InsightNotes: Summary-Based Annotation Management in Relational Databases*

Dongqing Xiao
Computer Science Department
Worcester Polytechnic Institute (WPI), USA
dxiao@cs.wpi.edu

Mohamed Y. Eltabakh
Computer Science Department
Worcester Polytechnic Institute (WPI), USA
meltabakh@cs.wpi.edu

ABSTRACT

In this paper, we address the challenges that arise from the growing scale of annotations in scientific databases. On one hand, end-users and scientists are incapable of analyzing and extracting knowledge from the large number of reported annotations, e.g., one tuple may have hundreds of annotations attached to it over time. On the other hand, current annotation management techniques fall short in providing advanced processing over the annotations beyond just propagating them to end-users. To address this limitation, we propose the InsightNotes system, *a summary-based annotation management engine in relational databases*. InsightNotes integrates data mining and summarization techniques into annotation management in novel ways with the objective of creating and reporting concise representations (summaries) of the raw annotations. We propose an extended *summary-aware* query processing engine for efficient manipulation and propagation of the annotation summaries in the query pipeline. We introduce several optimizations for the creation, maintenance, and zoom-in processing over the annotations summaries. InsightNotes is implemented on top of an existing annotation management system within which it is experimentally evaluated using real-world datasets. The results illustrate significant performance gain from the proposed techniques and optimizations (up to 100x in some operations) compared to the naive approaches.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*Query processing*

Keywords

Annotation Management; Summarization; Query Processing.

1. INTRODUCTION

Modern relational database systems provide backbone support to many emerging scientific applications in various disciplines such as in biology, healthcare, ornithology, among many others. In these applications, data curation and annotation is a vital mechanism for capturing users' observations, understanding and curating the data, highlighting erroneous or conflicting values, and freely describing the data outside the barriers of the rigid relational schemas. More importantly, in many cases, scientists assess the credibility and trustworthy of the data based on the annotations attached to it, e.g., based on the data's provenance information and any attached scientific articles. That is why annotation management has been extensively studied in the context of relational databases to support these applications [4, 6, 10, 13, 16, 22] .

However, with the increasing scale of collaboration and the extensive use of annotations among scientists, the number and size of the annotations may far exceed the size of the original data itself, e.g., one tuple in the database may have tens or even hundreds of annotations attached to it over time. This is especially true due to the increasing number of automated tools that annotate the data with various information [21], e.g., verification, quality assessment, and prediction tools. The following examples illustrate the increasing scale of scientific annotations.

Example 1−Biological Annotations (10x Scale-up): Numerous biological databases leverage the power of relational DBMSs for storing, querying, and annotating or curating their data, e.g., Genobase (http://ecoli.naist.jp/GB8/), EcoliHouse (http://www.porteco.org/), Ensemble project (http://www.ensembl.org/), and UniProt database system (http://www.ebi.ac.uk/uniprot). These systems either provide web-based SQL interfaces for directly executing complex queries over the data, e.g., Genobase, or allow full database download, e.g., Ensemble, EcoliHouse, and UniProt. According to the *geneontology.org* website, the ratio between the number of annotations to the number of biological records in several of these systems is more than an order of magnitude, e.g., in UniProt database there are around 28,239,042 data records compared to 186,648,155 annotation records, and in DataBank database there are 199,930 data records compared to 3,047,731 annotation records. These annotations range from function predictions of genes and proteins, provenance records carrying the sources' information, scientific articles about the data, and curation information such as bug fixes and version numbers. Thus, it is typical in these databases to have 10s of annotations attached to each data tuple.

Example 2−Ornithological Annotations (100x Scale-up): Ornithology is the branch in science that concerns the study of birds. Several large-scale relational databases are in use to store and query information related to 10s of thousands of birds worldwide, e.g., DBRC (http://www.dbrc.org.uk/), and AKN (http://www.avianknowledge.net/). In these systems, in addition to the base data, i.e., the birds' basic information such as scientific names, synonyms, geographic ranges, images, description, etc., there are more than 200,000 bird watchers and scientists who continuously provide observations and annotations on these birds from their trips. It is reported in [1] that, on average, bird watchers add 1.6 million observations (annotations) per month to the ebirds sys-

*This project is partially supported by NSF-CRI 1305258 grant.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

tem, which is part of the AKN network. These annotations are free-text values that may describe anything related to the observed birds, e.g., color, body shape or weight, certain behavior or sound, eating habits, geographic location, or observed diseases. Therefore, in these databases the number of annotations can be more than two orders of magnitude larger than the number of data records.

Given these motivating applications, it is evident that the amount of annotations reported back (propagated) to end-users along with queries' answers can be overwhelming and hard to interpret, e.g., it is extremely hard for a biologist to go over many reported annotations to find out which ones carry provenance information vs. bug fixes, which ones are obsolete or proven wrong, and what is the summary of an attached big article or document. Similarly, for an ornithologist or bird lover, it is almost impractical to go over 100s of annotations per tuple to find out which ones are duplicates or have similar content, which ones talk about a bird's eating habits vs. body anatomy, and which ones are more critical and highlight diseases. Therefore, delegating the extraction of such knowledge to end-users, especially in large-scale systems, is the wrong choice. Instead, we need more advanced annotation management systems that not only propagate and query the raw annotations, but also analyze, mine, and summarize them into more meaningful representations that provide the most insight possible to end-users.

In this paper, we propose the *"InsightNotes"* system; *a summary-based annotation management engine in relational databases*. InsightNotes advances the state-of-art in annotation management by exploring, utilizing, and propagating the annotations in novel ways even under complex relational queries and transformations, e.g., projection, join, grouping and aggregation, and duplicate elimination. InsightNotes integrates data mining techniques into annotation management with the objective of creating and reporting more concise representations of the raw annotations. For example, referring to Figure 1, instead of propagating the raw annotations with each output tuple from a query, InsightNotes will propagate various types of summaries, e.g., clustering the annotations having similar content into groups and reporting only a representative from each group, classifying the annotations according to user-defined classifiers and reporting each class label along with the number of associated annotations, and summarizing large-object annotations, e.g., big documents or articles, and reporting small snippets representing them. The mining techniques will operate on the tuple-level, i.e., the annotations on each tuple will be summarized and then the mining outputs will be attached back to their data tuples as illustrated in Figure 1.

InsightNotes addresses several core challenges, which include: ***(1) Extensibility and Efficient Maintenance:*** Each of the motivating examples described above warrant the need for different types of annotation summaries. Therefore, InsightNotes is designed as an extensible system, where the database admins define how to summarize the annotations in a way suitable for their applications. For example, in biological databases (Figure 1(a)), it can be meaningful to classify the annotations on genes into classes {'FunctionPrediction', 'Provenance', 'Comment'}, while in ornithological databases in is more meaningful to classify them into classes {'Behavior', 'Disease', 'Anatomy', 'Other'}. The system only defines some properties and requirements that the integrated mining techniques should obey for efficient and incremental maintenance of the annotation summaries. ***(2) Summary-Aware Query Processing:*** Unlike the raw annotations which are free-text objects, the annotation summaries will have well-defined structures and properties. The challenge is how to extend the query engine to efficiently manipulate the annotation summaries at query time without retrieving the raw annotations whenever possible. We present two strategies that compute the summaries at the last stage of query processing (*lazy evaluation*), and then we propose a more aggressive and efficient strategy that entails extending each of the query operators, e.g., projection, join, grouping, and aggregation, to directly operate on the summary objects attached to each tuple. And ***(3) Zoom-in Query Processing:*** Reporting the annotation summaries raises another challenge, which is: *What if the end-user is interested in zooming-in and retrieving specific raw annotations?* For example, referring to Figure 1(a), the end-user may be interested in retrieving the three annotations labeled with "FunctionPrediction" on the given tuple, or in retrieving all annotations in the cluster represented by annotation *"A1"*. Therefore, we propose novel *zoom-in* querying capabilities coupled with smart caching techniques for efficient execution.

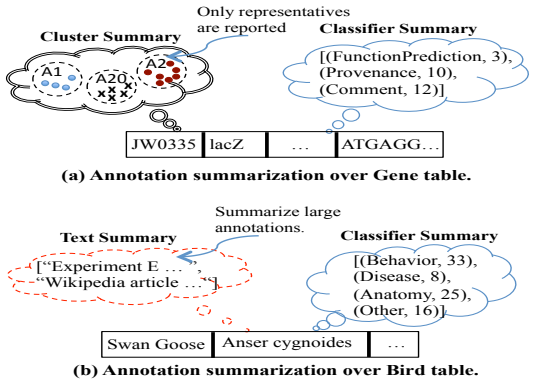

Figure 1: Annotation summarization in different domains.

The key contributions of this paper are summarized as follows:

- Proposing the *InsightNotes* system, a summary-based annotation management engine in relational DBs, that exploits and propagates the annotations in novel ways. InsightNotes propagates to end-users concise and meaningful representations, called *annotation summaries*, instead of the, possibly too numerous, raw annotations. InsightNotes provides efficient incremental maintenance of the annotation summaries, and it is extensible as it enables database admins to define their own summarization techniques.
- Proposing different strategies for optimizing the creation and propagation of the annotation summaries. They cover the spectrum of entirely postponing the creation of summaries until query time (*On-The-Fly* strategy), pre-computing and materializing the summaries but lazily integrating them at the last stage of query processing (*Lazy-Propagation* strategy) and aggressively integrating them at the early stage of query processing (*Summary-Aware* strategy). We studied the pros and cons of each strategy.
- Extending the semantics and algebra of the standard query operators to propagate the annotation summaries within the query pipeline. The extended operators can directly manipulate the summaries without retrieving the raw annotations. We also introduce new operators specific to the propagation of annotation summaries.
- Introducing a novel *zoom-in* query processing mechanism for expanding the annotation summaries and retrieving the raw annotations when desired. We propose caching techniques and several runtime optimizations for efficient execution.

The rest of the paper is organized as follows. In Section 2, we provide formal definitions for the annotation summaries, and present three strategies for developing InsightNotes. In Section 3, we cover the extended query processing engine including the *Lazy-Propagation* and *Summary-Aware* strategies as well as the *zoom-in* processing. The creation and maintenance of the annotation summaries is introduced in Section 4. The related work and experimental evaluation are presented in Sections 5 and 6, respectively. And finally the conclusion remarks are included in Section 7.

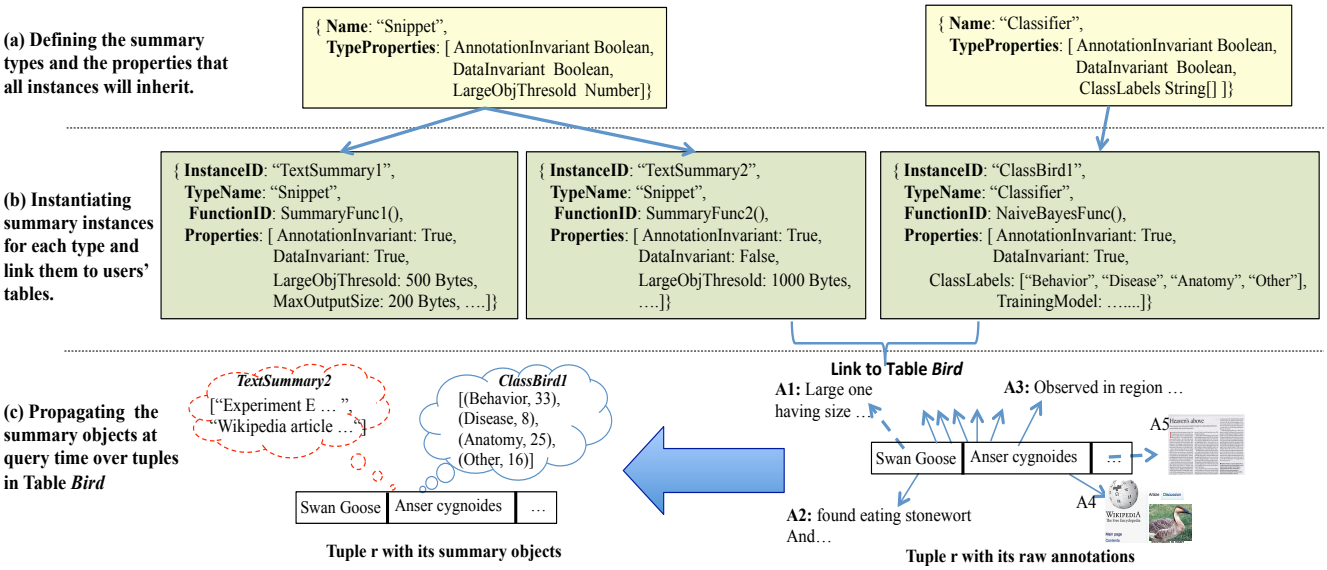

Figure 2: The hierarchy of annotation summaries: Types, Instances, and Objects.

2. ANNOTATION SUMMARIES: TYPES AND STRUCTURES

In this section, we formally present the data model for the annotation summaries, which includes the types of summaries supported in InsightNotes along with their structures and properties.

• **Summary Types:** InsightNotes supports three widely-used families (types) of data mining techniques for mining and summarizing the raw annotations, which are: (1) Text summarization techniques, e.g., [18], for summarizing large-object annotations, e.g., big text values and large documents, and creating concise snippets from them, (2) Clustering techniques, e.g., [17, 24], for clustering the annotations into distinct groups of similar content, And (3) Classification techniques, e.g., [9], for categorizing annotations according to user-defined classifiers. The definition of the summary types is as follows:

Definition 1– Summary Type: *Each annotation summary has a type that consists of a pair of values {Name, TypeProperties}, where Name is a unique identifier that defines the summary type, i.e., "Snippet", "Cluster", or "Classifier", and TypeProperties is a set of key-type pairs representing the type-level properties.*

The type-level properties will be inherited by all summary instances instantiated from a given type. Typically, InsightNotes will leverage some of these properties at execution time to optimize the creation and propagation of the annotation summaries. For example, all summary types will have two Boolean properties (See Figure 2(a)), which are: ***AnnotationsInvariant***, and ***DataInvariant***. The former property specifies whether or not the summarization of a newly added annotation over a given tuple t depends on t's existing annotations. In contrast, the latter property specifies whether or not the summarization depends on t's content. Either of these properties can be True or False independently. In the case where the two properties are set to True, then the Annotation Manager can execute the summarization algorithm for that type only once for each added annotation even if it will be attached to many tuples. The created summary can then be attached to as many tuples as needed. Each summary type may also have a set of type-specific properties. For example, the *Snippet* type may have a ***LargeObjThreshold*** property that specifies the threshold above which an annotation is considered a large object and needs to be summarized. Similarly, the *Classifier* type may have a ***ClassLabel*** property that specifies the possible class labels for a given classifier as shown in Figure 2(a).

• **Summary Instances:** A summary instance is a specific instance of a given summary type that defines the exact algorithm (function) used to implement the summary type as well as any instance-level properties. Each summary type may have many instances defined under it as illustrated in Figure 2(b). The definition of summary instance is as follows:

Definition 2– Summary Instance: *A summary instance defines a specific implementation of a summary type and consists of a four-ary vector {InstanceID, TypeName, FunctionID, InstanceProperties}, where the InstanceID is a unique identifier for each instance, TypeName references a summary type, FunctionID defines the function name implementing this instance, and InstanceProperties is a set of key-value pairs representing the instance-level properties.*

The instance-level properties may have an arbitrary number of user-defined properties—This is in addition to the type-level properties inherited from the summary type. Unlike the type-level properties, instance-level properties are not used for optimizations because their semantics are not known to the system. In contrast, they are used as a systematic way for storing and passing information in the system. For example, in Figure 2(b), there are two summary instances instantiated from the *Snippet* type, which are *TextSummary1* and *TextSummary2*. The former instance is both *AnnotationInvariant* and *DataInvariant*, whereas the latter instance is not *DataInvariant*, i.e., its function (*SummaryFunc2()*) may depend on the tuple's content while creating its snippets. Another example is the *ClassBird1* classifier instance presented in Figure 2(b). Any of the type- and instance-level properties can be accessed by the underlying functions to customize its execution as desired. For example, the *ClassBird1* classifier instances uses the *TrainingModel* property to pass the location of the model to the underlying function *NaiveBayesFunc()*.

Once summary instances are defined in the database, they can be linked to users' relations in a many-to-many relationship, i.e., each DB relation R may have many instances linked to it to summarize the annotations attached to each tuple $r \in R$ and to create the summary objects defined next. It is worth mentioning that the creation of the summary types and instances, and linking them to users' relations is the task of the database admin, which is entirely transparent from end-users querying the data.

• **Summary Object:** The summarization of the raw annotations attached to a given tuple r according to a given summary instance creates a *summary object* that will be attached to r and automatically maintained and updated by the system. Summary objects are the objects that will propagate in the query pipeline and be reported to end-users. They are defined as follows:

Definition 3– Summary Object: *A summary object summarizes the annotations attached to a given data tuple according to a specific summary instance. A summary object consists of a five-ary vector {ObjID, InstanceID, TupleID, Rep[], Elements[][]}, where ObjID is the objects's unique identifier, and the InstanceID and TupleID are references for the corresponding summary instance, and the data tuple, respectively. Rep[] is an array storing the representatives produced from the summarization algorithm, while Elements[][] is a two-dimensional array storing for each representative, the references (Ids) to its contributing raw annotations.*

The example in Figure 2(c) illustrates linking the two summary instances *TextSummary2* and *ClassBird1* to the Bird table, and hence the raw annotations on each tuple $r \in Bird$ will be summarized according to these instances and the resulting summary objects will be attached back to r as depicted in the figure.

In general, assume a user relation R that has n data attributes and k summary instances linked to it. Then, each tuple $r \in R$ has the following conceptual 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 annotation summary objects attached to r. For each summary object s_i, the structure of the summary representatives stored in Rep[]—which will be reported to the end-user— depends on s_i's summary type, e.g., clustering techniques may produce a representative for each cluster, while classification techniques may produce the class labels along with the count of annotations assigned to each label (Refer to Figure 2(c)). In InsightNotes, the structure of the output representatives is defined as follows (See Figure 2(c)):

Summary Type	Structure of Representatives (Rep[])
Cluster	[(*Text annotation, Number groupSize*)]
Classifier	[(*Text classLabel, Number annotationCnt*)]
Snippet	[(*Text snippet*)]

Finally, for a given representative, say *Rep[i]*, the corresponding entry in the *Elements* array, i.e., *Elements[i][]*, stores the Ids of the raw annotations contributing to Rep[i]. For example, in the *ClassBird1* summary object, the 1^{st} representative *Rep[1] = (Behavior, 33)* has the corresponding entry *Elements[1][] = [A2, ...]* referring to the raw annotations describing the bird's behavior. Similarly, the *TextSummary2* object has two entires in its Rep[] array with corresponding entires *Elements[1][] = [A5]*, and *Elements[2][]= [A4]* referring to the articles attached to the tuple, respectively. It is worth noting that the Elements[][] array will not be reported to end-users at query time, instead it will be used by the system during the query processing and zoom-in operation as described in Section 3.

2.1 InsightNotes Strategies

As presented above, the annotation summaries are fundamentally different from the raw annotations since they have well-defined structure and properties. Hence, existing annotation propagation techniques are not directly applicable in InsightNotes. In this section, we present three possible strategies (illustrated in Figure 3) for designing the InsightNotes engine. The pros and cons of each strategy are also summarized in Figure 4.

In Strategy I (*On-The-Fly*) depicted in Figure 3(a), the system maintains only the raw annotations without materializing or precomputing the annotation summaries. At query time, the raw annotations will propagate in the query pipeline according to the state-of-art annotation management techniques [4, 10, 13, 22], and then they will be mined and summarized on the fly using a newly introduced operator on top of the query plan as depicted in Figure 3(a). The new operator will basically receive each data tuple along with its attached raw annotations, execute the mining algorithms to summarize these annotations, and then report each data tuple along with its annotation summaries. As depicted in Figure 4, the advantages of this strategy are that it leverages the existing annotation management engines, does not require additional storage, and it may generate better cluster summaries since the clustering step is executed after all merging and dropping operations have been performed on the annotations in the query plan. However, Strategy I can be very expensive since the costly mining algorithms will be executed on-the-fly as part of each query. Nevertheless, it will not scale as the number of summary instances linked to the DB relations increases. Another critical limitation is that since the summaries are constructed only at the last stage of processing, then it is not applicable to apply advanced summary-based processing, e.g., selecting, joining, or ordering the data tuples based on their annotation summaries.

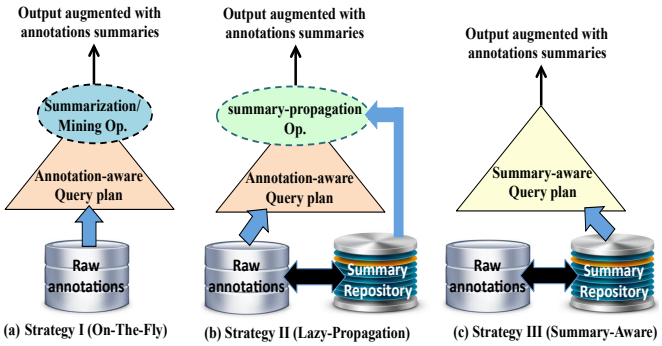

Figure 3: InsightNotes strategies.

	Strategy I	Strategy II	Strategy III
Query engine extension	Separate mining operator on top	Separate summary operator on top	Extended algebra for all operators
Query Algebra	Annotation-based algebra	Annotation-based algebra	Summary-based algebra
Additional Storage	No	Yes	Yes
Query performance*	slowest	slower	fastest
Scaling with number of summaries*	No	No, but better than Strategy I	Yes
Advanced summary-Based Querying**	No	No	Yes
Summaries quality*	Better clusters	Identical clusters in both strategies	
	Snippet & Classifier: Identical in all		

* Experimentally verified in Section 6.
** This feature is an on-going work and beyond the scope of this paper.

Figure 4: Comparison between InsightNotes Strategies.

In Strategy II (*Lazy-Propagation*) depicted in Figure 3(b), the system creates the annotation summaries and maintains them at the time of adding new annotations. These summaries will be stored in a repository called *Summary Repository*. However, the query operators will not be modified to operate on the annotation summaries, instead they will still operate on and propagate the raw annotations (similar to Strategy I). The key advantage of Strategy II is that the newly added operator on top of the query plan (the *summary-propagation* operator) will not execute the mining algorithms to create the summaries, instead it will access the *Summary Repository* to transform the raw annotations into their summarized

representations. Thus, Strategy II should yield better performance and scalability compared to Strategy I. However, this strategy has two main drawbacks. First, similar to Strategy I, since the summaries are integrated only at the last stage of processing, then advanced summary-based processing is not applicable. And second, the *summary-propagation* operator is still an expensive operator and involves high overheads in building the final summary objects. And hence, this strategy will not scale well under large number of summaries (See Figure 4).

Strategy III (*Summary-Aware*) depicted in Figure 3(c) will fundamentally extend the query processing engine to directly operate on the annotation summaries in the query pipeline, i.e., each of the relational operators will be extended to process and produce tuples carrying their summary objects (Refer to the left-side tuple in Figure 2(c)). Therefore, the query algebra will need to be extended to a summary-based algebra. Although this strategy is more complex, it has two main advantages. First, it is computationally the most efficient strategy w.r.t. performance and scalability, e.g., it is around 7x and 3x faster than Strategies I, and II, respectively as will be presented in Section 6. And second, it is now feasible to treat the annotation summaries as first-class citizens in the database and to develop more advanced summary-based query processing beyond just the propagation, which is part of our future work.

3. InsightNotes QUERY ENGINE

The implementation of Strategy I (*On-The-Fly*) is straightforward, and thus in this section, we focus on the other two strategies. We assume that the *Summary Repository* is already created and maintained by the *Annotation Manager* (The focus of Section 4).

3.1 Summary Propagation: The Lazy-Propagation Strategy

In this section, we present the *Lazy-Propagation* strategy for implementing InsightNotes (Refer to Figure 3(b)). In this strategy, the raw annotations will propagate in the query pipeline according to the common semantics used in current annotation management systems, e.g., [4, 10, 13]. The basic principles of the propagation are summarized as follows: (1) The *selection* operator propagates the annotations from an input tuple to the output tuple without modification, (2) The *join*, *grouping and aggregation*, *duplicate elimination* and *set operators*, all these operators merge two or more tuples together and produce an output tuple. The annotations on the output tuple will be the concatenation (union) of all annotations on the merged tuples. And (3) The *projection* operator has two common semantics, which are either to propagate the annotations attached only to the projected attributes and drop any other annotations, or to propagate all annotations attached to a tuple and treating all of them as tuple-level—Usually a keyword in the SQL query defines which semantics is desired.

On top of a given query plan, the system will add the newly introduced *summary-propagation* operator. An input tuple to the operator is in the following form:

$$r =< a_1, a_2, ..., a_n, \{w_1, w_2, ..., w_m\} >$$

where $a_1, a_2, ..., a_n$ are the data values of r, and $w_1, w_2, ..., w_m$ are the raw annotations attached to r.

In Figure 5, we present the algorithm of the *summary-propagation* operator. The operator maintains a memory buffer (*Buff*) for caching the input tuples received from the downstream operator until the buffer is full (Line 1). Then, the buffered tuples are joined with the tables in the *Summaries Repository* using a semi-join algorithm, i.e., the unique annotation ids from *Buff* are selected, and then joined with the table containing the summary objects (note that the relationship between raw annotations and the summary objects is many-to-many) (Lines 3-4). Since different annotations may reference the same summary object, we eliminate the duplicates among the retrieved objects before grouping them per data tuple (Lines 5-6). The next step is to modify and merge the summary objects for each data tuple. The update step (Line 8) is needed only if the query plan involves a projection operation, which may drop annotations from the data tuples. Thus, the summary objects retrieved from the *Summary Repository* will need to be updated. The details of such updating will be discussed in Section 3.2 (Table 1). The last step is to merge the summary objects having the same instance Ids. Recall that a data tuple may be generated from operations such as join, distinct, and grouping, and hence it carries annotations from several tuples.

***Summary-Propagation* Operator**
Inputs:
- Data records in the form of: $r_{in} = <a_1, a_2, ..., a_n, \{w_1, w_2, ..., w_m\}>$
  //$w_1, w_2, ..., w_m$ are the raw annotations attached to r_{in}

Outputs:
- Data records in the form of: $r_{out} = <a_1, a_2, ..., a_n, \{s_1, s_2, ..., s_k\}>$
  //$s_1, s_2, ..., s_k$ are the summary objects corresponding to r_{in}'s raw annotation

Operator's Algorithm:
1. - Buffer the input data records into a memory buffer (*Buff)* until full
2. - Join *Buff* with *Summaries Repository* (using semi-join)
3. - Anno-IDs = Extract the distinct annotation ids from Buff.
4. - SummaryObjs = Semi-join the Ann-IDs with Summaries Repository and retrieve the corresponding summary objects.
5. - Eliminate duplicates from SummaryObjs and group them per data tuple
6. - Update and merge the summary objects on each data tuple *r*
7. - **For each** SummaryObj *o* on *r* **Loop**
8. - update *o* to reflect the effect of un-projected (dropped) annotations
9. - Merge the summary objects having the same instance Ids.
10. - Report one output tuple at a time until Buff is empty.
11. - Go to Step 1 until all input tuples from the downstream operator are consumed.

Figure 5: The algorithm for the *summary-propagation* operator.

3.2 Summary Propagation: The Summary-Aware Strategy

In this section, we present the *Summary-Aware* propagation strategy (Figure 3(c)). This strategy is driven by three objectives: (1) Avoid executing the mining algorithms at query time since that is highly expensive and will not scale as the number of summary instances increases, (2) Allow the query operators to directly manipulate the summary object in a pipelined fashion without retrieving the raw annotations, which will achieve significant speedup and enable querying the data based on the annotation summaries at any processing stage, and (3) The final results from both *Summary-Aware* and *Lazy-Propagation* should be identical such that the above benefits are obtained while maintaining the same quality of the generated annotation summaries.

At query time, each data tuple $r \in R$ in the query pipeline will have the following conceptual 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.

- **Selection Operator (σ):** The selection operator $\sigma_p(R)$ applies a set of predicates *p* over the data part of the tuples, and selects only the ones that satisfy these predicates. The summary objects attached to the selected tuples will propagate without modification.

$$\sigma_p(R) = \{r \in R,\ r =< a_1, a_2, ..., a_n, \{s_1, s_2, ..., s_k\} > \mid p(< a_1, a_2, ..., a_n >) = True\}$$

- **Projection Operator (π):** The projection operator $\pi_{a_1,a_2,...,a_m}(R)$ selects the specified attributes from R's tuples. The summary objects associated with these attributes should

Type	$\pi^F_{a_1,a_2,...,a_m}()$ // s_i (input object) and s'_i (output object)
Classifier	For j = 1 to number of representative in s_i.Rep[] Loop s'_i.Elements[j][ ] = Filter(s_i.Elements[j][], $\{a_1, ..., a_m\}$) s'_i.Rep[j].classLabel = s_i.Rep[j].classLabel s'_i.Rep[j].annotationCnt = $\|s'_i$.Elements[j][]$\|$

Snippet	For j = 1 to number of representative in s_i.Rep[] Loop
	s'_i.Elements[j][] = Filter(s_i.Elements[j][], $\{a_1, ..., a_m\}$)
	If (s'_i.Elements[j][] is Null) Then
	Delete s'_i.Elements[j] and s'_i.Rep[j]
	Else
	s'_i.Rep[j] = s_i.Rep[j]

Cluster	For j = 1 to number of representative in s_i.Rep[] Loop s'_i.Elements[j][] = Filter(s_i.Elements[j][], $\{a_1, ..., a_m\}$) If (s'_i.Elements[j][] is Null) Then //Group will be deleted Delete s'_i.Elements[j] and s'_i.Rep[j] Else if (s_i.Rep[j].annotation $\in$ s'_i.Elements[j][]) Then s'_i.Rep[j].annotation = s_i.Rep[j].annotation s'_i.Rep[j].groupSize = $\|s'_i$.Elements[j][]$\|$ Else //Need to select a new representative s'_i.Rep[j].annotation = Most recent anno in s'_i.Elements[j][] s'_i.Rep[j].groupSize = $\|s'_i$.Elements[j][]$\|$

Table 1: Fine-grained projection on annotation summaries.

be selected accordingly. There are two different semantics covered in literature [10, 13], which are both adopted in InsightNotes.

- Fine-Grained Projection ($\pi^F_{a_1,a_2,...,a_m}(R)$): Under this semantics, only the annotations on the projected attributes (as well as the tuple-level annotations) are selected and propagated. Annotations on the non-projected attributes are discarded. Therefore, π^F over tuple r is defined as follows:

$$\pi^F_{a_1,a_2,...,a_m}(r) \rightarrow r' =< a_1, a_2, ..., a_m, \{s'_1, s'_2, ..., s'_k\} >$$

where r' is the output tuple, and s'_i ($\forall\ 1 \leq i \leq k$) is the updated summary object corresponding to s_i in r after dropping the non-projected annotations. More formally, the π^F operator will update the summary objects as presented in Table 1. For the *Classifier* type, the raw annotations (only the Ids) corresponding to each class label, i.e., $Elements[j][]\ \forall j$, will be filtered (using the *Filter()* function) to drop all cell-level annotations on the non-projected attributes. The *Rep[]* array will maintain the same set of class labels, but the count of annotations assigned to each label, i.e., *Rep[j].annotationCnt*, will be updated to the cardinality of the new $s'_i.Elements[j][]$. For the *Cluster* type, we want to avoid re-clustering the annotations after filtering out the non-projected ones because this would violate the first two objectives of our strategy. Instead, our algorithm refines the existing clusters as presented in Table 1. Each group within the cluster object (Figure 1(a) shows a cluster summary object with three groups) will have its raw annotations ($s_i.Elements[][]$) filtered to drop the un-projected annotations. If a group becomes empty, then it will be deleted from both the $Elements[][]$ and $Rep[]$ arrays. Otherwise, if the representative annotation is still projected, then it will remain as the representative of its group in the new s'_i object. This simple criteria has two main advantages; first it is deterministic (unlike random selection), and second it does not require retrieving the annotations' content. Notice that the same algorithm will be used by the *summary-propagation* operator in the *Lazy-Propagation* strategy (Refer to Line 8 in Figure 5).

- Promoted Projection ($\pi^P_{a_1,a_2,...,a_k}(R)$): Under this semantics, the annotations attached to the tuple, regardless to which attribute, will propagate along with the projected attributes (as if all annotations are promoted to the tuple-level). Therefore, π^P over tuple r is defined as follows:

Merge operator: $\Omega(S_r, S_t) \rightarrow S_{rt}$
1- $S_r = \{s_1, s_2, ..., s_k\}$ // The first input set of summary objects $S_t = \{s'_1, s'_2, ..., s'_m\}$ // The second input set of summary objects $S_{rt} = \{\}$ // The output set of summary objects
2- Copy the summary objects with unique instance Ids from S_r and S_t to S_{rt} $\forall\ s_i \in S_r \mid \nexists\ (s'_j \in S_t\ \&\ s_i$.InstanceID = s'_j.InstanceID) $\rightarrow$ *copy* s_i *to* S_{rt} $\forall\ s'_i \in S_t \mid \nexists\ (s_j \in S_r\ \&\ s'_i$.InstanceID = s_j.InstanceID) $\rightarrow$ *copy* s'_i *to* S_{rt}
3- Union the summary objects having the same instance Id from S_r and S_t If (s_i.InstanceID = s'_j.InstanceID) Then – Create a new summary object s''_{ij} in S_{rt} – s''_{ij}.InstanceID = s_i.InstanceID – s''_{ij}.TupleID = null // Null for the on-the-fly tuples
Classifier: For x = 1 to number of representatives in Rep[] Loop s''_{ij}.Rep[x].classLabel = s_i.Rep[x].classLabel s''_{ij}.Elements[x][] = s_i.Elements[x][] $\cup$ s'_j.Elements[x][] s''_{ij}.Rep[x].annotationCnt = $\|s''_{ij}$.Elements[x][]$\|$
Snippet: For x = 1 to number of representatives in s_i.Rep[] Loop s''_{ij}.Rep[x] = s_i.Rep[x] s''_{ij}.Elements[x][] = s_i.Elements[x][] – Add the snippets from s'_j that does not exist in s_i to s''_{ij}
Cluster: For x = 1 to number of representatives in s_i.Rep[] Loop If s_i.Elements[x][] do not overlap with groups in s'_j Then s''_{ij}.Rep[x] = s_i.Rep[x] s''_{ij}.Elements[x][] = s_i.Elements[x][] Else //Merge $s_i[x]$ with overlapping groups (say $s'_j[k]$) s''_{ij}.Elements[x][] = s_i.Elements[x][] $\cup$ s'_j.Elements[k][] $\forall k$ s''_{ij}.Rep[x] = Most recent ann in s''_{ij}.Elements[x][] – Add s'_j's groups that did not merge with groups in s_i to s''_{ij}

Table 2: The *merge* operator on two sets of summary objects.

$$\pi^P_{a_1,a_2,...,a_m}(r) \rightarrow r' =< a_1, a_2, ..., a_m, \{s_1, s_2, ..., s_k\} >$$

where $s_i\ \forall\ 1 \leq i \leq k$ on the output tuple r' are the same summary objects as on the input tuple r.

• **Merge Operator (Ω):** Several of the relational operators, e.g., duplicate elimination, set operators, and grouping, involve merging the identical tuples together into one tuple, and hence the annotations on these tuples need to be also merged together. The commonly used semantic in annotation management systems is to union the attached annotations. Although this operation is straightforward in the case of the raw text annotations, it is more challenging in the case of the annotation summaries. In general, assume we have two data tuples r and t, where r has a set of attached summary objects S_r corresponding to r's raw annotions A_r. Similarly, t has a set of attached summary objects S_t corresponding to t's raw annotions A_t. It is worth noting that the summary objects in S_r and S_t may have different types and structures because r and t may belong to different tables having different summary instances. our objective is to compute S_{rt} which corresponds to $A_r \cup A_t$ directly from S_r and S_t.

In Table 2, we introduce the new *merge* operator $\Omega(S_r,\ S_t)$ that merges (union) two sets of summary objects S_r and S_t and produces an output set S_{rt}. The merge operator is a *logical* operator that will be used within other relational operators. In Step 2 in Table 2, the algorithm copies the summary objects having unique instance Ids from S_r and S_t to S_{rt}. These objects do not have matching counterpart objects with the same instance Id from the other set, and hence they will not change. In Step 3, the summary objects having the same instance Ids (say, s_i and s'_j) from S_r and S_t, respectively, will be merged together. The new object (s''_{ij}) will have the same summary instance Id, but the *Rep* and *Elements* arrays will be updated based on the summary type. For example, for

the *Classifier* type, the summary representatives, i.e., the classifier labels in the *Rep* array, will remain the same for s''_{ij}, and for each class label, the Ids of the raw annotations from both s_i and s'_j (in *Elements* array) will be merged together to eliminate any duplicates, and then the count for each class label is updated accordingly. For the *Cluster* type, assume that s_i has groups $s_i = \{g_1, g_2, ..., g_k\}$ while s'_j has groups $s'_j = \{g'_1, g'_2, ..., g'_m\}$, then for each group in s_i if it is not overlapping with groups in s'_j, then it will be copied as is to s''_{ij} (same elements and representative). Otherwise, the overlapping groups will be merged together, i.e., getting the union of their elements and and the representative will be the most recent annotations among the old representatives. Finally, as shown in Table 2, the remaining groups from s'_j will be added to s''_{ij}.

The merge algorithm in Table 2 is designed not only to satisfy the first two objectives of the *Summary-Aware* strategy, but also to achieve the following two crucial properties for the *Merge* operator that will help achieving the third objective.

Lemma 1: *The Merge operator retains the two properties:*

Commutativity: $\Omega(S_{r_1}, S_{r_2}) = \Omega(S_{r_2}, S_{r_1})$

Associativity: $\Omega(\Omega(S_{r_1}, S_{r_2}), S_{r_3}) = \Omega(S_{r_1}, \Omega(S_{r_2}, S_{r_3})) = \Omega(S_{r_1}, S_{r_2}, S_{r_3})$ □

Proof: The proof of Commutativity is straightforward since the algorithm in Table 2 is not sensitive to the order of S_{r_1} and S_{r_2}. The proof of Associativity relies on that the primitive operations for merging summary objects are themselves associative. For example, if S_{r_1}, S_{r_2}, and S_{r_3} have their summary objects belong to distinct summary instances, then the output result is the union of the input objects (Step 2 in Table 2), and the union operation is associative. If there are objects having the same instance Ids and will be merged, then in the case of the Classifier type, for example, the output class labels will be the same regardless of the order of the merge. Moreover, the Elements[][] arrays will be merged together using a union operation (which is associative), and hence the merge operation itself is associative. The same rules apply for the Snippet and Cluster types.

• **Duplicate Elimination and Set Operators:** Two annotated tuples $r =< a_1, a_2, ..., a_n, \{s_1, s_2, ..., s_k\} >$ and $t =< b_1, b_2, ..., b_n, \{s'_1, s'_2, ..., s'_m\} >$ are said to be identical (denoted by $r \equiv t$) iff the data part of both r and t are identical. That is:

$$r \equiv t \quad \text{iff} \quad r.a_i = t.b_i, \quad \forall\, 1 \leq i \leq n$$

The summary objects of r and t will be then merged together using the *merge* operator and attached to the output tuple, i.e., if $r \equiv t$, then the output from the duplicate elimination $\delta(r, t)$ is:

$$\delta(r, t) =< a_1, a_2, ..., a_n, \Omega(\{s_1, s_2, ..., s_k\}, \{s'_1, s'_2, ..., s'_m\}) >$$

The same semantics apply for the set operators that merge identical tuples together, e.g., *union* and *intersect* operators.

• **Grouping & Aggregation Operators:** The grouping operator $\gamma_{a_1,a_2,...,a_m,F_1(\beta_1),...F_z(\beta_z)}(R)$ groups a set of tuples of identical values w.r.t the grouping columns $(a_1, a_2, ..., a_m)$ and merges them into one tuple. The propagation of the annotation summaries under the grouping operator is based on the semantics of the *merge* operator. That is, if $r_1, r_2, ..., r_n$ belong to the same group, and they have the sets of summary objects $S_{r_1}, S_{r_2}, ..., S_{r_n}$, respectively, then the output set of summary objects, say S_g, that will be attached to the group output is:

$$S_g = \Omega(...(\Omega(\Omega(S_{r_1}, S_{r_2}), S_{r_3}), ..., S_{r_n}) = \Omega(S_{r_1}, S_{r_2}, S_{r_3}, ..., S_{r_n})$$

• **Join Operator ($\bowtie$):** The join operation $\bowtie_p (R, T)$ joins tuple $r \in R$ with $t \in T$ iff they jointly satisfy the set of predicates p. Consistent with the semantics of the other operators, the sets of summary objects on r (S_r) and t (S_t) will be merged together. That is, a joined output tuple will have a set of summary objects S_{rt} as:

$$S_{rt} = \Omega(S_r, S_t)$$

Lemma 2: *The relational operators using the Merge operator are not sensitive to the order of processing their input tuples.* □

Proof: The join operator is not sensitive to the order of the joined tuples because of the commutativity of the merge operator (Lemma 1). Moreover, the duplicate elimination, grouping, and set operators are not sensitive to the order of processing their identical tuples because of the associativity of the merge operator (Lemma 1).

If a query plan involves both a fine-grained projection and merge operations, then to guarantee that the *Summary-Aware* and *Lazy-Propagation* strategies will produce the same results the fine-grained projection has to be performed before any merge operation. Recall that the *Lazy-Propagation* performs all merges at the last operator (after the projection is done). Therefor, the *Summary-Aware* strategy always pushes the projection as early as possible. Even if some of the projected-out attributes are temporarily needed for other operations, e.g., join or grouping, then the early projection will only apply to the summary objects, and then the data projection can be applied later in the query plan.

Theorem 1: *Given a query plan P_Q under no annotation propagation, and P_{Q-lazy}, and $P_{Q-summary}$ are its extended versions under the Lazy-Propagation, and Summary-Aware strategies, respectively, then P_{Q-lazy} and $P_{Q-summary}$ are guaranteed to produce identical results.* □

Proof: The proof follows from the properties of the query operators. The selection and promoted-projection operators have no effect on the annotation summaries. Also, according to Lemmas 1 & 2 applying the merge operations in $P_{Q-summary}$ in a pipelined fashion (within one operator or across operators) will yield the same final result as performing all the merge operations at the last operator in P_{Q-lazy}. Finally, the sequence of applying the fine-grained projection and merge operations is the same in both P_{Q-lazy} and in $P_{Q-summary}$, which concludes that the final results from both query plans are identical.

Based on Theorem 1, we can also derive the following theorem.

Theorem 2: *Given a query Q and two equivalent, but different, execution plans P_Q and P'_Q generated from the query optimizer, then the annotation summaries produced from each of Strategies I, II, and III will be identical under both plans.* □

Proof: Considering Strategy I, since the annotation summaries are created at the last stage of processing, i.e., immediately before reporting to end-users, then the input tuples (data and raw annotations) to the *Summarization-Mining* operator are guaranteed to be identical under P_Q and P'_Q. And hence, the output results (data and annotation summaries) from Strategy I will be identical under both execution plans. The same proof applies to Strategy II. For Strategy III and based on Theorem 1, we can infer that this strategy also produces identical results under P_Q and P'_Q.

3.3 Zoom-In Query Processing

One of the interesting challenges that arise in InsightNotes is that: *What if the end-user is interested in zooming-in and retrieving the detailed annotations of a certain summary*. For example, in ornithological DBs, the annotations on a given table can be classified as { 'Disease', 'Anatomy', ...} as depicted in Figure 6. An end-user may be interested in retrieving the raw annotations about diseases, or the full articles of specific reported snippets.

We formulate the *zoom-in* operation as follows. Given a query Q and its corresponding answerset $A_Q = \{r_1, r_2, ..., r_x\}$, where

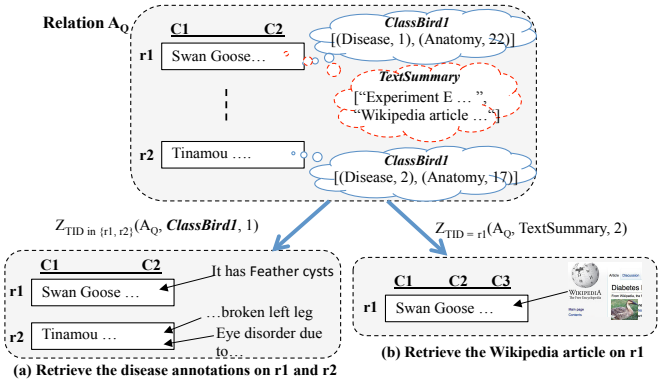

Figure 6: Examples of the zoom-in operator.

each tuple r_i has the schema containing its data values and the set of summary objects, i.e., $r_i =< a_1, a_2, ..., a_n, \{s_1, s_2, ..., s_k\} >$

We introduce a new *zoom-in* operator $\mathbb{Z}_p(A_Q, instanceId\ [, index])$, where A_Q is an input relation, p is a set of predicates over A_Q data attributes to optionally select a subset of interest from A_Q's tuples, $instanceId$ is the id of the summary instance that the user wants to retrieve its details, and the optional *index* argument specifies an index within the *Rep[]* array for which the raw annotations will be retrieved. If omitted, then all raw annotations from the summary object will be retrieved. Recall that for a summary object O, the ids of the raw annotations corresponding to a given representative $O.Rep[i]$ are stored in *O.Elements[i][]*. Thus, the output from the zoom-in operator is formalized as follows:

$$\mathbb{Z}_p(A_Q, instanceId\ [, index]) = \{r'_i =< a_1, ..., a_n, \{w_1, ..., w_m\} > \mid p(r'_i) = True\ \&\ \{w_1, ..., w_m\} = Retrieve(s.Elements[index][]), where\ s.InstanceID = instanceId\ \}$$

where each output tuple r'_i must satisfy the input predicates p, i.e., $p(r'_i)$ = *True*, s is the summary object attached to r_i having the specified *instanceId*, and $\{w_1, w_2, ..., w_m\}$ are the raw annotations corresponding to the Ids in *s.Elements[index][]*. The example in Figure 6 demonstrates how the *zoom-in* operator works. Assume that tuples r_1 and r_2 in the given relation A_Q have summary objects of type *Classifier* with instance id *ClassBird1*. Moreover, r_1 has a summary object of type *Snippet* and instance id *TextSummary*. Then, the algebraic expression $\mathbb{Z}_{TID\ in\ \{r_1,r_2\}}(A_Q, ClassBird1, 1)$ retrieves r_1 and r_2 along with the raw disease annotations as depicted in Figure 6(a) (notice that *index* 1 in the expression references the *Disease* label within the classifier). Similarly, the algebraic expression $\mathbb{Z}_{TID\ =r_1}(A_Q, TextSummary, 2)$ selects tuple r_1 along with the complete Wikipedia article corresponding to the 2^{nd} snippet within its summary object (Figure 6(b)).

3.3.1 Zoom-in Execution Optimizations

The zoom-in operator is typically a follow-up operation on previously executed queries, i.e., users first execute a query, analyze its results, and then get interested in zooming-in over specific annotation summaries. Therefore, we implement the zoom-in operator as a new stand-alone SQL command that can reference previously executed queries. To enable this mechanism, InsightNotes assigns for each executed query a unique identifier *QID* that users can reference as the 1^{st} argument in the zoom-in operator. To optimize the execution of the operator, we combine two execution modes:

Re-Execution Mode: In this mode, the system stores only the definition of the executed queries, and when the end-user issues a zoom-in operation $\mathbb{Z}_p(QID, instanceId\ [, index])$, the query corresponding to QID is re-executed. However, before the re-execution, the query is optimized by pushing down the predicates p as well as the other arguments, i.e., $instanceId$ and $index$, as much as possible in the query plan. Thus, irrelevant data tuples and summary objects are filtered out as early as possible. Although straightforward, this execution mode may involve high overhead especially if the referenced query is expensive or if multiple zoom-in operations are executed over the same query.

Materialization Mode: In this mode, the queries' results are *temporarily* materialized and stored in *temp* tables, and hence the zoom-in operations can be directly applied to these tables. However, since materializing all the executed queries is impractical, the system deploys a caching mechanism that allocates a disk space in the database, called *TempSpace*, for the temporarily materialized result sets, and queries will compete for this space using a data replacement policy. We developed a replacement policy called *RCO* (stands for Recency, Complexity, and Overhead), which is a modified version of the LRU replacement policy. First, LRU is well suited for the problem at hand since it matches the expected execution pattern of the zoom-in operations, i.e., a query is executed, followed by a sequence of zoom-in operations (if any), and then discarded. However, LRU does not take into account the complexity of the queries, i.e., the execution cost and output overhead. In contrast, the RCO policy takes into account three factors, which are: *Recency* (captured by the timestamp of the last execution or reference to a query), *Complexity* (captured by the execution time of the query—It is known since the query is executed in the system), and *Overhead* (captured by the result set size of the query). When prioritizing the queries in *TempSpace* for replacement, these factors are normalized for each query Q and then summed up to get Q's final score as follows:

Q.RecencyNorm = (currentTime - Q.lastReference) / MaxInterval
Q.OverheadNorm = Q.outputSize / MaxSize
Q.ComplexityNorm = MinExecutionTime / Q.executionTime
Q.Score = Q.RecencyNorm + Q.OverheadNorm + Q.ComplexityNorm

where *MaxInterval* is the largest interval of all available queries, i.e., $Max(currentTime - Q_i.lastReference)\ \forall i$, *MaxSize* and *MinExecutionTime* are the largest result set size, and the smallest execution time among the available queries, respectively. Note that each of these normalized factors produces a measure between $(0, 1]$, where closer to 0 means higher priority to stay in *TempSpace*. Thus, the final score is within the range of $(0, 3]$ with score 3 to be the highest priority to be replaced. Now, given a new query Q_{new}, it will be added to *TempSpace* iff we can remove one (or more) existing queries to make room for Q_{new}'s output such that the final score of each of these removed queries is larger than $Q_{new}.Score$. Otherwise, Q_{new} is not materialized.

4. EXTENSIBILITY AND MAINTENANCE OF ANNOTATION SUMMARIES

In this section, we present several issues and optimizations related to the creation and maintenance of the *Summary Repository*, which takes place at the time of adding or deleting annotations.

Properties of Summarization Algorithms: The only property that we mandate in the summarization algorithms is to be *incremental*, i.e., given a new annotation on a specific tuple the summaries can be updated incrementally without the need for re-building. This property is critical for efficient execution since annotations are continuously added to the data tuples. For the Snippet and Classifier summary types, the incremental processing is inherent in most of their algorithms, e.g., text summarization in [18], and classification techniques in [9]. For the clustering techniques, there are numerous incremental algorithms for clustering evolving data [2, 17, 23, 24]. Several of these algorithms have shown to achieve the *stability*

property, which is that the created clusters are stable (with a small variance) under the different permutations of a given set of input objects [23, 24]. Therefore, any of these algorithms can be efficiently integrated in InsightNotes. InsightNotes also mandates the generated representatives from the summarization techniques to adhere to the structure of representatives (the Rep[] array) presented in Section 2.

Maintenance Algorithm: The algorithm for adding new annotations and maintaining their summaries is presented in Figure 7. This algorithm is carried out by the *Annotation Manager*, which is responsible for adding the annotations and for maintaining the *Raw-Annotation Repository* and *Summary Repository* up-to-date. As inputs, the algorithm takes a new annotation *A* and a simple *Select-Project* query *Q* on a user relation *R*, where *A* will be attached to *Q*'s output. It is worth highlighting two key optimizations in the algorithm that significantly reduce the overheads involved in adding annotations and updating their summaries. The first optimization is that the Annotation Manager has two execution modes, i.e., *Eager Attachment* and *Lazy Attachment*. If query *Q* can be efficiently executed, i.e., using an index and it has estimated high selectivity, then annotation *A* will be instantaneously attached to its tuples (Lines 3-4), and the annotation summaries will be updated (Lines 5-13). If *Q* would require an expensive table scan, then the Annotation Manager will store *A* and *Q* in a system table for later execution with the aim for buffering multiple of these annotations, and then attaching them using a single table scan (Lines 14-20). In the Lazy Attachment mode, the buffered annotations will be refreshed on relation *R* when a user's query touches *R* (Line 16).

The second optimization is used when updating the summaries (Lines 5-13), where the Annotation Manager checks the properties of each summary instance linked to relation *R*. If a given instance has both of its properties *AnnotationInvariant* and *DataInvariant* set to True, then this means that the summarization technique works independent of any other annotations or the data content. Therefore, the corresponding summarization technique is called once independent of the number of tuples returned from *Q* (Lines 7-8), and the summarization output is then attached to all of *Q*'s output tuples in the Summary Repository (Line 9). Otherwise, the summarization technique has to be called for each tuple in *Q* (Lines 10-13 & 19-20).

Extensibility Feature: InsightNotes is not limited to specific algorithms or techniques. Instead the DB admins can integrate the desired mining or summarization techniques into InsightNotes as UDFs, e.g., PostgreSQL, which is the underlying database engine of InsightNotes , supports UDFs in different languages such as SQL, C, and Java. The metadata information, e.g., training datasets, models, or configuration parameters, needed for a given technique can be passed—either directly or as pointers to locations— through the *instance-level* properties attached to each summary instance (Refer to Figure 2). For example, assume we want two Naive Bayes classifiers, one for classifying annotations as *{approve, refute, neutral}*, and the other one for classifying them as *{provenance, comment}*. Then, the Naive Bayes algorithm will be implemented as a UDF in the database, and the two statistical models for the classifiers will be also stored in the database, e.g., tables *Model-1* and *Model-2*. Then, two summary instances will be created, one for each classifier, and they will both refer to the same UDF, but they will differ in the instance-level properties that point to the location of the input model and that define the class labels. At the system level, InsightNotes provides a generic interface function, called *getPropertyValue(<propertyName>)*, that can be called from the UDFs to return the value of the given property name within the context of the active summary instance. Thus, the UDFs can operate differently according to properties defined in each summary instance.

Adding New Annotation

Inputs: // annotation A will be attached to the output from query Q
- The new annotation ***A*** (timestamp, curator, value)
- Select-Project query ***Q*** on relation ***R***

1. ***Eager Attachment (If Q has efficient execution plan using an index on R):***
2. // Insert the raw annotation
3. - Execute Q to find the target tuples being annotated → $\{r_1, r_2, \ldots, r_n\}$
4. - Insert A into the Raw-Annotations Repository over $\{r_1, r_2, \ldots, r_n\}$
5. // Update the summaries
6. - For each summary instance (say *s*) on R Loop
7. - If (s.AnnotationInvariant = True) and (s.DataInvariant = True) Then
8. - Execute s.FunctionID on A to produce A's summary
9. - Update (or create if not exists) the summary object of s on $\{r_1, r_2, \ldots, r_n\}$ in the Summary Repository.
10. - Else
11. - For each tuple (say r_x) in $\{r_1, r_2, \ldots, r_n\}$ Loop
12. - Execute s.FunctionID on (A, r_x and/or the summary object) to produce A's summary on r_x
13. - Update (or create if not exists) the summary object of s on r_x in the Summary Repository.
14. ***Lazy Attachment (If Q's execution plan is table-scan on R):***
15. - Buffer A and its query Q in a system table *Buff*
16. - When a user query is issued over R, then R's annotations are refreshed
17. - Scan R once and identify the tuples satisfying each query in Buff
18. - Attach the annotations to their corresponding tuples
19. - For each annotation A in Buff Loop
20. - Execute Lines 5-13 to update the summaries.

Figure 7: Creation and maintenance of annotation summaries.

5. RELATED WORK

Annotation management is widely applicable to a broad range of applications, yet it gained significant importance within the context of scientific applications [3, 15, 19]. Therefore, to help scientists in their scientific exterminations and to boost the discovery process, several generic annotation management frameworks have been proposed for annotating and curating scientific data in relational DBMSs [4, 8, 13, 14, 22]. Several of these systems, e.g., [4, 8, 13, 22], focus on extending the relational algebra and query semantics for propagating the annotations along with the queries' answers at query time. The work in [10] proposes compact storage mechanisms for storing multi-granular annotations at the raw-, cell-, column-, and table-levels, as well as defining behaviors for annotations under the different database operations. Other systems have addressed special types of annotations, e.g., [7, 12]. For example, the work in [7] addresses the ability to annotate the annotations, and hence it proposes a hierarchal approach that treats annotations as data.

Although the above systems provide efficient query processing for annotations, they all share a common limitation, which is that they all manipulate the raw annotations, and they report to end-users all ones relevant to their queries even if each output tuple has hundreds of annotations. There are no previous attempts to provide advanced and optimized query processing on top summarized forms of the annotations. Thus, as annotations scale up over time, existing systems will fall short in providing real insights and useful information to end-users. The InsightNotes system is proposed to address such critical limitations.

Scientific systems and workflows have also leveraged the concept of semantic and ontology-based annotations, e.g., [3, 5, 19]. For example, the work in [19] uses the annotations drawn from a specific ontology to relate and measure the similarity among the scientific entities in the database. And hence, it uses the annotations as a similarity metric among the data entires. In contrast, the work in [3, 5] uses semantic annotations to either summarize complex workflows [3], or help in building and verifying

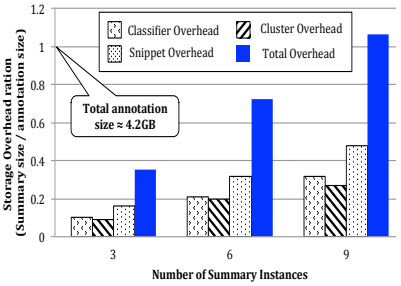

Figure 8: Storage overhead.

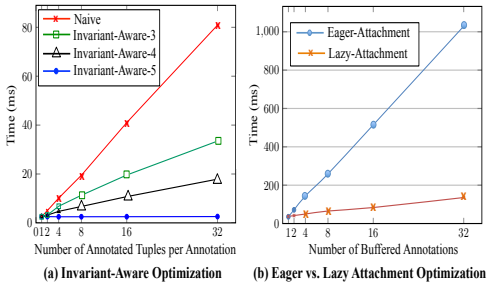

Figure 9: Maintenance Overhead.

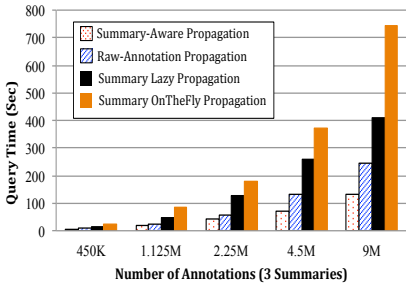

Figure 10: Propagation Performance (SP).

workflows [5]. Therefore, these systems are based on workflow- and process-centric annotations, e.g., annotations capturing the semantics of each function in a workflow, the structure of their input and output arguments, etc. In contrast, InsightNotes manages data-centric annotations that are independent from how the data is processed—As highlighted in the motivating applications in Section 1. Hence, the objective and approaches proposed in InsightNotes are different from those in the other systems.

In the domains of e-commerce, social networks, and entertainment systems, e.g., [11], the annotations are usually referred to as *tags*. These systems deploy advanced mining and summarization techniques for extracting the best insight possible from the annotations to enhance users' experience. However, unlike relational DBs, the retrieval mechanisms in these systems are typically straightforward and do not involve complex processing or transformations, i.e., objects (products in Amazon or movies in Netflix) are usually queried as individual instances without going through a complex pipeline of query operators, e.g., projection, join, grouping, and aggregation operators. Therefore, no advanced query processing is required over the annotations summaries once created.

6. EXPERIMENTS

InsightNotes is developed on top of an existing annotation management system. The experiments are executed using a Dell OptiPlex 990 Desktop machine having Intel dual core i5 2400 Processor (3.1GHz, 6M), 4GB DDR3 memory, and 250GB SATA hard drive.

Application Datasets: We use real-world annotated scientific database available from the AKN system (http://www.avianknowledge.net/) that stores information related to 10s of thousands of birds worldwide. The database schema consists of several tables including the main table *Birds*, and several dimension tables for regions, taxonomies, and synonyms. We used the *Birds* table since it is the largest table and it is the main annotated tables by 100s of thousands of bird watchers and scientists. The table consists of 45,000 tuples holding birds' information over 12 attributes, e.g., scientific name, Ids across different systems, description, genus, family, and habit. The table size in the database is 450MBs. The collected number of annotations is 9.2×10^6 that describe a wide range of bird related information, e.g., color, body shape or weight, certain behavior or sound, eating habits, geographic location, or observed diseases. All tuples in the Birds table are annotated with the following statistics on the number of annotations attached to each tuple: *avg = 204*, *min = 62*, *max = 321*, and *stdev= 76*. The total size of the annotation table is 4.2GBs. All annotations in the dataset are at the tuple-level, i.e., attached to the entire tuple. For the purpose of our experiments, we developed a tool that attaches some of these annotations to specific attributes in the tuples. The tool searches for the column names in a given annotation, and based on that, it attaches an annotation to its referenced columns.

Summarization Techniques: We integrated several data mining techniques within InsightNotes for the annotation summarization. We used the Naive Bayes [9] technique for annotation classification, the CluStream technique [2] for clustering the annotations in an incremental way, and the LSA (Latent Semantic Analysis) technique [18] for text summarization and snippet creation. In the experiments, we created several summary instances that use the same underlying techniques but differ in their outputs. For example, several classifiers have been instantiated to classify annotations as either {*'Question", 'Answer', 'Comment*}, {*'Disease', 'Anatomy', 'Behavior', 'Other'*}, or {*'Geographic', 'Non-Geographic'*}. Moreover, several snippet instances have been created that differ in whether or not they are data dependent, the threshold above which an annotations is considered a large-object, and the snippet size to create.

Storage Overhead: In Figure 8, we present the storage overhead associated with the annotation summaries. The *y-axis* shows the ratio of the summaries' size to the size of the raw annotations ($\simeq$ 4.2 GBs). In the experiment, we vary the number of annotation summaries over 3, 6, and 9 divided equally between the three summary types: *Snippet*, *Classifier*, and *Cluster*. As the figure shows, the average size of each summary type is around 13% of the raw annotations' size, where snippets have slightly higher overhead while classifiers have the least overhead. The reason is that the representatives in the case of the *Snippet* type are larger than those in the *Cluster* and *Classifier* types. Moreover, the representatives in the *Classifier* type are smaller than those in the *Cluster* type (The class labels in classifiers are just a single word). And as expected, as the number of summary instances increases, the storage overhead increases proportionally.

Creation and Maintenance of Annotation Summaries: We proposed in Figure 7 two key optimizations for scalable creation and maintenance of annotation summaries. In Figures 9, we evaluate the effectiveness of these two optimizations. The *x-axis* in Figure 9(a) shows the number of tuples that a single annotation can be attached to (ranging from 1 to 32). In this experiment, we link five summary instances (two Snippet, two Classifier, and one Cluster) to the database relation. The native approach (labeled as *"Naive"*) would compute the summaries of the new annotation with each related data tuple regardless of whether or not the summary instances are defined as *DataInvariant* or *AnnotationInvariant*. In contrast, the proposed algorithm (labeled as *"Invariant-Aware"*) will take these properties into account. Therefore, if all five summary instances are *DataInvariant* and *AnnotationInvariant*, then the summaries are computed only once for each annotation, and then the summary objects of each related tuple are updated (See label *"Invariant-Aware-5"*). We also studied the cases between these two extremes, e.g., when one or two of the five summary instances are data dependent while the rest are *DataInvariant* and *AnnotationInvariant*. The performance of these cases is illustrated in Figure 9(a) as *"Invariant-Aware-4"* (For one data-dependent instance) and *"Invariant-Aware-3"* (For two data-dependent instances). The results indicate that even in the cases where not all summary in-

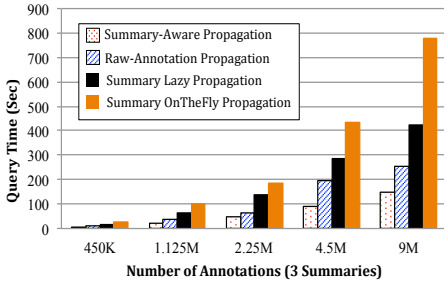

Figure 11: Propagation Performance (SPG).

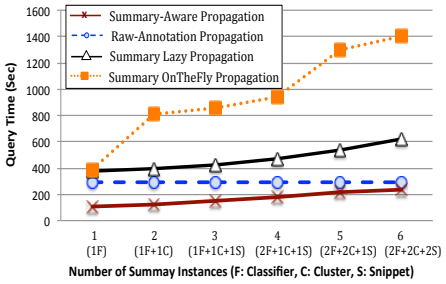

Figure 12: Propagation vs. Instance Num.

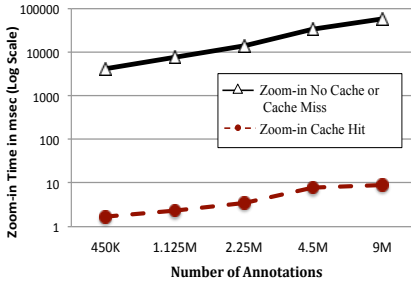

Figure 13: Zoom-in Performance.

stances are *DataInvariant* or *AnnotationInvariant*, the proposed *"Invariant-Aware"* algorithm still achieves significant savings.

The performance of the *Eager* vs. *Lazy* evaluation approaches is presented in Figures 9(b). We vary the number of annotations added to the data table between two consecutive user's queries between 1 and 32 as illustrated over the *x-axis*, and the *y-axis* shows the total time needed to add these annotations and update their corresponding summaries. In the Eager-Attachment approach each added annotation will trigger a table scan to select the target tuples to be annotated and to update their summaries. In contrast, in the Lazy-Attachment approach only one table scan is needed (at the time of the user's query) to attach all the buffered annotations to their data tuples. Notice that in the Lazy-Attachment approach a user query may be penalized and get delayed until all buffered annotations are refreshed. Therefore, to avoid potential large delays, InsightNotes uses a threshold (currently 50), which if reached, then the buffered annotations will be automatically refreshed even without the presence of a user's query.

Propagation at Query Time: In Figures 10 and 11, we study the propagation performance of the annotation summaries under different query types. Figure 10 shows the performance of a *Select-Project* query, while Figure 11 shows the performance of a *Select-Project-Grouping* query. We vary the number of annotations in the database over the range between 450,000 (10 annotations per tuple) to 9x10^6 (200 annotations per tuple) as depicted over the *x-axis* in the figures. The number of summary instances linked to the database relation are three (one of each type, Snippet, Cluster, and Classifier). The four techniques in comparison are the *Raw-Annotation Propagation*, which is a standard propagation of the raw annotations without any summaries involved, and the three summary propagation strategies proposed in Section 2.1.

The results in Figure 10 show that the *Summary-Aware* strategy is the most efficient even when compared to the standard propagation of raw annotations. The reason is that the size of the summaries is much smaller than the size of the raw annotations (around 40% as illustrated in Figure 8), and hence the summary propagation is up to 2x faster. Moreover, the summary-aware strategy manipulates the summaries in an efficient pipelined fashion without the need for any special-purpose expensive operators. In contrast, the *Lazy Propagation* strategy uses the standard propagation as a black-box, and on top of that it adds the overheads involved in the newly introduced *summary-propagation* operator. The *On-The-Fly Propagation* strategy is the most expensive one since the mining algorithms are executed at query time. The figure illustrates that it is 7x slower than the *Summary-Aware* strategy. The performance under a more complex SPG query inherits the same trend as depicted in Figure 11 with the exception that the query execution is slightly more expensive. The figure confirms that the summary-aware is around 2x, 3x, and 7x faster than the standard, lazy, and on-the-fly propagation strategies, respectively.

In Figure 12, we study the propagation performance while varying the number of summary instances linked to the database relation. In this experiment, the number of annotations is set to 9x10^6 and the number of instances varies from 1 to 6 as illustrated in the figure. We use the same SPG query used in Figure 11. The reference fixed performance is the propagation of the raw annotations (*Raw-Annotation Propagation*), which is independent from the summary instances. As the figure shows, the *Summary-Aware* strategy scales better than the *Lazy Propagation* and *On-The-Fly Propagation* strategies when scaling up the number of summary instances. Moreover, the *On-The-Fly* strategy is more sensitive to the mining algorithms used since they are executed at query time.

Quality of Generated Annotation Summaries: The three summary propagation strategies, i.e., *On-The-Fly*, *Lazy-Propagation*, and *Summary-Aware*, will generate identical summaries w.r.t the *Snippet* and *Classifier* types. The only difference will be in the *Cluster* type summaries since the latter two strategies will generate an approximated cluster summaries compared to the *On-The-Fly* strategy. The reason is that in the latter strategies some operators may drop annotations from the pre-computed clusters, e.g., the *fine-grained projection* operator, while others may merge pre-computed clusters, e.g., the *join* and *grouping* operators. In contrast, the *On-The-Fly* strategy will build the clusters from the raw annotations only at the last stage of processing. To measure the closeness (similarity) of the approximated clusters to those generated from the *On-The-Fly* strategy, we used the Rand Index [20] that takes two groups of clusters and returns a measure of their similarity between [0,1], where closer to 1 means higher similarity. In the following table, we report the RI measures over the two types of queries we used in Figures 10 and 11, namely a select-project (SP) query, and a select-project-group (SPG) query. We used the CluStream technique [2] with varied K parameter (# of clusters), and we calculated the average quality over two sizes of tuples 10, and 100 as presented in Table 3. The results show that the quality ranges between 70%-75% in most cases, and it gets slightly better with large K parameter. In many applications, especially with large-scale annotations, this approximation can be acceptable to achieve better scalability and queries' response time.

	SP Query			SPG Query		
Tuple Count	K= 5	K=7	k=10	K=5	K=7	K=10
Avg over 10	0.75	0.75	0.75	0.69	0.72	0.74
Avg over 100	0.71	0.73	0.75	0.68	0.71	0.76

Table 3: The Rand-Index Measure of Cluster Summaries.

Zoom-in Query Processing: In Figure 13, we illustrate the effectiveness of the materialization and caching of the query results to answer a zoom-in query vs. the re-execution of the user's query. In this experiment, we use the same SPG query used in Figure 11, and the number of summary instances is set to 3 (one of each type). The zoom-in operation will retrieve the raw annotations classified as *'Anatomy'*. As the figure shows, if the results of the user's query is not in the cache, i.e., *Cache Miss*, then the query will be re-executed (which is in the order of 10s of seconds). Whereas, if the results are in the cache, then we only need to probe the raw-

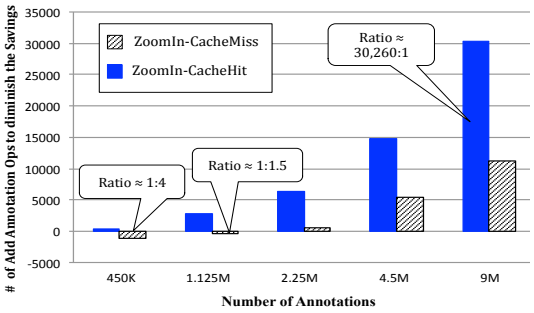

Figure 14: Summary-Aware vs. Standard Propagation: Total Savings-to-Overheads ratio.

annotation table (using an index) to retrieve the detailed information of specific annotations (which is in the order of milliseconds). In practice, the huge savings from the caching will even intensify since multiple zoom-in operations can be performed on the same query over a short period of time.

Overall Savings vs. Overheads: Although the *Summary-Aware* strategy has shown superior performance in query response time, it encounters overheads in two operations: (1) The creation and maintenance of the annotation summaries, and (2) The zoom-in operations. Thus, in the following experiment (Figure 14) we study the tradeoff between the total savings vs. overheads compared to the standard propagation (no summaries). We consider the SP query presented in Figure 10, and we assume that the saving achieved by the *Summary-Aware* strategy in the query's response time is denoted by S. Then, we assume that users will perform, on average, 20 zoom-in operations on the results, which adds an overhead denoted by Z. Therefore, the overall saving becomes *"S - Z"*. In Figure 14, we consider the two cases where the 20 zoom-in operations will have a cache hit (and thus Z is very small), and the case where the first zoom-in operation will be a cache miss, but the rest will be a cache hit (and thus Z will be larger).

Since inserting each new annotation adds some overhead for maintaining the summaries, we measured in Figure 14 the number of annotations, say X, that need to be inserted to balance the saving *"S - Z"*. For example, in the case of the largest dataset ($9x10^6$ annotations), the X is 30,260 (in the case of cache hits) and 11,300 (in the case of a cache miss). The figure shows that this ratio *X (annotation): 1 (query)* is large in most cases, which means that in typical applications the overall savings are higher than the overheads. The only extreme cases where the overheads are higher than the savings are for the smallest datasets under a cache miss for the zoom-in operation, e.g., for the size of 450K annotations, the overhead from adding one annotation and 20 zoom-in operations—having one cache miss— can be redeemed after 4 users' queries having no cache-miss zoom-in operations. These extreme cases can be even avoided with the use an appropriate cache size combined with the proposed *RCO* replacement policy.

7. CONCLUSION

We proposed the *InsightNotes* system for exploiting and propagating the annotations in relational DBs in novel ways that have not been addressed before. The novel features of InsightNotes include: (1) The integration of mining and summarization techniques with the annotation management with the objective of creating concise and meaningful representations of the raw annotations, (2) The development of summary-aware query processing engine that enables efficient and seamless manipulation of the annotation summaries within the query pipeline, and (3) The zoom-in query processing for retrieving the raw annotations when desired. We introduced several optimizations in each of these features for efficient execution and scalable performance. InsightNotes is developed on top of an existing annotation management system. The experimental evaluation indicates the practicality of InsightNotes's design and the effectiveness of its proposed optimizations.

8. REFERENCES

[1] eBird Trail Tracker Puts Millions of Eyes on the Sky. `https://www.fws.gov/refuges/RefugeUpdate/MayJune_2011/ebirdtrailtracker.html`.

[2] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A Framework for Clustering Evolving Data Streams. In *VLDB*, pages 81–92, 2003.

[3] P. Alper, K. Belhajjame, C. Goble, and P. Karagoz. Small Is Beautiful: Summarizing Scientific Workflows Using Semantic Annotations. In *IEEE BigData Congress*, pages 318–325, 2013.

[4] D. Bhagwat, L. Chiticariu, and W. Tan. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[5] S. Bowers and B. LudŁscher. A Calculus for Propagating Semantic Annotations through Scientific Workflow Queries. In *In Query Languages and Query Processing (QLQP)*, 2006.

[6] P. Buneman and et. al. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

[7] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, pages 177–188, 2013.

[8] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.

[9] P. R. Christopher D. Manning and H. Schutze. Book Chapter: Text classification and Naive Bayes, in Introduction to Information Retrieval. In *Cambridge University Press*, pages 253–287, 2008.

[10] M. Eltabakh, W. Aref, A. Elmagarmid, and M. Ouzzani. Supporting annotations on relations. In *EDBT*, pages 379–390, 2009.

[11] A. Gattani and et. al. Entity extraction, linking, classification, and tagging for social media: a wikipedia-based approach. *Proc. VLDB Endow.*, 6(11):1126–1137, 2013.

[12] W. Gatterbauer, M. Balazinska, N. Khoussainova, and D. Suciu. Believe it or not: adding belief annotations to databases. *Proc. VLDB Endow.*, 2(1):1–12, 2009.

[13] F. Geerts and et. al. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, pages 82–93, 2006.

[14] F. Geerts and J. Van Den Bussche. Relational completeness of query languages for annotated databases. In *Proceedings of the 11th international conference on Database Programming Languages (DBPL)*, pages 127–137, 2007.

[15] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[16] Q. Li, A. Labrinidis, and P. K. Chrysanthis. ViP: A User-Centric View-Based Annotation Framework for Scientific Data. In *Proceedings of the 20th international conference on Scientific and Statistical Database Management (SSDBM)*, pages 295–312, 2008.

[17] Y.-B. Liu, J.-R. Cai, J. Yin, and A. W. Fu. Clustering Text Data Streams. *Journal of Computer Science and Technology*, 23(1):112–128, 2008.

[18] A. Nenkova and K. McKeown. A Survey of Text Summarization Techniques. In *Book: Mining Text Data*, pages 43–76, 2012.

[19] G. Palma and et. al. Measuring Relatedness Between Scientific Entities in Annotation Datasets. In *International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, pages 367:367–367:376, 2013.

[20] W. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[21] M. Stonebraker and et. al. Data Curation at Scale: The Data Tamer System. In *CIDR*, 2013.

[22] W.-C. Tan. Containment of relational queries with annotation propagation. In *DBPL*, 2003.

[23] S. Young and et. al. A Fast and Stable Incremental Clustering Algorithm. In *International Conference on Information Technology: New Generations*, pages 204–209, 2010.

[24] S. Zhong. Efficient Streaming Text Clustering. *Neural Networks*, 18(6), 2005.