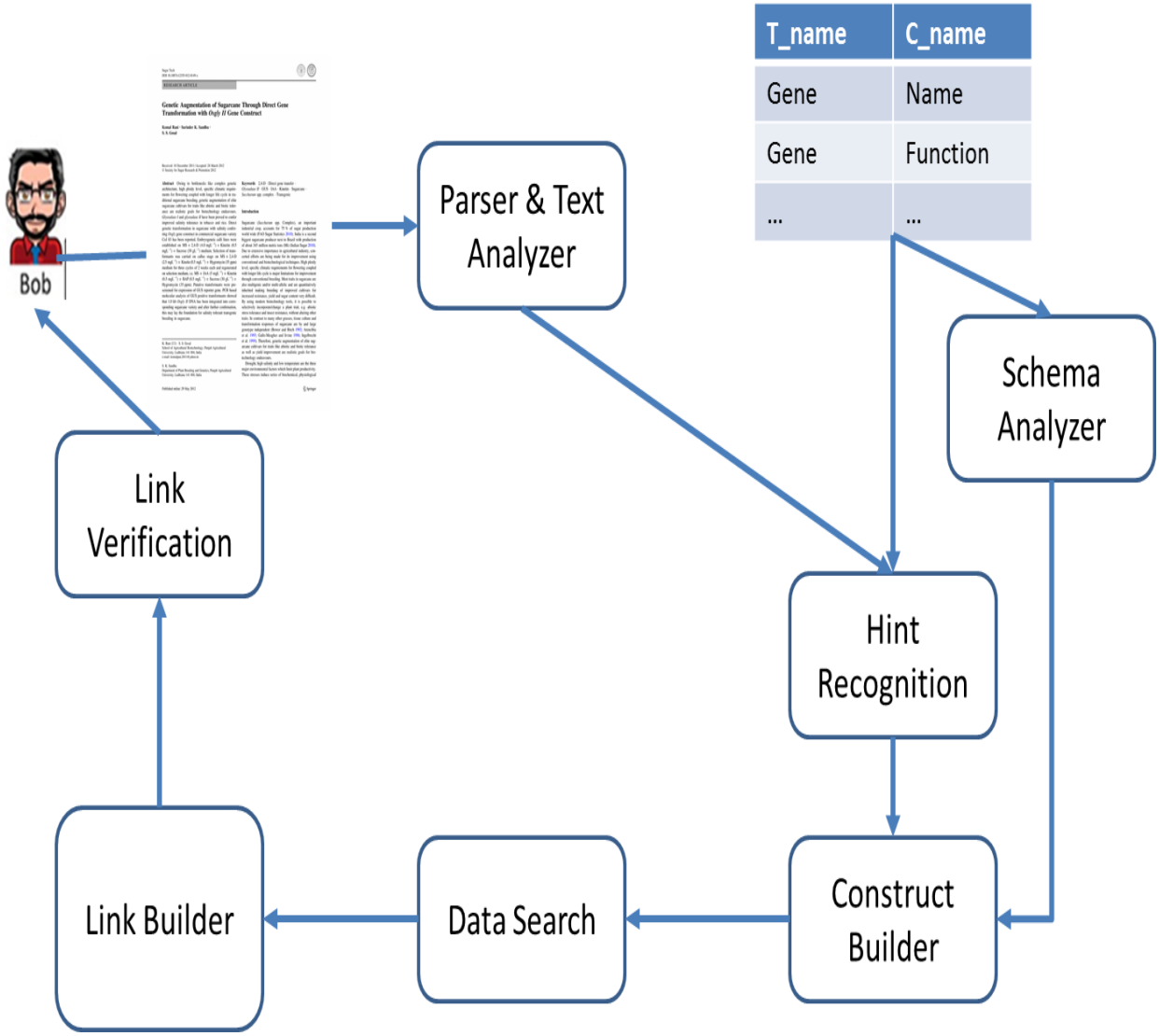


Figure 9: Proactive Annotation Management Architecture

3.2 System Components

3.2.1 Parser & Text Analyzer:

Searching words is not a straight forward task, as words take different forms as plural for nouns and present continuous form for verbs. We need to transform the words to their lexeme (unit of lexical meaning that exists regardless of the number of inflectional endings it may have or the number of words it may contain). In order to get the lexeme of any word, a dictionary is looked up to search for the basic word and its possible mutations. PostgreSql offers the option to analyze the text, convert each word to its original lexeme, remove the stop words (words which are filtered out prior to processing of natural language data) and recognize the position of each word in the text. 'To_Tsvector' is a function that takes text as input and transform to vector of words lexemes and their positions.

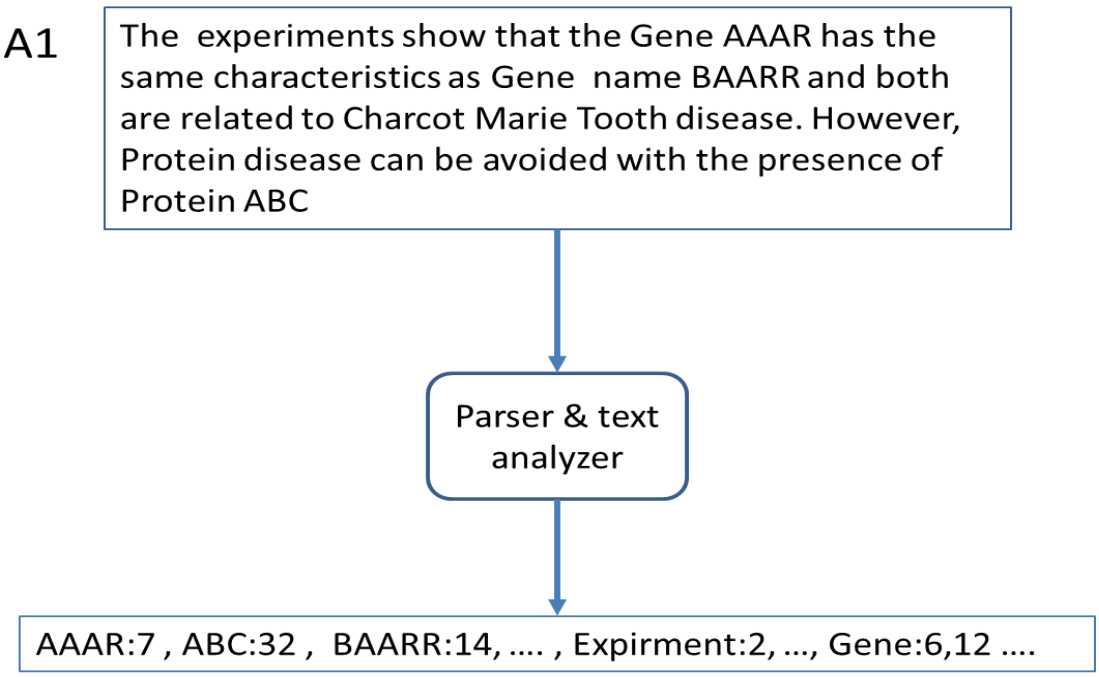


Figure 10: Parser & Text Analyzer Example

3.2.2 Schema Analyzer:

The schema analyzer is responsible for creating to list to be used in the context analyzer. The schema analyzes the tables and columns that are highlighted by the database administrator before (blue table). Then, for each table the system get its schema (orange tables) and classify each column either to be searched or to be ignored. If the column name is highlighted by the database admin then it is to be searched otherwise to be ignored. Here is the query to get a table column names:

```
SELECT  pg_catalog.textin(attname)
FROM   pg_attribute
WHERE  attrelid ='Gene'::regclass  AND    attnum > 0 ;
```

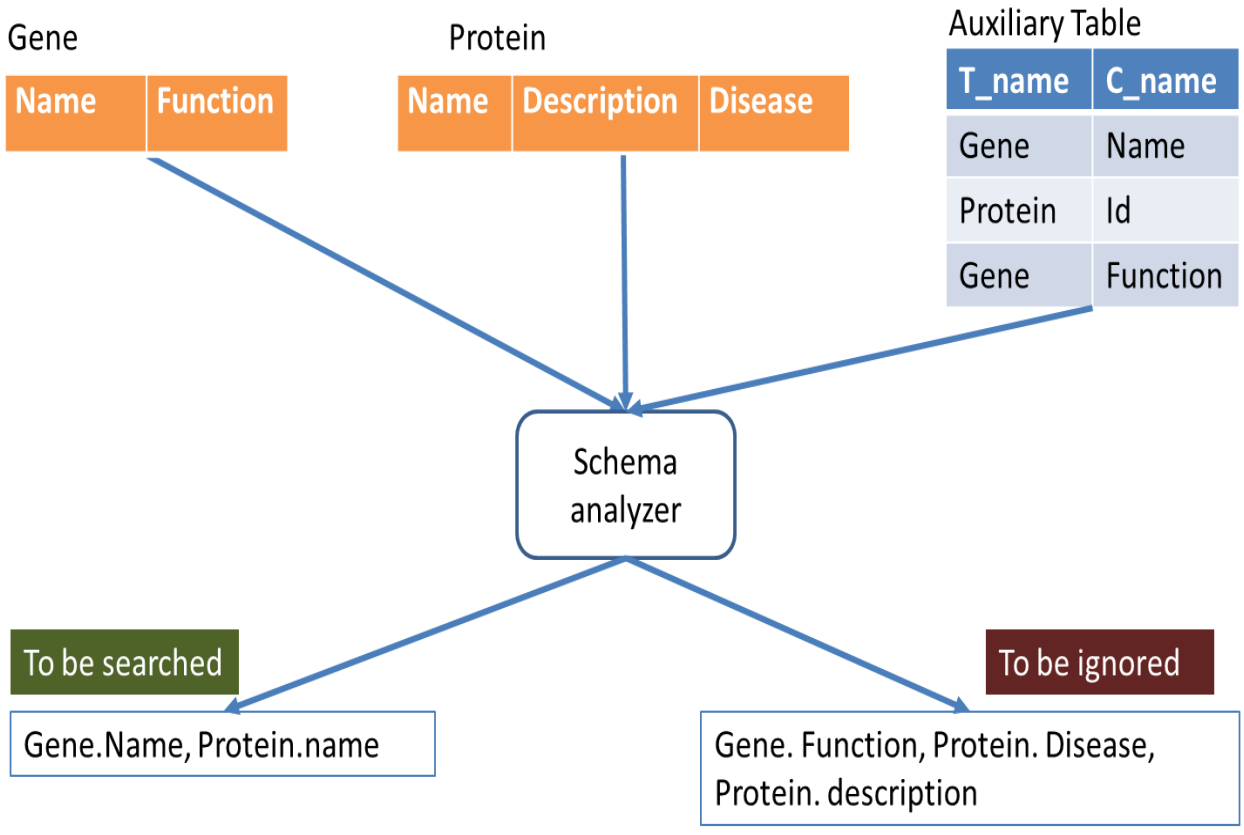


Figure 11: Schema Analyzer Example

3.2.3 Hint Recognition:

In the hint searching, the system checks if any of the table or column names where mentioned in the annotation so they can act as hints. Tsvector index the words so it can support fast querying of words by using operator '@@' and to_tsquery. tsquery is not just raw text, any more than a tsvector is. A tsquery contains search terms, which must be already-normalized lexemes, and may combine multiple terms using AND, OR, and NOT operators. For example, query 'SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');' would return true. So, we use the same query to search for each tuple specified by the database administrator that contain a table and column name. If any of them was found, then we get their position(s). In order to build the context, we build list of words that are three positions away before or after the hints.

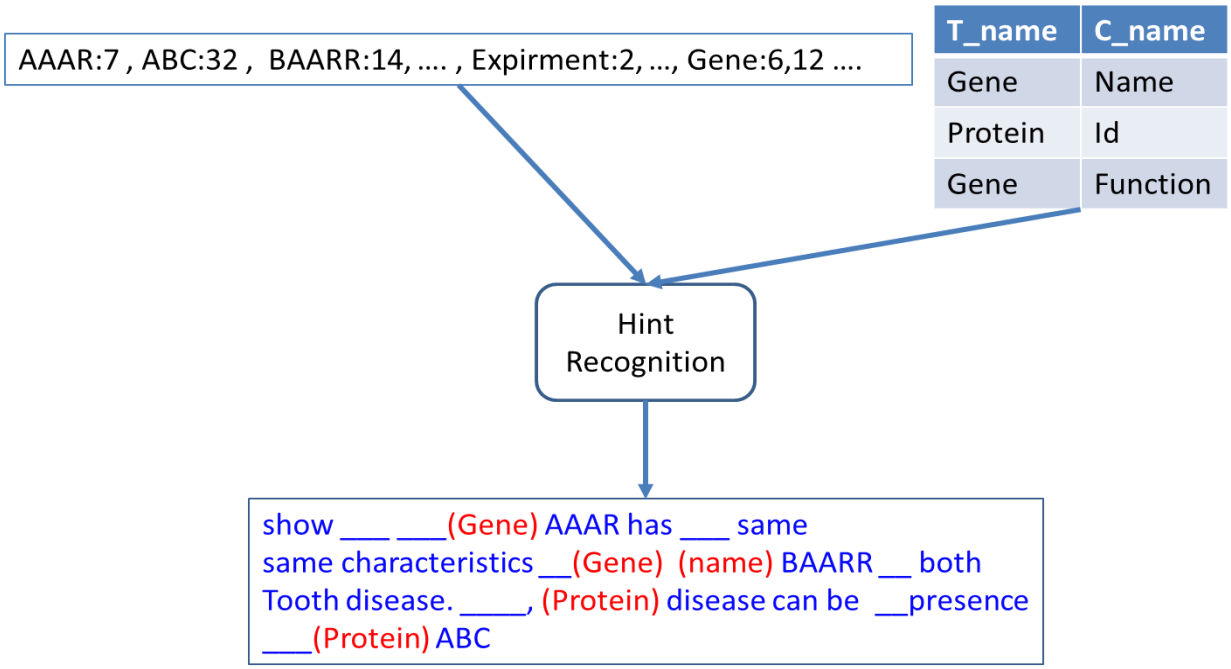


Figure 12: Hint Recognition Example

3.2.4 Construct Builder:

In this stage, the system has list of hints and their context, list of to be searched and list of to be ignored. Basically, if the hint is a table name then we start look in the context to keyword in the list of to be searched and list of to be ignored. If any context words belong to the first case (to be searched list), that means we have double hints. So, we will discard one of them in order to prevent double searching and typically we discard the table name hint. If any context words belong to the second case (to be ignored), then that means the other words in context mostly refer to data in column that the database administrator doesn't want to search. In that case, this hint and its context are discarded from the searching step.

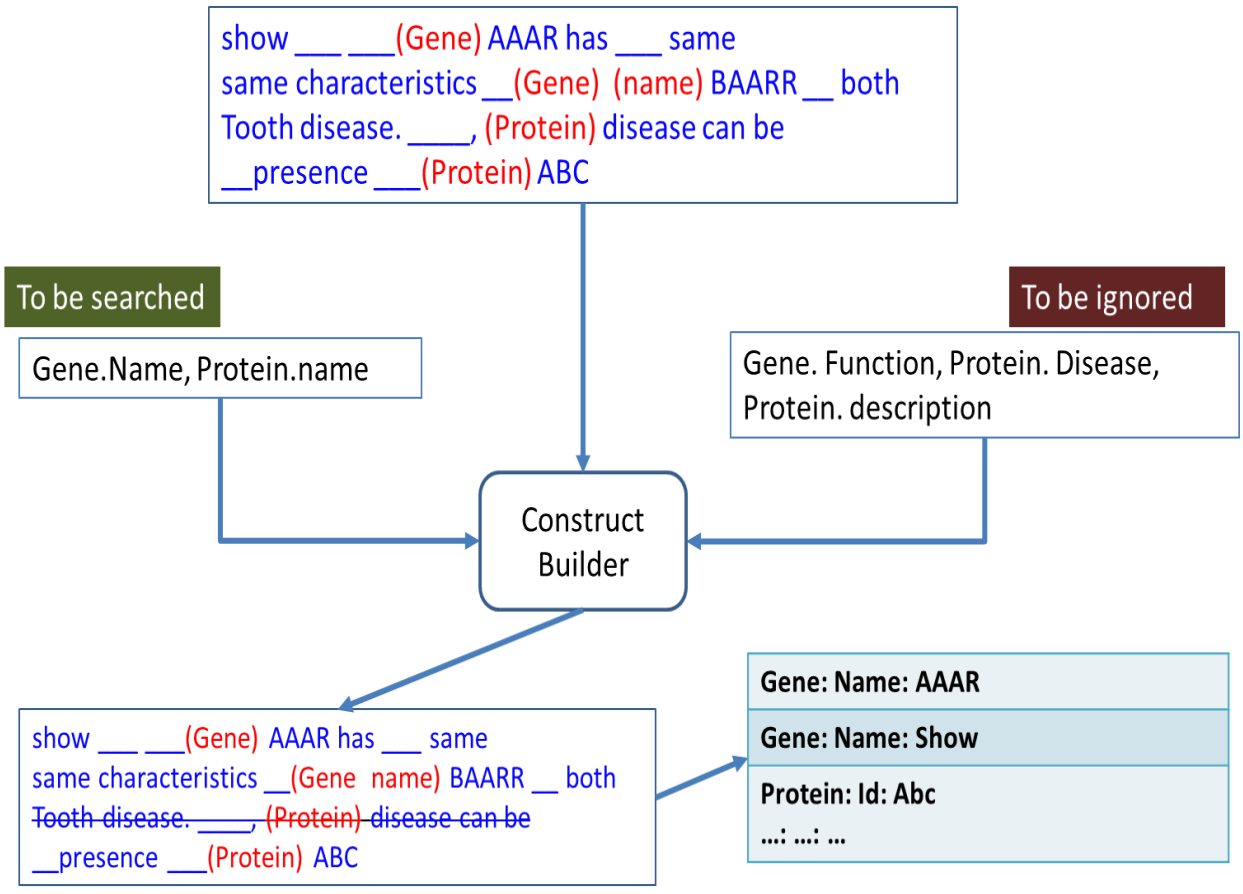


Figure 13: Construct Builder Example

3.2.5 Data Search:

Last step is to search the context words in the columns that were highlighted by the database administrator before. If the table name is used as hint then all the columns that are in the 'to be searched' list. If a column name is the hint (and belong to different tables) and no table name is mentioned in the context then context words are searched in every table.

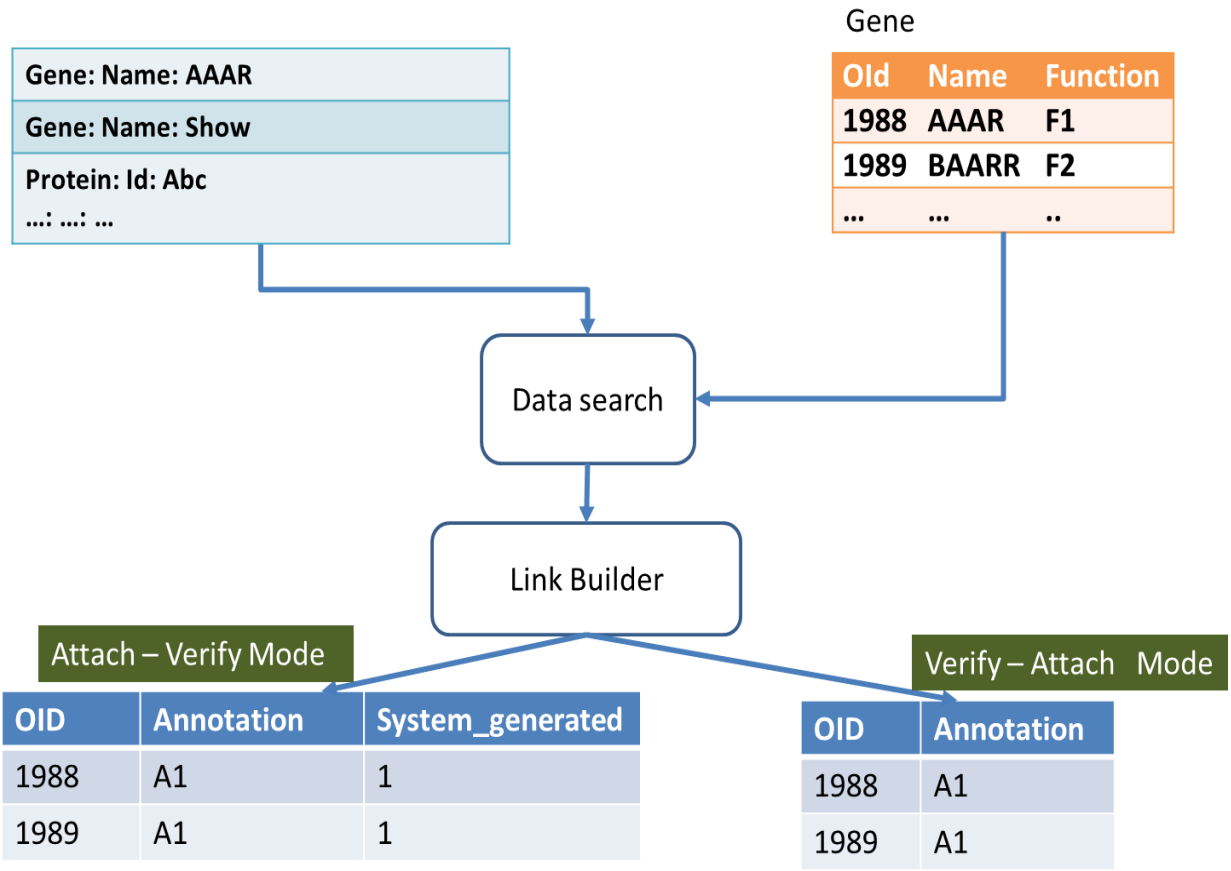


Figure 14: Data Search & Link Builder Example

3.2.6 Link Builder:

For each match found in the data search, new link is established to connect between the data and the annotation. In the first mode ‘Attach Verify’, the new link is added to the many-to-many table that join the data and the annotations with a flag that it’s a system generated link. In the second mode ‘Verify Attach’, the link is save in temporary table till the user verify it and then transform to join the annotation and the data. Eventually, the user show the generated links and verify each data-annotation connection. If the user confirms a connection, the link’s system generated flag is changed to false else if the user rejects the link then the link is deleted from the data-annotation join table or the temporary table.

3.3 Special cases:

Given the following tables:

Tb_1(C_1,C_2), Tb_2(C_1,C_3), Tb_3(C_1,C_4), C_2(C_5,C_6)

And columns to be searched Tb_1(C_2),Tb_2(C_1),C_2(C_6),Tb_3(C_1)

Tb_1

C_1	C_2

Tb_2

C_1	C_3

Tb_3

C_1	C_4

C_2

C_5	C_6

Id	Table	Column
1	Tb_1	C_2
2	Tb_2	C_1
3	C_2	C_6
4	Tb_3	C_1

1	2	3	4	Case
✓	✗	✗	✗	___ ___ Tb_1 C_2 ___ ___
✓	✗	✗	✗	___ ___ Tb_1 ___ ___
✓	✗	✓	✗	___ ___ C_2 ___ ___
✗	✓	✗	✓	___ ___ C_1 ___ ___
✗	✓	✗	✗	___ ___ Tb_2 C_1 ___ ___
✗	✓	✗	✓	___ ___ C_1 Tb_2 ___ ___
✗	✓	✗	✗	___ ___ Tb_2 ___ ___
✗	✗	✓	✗	___ ___ C_2 C_6 ___ ___
✗	✗	✗	✗	___ ___ C_2 C_5 ___ ___

Figure 15: Special cases

3.4 Summary:

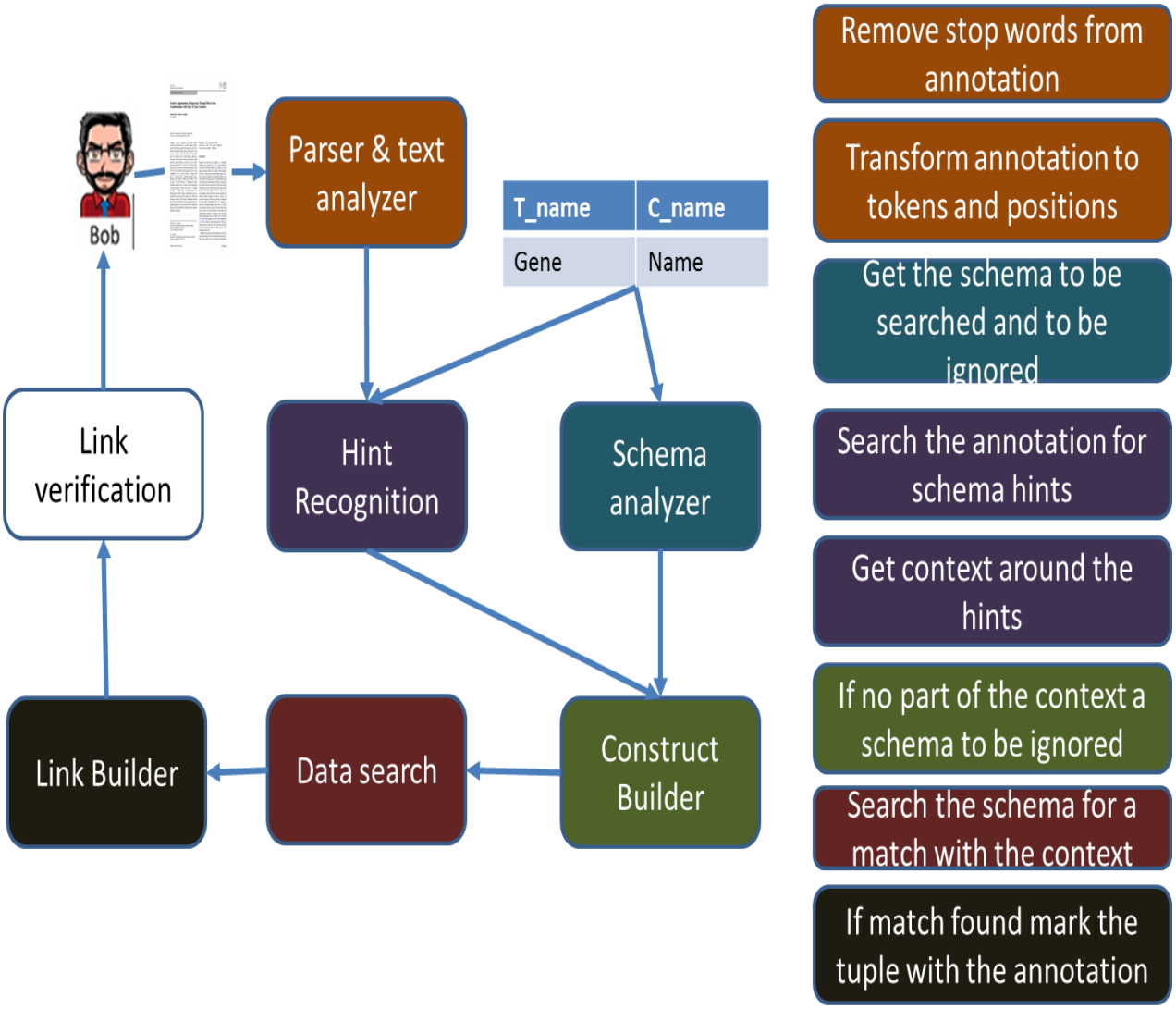


Figure 16: Proactive Annotation Management summary

The process is divided into three major phases:

Phase 1: Annotation and schema analysis

Phase 2: Hints and context extraction

Phase 3: Data-annotation search and link construction

Chapter 4: Summary-Based Annotation Management: Advanced Querying

The increasing scale and complexity of these annotations makes it very challenging for end-users and scientists to extract the hidden. In our previous work [], they initiated an effort for addressing these challenges in novel ways by proposing the "InsightNotes" system, which is a summary-based annotation management engine in relational databases. InsightNotes is based on creating concise and meaningful representations of the raw annotations, called annotation summaries. We addressed several unique challenges that arise from creating, maintaining, propagating, and zooming-in over these annotation summaries at query time, e.g., how to incrementally maintain the annotation summaries when new annotations are added, and how to extend the query engine and relational operators to manipulate the summaries in a pipelined fashion and propagate them along with the queries' answer.

We take one step further towards building an end-to-end and full-fledged summary-based annotation management engine by addressing the following challenge: Summary-Based Query Processing: That is, how to query the data records, e.g., selection, join, or ordering, based on their attached summaries. The annotation summaries have well-defined structures and properties that need to be seamlessly integrated into the query engine to enable full querying capabilities beyond just the propagation feature supported in [19].

To address these challenges and to enable seamless summary-based query processing, we introduce new query operators that operate on the annotations summaries attached to each tuple instead of its data attributes. For example, referring to Figure 17, among the many summaries attached to the data tuple, we may want to report only the summaries of type classifier, select the data tuples having more than two refuting annotations, i.e., referencing RefuteApprove, (A Summary-Based Selection operator), or ordering the output tuples according to the number of provenance annotations attached

to each tuple, i.e., referencing TextSummary2, (A Summary-Based Sort operator). To achieve efficient execution of these summary-based operators, we propose new indexing structures on top of the annotation summaries and integrate them within the query optimizer and execution engine. The key contributions in our system are:

Introducing new summary-based query operators that operate on the annotations summaries attached to each tuple instead of its data attributes, e.g., summary-based filter, selection, join, and ordering. The new operators enable querying and manipulating the annotation summaries as first-class citizens at query time beyond just propagating them along with queries' answers.

Proposing new indexing structures over the annotation summaries for efficient query execution. Although the annotation summaries and the base data are stored in separate tables, the indexing scheme will allow direct access from the summaries to their corresponding data tuples without the need for expensive join operations. We extend the query optimizer and execution engine to integrate these indexes into their cost model and execution plans, respectively.

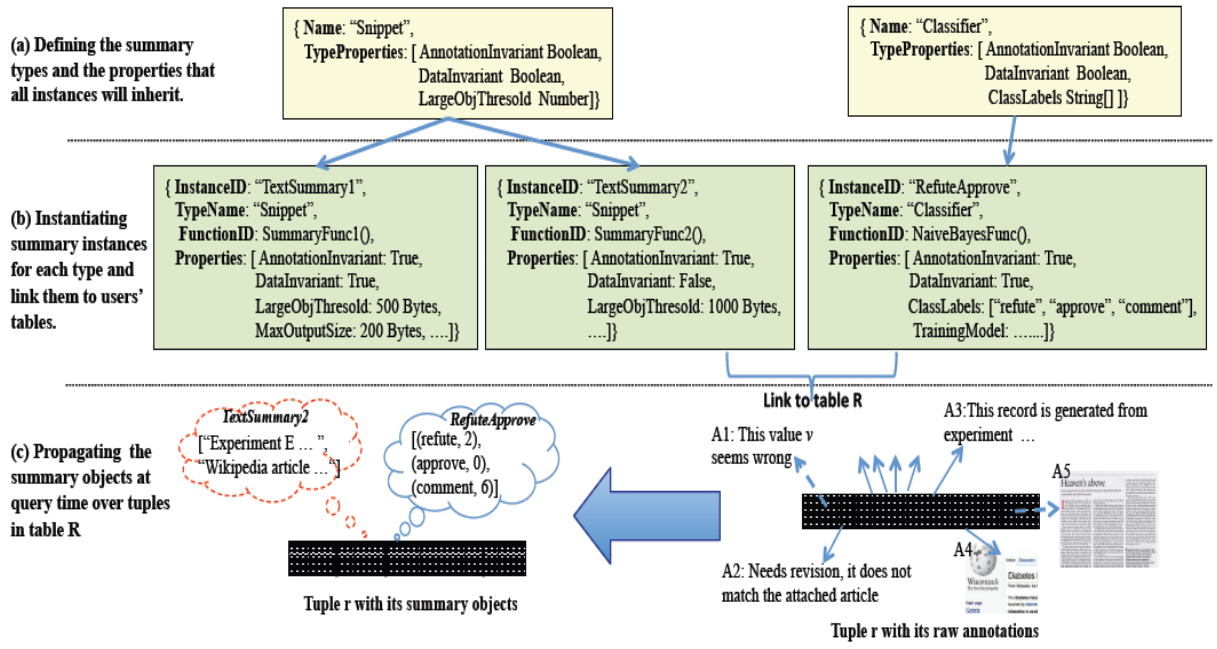


Figure 17: InsightNotes Summary Structure

4.1 SUMMARY-BASED QUERY FUNCTIONS AND OPERATORS

The summary objects attached to each data tuple carry useful mined information from the raw annotations that is organized in a structured format, i.e., the Rep array within each summary object. Therefore, it is very beneficial to end-users and applications to enable full-fledged query processing over these summary objects, e.g., filtering and reporting only specific summary types, applying predicates and selection/join based on the attached summaries, and ordering data tuples based on values within the summary objects. However, without a system's support, querying and manipulating the summary objects—beyond propagation—will not be feasible. First, because there are no interfaces or functions provided to end users to manipulate the annotation summaries. And second, because there are no system-level optimizations or summary-specific operators that operate on the annotation summaries as first-class citizens in the database. In this section, we overcome these limitations by introducing new summary-based manipulation functions and relational operators.

4.1.1 Summary-Based Manipulation Functions

A data tuple in the query pipeline has the following schema:

$r =< a_1; a_2;..; a_n; \{s_1; s_2;..; s_k\}>$

where $a_1; a_2; ..; a_n$ are the data values of r, and $s_1; s_2; ..; s_k$ are the summary objects attached to r. The following set of manipulation functions are defined to operate on the summary set as well as the objects within the set.

4.1.1.1 Summary Set Functions:

We introduce a special variable (attribute) "$" for each data tuple that represents the set of summary objects attached to this tuple, i.e., r.$ represents the set of summary objects attached to r. Then, several interface functions are defined over the $ variable, which include:

Int $.getSize(): Returns the number of summary objects within the set. For example, referring to relation R in Figure 17 r1:$:getSize() returns value two.

SummaryObj $.getSummaryObject(String instanceName): This function takes a summary instance name and returns the summary object of that instance. For example,

r1:$:getSummaryObject(`RefuteApprove') returns the classifier summary object with that instance name.

SummaryObj $.getSummaryObject (Int i): This function takes a position within the summary set and returns the summary object at that position. Since the objects in the set do not follow a pre-defined order, this function is more useful when used within UDFs that, for example, take the summary set as input and loops over its summary objects to perform a certain functionality.

4.1.1.2 Classifier Type Functions:

For a summary object O of type Classifier, the following functions are defined:

Int O.getSize(): Returns the number of class labels within this object. For example, the RefuteApprove and ProvenanceQuestion classifier objects attached to r1 in Figure have four and three class labels, respectively.

String O.getLabelName(Int i): Returns the class label at position i, i.e., Rep[i].classLabel. The order among the class labels is pre-defined based on the order specified by users while creating the classifier summary instance in the system.

Int O.getLabelValue([Int i | String label): This function takes either an index i or a class label label and returns the corresponding value, i.e., Rep[i].annotationCnt (for input i), or Rep[j].annotationCnt, where Rep[j].classLabel = label (for input label).

4.1.1.3 Snippet Type Functions:

For a summary object O of type Snippet, the following functions are defined:

Int O.getSize(): Returns the number of snippets within this object. For example, the number of snippets in the TextSummary object attached to tuple r1 in Figure 17 is two.

String O.getSnippet(Int i): Returns the snippet value at position i. The order among the snippets is arbitrary and does not follow a pre-defined order.

Boolean O.contain(String kw1 [, String kw2, ...]): Returns True if all of the given keywords kw1; kw2; .. are contained within any one of O's snippets or the raw annotations. The system expands the search to the raw annotations to avoid producing false negative results.

Boolean O.fullSearch(String kw1 [, String kw2, ...]): Returns True if all of the given keywords kw1; kw2; .. are contained within the union of O's snippets or the raw annotations. In this function, the keywords may span multiple annotations that are summarized by the summary object O.

4.1.1.4 Cluster Type Functions:

For a summary object O of type Cluster, the following functions are defined:

Int O.getSize(): Returns the number of clusters within this object.

String O.getRepresentative(Int i): Returns the representative annotation of cluster i. The order among the clusters is arbitrary and does not follow a pre-defined order.

String O.getCount(Int i): Returns the count of annotations in cluster i.

Moreover, each summary object O has additional functions O.getSummaryType() and O.getSummaryName(). The former function returns the type of the summary object as either "Classifier", "Snippet", or "Cluster", while the latter returns the summary instance name of that object. Internally in InsightNotes system—which uses PostgreSQL as its underlying DBMS— the summary objects are defined as composite data types on top of which the manipulation functions presented above are defined. Users' can create UDFs that leverage these basic interfaces to perform other complex functionalities.

4.1.2 Summary-Based Relational Operators

We introduce several summary-based operators that operate on the summary objects attached to each tuple instead of its data values. These operators include:

4.1.2.1 Filter Operator (Fp(R)):

The filter operator takes a set of summary-based predicates (conditions) p, and applies p over each summary object attached to r ∈ R. The operator returns r along with only the summary objects satisfying p. The is, for each tuple r with schema r =< a_1; a_2; ..; a_n; {s_1; s_2; ..; s_k} >, the operator output will be:

$F_p(r) = \{ r` =< a_1; a_2; ..; a_n; \{s_i; ..\} > | p(s_i) = True \}$

For example, the predicate “getSummaryName() =‘RefuteApprove’ ” returns with each tuple only the summary object having instance name RefuteApprove. In contrast, the predicate “getSummaryType() = ‘Snippet’ & getSize() >= 2” returns with each tuple any summary object of type Snippet that contains two or more snippet values. Notice that the predicates p leverage the built-in functions developed on the summary objects.

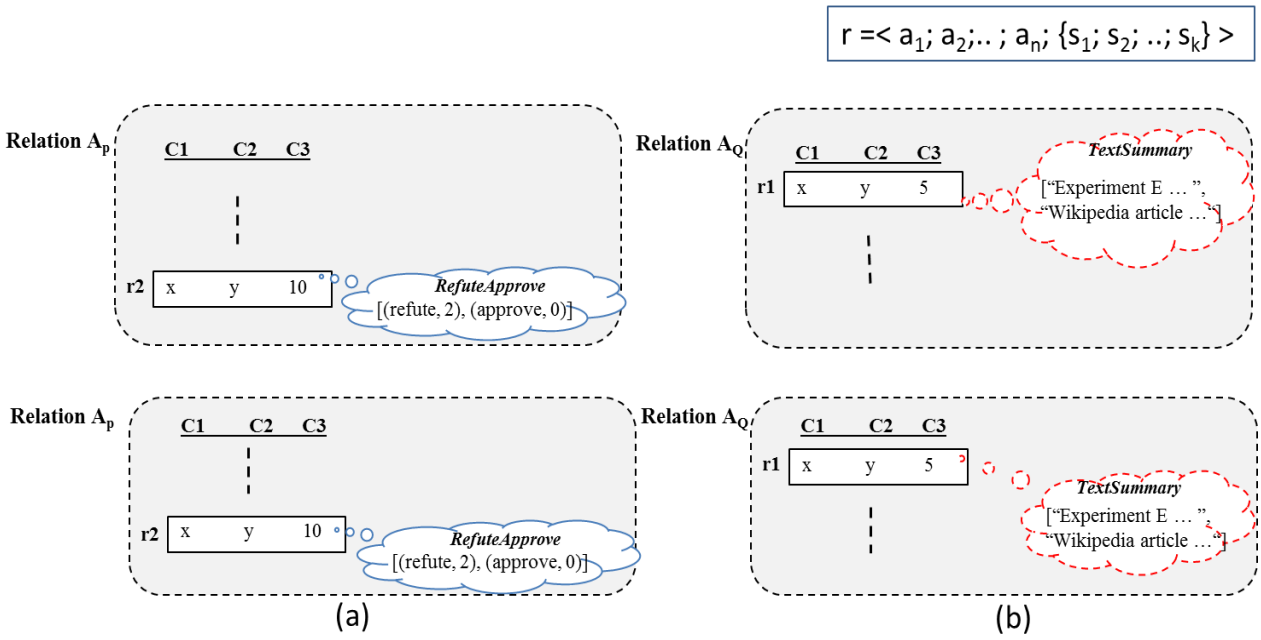


Figure 18: Summary Filter example

4.1.2.2 Selection Operator (Sp(R)):

The summary-based selection operator takes a set of summary-based predicates (conditions) p, and returns only the data tuples r ∈ R having summary objects satisfying p. Otherwise, r is dropped. The algebraic expression of the operator is as follows:

$S_p(R) = \{r \in R ; r =< a_1; a_2; ..; a_n; \{s1; s_2; :::; s_k\} > | \ p(r:\$) = True\}$

The summary-based predicates may range from black-box UDFs that take r:$ as a parameter and returns a Boolean value. Or they can be explicit predicates that the system can reason about and optimize. For example, the predicate "r.$.getSummaryObject('ProvenanceQuestion').getLabelValue('Provenance') = 0" returns only the data tuples in R that have no provenance records attached to them (according to the ProvenanceQuestion classifier summary object). While the predicate

"r.$.getSummaryObject('TextSummary').fullSearch ('Wikipedia', 'hormone')" returns only the data tuples in R whose annotations have the specified keywords. These explicit predicates can be efficiently executed using the summary based indexing schemes and query optimizations.

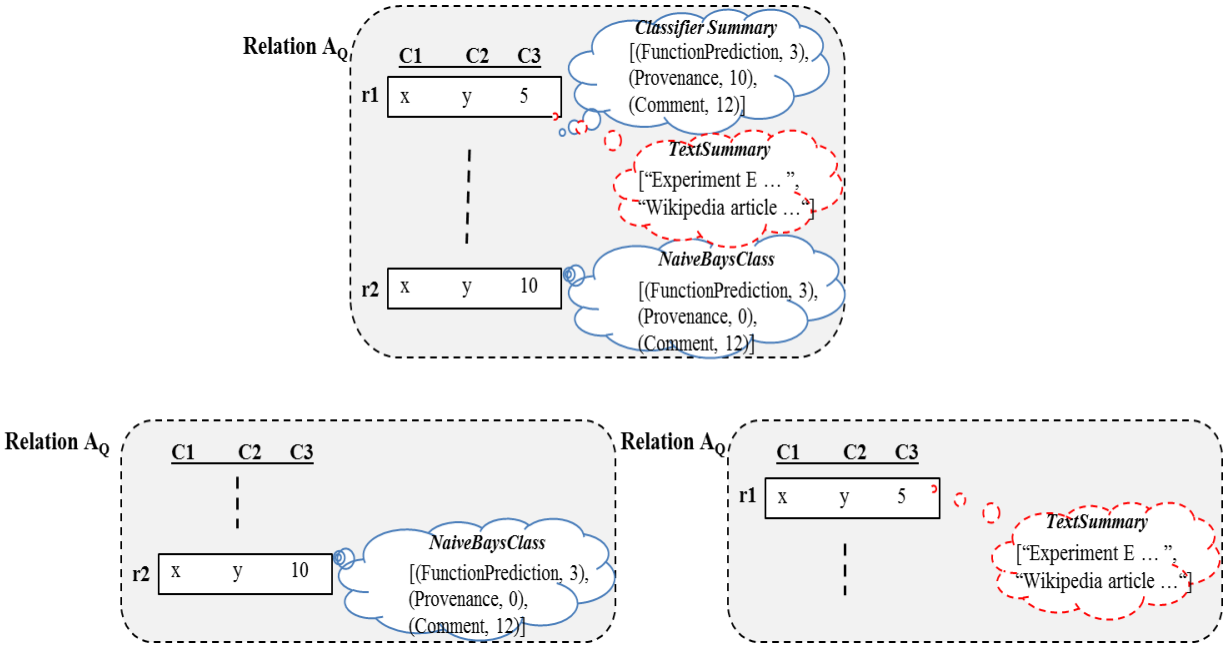


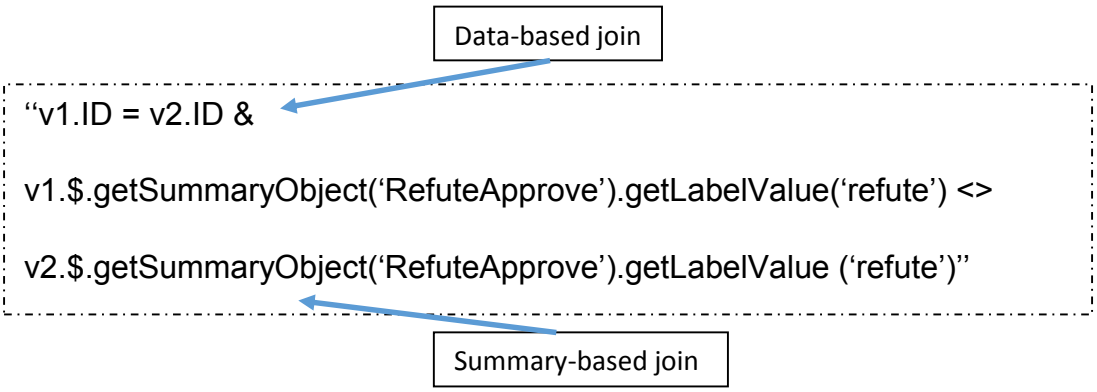
Figure 19: Summary Selection example

4.1.2.3 Join Operator ($J_p(R; S)$):

The summary-based join operator joins two input tuples r ∈ R and s ∈ S iff the summary-based predicates p evaluate to True over r:$ and s:$. The algebraic expression of the operator is as follows:

$J_p(R; S) = \{< r; s >; \text{where } r \in R \ \& \ s \in S \mid p(r:\$; s:\$) = True\}$

For example, if a dataset have two versions V1 (after Revision1) and V2 (after revision 2), then to report the data tuples whose number of disapproving annotations has changed between the two revisions, we may use an expression that combines both the data and summary-based joins as follow:



As will be discuss, based on the available indexes and statistics, the query optimizer may decide to join based on the data values (R ⋈ S), and then uses a summary-based selection operator (S). Alternatively, it may join based on summaries (R ⋈ S), and then uses a standard selection operator (ρ).

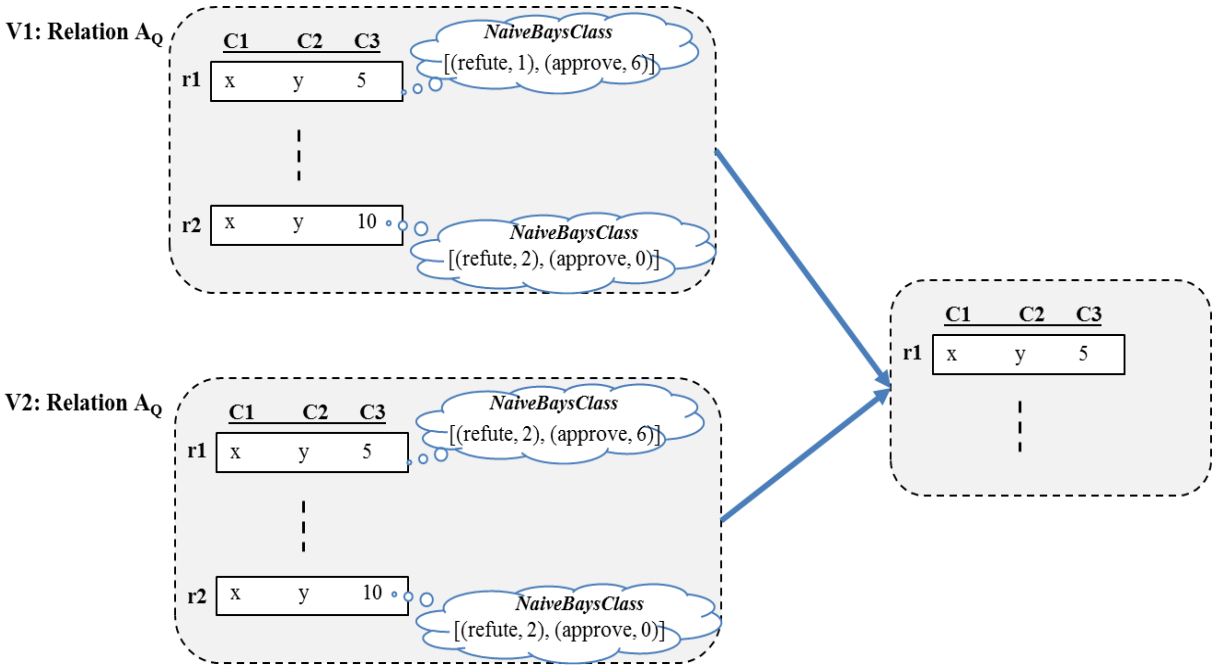


Figure 20: Summary Join example

4.1.2.4 Sort Operator O(f,direction(R)):

The summary-based sort operator orders the tuples in R according to the summary based function f(r.$). Function f must return values of a data type with full ordering, e.g., number, string, and Boolean types. For example tuples can be ordered by the number of summary objects attached to them, e.g., “r.$.getSize()”, or by the number of approving annotation, e.g., “r.$.getSummaryObject(‘RefuteApprove’). getLabelValue(‘approve’)”.

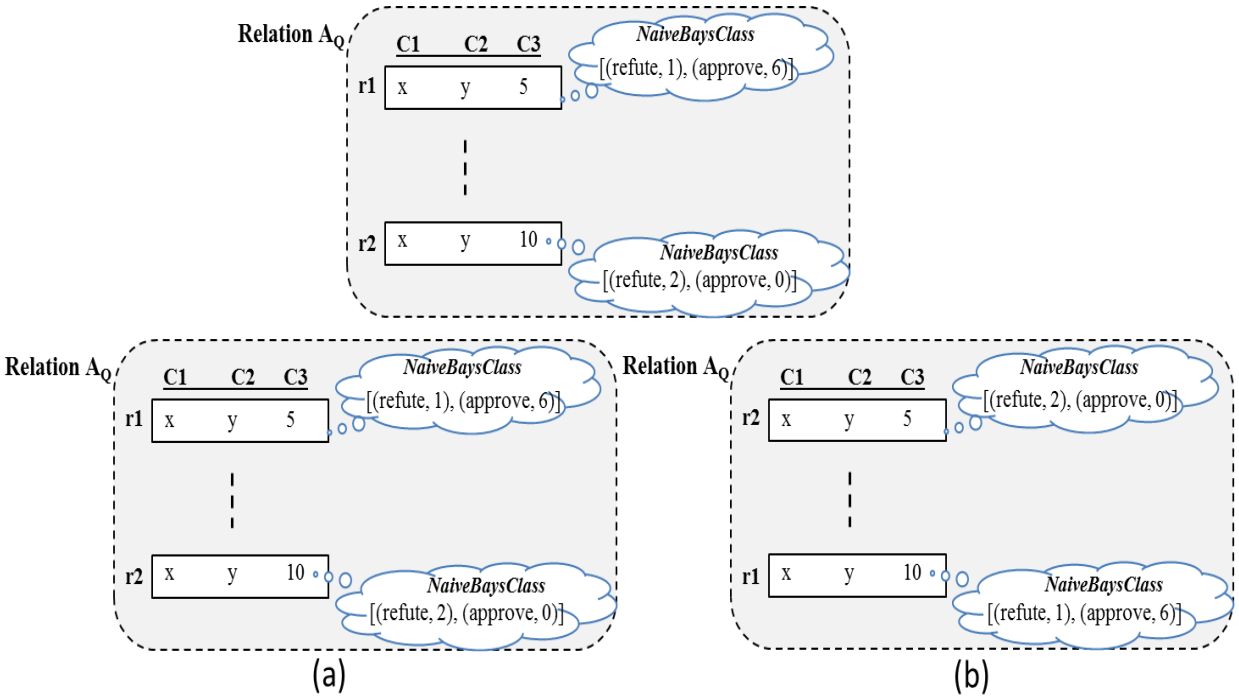


Figure 21: Summary sort example

4.2 SUMMARY-BASED INDEX SCHEME

To enable efficient execution of the summary-based relational operators, we propose a summary-based indexing scheme over the annotation summary objects. To create a summary-based index, we extended the mechanism that enables the database

admins to link summary instances with users' relations to also define whether or not that instance is indexable. This mechanism uses the Alter Table command as follows:

Alter Table <tableName>

[Add [Indexable] | Drop] <InstanceName>;

When adding a summary instance SI over table R, the optional clause Indexable will be used to inform the system to build an index on SI's summary objects created over R's tuples. In that case, predicates over these summary objects can be efficiently handled. Otherwise, no indexes will be created. For example, the following command: "Alter Table R Add Indexable RefuteApprove" indicates that summary instance RefuteApprove is linked to relation R, and the DB admin specifies indexing the resulted summary objects. In our work, we focus on indexing the summary objects of type Classifier only. Whereas, for the Cluster type, users usually do not know what's inside each cluster, and thus it is less likely to have predicates on these summary objects. Developing an efficient indexing scheme for the summary objects is a challenging task because the traditional indexing structures, e.g., B-Tree and GiST, will not be effective for two reasons: First, unlike primitive attributes, e.g., number, string, or date, summary objects have complex structures. For example, for a Classifier-type object, its Rep array consists of an array of pairs consisting of (String classLabel, Integer AnnotationCnt) . Therefore, predicates in the form of "classLabel<op> constant", where op is any of the comparison operators, e.g., = or <, would suggest breaking the summary objects into their primitive components to be easier for indexing. However, by doing so, we will significantly penalize the propagation operation—which is a core operation in the system— since it will require joining many primitive components (instead of a single summary object) with their data tuples, and then re-building the summary objects each time. Second, the annotation summaries are typically stored in system tables separate from the users' tables. And thus, expensive join operations will be needed to retrieve the data tuples satisfying some predicates on their summary objects. To overcome these two limitations, we propose a variant of the B-Tree for indexing the Classifier summary objects. This variant will retain the same storage scheme used for optimizing the propagation of summary objects at

query time, eliminate the need for expensive join operations when predicates are applied over summary objects, and encounter minimal storage overhead.

4.2.1 Summary-BTree Index

The Summary-BTree index is a variant of the B-Tree index structure for indexing the Classifier-type summary objects (Refer to Figure 22). The target predicates for optimizations are in the form of: “classLabel <op> constant”, where op is any of the comparison operators in {=, <, >, ≤, ≥}.

4.2.2 Summary-BTree Index Structure:

The structure of the index is depicted in Figure 22. Given the annotated user’s relation R with three summary instances linked to it, i.e., the Classifier instances “RefuteApprove” and “ProvenanceQuestion”. The first step in the figure is the default step in the InsightNotes system [19], where the summary objects attached to each data tuple are stored in a representation optimized for efficient propagation at query time. That is, the summary objects are stored in their complex structures that will propagate to end-users along with queries’ answers. For each indexable Classifier summary instance, a Summary- BTree index will be created on the corresponding column. Referring to our example, assume that the RefuteApprove summary instance is defined as indexable over relation R, then the system will automatically create a Summary-BTree index over column RefuteApprove in the summary storage table. Building the index will involve three steps as illustrated in the figure, which are: Itemization, Indexing, and Back Referencing, described as follows:

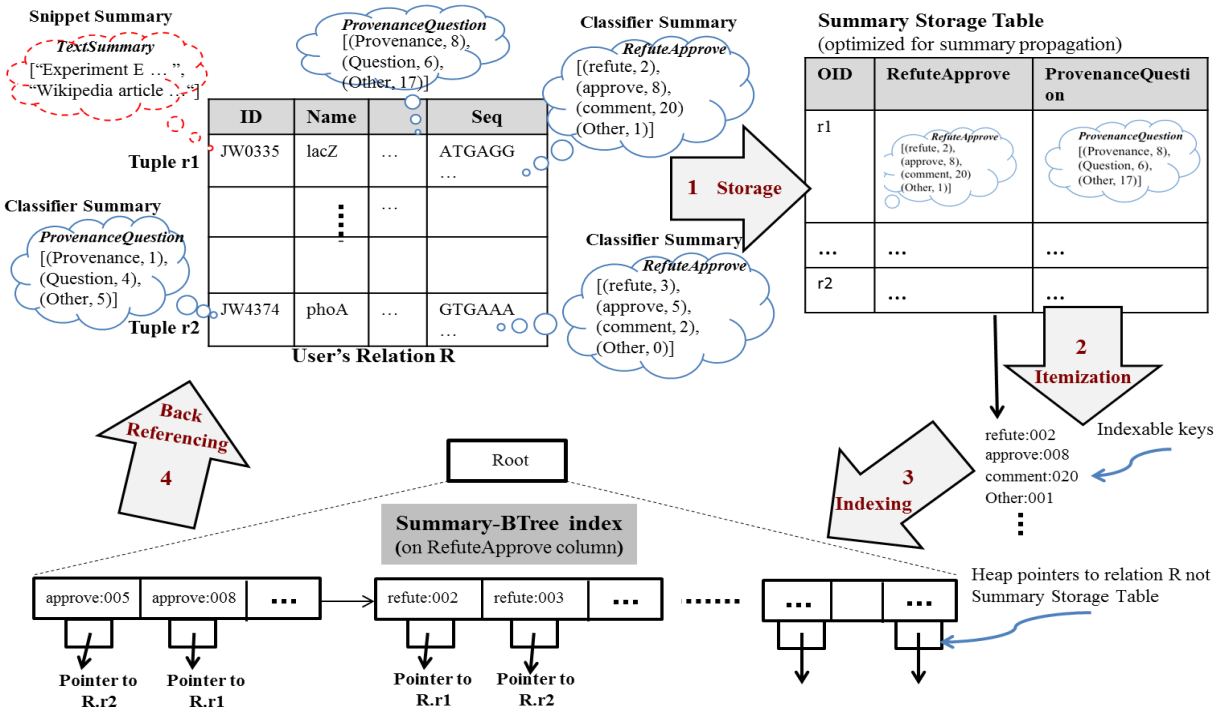


Figure 22: Summary B-tree

Itemization:

To index a Classifier summary object, the Rep array within the object will be itemized by converting the array elements (String classLabel, Integer AnnotationCnt) to a sequence of text values in the form of "classLable:AnnotationCnt" as illustrated in Figure 22 Step 2. The AnnotationCnt field when converted into string, it will have 3-digit format to preserve the order among the integer values even after the conversion to string values. We assume that no class label will have more than 999 annotations. For example, the RefuteApprove summary object attached to tuple r_1 will be itemized to the sequence of values depicted in Figure 22 Step 2.

Indexing:

Each of the text values generated from the Itemization step will be inserted into the Summary-BTree index. The index will follow the same structure and operations of the

standard B-Tree index. For example, the indexed values will all appear in the leaf nodes of the index tree sorted alphabetically. The only exception will be in the pointers that these values will point to, which are called back references described next.

Back Referencing:

A key trick in the Summary-BTree index is that the leaf nodes will point back to the annotated data tuples in the user's relation instead of pointing back to the Summary Storage Table. Recall that apart from the user's relations, all other summary tables and related indexes are totally transparent from (and not directly query-able by) the end-users. This gives us the opportunity to optimize these structures for their targeted operations. Referring to Figure 22, the indexed values "approve:005" and "approve:008" will point back to their data tuples R:r_2 and R:r_1, respectively. These back pointers will be created and maintained under the different operations as described in sequel.

Summary-BTree Index Operations:

The index structure needs to be maintained under two basic operations: (1) Adding a new annotation on relation R, which will either insert a new tuple in the Summary Storage table (if that is the first annotation on the data tuple), or update an existing tuple in the Summary Storage table. And (2) Deleting a data tuple from relation R, which should delete all summary objects associated with that tuple. No other operations will affect the index4. To enable the back referencing mechanism, we developed a function inside the database engine (on the "Relation" structure) that takes the unique identifier of a tuple (OID) and returns its storage location on disk. We then, created an external SQL interface, called diskTupleLoc(), for seamless invocation of this function using the standard SQL Select statement, e.g., the query "Select diskTupleLoc(OID) From R Where OID = r1;" would return the disk location of tuple r1 in a text representation.

Adding Annotation-Insertion:

Adding a new annotation on an un-annotated tuple in R will result in inserting a new tuple in the Summary Storage table. Then, to insert the indexed values in the Summary-BTree, the system will itemize the Rep array in the summary object, retrieve the disk location of the data tuple (using diskTupleLoc() function), and associate this location as an auxiliary information to the indexed values (keys). The Summary-BTree index will then insert these keys by following the standard B-Tree processing with the exception of using the auxiliary information as the data pointers instead of pointing to the tuples in the Summary Storage table.

Adding Annotation-Update:

Adding a new annotation on an already-annotated tuple in R will result in updating the corresponding summary objects in the Summary Storage table. For example, if a new annotation is added disapproving the content of tuple $R.r_1$, then in the RefuteApprove summary object, the first entry will be (refute, 3) instead of (refute, 2) . To update the index, the system will first trigger a deletion and then re- insertion only for the modified class label not to all labels within the object. For example, “refute:002” will be deleted and “refute:003” will be inserted, and both keys will be augmented with r_1’s disk location for correct index modification.

Deleting Tuple:

The deletion operation follows the same mechanisms as described above. If, for example, tuple $R:r_1$ is deleted, then the corresponding entry in Summary Storage table will be deleted. This will trigger a sequence of deletions to the Summary-BTree index corresponding to the itemized keys generated from the deleted object.

4.3 SUMMARY-AWARE QUERY-TIME OPTIMIZATIONS

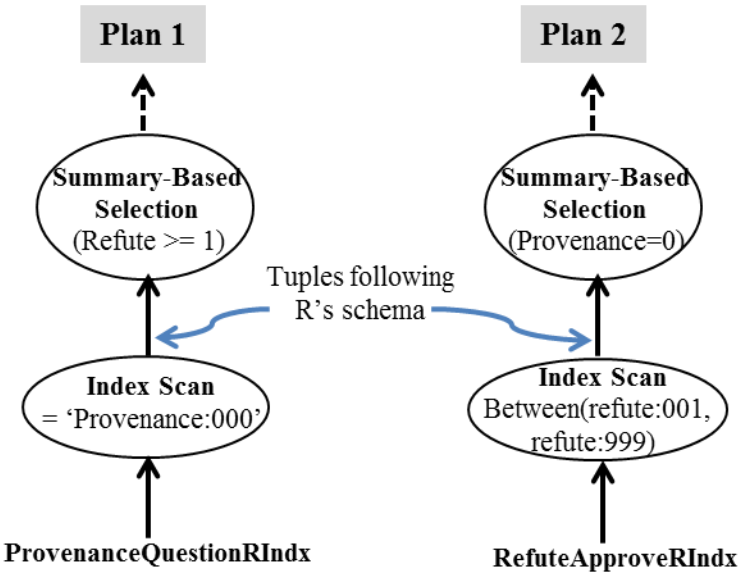
Based on the summary-based indexes, we extended the query optimizer to create efficient query plans that make use of these indexes and their inherent properties, i.e., ordering properties. The following scenario will highlight these optimizations.

4.3.1 Index Scan Based on Selectivity:

In Figure 23, we demonstrate a query that selects the data tuples from a user's relation R satisfying two summary-based selection predicates on the RefuteApprove and ProvenanceQuestion classifier summary instances. Assuming that both instances have corresponding indexes, then the query optimizer has two possible query plans to select from as illustrated in Figure 23. Notice that the index scan operator is not modified in any way, i.e., it operates as any standard index scan operator that takes a key (or key range) and returns the corresponding tuples. We only set the output schema corresponding to the tuples produced from these operators to R's schema. In Plan 1, the Summary-BTree index on the ProvenanceQuestion summary instance is used (using key "Provenance:000"), and then to satisfy the second predicate, a summary-based selection operator is added to the query plan. In contrast, in Plan 2 the index on the RefuteApprove summary instance (using a key range between "refute:001" and "refute:999"), and then add summary-based selection operator for the other predicate. The construction of the index key is part of the comparison operators, e.g., = or ≥ in our query, defined on the summary objects. In order to decide which plan is cheaper, the system needs to estimate the selectivity of each predicate. To enable such decision, we maintain a set of histograms for each indexed summary instance, i.e., for a summary instance with k class labels, the system maintains k histograms; one for each label. For each bucket in a histogram, the total number of annotations falling in that bucket is maintained, and we assume uniform distribution within each bucket. Therefore, in query Q1 in Figure 23, the two buckets corresponding to labels refute and Provenance will be checked to estimate their selectivity. And then, the most selective one will be used for the index scan operator.

Q1: Select tuples from R that have no provenance records and 1 or more comments disapproving their content.

```
Select  *
From    R
Where   ….refute >= 1
And     ….Provenance = 0;
```



(a) Index Scan Based on Selectivity

Figure 23: Summary optimization in Selectivity

Chapter 5: Performance Evaluation

5.1 Experiment Setup and Methodology

We run our experiments on virtual machine that has two CPUs and a gigabyte of memory. Hard drive a 10 gigabytes, two gigabytes of which are dedicated to swap space. The Virtual machine is only as quick as the hardware it runs on though.

Synthetic Data We also implemented a data generator to create dataset containing 11,100,000 objects produced by a data generator. This dataset is composed of three tables with sizes of 10, 1, 0.1 millions respectively.

We use real dataset for summaries of annotations. The annotation size is the raw size and varies from 9 millions to 450,000. For the summaries, each type is around 12% of the raw annotations.

5.2 User-Centric Annotation Propagation

We measure the response time for each propagation type, with different filtering. We vary the number of tuples and number of annotations attached to each tuple. Default represents the propagation of all annotations attached to the tuples plus the rank of the annotation without sorting as shown in Figures 24, 25, 26.

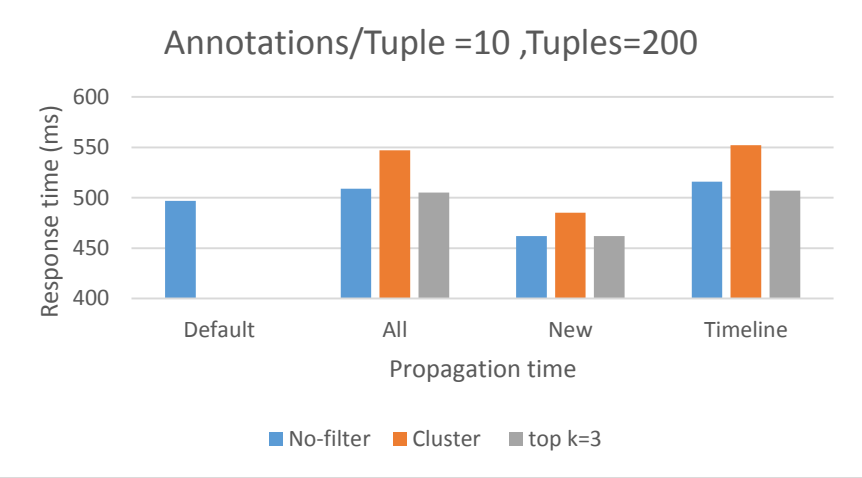


Figure 24: User-Centric Annotation Propagation Experiment 1

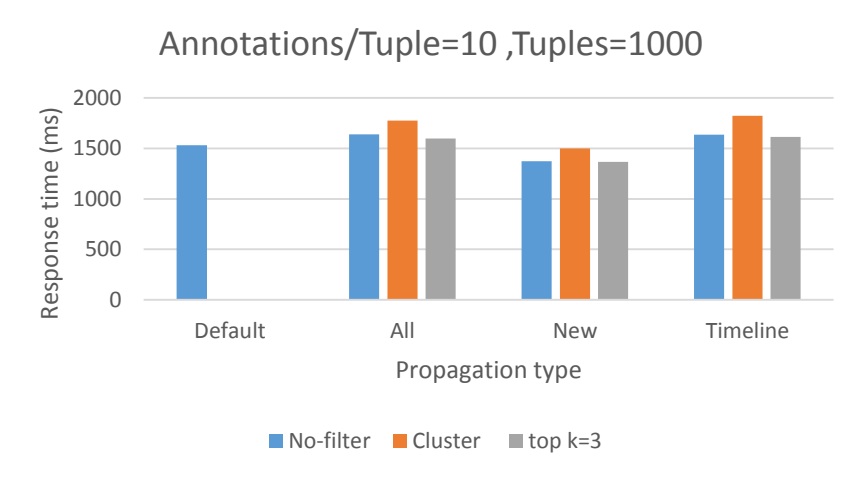


Figure 25: User-Centric Annotation Propagation Experiment 2

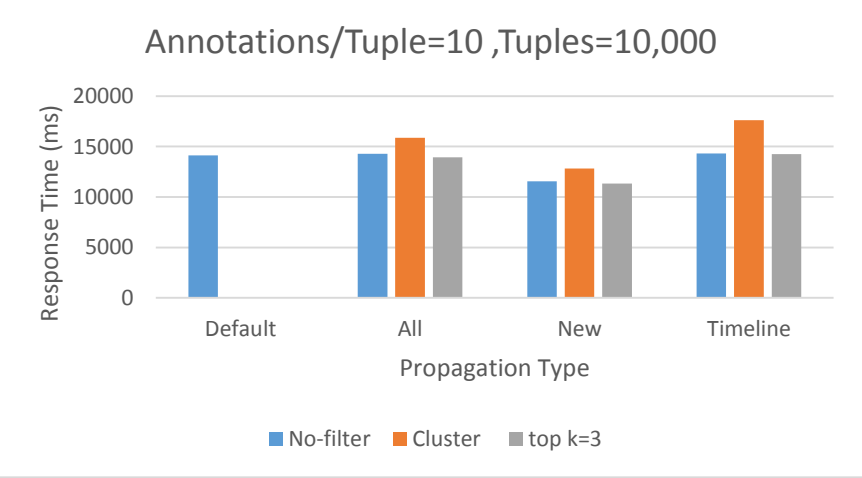


Figure 26: User-Centric Annotation Propagation Experiment 3

However, if the number of tuples increases the response time increases as well. Basically, each tuple is blocked until all the annotations attached to the tuple has arrived and processed. After the processing and sorting of the annotations according their ranks, then the tuples start to be propagated as shown in Figures 27, 28, 29.

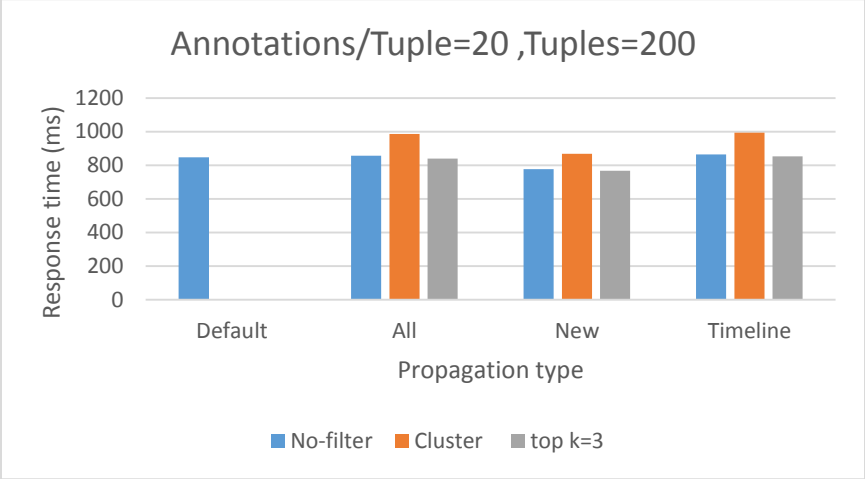


Figure 27: User-Centric Annotation Propagation Experiment 4

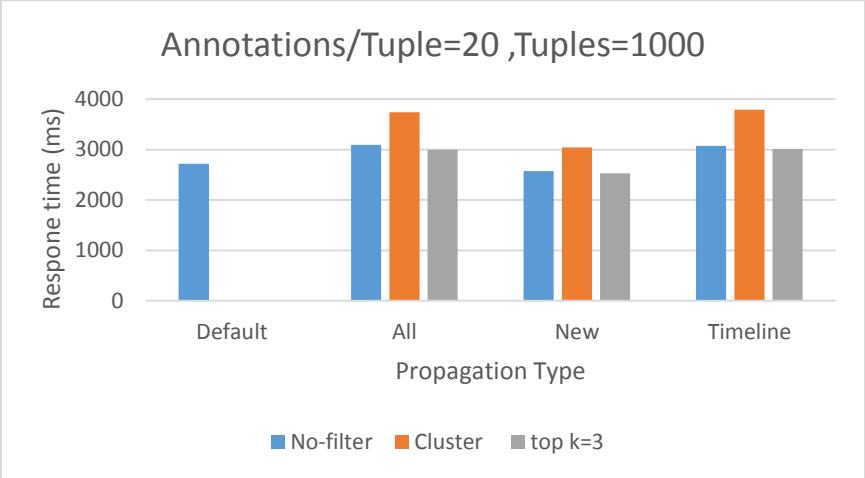


Figure 28: User-Centric Annotation Propagation Experiment 5

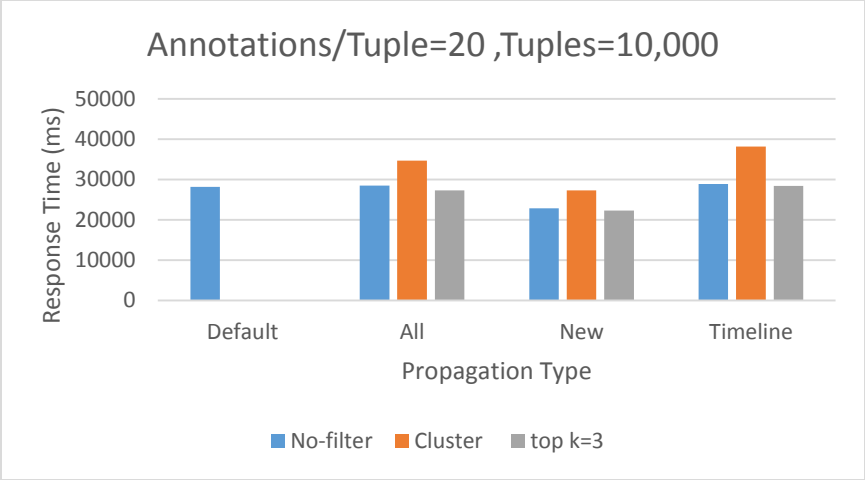


Figure 29: User-Centric Annotation Propagation Experiment 6

In Figures 30, 31, 32, as the number of annotations increases, the response time increases. The reason for that, the number of annotation to be sorted per tuple increases hence, it takes more time for each tuple and my summing processing time for each tuple the overall response time increases. We can see from the experiments that the clustering filtering cost much more than the Top K filtering, as in order to build cluster you need to compare pair annotations until you join an existing cluster or build a new one. Top K filtering is almost the same as no filter and that's because all the sorting and processing of annotations is done first and the last step is to choose the Top K of the sorted list of annotations.

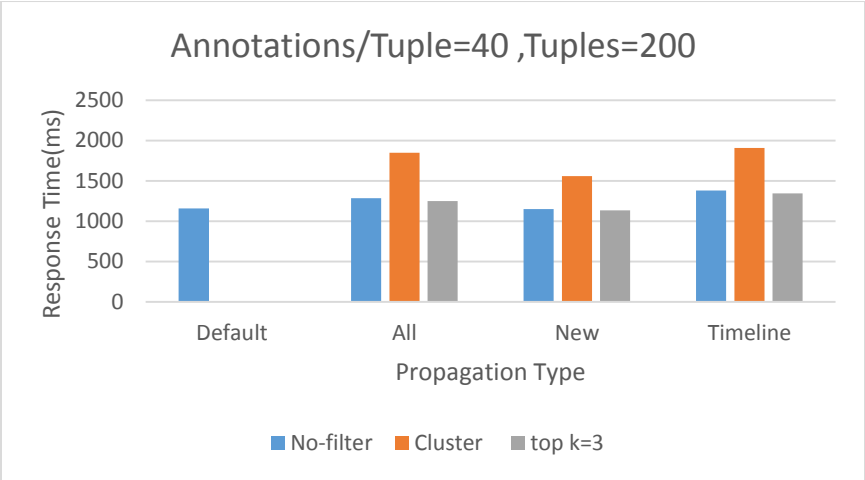


Figure 30: User-Centric Annotation Propagation Experiment 7

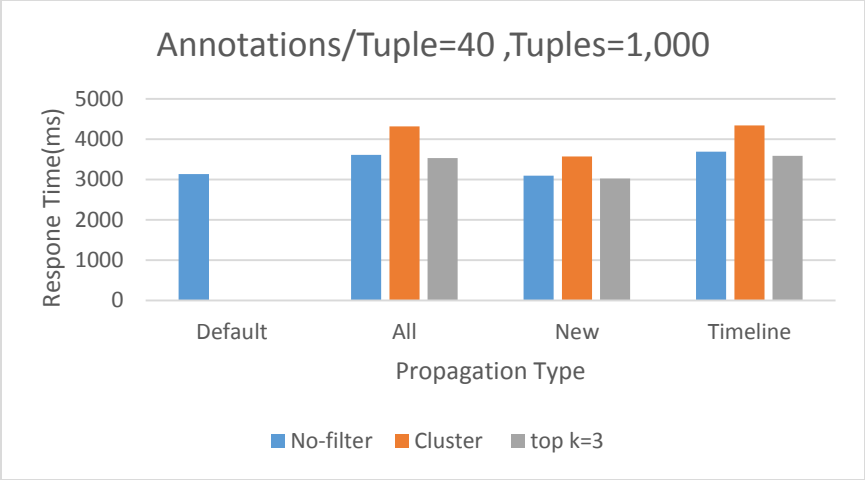


Figure 31: User-Centric Annotation Propagation Experiment 8

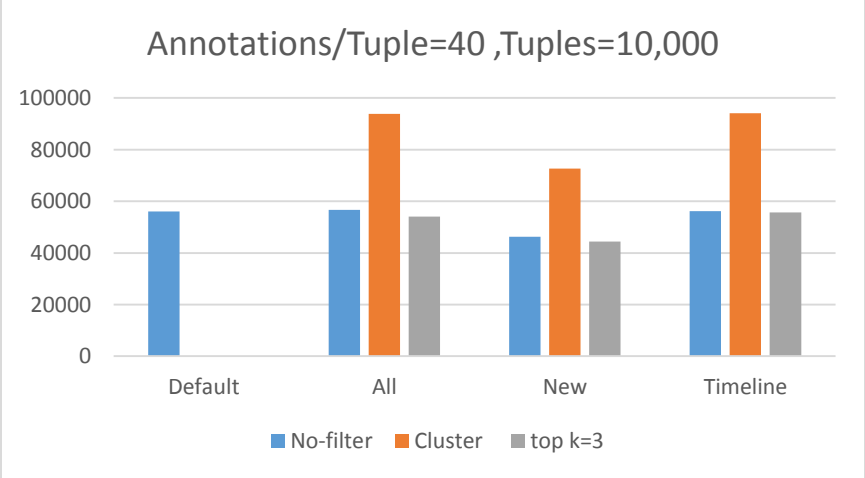


Figure 32: User-Centric Annotation Propagation Experiment 9

In Figure 33, the last experiment is to show that varying the K in the Top K filtering has minimum impact on the response time. The reason is that Top K is done after ranking and sorting all annotations per tuple and K define how many annotations to be shown in the propagation phase. Therefore, increasing the K has very slight increase in the response time.

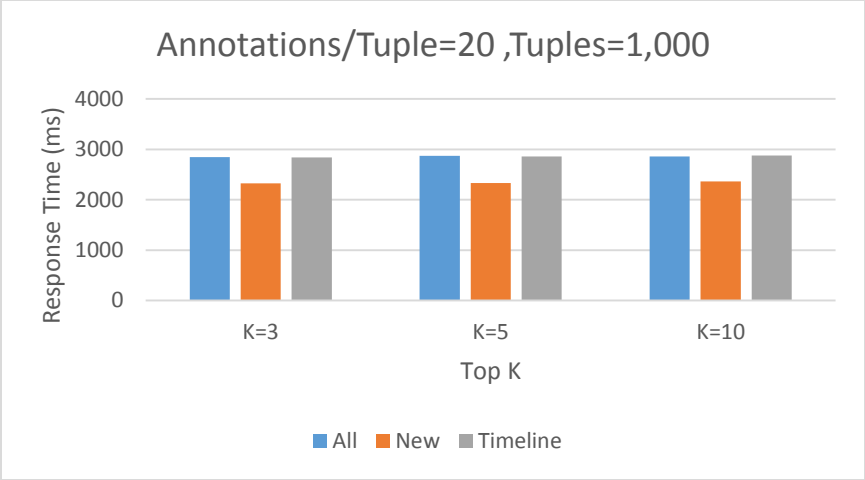


Figure 33: User-Centric Annotation Propagation Experiment 10

5.3 Proactive Annotation Management

In proactive annotation, we conducted a lot experiments varying the number of matches (Fan-outs) , the size of context around each hint , the overhead of each component in the workflow and last we compared the difference be using index (guided search) and no index (unguided search). The table sizes we used Table_1 =0.1 M, Table_2 =1M and Table_3 =10M as shown in Figure 34.

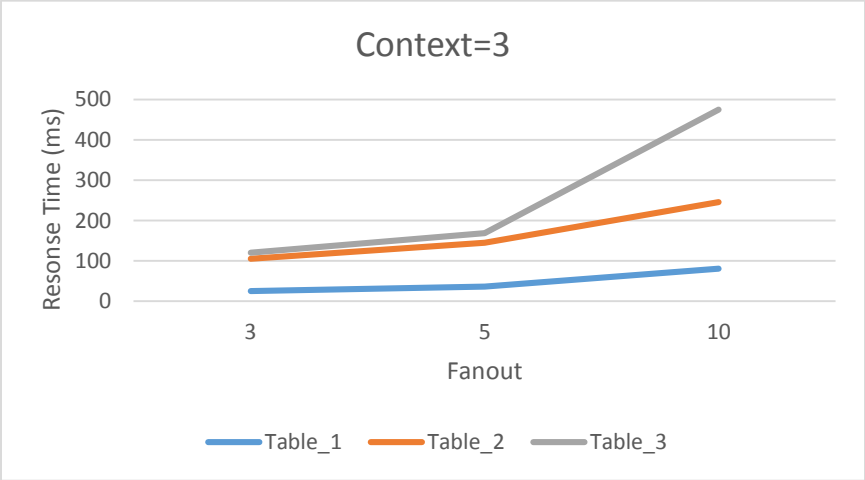


Figure 34: Proactive Annotation Management Experiment 1

In Figure 35, we measure the effect of the context size around the hints that were highlighted. We can see from the experiment that as the size of the context decreases the number of constructs decreases, therefore the response time decreases.

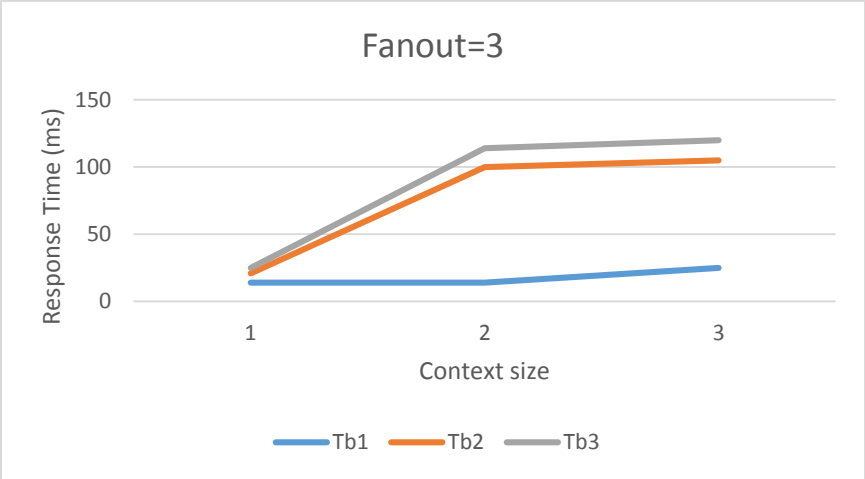


Figure 35: Proactive Annotation Management Experiment 2

The following experiment in Figure 36 shows the overhead of each component in proactive workflow. The data search is the most expensive component that's why we try to filter and eliminate unnecessary searches like stop words (parser) and to be ignored schema (schema analyzer).

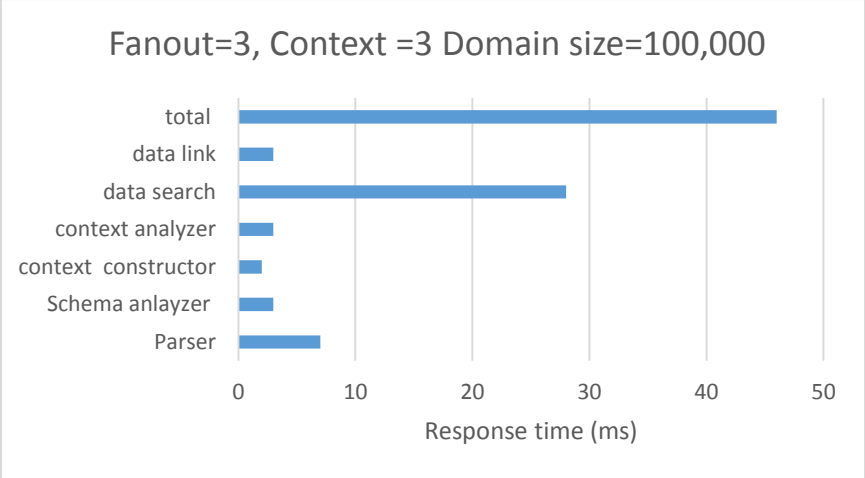


Figure 36: Proactive Annotation Management Experiment 3

In Figure 37, we conducted an experiment on five different annotations and measure the relationship between the number of constructs and the size of context. We observe that ratio of increase of the size is not the same as the ratio of increase of the

number of constructs basically because of the stop words. As the context size increases the chance of having stop words increases.

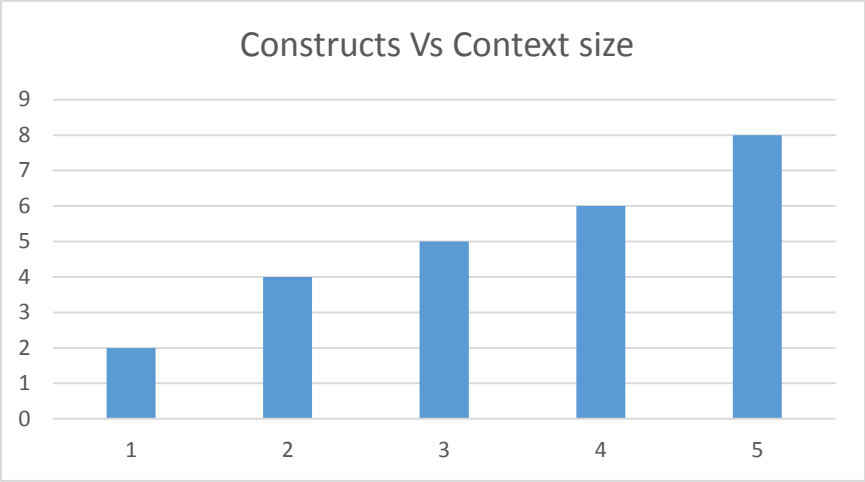


Figure 37: Proactive Annotation Management Experiment 4

The last experiment we conducted in Figure 38, is to measure the response time if DBA define a columns to be searched (guided) and if DBA don't define any column therefore search the whole table (unguided). If a column is highlighted by the DBA an index is created over this specific column and that improve the performance significantly.

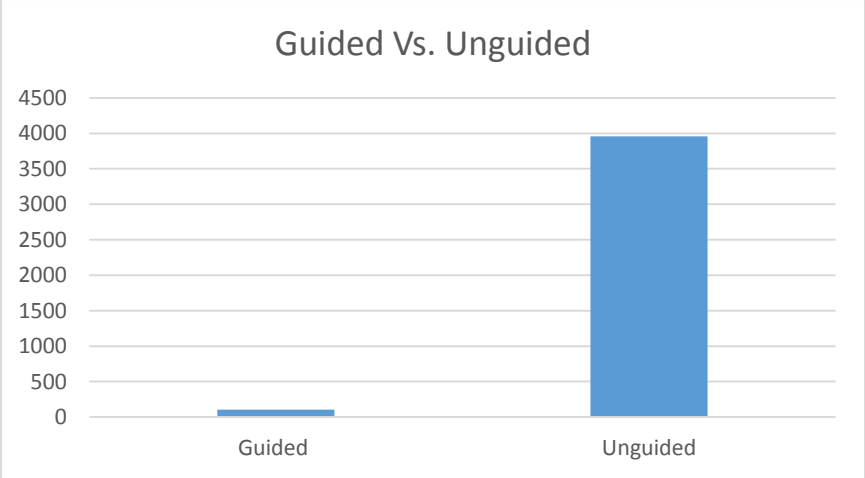


Figure 38: Proactive Annotation Management Experiment 5

5.4 Summary-Based Annotation Management: Advanced Querying

In this chapter, we conducted two experiments to show the effect of using index to the response time and how efficient it is. We used real data of annotations from http://www.dbrc.org.uk/ and vary the size of annotations from 450,000 to 9 million annotations. The size of summaries is approximately 12% of the annotation size. Our test query is a select query that select 1% of the data tuples which is almost 400 tuples. The query is (select * from Table_1 Where "ClassBird1.Disease = 0"). In this query we propagate the annotation summaries with the data. So, after the selection using an index, there is a join between the selected tuples and the summary table to get its summaries. From the experiments, we can see that using index and varying the size of annotations the response time in the hundreds milliseconds while without using index response time in ten thousands milliseconds as shown in Figure 39.

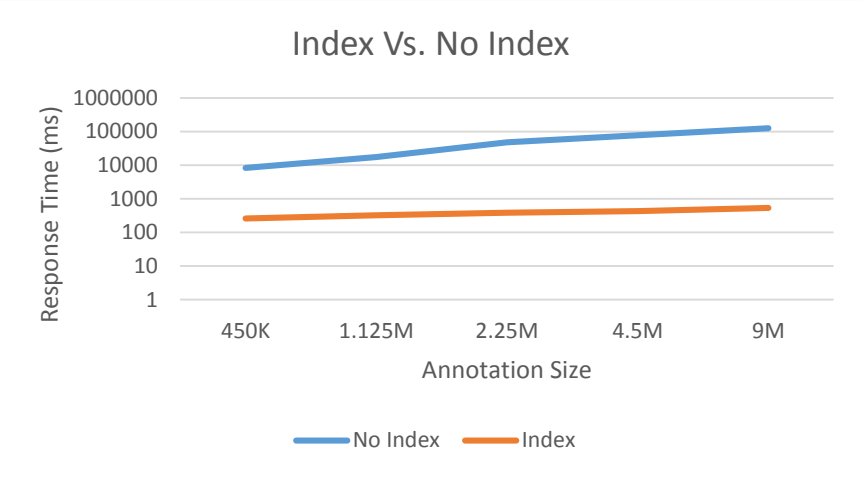


Figure 39: Summary-Based Annotation Management Experiment 1

In Figure 40, the second experiment showing the importance of back referencing in the B-tree index. The query is summary-based selection (select * from table Where “ClassBird1.Disease = 0”). The query selectivity is 1% , we return 1% of the data records almost 400 tuples. In this query, we do not propagate the annotation summaries with the data. No index is the same like previous experiment since it performs the expensive join between the data and summaries, and then applies a summary-based selection. Index

with No back pointers are pointers point back to summary table. In this case, we still need a join between summaries returned from the index and data tuples. Index with the back pointers which returns the data tuples directly. Therefore, no join operation is needed.

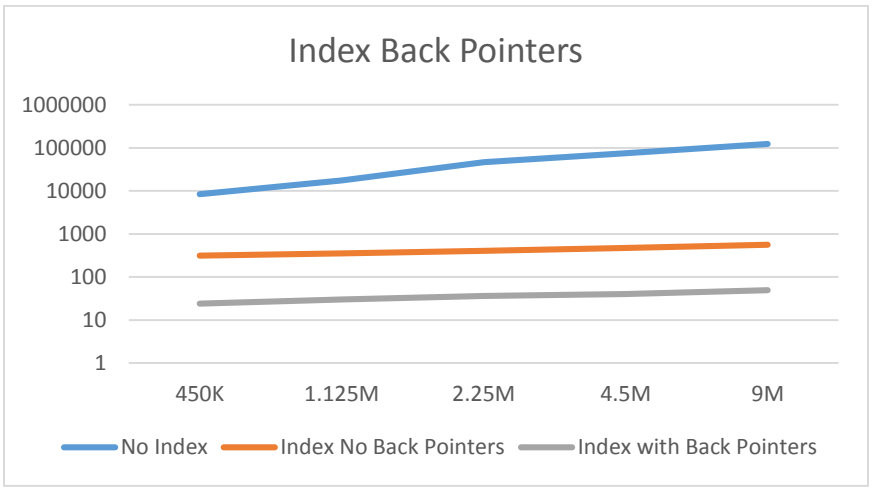


Figure 40: Summary-Based Annotation Management Experiment 2

Chapter 6: Related Work

6.1 User-centric Annotation related work:

Combining Dependent Annotations for Relational Algebra [13]

In their work, they assume that annotations have some structure in order to determine how to propagate them. In previous work, different kinds of annotations forms are treated in isolation of the other forms. Their goal is to combine different forms of annotations that depend on each other. They provide a method for combining different forms and provide a normal form which is useful in deciding whether two or more combined annotations are equivalent.

Annotation are relative [2]

In this paper they describe a hierarchical model of annotation where annotation is treated as 1st class citizen as user can explore the annotations. They assume that annotation can be applied to two or more data values with some shared structure. Given the annotation schema, they provide a mechanism to query the annotation hierarchy. They consider according to the query and the schema, a level is defined and everything below that level is treated as data and everything above that level is annotation which is propagated with the query according to certain rules.

Propagation of multi-granularity annotations [1]

In this work, they refer to the model for implicitly associating annotations (provenance) – the annotations are associated with data with arbitrary granularity – as "multi-granularity annotation" model. In the previous work no provenance management on multi-granularity annotation is reported. They define set of rules that show how to

propagate the annotation with each algebra operator and define the scopes of the annotations associated. They also show that during data derivation how to preserve the annotation association with data without losing significant information.

Google new personalization: Scalable online collaborative filtering [5]

In this paper, they describe collaborative filtering for personalizing google news for users. They use three main techniques in order generate the personalized recommendation which are: collaborative filtering using MinHash clustering, Probabilistic Latent Semantic Indexing (PLSI), and co-visitation counts. They describe the algorithm used and system setup where is general and can be applied in different application as their method is content agnostic and consequently domain independent.

YMALDB: A Result-Driven Recommendation System for Databases [6]

In this paper, they present to users additional information that may be of interest to them called YMAL “You May Also Like”. They try to find correlation among different attributes in various relations. They compute the correlation using the most interesting set of attribute values used FaSets. The interestingness of a faSet is based on its frequency both in the query result and in the database

Believe It or Not: Adding Belief Annotations to Databases [14]

A model is implemented to for the relational database to capture the users’ believes and save it as annotations. Believes can be added to data or other belief statements. They introduce new semantics based on multi-agent epistemic logic and new query language to propagate believes with the data. They use structured annotation to represent a belief in the database and transform the belief queries to traditional sql queries.

They define belief database as” A belief database contains base information in the form of ground tuples, annotated with belief statements. It represents a set of different

belief worlds, each one for one type of belief annotation, i.e. the beliefs of a particular user on ground tuples, or on another user's beliefs." They use canonical Kripke structure to encode a belief. They use this structure to describe a relational representation of belief databases, and give an algorithm for translating queries over the belief database into standard relational queries. Believes can be overlapping or conflicting, which is taken into consideration in the propagation process.

They also introduce belief conjunctive queries that serves as interface to a belief database. Basically by using the hierarchical structure of believes (annotations), users can figure out the agreements or disagreements with other users.

MONDRIAN: Annotating and querying databases through colors and block [9]

The main focus is this paper to enable the users to query and propagate annotations with the data. They introduce a model that associate annotations to set of values of data and efficiently query their information. Annotations are represented colors and the set of values are as blocks which are marked by a color or multiple colors. They introduce new query language that they prove to be complete as users can use all possible queries over the annotated data.

They prove that their mechanism is more efficient than the previous work. In the previous work in order to annotate data, they alter the table with new annotation column to correspond to data column. For instance, table Gene has column gene_name, a new column annot_gene_name is altered to the table to hold the annotations.

Another part of the paper, they emphasis that annotations should be treated as first class citizen and annotations shouldn't be hidden from the user as in the previous work. They claim "annotations are of equal or even greater importance than values". They succeeded in expressing queries include predicate as data that are annotated or not.

Supporting Annotations on Relations [8]

They present a new efficient storage technique that enable to user to attach annotations in different granularities like column, row, or cell. They advantage of using

their storage mechanism that it saves up to 70 % of the execution time compared to storing annotation for cell level. They also, introduced different types of annotations as annotations that applied to newly inserted data or modified data. They extend the standard SQL to enable add, archive, query, and propagate annotations along with the data.

A compact representation of annotations is implemented that reduce the storage overhead and I/O cost of queries. They also introduced three types of annotations: snapshot, view, and join annotations. Snapshot annotations is applied to an instance of data, view annotations annotate newly inserted data if they satisfy certain conditions, join annotations are attached to data from different relations. There are other querying and storage challenges they addressed in the paper.

An annotation management system for relational databases [4]

They present an extension to SQL that enable the users to propagate the annotation using three different schemes. First, the default scheme propagate annotations with respect to where the data are copied from. Second, the default-all scheme propagate annotations with respect to where the data are copied from all equivalent formulations of a query. Third, the custom scheme enable the user to have the control over the propagation of the annotation over the data. A storage scheme for annotations is introduced that contains algorithms that translate predefined query language to standard SQL queries and propagate the relevant annotations according to one of the three defined schemes.

DBNotes: A Post-It System for Relational Databases based on Provenance [3]

They refer the annotations in the relational data as a note that hold information about the provenance or the linage of data. Every value in the table is marked with a note that shows the provenance. They use special query language to propagate and query the annotations in different ways. They enable the users to view a high-level explanation of the provenance of a value that may be the result of query transformation steps. Also, they

provide a detailed explanation at each transformation step to explain why a value has allocated from a source to a target database.

6.2 Proactive Annotation Management related work:

Keyword Search over Relational Databases: A Metadata Approach [16]

They introduce a novel techniques to transform keyword queries to standard SQL based on Hungarian algorithm. They identify a meaningful SQL queries that satisfy the intended keyword query semantics. They have three main contributions, first they defined that one of the major problem in keyword searching over the relational databases is the lack of access to the database instance. Second, in order measure the likelihood of the semantics of a keyword and the database structure as table, attribute or value, they introduce the notion of weights. Weights are divided into intrinsic (isolation of keywords) and contextual (semantics with respect to neighbor keywords). Third, they extend the Hungarian algorithm to perform the necessary computations for contextual weights that leads to different interpretations of the keyword query then transform to different SQL queries based on the ranking of each interpretation.

Keyword Search in Databases: The Power of RDBMS [17]

In their work, in order to answer a keyword query they use standard SQL to compute all the interconnected tuples that satisfies the given query. They prove that current commercial database management systems can perform keyword search efficiently and without any additional indexing to be build or maintained by the users by tuple reduction. The first step in tuple reduction is to exclude the tuples that don't participate in any results. Then using join operator more tuples are excluded.

6.3 Summary-based Annotation Management: Advanced Querying

InsightNotes: Summary-Based Annotation Management in Relational Databases [19]

In this paper, they provide a mechanism to summarize annotation and manage the propagation of summaries with data. In previous annotation management techniques, they fall short in providing advanced processing over the annotations beyond just propagating them to end-users. To address this limitation, they propose the InsightNotes system, a summary-based annotation management engine in relational databases. InsightNotes integrates data mining and summarization techniques into the annotation management in novel ways with the objective of creating and reporting concise representations (summaries) of the raw annotations. They propose an extended summary-aware query processing engine for efficient manipulation and propagation of the annotation summaries in the query pipeline. They also introduce several optimizations for the creation, maintenance, and zoom-in processing over the annotations summaries. In the below figure it shows the hierarchy of annotation summaries attached to a data tuple.

Chapter 7: Conclusion

Annotations management is important topic and it is widely used in many applications. Our focus in this thesis about annotation management in scientific databases in relational database management system. With the massive collaboration between scientists using the annotation, annotations become Big with large scale and that imposes new challenges. We tackle the Big annotation problem from three different angles. First, we enable the users to build profiles of their interests and based on their profile, the system personalize the propagation of the annotations. The users have the option to view the annotation in a friendly way that ease the searching process for them. Second, we Big annotations added by the users like scientific article, we don't expect that the user is going to annotate every data mentioned in the article. Meanwhile, annotations are trustworthy and all the data related to the annotation should be annotated. Therefore, we introduce proactive annotation management to fill in this gap which search for hints in the annotation and search for possible hidden links in the annotation and data. Third, InsightNote system provide summaries for annotations to be propagated instead of the raw annotations. However, users can not access the summaries and add predicates on summaries. We extended InsightNotes to treat the summaries as 1st class citizen and enable the user to search the data based on summaries by adding predicates to summaries. We introduced special type of B-tree index on classifiers instances that improved the performance significantly

Chapter 8: References

[1] Ryo Aoto, Toshiyuki Shimizu, and Masatoshi Yoshikawa. Propagation of multi-granularity annotations. 22nd international conference on Database and expert systems applications, September 2011.

[2] Peter Buneman, Egor V. Kostylev, and Stijn Vansummeren. Annotations are relative. EDBT, 2013.

[3] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. SIGMOD, May 2005.

[4] W. Tan D. Bhagwat, L. Chiticariu and G. Vijayvargiya. An annotation management system for relational. VLDB Journal, 2005.

[5] Abhinandan Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization:Scalable online collaborative ltering. WWW Track: Industrial Practice and Experience, pages 271-280,2007.

[6] Marina Drosou and Evaggelia Pitoura. Ymaldb: A result-driven recommendation system for databases. EDBT/ICDT, March 2013.

[7] M. Eltabakh, M. Ouzzani, W. Aref, A. Elmagarmid, Y. Laura-Silva, M. Arshad, D. Salt, and I. Baxter. Managing biological data using bdbms. ICDE, 2008.

[8] Mohamed Y. Eltabakh, Walid G. Aref, Ahmed K. Elmagarmid, Mourad Ouzzani, and Yasin N. Silva.Supporting annotations on relations. EDBT, pages 379-390, 2009.

[9] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. Mondrian: Annotating and querying databases through colors and blocks. 22nd International Conference on Data Engineerin (ICDE),March 2006.

[10] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations item-to-item collaborative ltering. The IEEE Computer Society, pages 76-80, February 2003.

[11] W. C. Tan. Containment of relational queries with annotation propagation. DBPL, 2003.

[12] Maria Theodoridou, Yannis Tzitzikas, Martin Doerr, Yannis Marketakis, and Valantis Melessanakis. Modeling and querying provenance by extending cidoc crm. Distributed and Parallel Databases, 27(2):169-210, April 2010.

[13] Egor V. Kostylev, Peter Buneman. Combining dependent annotations for relational algebra. ICDT Proceedings of the 15th International Conference on Database Theory, pages 196-207, ICDT 2012

[14] Wolfgang Gatterbauer, Magdalena Balazinska, Nodira Khoussainova, Dan Suciu. Believe it or not: adding belief annotations to databases. Proceedings of the VLDB Endowment. Volume 2 Issue 1, VLDB August 2009

[15] Sonia Bergamaschi, Elton Domnori, Francesco Guerra, Raquel Trillo Lado, and Yannis Velegrakis. 2011. Keyword search over relational databases: a metadata approach. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data SIGMOD 2011.

[16] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. 2009. Keyword search in databases: the power of RDBMS. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data SIGMOD 2009.

[17] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das DBXplorer: A System for Keyword-Based Search over Relational Databases. In Proceedings of the 18th International Conference on Data Engineering ICDE 2002.

[18] Vagelis Hristidis and Yannis Papakonstantinou. 2002. Discover: keyword search in relational databases. In Proceedings of the 28th international conference on Very Large Data Bases VLDB 2002.

[19] Dongqing Xiao, Mohamed Y. Eltabakh. InsightNotes: Summary-Based Management in Relational databases. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of data SIGMOD 2014.