Elevating Annotation Summaries To First-Class Citizens In InsightNotes

Karim Ibrahim, Dongqing Xiao, Mohamed Eltabakh

Computer Science Department, Worcester Polytechnic Institute (WPI)
100 Institute Rd., Worcester, MA, USA
{kaibrahm, dxiao, meltabakh}@cs.wpi.edu

ABSTRACT

Most scientific and modern applications generate—in addition to the base data—valuable annotations and metadata information at unprecedented scale and complexity. Such annotations warrant the need for advanced annotation management techniques that not only propagate the raw annotations to end-users, but also mine, summarize, and extract useful knowledge from them. Towards this goal, we proposed the *InsightNotes* system, the first summary-based annotation management engine in relational databases [22]. InsightNotes relies on creating concise representations of the raw annotations, called *annotation summaries*. InsightNotes addresses several unique challenges related to the maintenance, propagation, and zooming of these summaries. However, a key limitation is that the annotation summaries are treated as *propagate-only* (*report-only*) objects that cannot be directly queried or manipulated. This limitation hinders higher-level applications from applying complex processing over both the base data and its attached annotation summaries even within a single query. In this paper, we propose new extensions to InsightNotes for treating the annotation summaries as *first-class citizens*. We address the challenges of: (1) Developing new manipulation functions and query operators specific for the annotation summaries, (2) Designing summary-based index structures and access methods for efficient retrieval and predicate evaluation, and (3) Extending the query optimizer to optimize queries accessing both the data and the annotation summaries. The proposed extensions not only make it feasible to natively query and manipulate the annotation summaries, but also achieve more than two orders of magnitude speedup in query evaluation.

1. INTRODUCTION

Metadata—usually referred to as *"annotations"*— is gaining an increasing importance in most modern database applications as a valuable source of information. Applications in many science domains, e.g., in biology, healthcare, earth sciences, and ornithology, create and manage annotations and metadata information in orders of magnitude larger than the base datasets as reported in [5, 20, 22]. For example, according to the *geneontology.org* website, several biological databases, e.g., Genobase (http://ecoli.naist.jp/GB8/), EcoliHouse (http://www.porteco.org/), and UniProt (http://www.ebi.ac.uk/uniprot), manage annotations in a 10x scale compared to the number of genes and proteins in the database. Moreover, in ornithological databases, e.g., DBRC (http://www.dbrc.org.uk/), and AKN (http://www.avianknowledge.net/), the number of annotations collected from the bird watchers and scientists all over the world is around 200x larger than the number of birds' collection stored in these repositories [1].

It is not only the scale of annotations that poses challenges, but also the need for transparent processing and propagation of annotations, and their combinatorial relationship with the data, e.g., annotations can be attached to single table cells (attributes), rows, columns, arbitrary sets and combinations of them, or even attached to sub-attributes. That is why annotation management has been extensively studied in RDBMSs to address some of these challenges [4, 7, 11, 14, 17, 21]. However, all of the existing techniques have the common limitation of manipulating only the raw annotations, and hence reporting back to end-users 100s of annotations attached to each output tuple. Nevertheless, any advanced processing of mining, summarizing, and extracting useful knowledge from the annotations is entirely delegated to end-users.

As a first step towards addressing the above limitations, we proposed the *"InsightNotes"* system, *a summary-based annotation management engine in relational databases* [22]. InsightNotes is based on integrating data mining and summarization techniques with annotation management in novel ways with the objective of creating concise and meaningful representations of the raw annotations, called *"annotation summaries"*. For example, the R.H.S in Figure 1 illustrates a data tuple with 100s of attached raw annotations, while the L.H.S illustrates the tuple with its attached summary objects using InsightNotes. The summary objects include, for example, Classifier-type objects, e.g., *ClassBird1* and *ClassBird2*, that classify the raw annotations into user-defined classes, Snippet-type objects, e.g., *TextSummary1*, that summarize the attached big articles and report snippets on each, and Cluster-type objects, e.g., *SimCluster*, that group similar annotations into groups and reports only a representative from each group. An overview on the system will be presented in Section 2.

1.1 Case Study: Effectiveness and Motivation

We performed a usability case study to demonstrate the effectiveness of InsightNotes and motivate the new extensions proposed in this paper. We used a small subset of 100 data tuples from the AKN ornithological database, each has a number of raw annotations ranging between 75 to 380. The annotations describe anything related to birds, e.g., color, body shape or weight, certain behavior or sound, eating habits, geographic location, or observed diseases. And then, we asked 20 students to query the data, and an-

© 2015, Copyright is with the authors. Published in Proc. 18th International Conference on Extending Database Technology (EDBT), March 23-27, 2015, Brussels, Belgium: ISBN 978-3-89318-067-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

10.5441/002/edbt.2015.06

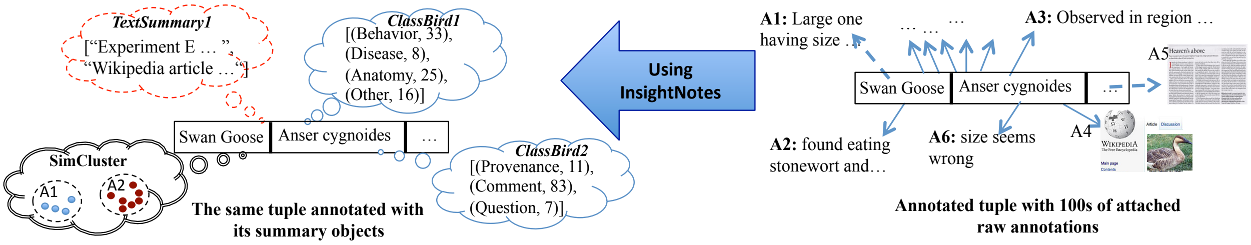


Figure 1: Summary-Based Annotation Management in InsightNotes.

Query Semantics	# Qualifying data tuples	InsightNotes Group	Raw-Annotations Group
Q1: Report the *disease-related* annotations attached to birds with name like "Swan*".	5	Time: 47 sec Accuracy: 100%	Time: 21 mins False Positives: 17% False Negatives: 25%
Q2: Aggregate based on the bird's family column, and report the number of behavior-related information on each group.	3	Time: 47 sec Accuracy: 100%	Time: 45 mins False Positives: 18% False Negatives: 34%
Q3: Report the data tuples sorted based on the number of attached disease-related annotations	100	Time: 5.2 mins Accuracy: 100%	-----

Figure 2: Usability Case Study using InsightNotes.

swer the three questions highlighted in Figure 2. These are simple annotation-based analytical queries that scientists or end-users may ask over their datasets. Half of the students use the InsightNotes engine, while the other half uses an existing annotation management engine that reports the raw annotations [11]. We then measured the average time taken by each group (including writing the query) as well as the results' accuracy.

To answer $Q1$, the InsightNotes group needs to submit a single SQL query to get the 5 expected data tuples (similar to the L.H.S in Figure 1). And then, they need to issue another follow-up command, i.e., a zoom-in command, to retrieve the raw disease-related annotations over these tuples. In contrast, the *Raw-Annotations* group will get the 5 tuples along with their raw annotations (similar to the R.H.S in Figure 1). And then, they need to manually read the annotations and extract the desired ones. It took them, on average, 21 minutes and they reported the results with high false-positive and false-negative ratios as indicated in the figure.

To answer $Q2$, the InsightNotes group needs to only retrieve the number of the behavior-related annotations from the answer, i.e., *ClassBird1.Behavior*. It took them few seconds for writing and executing the query. In contract, the other group took very long time and still produced erroneous results—Notice that $Q2$ is an aggregation query, and thus each output tuple may have many annotations collected from multiple base tuples.

The $Q3$ query is more challenging because InsightNotes does not provide mechanisms for sorting the data based on their attached summaries. Thus, the InsightNotes group needed to go over the 100 reported tuples, and manually sort them according to the *ClassBird1.Disease* field. For the other group, it was not even feasible to analyze 100s of annotations over each of the reported tuples to figure out the number of disease-related annotations, and then sort based on that.

1.2 Limitations and Proposed Extensions

The results from the our study show that InsightNotes opens a promising direction for better understanding of large-scale annotations and extracting useful knowledge from them. However, the results also show that InsightNotes has the critical limitation of treating the annotation summaries as *"propagate-only objects"*. This limitation hinders the applications from mixing operations over both the data content and annotation summaries even within a single query (Refer to $Q3$ in Figure 2). In this paper, we propose extending the InsightNotes system by elevating the annotation summaries to be first-class citizens, where end-users and applications can manipulate them in various ways, e.g., selecting, joining, or ordering the data tuples based on their attached annotation summaries. To build such full-fledged summary-based annotation management engine, we propose the following contributions:

- **Seamless Manipulation of Diverse Summary Types:** InsightNotes supports three types of summarization techniques, i.e., clustering, classification, and text summarization. And hence, the summary objects attached to the data tuples can have diverse types, structures, and properties (Refer to Figure 1). Therefore, we propose manipulation functions at different granularities, e.g., at the *tuple-level* to manipulate the entire set of attached summary objects, and at the *object-level* to manipulate the individual summary objects according to their types.

- **Summary-Based Query Processing:** We propose building an extended query engine, where end-users can process both the data and their attached annotation summaries seamlessly in a single query plan. For example, $Q3$ in Figure 2 involves a *summary-based ordering* operation. Another query may be interested in retrieving only the data tuples with zero provenance-related annotations, i.e., `ClassBird2.Provenance` = 0, which involves a *summary-based selection* operation. Therefore, we extend the InsightNotes's query engine by introducing new summary-based query operators, e.g., filter, selection, join, and sort, that operate on the summaries' content. We define their algebraic semantics and integrate them with the standard operators in a single query plan.

- **Efficient Access Methods and Retrieval Mechanisms:** Applying predicates and operators on top of the annotation summaries warrants the need for efficient retrieval mechanisms and indexing techniques to achieve scalable performance. For example, how could the system efficiently answer the two queries mentioned above, i.e., a selection query based on `ClassBird2.Provenance = 0`, and an ordering query based on `ClassBird1.Disease`. We propose summary-based indexing techniques that achieve efficient execution for the summary-based queries, while retaining the optimal summary-propagation performance.

- **Extended Summary-Based Query Optimizer:** The integration between the summary-based and the standard SQL operators within a single query engine opens several new opportunities for query optimization. For example, one query may now involve joins and selections based on both the data and the summaries, and thus the query optimization becomes even more challenging. Therefore, we introduce several new equivalence and transformation rules as well as an extended cost model that guide the query optimizer in generating efficient execution plans.

• **Realization and Evaluation:** We developed the proposed extensions within the InsightNotes prototype engine [22]. The experimental analysis demonstrates the value-added functionalities of directly manipulating and querying the annotation summaries, e.g., enabling a seamless expression of more complex annotation-based analytical queries, and the significant performance gain from the proposed optimizations.

The rest of the paper is organized as follows. In Section 2, we overview the InsightNotes system. In Section 3, we present the new summary-based functions and query operators. Sections 4 and 5 introduce the summary-based indexing scheme, and the extended query optimizer, respectively. The experimental evaluation is presented in Section 6, while the related work is presented in Section 7. Finally, the conclusion remarks are included in Section 8.

2. OVERVIEW ON InsightNotes SYSTEM

InsightNotes addresses several challenges related to managing the annotation summaries, which include: (1) Designing an extensible engine where domain experts and database admins can define how to summarize and mine their annotations, (2) Developing efficient and incremental mechanisms for the maintenance of annotation summaries to scale up with large number of annotations, (3) Extending the query engine and relational algebra to operate on and propagate the annotation summaries along with the queries' answers, and (4) Building zoom-in query processing mechanisms that enable end-users to zoom-in and retrieve the raw annotations of specific summaries of interest. In this section, we overview the basic functionalities of InsightNotes needed for this paper.

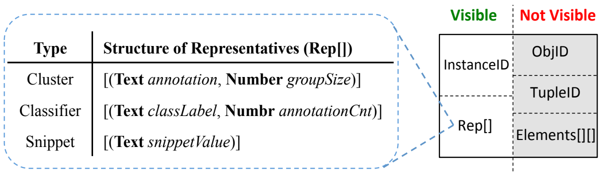
2.1 InsightNotes's Data Model

The system supports three widely-used families (types) of mining and summarization techniques, which are: *Text Summarization*, *Clustering*, and *Classification* techniques. The system is extensible such that the database admins can customize these techniques—and instantiate what is called *Summary Instances*—to fit their domains and produce the desired summaries. Each user relation R can be linked to as many summary instances as needed. For example, Figure 1 illustrates Table `Birds` having four summary instances linked to it (2 Classifiers, 1 Snippet, and 1 Cluster). Therefore, the raw annotations attached to each data tuple in this table (the R.H.S) will be summarized according to these four summary instances. This will result in creating the *Summary Objects*, which will be attached back to the corresponding data tuple (the L.H.S).

Assume a user's relation R having 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 summary objects attached to r. Each summary object consists of a five-ary vector {ObjID, InstanceID, TupleID, Rep[], Elements[][]} as depicted in the following figure:



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

The *ObjID* is the objects's unique identifier, and the *InstanceID* and *TupleID* are references for the corresponding summary instance, and the data tuple, respectively. The *Rep[]* array stores 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. At query time, end-users will see only the *InstanceID* and *Rep[]* fields of each propagated summary object as illustrated in Figure 1.

For each summary object s_i, the structure of its representatives stored in Rep[] depends on s_i's type as depicted in the above figure. For example, in the case of the *Cluster* type, each cluster (group) will report an annotation as its representative as well as the number of annotations in that group. Hence, the Rep[] array consists of a list of representatives in the form of pairs [(*Text annotation, Number groupSize*)]. In the case of the *Classifier* type, each representative will have a class label along with the number of annotations assigned to this label. For the *Snippet* type, each large annotation will have a corresponding short snippet as its representative.

2.2 Summary-Aware Query Processing and Propagation

InsightNotes's query engine has several extensions that enable efficient and seamless propagation of the summary objects under complex transformations, e.g., projection, join, grouping and aggregations, and duplicate elimination. We proposed extensions to the semantics and algebra of each query operator to manipulate the summary objects on-the-fly without the need for accessing the raw annotations. The following example demonstrates a Select-Project-Join (SPJ) query involving summary propagation in InsightNotes. The formal semantics of all query operators can be found in [22].

Example 1: *Assume an SQL query* `"Select r.a, r.b, s.z From R r, S s Where r.a = s.x And r.b = 2"` *over the two tuples r and s presented in Figure 3. Tuple r has four summary objects attached to it, while tuple s has only two attached summary objects. We proved in [22] in Theorems 1 and 2 that to guarantee identical summary propagation under different—but equivalent—query plans, InsightNotes needs to project out the un-needed annotations before any merge operation over the summary objects. Therefore, the projection operator in Step 1 in Figure 3 projects out attributes $r.c$ and $r.d$ and eliminates the effect of their annotations from r's summary objects. For example, the* `annotationCnt` *field in the classifier objects is decremented, the wikipedia article in the snippet object is deleted, and the cluster objects are modified, e.g., some annotations are dropped from each cluster, and hence the* `groupSize` *field is decremented. Moreover, if a cluster's representative is dropped, then another representative is elected (See $A5$ representative replacing the dropped $A2$ representative). The same operation takes place over tuple s, where the effect of all annotations attached to both $s.x$ and $s.y$ is removed from s's summary objects. The only difference is that $s.x$ attribute will not be projected out because it is needed in the subsequent join operator.*

The next operator in the query plan is the selection operator over r (Step 2). Based on the query's predicate, r will pass the operator and all its summary objects will propagate without any change. Then, the produced tuples will join and their summary objects will be merged (Step 3). According to the merge procedure, r's summary objects `ClassBird1` *and* `TextSummary1` *will propagate without any change since they do have no counterpart objects over s. Whereas summary objects* `ClassBird2` *and* `SimCluster` *will be combined. This action takes into account the case where the same annotation may be attached to both tuples r and s, and hence the annotation's effect on the summary objects should not be double counted. For example, assuming that there are five common annotations on both r and s classified as* `Comment`*, then when the two objects are merged the sum of that classifier label will be 22 instead of 27 as illustrated in the figure. The merge of the clus-*

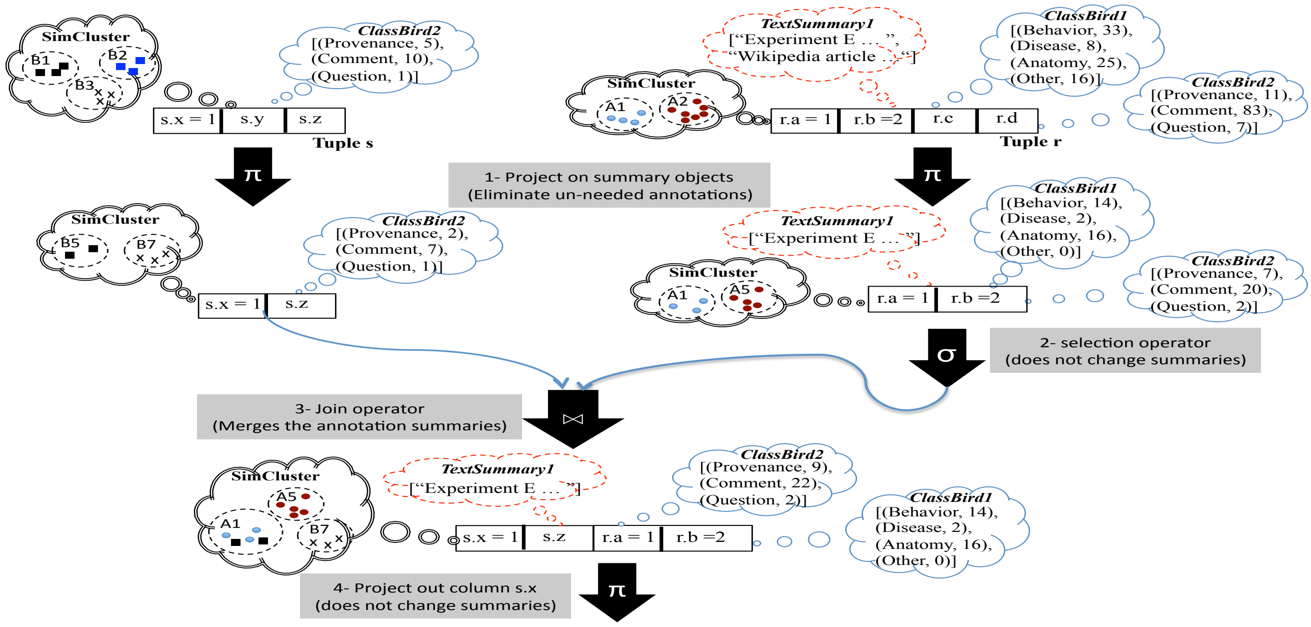


Figure 3: Example Query in InsightNotes.

ter summary objects is slightly more complex. The main idea is that the overlapping groups from both sides, e.g., the groups represented by $A1$ and $B5$, will be combined together, whereas the non-overlapping groups, e.g., the groups represented by $A5$ and $B7$, will propagate separately as illustrated in the figure. Finally attribute $s.x$ will be projected out before producing the output.

3. SUMMARY-BASED FUNCTIONS & OPERATORS

3.1 Summary-Based Manipulation Functions

The first step in treating the annotation summaries as first-class citizens is to design a set of interfaces and manipulation functions on top of them. In the following, we demonstrate few of the developed functions, which we use throughout the paper. We also expect the end-users to leverage these basic functions to create more semantic-rich summary-based UDFs.

- **Summary Set Functions:** We introduce a special variable “\$” 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, we define interface functions over the \$ variable, which include:

○ **Int \$.getSize():** Returns the number of summary objects within the set. For example, referring to tuple r in Table Birds in Figure 1(c), $r.\$.getSize() = 4$.

○ **SummaryObj \$.getSummaryObject(String *InstName*):** The function takes a summary instance name as input, and returns the summary object corresponding to that name, otherwise it returns Null. For example, $r.\$.getSummaryObject(‘ClassBird1')$ and $r.\$.getSummaryObject(‘TextSummary1')$ return the Classifier and Snippet summary objects attached to tuple r.

○ **SummaryObj \$.getSummaryObject(Int i):** This function takes a position within the summary set as input, 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, e.g., to iterate over the objects within a summary set and apply a certain functionality.

We then define a set of manipulation functions over each summary object O according to its summary type. Some functions are common to all types. For example, O**.getSummaryType()** and O**.getSummaryName()**, return the type of the summary object—as either “Classifier”, “Snippet”, or “Cluster”—, and the summary instance name, respectively. Another common function to all types is O**.getSize()**, which returns the number of representatives within object O, i.e., the size of $O.Rep[]$. For example, referring to Figure 1, the *ClassBird1* classifier object has 4 representatives, while *SimCluster* cluster object has 2 representatives. Other functions are specific to each summary type. For example:

- **Classifier Type Functions:** For a summary object O of type Classifier, the defined functions include:

○ **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 when 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* as input, 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*).

- **Snippet Type Functions:** For a summary object O of type Snippet, the defined functions include:

○ **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.containsSingle(String kw_1 [, String kw_2, ...]):** Returns True if all of the given keywords $kw_1, kw_2, \ldots$ are contained within any one of O's snippets or the raw annotations. As we studied in [16], there is a tradeoff—w.r.t accuracy and performance—between searching the snippets vs. searching the raw annotations.

○ **Boolean O.containsUnion(String kw_1 [, String kw_2, ...]):** Returns True if all of the given keywords $kw_1, kw_2, ...$ are contained within the union of O's snippets or O's raw annotations. In this function, the keywords may span multiple annotations attached to the same tuple.

Internally, InsightNotes—which uses PostgreSQL as its underlying DBMS— implements the summary objects as composite data types. On top of these types, the manipulation functions presented above are defined.

3.2 Summary-Based Relational Operators

We now introduce several summary-based relational operators. Unlike the standard SQL operators, these operators operate on the summary objects attached to each tuple instead of its data content. The summary-based operators can be mixed with other standard relational operators in a single query pipeline for seamless processing. The new operators include:

• **Filter Operator ($\mathcal{F}_p(R)$):** The filter operator takes a set of summary-based predicates p, and returns each tuple $r \in R$ along with only its summary objects satisfying p. The operator is formally defined as:

$$\mathcal{F}_p(r) = \{r' =< a_1, a_2, ..., a_n, \{s_i, ...\} > \mid p(s_i) = True, \text{ where } 1 \leq \text{i} \leq \text{k} \}$$

For example, referring to Figure 1(c), the predicate `(getSummaryName() = 'SimCluster')` returns r along with only the the specified cluster summary object. In contrast, the predicates `(getSummaryType() = 'Classifier')` return r along with only the two classifier summary objects *ClassBird1* and *ClassBird2*.

• **Selection Operator ($\mathcal{S}_p(R)$):** The summary-based selection operator takes a set of summary-based predicates p, and returns the data tuples $r \in R$ having summary objects satisfying p. Otherwise, r is dropped. For qualifying tuples, all their summary objects will pass without change. The algebraic expression of the operator is as follows:

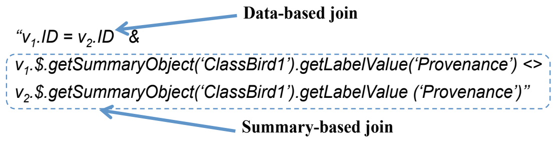
$$\mathcal{S}_p(R) = \{r \in R,\ r =< a_1, a_2, ..., a_n, \{s_1, s_2, ..., s_k\} > \mid p(r.\$) = True\}$$

The summary-based predicates may range from black-box UDFs that take $r.\$$ as a parameter and return a Boolean value, to explicit predicates based on the system-defined manipulation functions presented in Section 3.1. In the latter case, the system can reason about and optimize the execution of these predicates as will be presented in Section 4. For example, the predicate `(r.$.getSummaryObject('ClassBird2'). getLabelValue('Provenance') = 0)` returns only R's tuples having no provenance-related annotations attached to them. In contrast, the predicate `(r.$.getSummaryObject('TextSummary1').contains Single('Wikipedia', 'hormone'))` returns R's tuples that have at least one annotation containing both keywords. Such predicates can be efficiently evaluated using the the summary-based indexes presented in Section 4.

• **Join Operator ($\mathcal{J}_p(R, S)$):** The summary-based join operator joins two input tuples $r \in R$ and $s \in S$ iff the summary-based join predicates p evaluate to True over $r.\$$ and $s.\$$. The algebraic expression of the operator is as follows:

$$\mathcal{J}_p(R, S) = \{< r, s >, where\ r \in R\ \&\ s \in S \mid p(r.\$, s.\$) = True\}$$

For example, referring to Table *Birds* in Figure 1, assume we have two revisions of this table, V_1 (after Revision 1) and V_2 (after Revision 2). Then, reporting the data tuples whose number of provenance annotations has changed between the two revisions would involve the following expression:



"v_1.ID = v_2.ID &
v_1.$.getSummaryObject('ClassBird1').getLabelValue('Provenance') <>
v_2.$.getSummaryObject('ClassBird1').getLabelValue ('Provenance')"

The expression combines both data- and summary-based join operators. As will be discussed in Section 5 and based on the available indexes and statistics, the query optimizer may decide to join the tuples based on the data values and then applies a summary-based selection operator, i.e., ($\mathcal{S}(R \bowtie S)$). Alternatively, it may join the tuples based on the summary objects and then applies a standard selection operator ($\rho(\mathcal{J}(R, S))$).

• **Sort Operator ($\mathcal{O}_{f\,[,direction]}(R)$):** The summary-based sort operator orders the data tuples in R according to the summary-based function $f(r.\$)$. Function f must return values of a data type having a *full-ordering* property, e.g., number, string, and Boolean. Using the $\mathcal{O}$ sort operator, the $Q3$ query in the case study (Figure 2) can now be fully automated and answered in few seconds.

It is worth highlighting that these summary-based operators are new physical operators introduced to the InsightNotes engine. They are not implemented as UDFs within PostgreSQL DBMSs for the following fundamental reasons: (1) If the summary-based operators are implemented as UDFs, then their execution will be carried out and encapsulated within the standard SQL operators. As a result, none of the summary-based optimizations proposed in Section 5 would have been possible. (2) The annotation summaries are not like any other user-defined data types created through PostgreSQL extensibility. They are special tuple-based metadata information that requires extending the semantics of the core query operators [22]. That is why the core operators of InsightNotes in [22] do not manipulate the summaries through UDFs, and consequently, the newly proposed operators cannot be implemented as UDFs. And (3) The design choice of implementing the summary-based operators as new physical operators does not limit the extensibility of InsightNotes because the operators are defined at the summary-type level, i.e., Classifier, Snippet, and Cluster types. And thus, they apply to any driven instance under those types.

4. SUMMARY-BASED INDEX SCHEME

To enable efficient execution of the summary-based relational operators, we need to build a summary-based indexing scheme over the summary objects. In this paper, we will focus only on the Classifier-Type indexing scheme. The InsightNotes system will not automatically index all summary instances defined in the database. Instead, this process is triggered by DB admins using the following command:

```
Alter Table <tableName>
  [Add [Indexable] | Drop] <InstanceName>;
```

This extended SQL command is used in InsightNotes to link a Summary Instance SI to a given user's relation R (Refer to Section 2.1). The newly added optional clause `Indexable` will inform the system to build an index on SI's summary objects created over R's tuples.

Before we investigate possible indexing scheme, we briefly explain how the summary objects are currently stored in InsightNotes to optimize their propagation at query time. Referring to Figure 4(a), given a user's relation R, each tuple in R may have one or more summary objects attached to it according to number of summary instances linked to R. To optimize the propagation of the annotation summaries at query time, R's summary objects are stored in a de-normalized form in a corresponding catalog table *R_SummaryStorage*, as illustrated in Figure 4(b). Each tuple in R has a corresponding unique tuple in *R_SummaryStorage* linked together through unique tuple identifiers (OIDs). This scheme has two main advantages: (1) Since the summary objects are stored in tables separate from the data tables, there is no I/O or CPU overheads added to users' relations when queried in isolation, i.e., when the data is queried without annotation propagation, and (2) Since the summary objects are stored in a de-normalized form, there is no additional I/O or CPU overheads at query time to re-construct them from their primitive components. Thus, their propagations becomes more efficient as studied in [22].

4.1 Classifier-Type Indexing Scheme

Target Query: *The index will speedup summary-based selection operators in the form of* "classLabel <Op> constant", *where* classLabel *is a classifier label within a classifier summary object, and Op is a comparison operator including {=, >, <, ≤, ≥}. The output are the data tuples whose classifier summary objects satisfy the given predicates. The index will also speedup summary-based join and sorting operators involving the indexed classifier column.*

Example 2: *Referring to Figure 4(a), assume we want to retrieve the data tuples having more than 5 associated questions. The SQL query will be:*

```
Select * From R r
   Where r.$.getSummaryObject('ClassBird2').
   getLabelValue('Question') > 5;
```

Baseline Indexing Scheme: A straightforward indexing strategy over the Classifier-type summary objects is to normalize their representation by replicating their components, i.e., the class labels and their counts, and storing them in a separate table (See Figure 4(c)). And then, we can build a standard B-Tree index on each of the columns, i.e., the ClassLabel and Cnt columns. Moreover, since most predicates over the Classifier-type objects will reference both columns, we may create a third system-maintained (derived) column that concatenates these two columns, and then index its values using the B-Tree index as illustrated in Figure 4(c).

The advantage of this scheme is that it uses the standard indexes without modifications. However, it has two major drawbacks. First, the storage overhead of the summary objects is doubled; one replicate is for efficient propagation, and another replica is for indexing. And second, starting from the index to reach the actual data tuples in relation R, we will need several join operations among multiple tables, which certainly degrades the query performance. The proposed Summary-BTree indexing scheme will overcome these limitations.

4.1.1 Summary-BTree Index Structure

The proposed *Summary-BTree* index is a variant of the standard B-Tree that can be directly built over the de-normalized representation of the Classifier-type summary objects. The structure of the index is depicted in Figure 4(d). Assume the index is built on top of the summary instance *ClassBird1* defined on Relation R. The creation of the index involves three steps:

○ **Itemization:** 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:ExtendedAnnotationCnt" as illustrated in Figure 4(c) Step 1. The *ExtendedAnnotationCnt* will have an initial 3-character format to preserve the order among the integer values even after converting them into strings[1]. In Figure 4(d) Step 1, we illustrate the itemization of the *ClassBird1* summary object attached to tuple $r1$.

○ **Indexing:** The text values generated from the Itemization step will be inserted into the Summary-BTree index. The index follows the same structure and operations of the standard B-Tree. And hence, the B-Tree's maintenance algorithms, i.e., insertion and deletion, are all leveraged in the Summary-BTree index. The indexed values will appear in the leaf nodes of the index sorted alphabetically as depicted in the figure. The only exception compared to the standard B-Tree will be in the heap pointers stored in the leaf nodes, which are called *backward pointers* and described next.

○ **Backward Referencing:** We make use of the fact that the storage of the annotation summaries is entirely transparent from (and not directly query-able by) the end-users. Hence, we have the opportunity to optimize the internal structure of the proposed tables and indexes for efficient performance. A key trick in the Summary-BTree index is that the leaf nodes will point back to their annotated data tuples in Relation R instead of pointing back to the *R_SummaryStorage* table. For example, the index entries "Disease:002" and "Disease:008" will point back to tuple $R.r2$ and $R.r1$, respectively. These backward pointers will be created and maintained under the different operations as described in sequel [2].

The advantages of the Summary-BTree index are two fold: (1) It builds on the existing storage scheme of InsightNotes without the need to replicate or normalize the summary objects, and hence the optimized propagation performance is not affected. And (2) As the experimental evaluation will confirm (Section 6), the backward-referencing mechanism achieves up to 11x speedup in query performance compared to the baseline indexing scheme.

4.1.2 Summary-BTree Index Operations

To enable the backward-referencing mechanism, we developed an internal function, called *diskTupleLoc()*, within the database engine, which takes a tuple's identifier (OID) and returns its heap location. This function will be used inside the index's maintenance algorithms to create the correct backward pointers. Notice that this mechanism does not break the transparency concept in database systems since it is entirely encapsulated within the index structure and not exposed to the outside world (neither end-users nor database developers). The index is maintained under the following operations:

○ **Adding Annotation−Insertion:** Adding a new annotation on an un-annotated tuple in R results in inserting a new tuple in *R_SummaryStorage*. The system will then retrieve the heap location of the data tuple, itemize the indexed classifier summary object, and insert them into the Summary-BTree index as illustrated in Figure 4(d).

○ **Adding Annotation−Update:** Adding a new annotation on an already-annotated tuple in R will result in updating the corresponding summary objects in *R_SummaryStorage*. For example, if a new annotation highlighting a disease is added to $R.r1$, then the *ClassBird1*'s summary object will be updated by incrementing the count

[1] If the number of annotations assigned to a single classifier's label exceeds 999, then InsightNotes automatically increments the number of allocated characters and re-builds the index. However, it is a very rare operation.

[2] The SummaryStorage tables are still directly accessible using SQL queries, but for administrative tasks only. Such administrative queries use table-scan plans instead of index-scan plans.

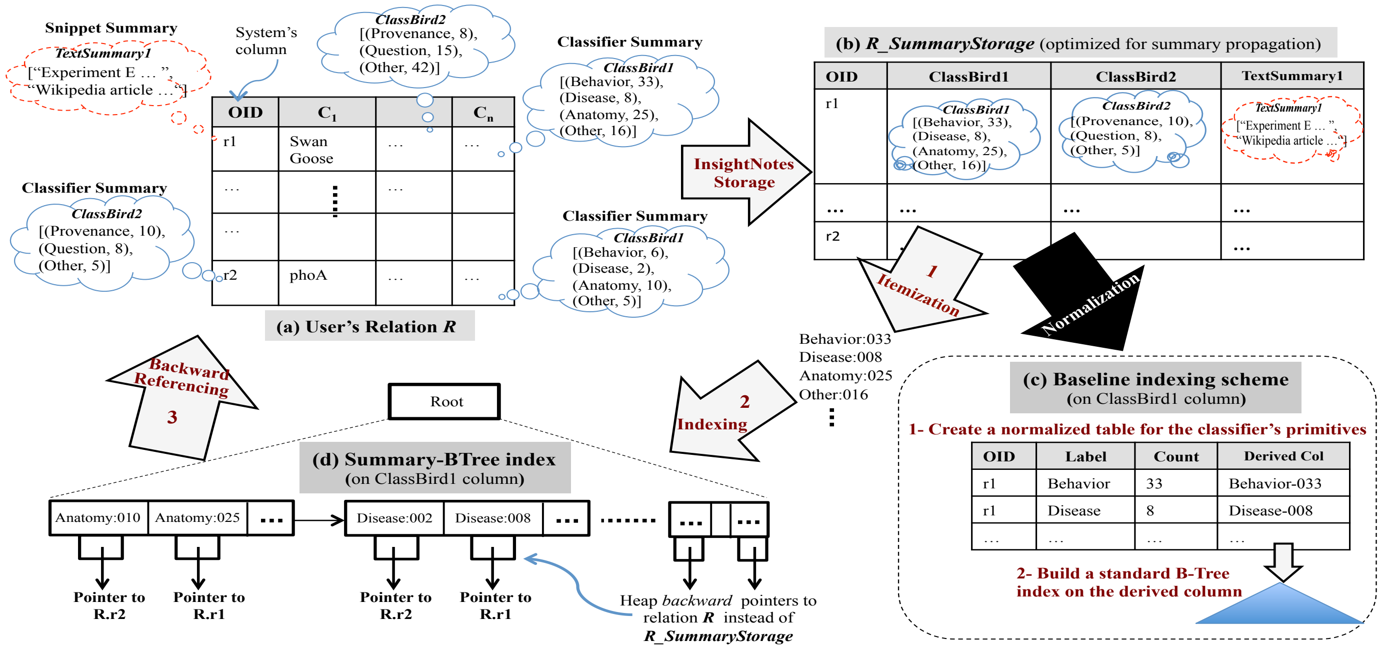

Snippet Summary
TextSummary1
["Experiment E … ", "Wikipedia article …"]
System's column
ClassBird2
[(Provenance, 8), (Question, 15), (Other, 42)]
Classifier Summary
ClassBird1
[(Behavior, 33), (Disease, 8), (Anatomy, 25), (Other, 16)]
OID
C1
Cn
r1
Swan Goose
r2
phoA
Classifier Summary
ClassBird2
[(Provenance, 10), (Question, 8), (Other, 5)]
Classifier Summary
ClassBird1
[(Behavior, 6), (Disease, 2), (Anatomy, 10), (Other, 5)]
(a) User's Relation R
InsightNotes Storage
(b) R_SummaryStorage (optimized for summary propagation)
OID
ClassBird1
ClassBird2
TextSummary1
r1
ClassBird1
[(Behavior, 33), (Disease, 8), (Anatomy, 25), (Other, 16)]
ClassBird2
[(Provenance, 10), (Question, 8), (Other, 5)]
TextSummary1
["Experiment E … ", "Wikipedia article …"]
r2
1 Itemization
Normalization
Behavior:033
Disease:008
Anatomy:025
Other:016
Backward Referencing
3
Root
2 Indexing
(c) Baseline indexing scheme
(on ClassBird1 column)
1- Create a normalized table for the classifier's primitives
OID
Label
Count
Derived Col
r1
Behavior
33
Behavior-033
r1
Disease
8
Disease-008
2- Build a standard B-Tree index on the derived column
(d) Summary-BTree index
(on ClassBird1 column)
Anatomy:010
Anatomy:025
Disease:002
Disease:008
Pointer to R.r2
Pointer to R.r1
Pointer to R.r2
Pointer to R.r1
Heap backward pointers to relation R instead of R_SummaryStorage

Figure 4: Summary-Btree Index For Indexing The *Classifier* Type Summary Objects.

of the *Disease* class label to be 9. To update the index, the system will trigger a deletion and then re-insertion only for the modified class label—The other class labels within the object will remain untouched. For example, a deletion of key `"Disease:008"` and insertion of key `"Disease:009"` will take place.

○ **Deleting Annotation or Tuple:** The deletion of an annotation will result in updating the corresponding summary objects in *R_SummaryStorage*. Therefore, the same procedure described above will be applied. Similarly, the deletion of a data tuple from R will result in deleting the corresponding tuple in *R_SummaryStorage*, and all its index entries will be deleted.

○ **Summary-BTree Querying:** To answer an equality query over the index, e.g., `"classLabel = constant"`, a probing key will be formed by concatenating the two operands, i.e., `"classLabel:Extended_constant"`, where `Extended_constant` is the 3-character format of the constant value. In the case of a range query, e.g., `"constant1 > classLabel > constant2"`, two probing keys will be formed; a *starting key* as `"classLabel:Extended_constant1"`, and a *stopping key* as `"classLabel:Extended_constant2"`. All the keys in between will lead to the qualifying data tuples. If either of the starting or stopping keys is missing, then it will be replaced by `"classLabel:000"`, or `"classLabel:999"`, respectively.

4.1.3 *Summary-BTree Theoretical Bounds*

The Summary-BTree inherits the efficient logarithmic performance from the B-Tree index since they have similar structure. The following Theorem states the theoretical bounds of the index.

Theorem: *Assuming that the number of data tuples in the user's relation R is M, the number of Classifier-type summary objects is N, the number of class labels per summary object is k, and the disk page size in records is B, then the following theoretical bounds hold for a Summary-BTree index:*

○ *Adding Annotation—Insertion is O($kLog_B kN \ + \ Log_B M$)*

○ *Adding Annotation—Update is O($2Log_B kN \ + \ Log_B M$)*

○ *Deleting data tuple is O($kLog_B kN \ + \ Log_B M$)*

○ *Equality search is O($Log_B kN$)* □

Proof: Assuming N summary objects and each object has k class labels, then the number of indexed keys is O(kN). Therefore, any single search, insertion or deletion will be bounded by O($Log_B kN$). When adding a new annotation that triggers a new insertion in the SummaryStorage table, the k class labels will be inserted into the index which will cost O($kLog_B kN$). In contrast, if the added annotation will trigger an update of an existing class label, then only that label is deleted and then re-inserted, which will cost O($2Log_B kN$). Finally, when inserting into or deleting from the index tree, the system needs to retrieve the heap location of the data tuple. This operation uses a B-Tree index on the OID column in R with the cost of O($Log_B M$).

5. SUMMARY-BASED QUERY OPTIMIZATION

When a query involves both the summary-based and the standard SQL operators, then the traditional transformation and equivalence rules alone will be of a very limited use. This is because the semantics of the new operators are unknown to current optimizers. For example, the current optimizer may not be able to use the standard *selection-pushdown* rule to push a selection operator below a join operator because there is a summary-based operator in-between. Another example is illustrated in Figure 5(a), where the query plan involves a summary-based sort $\mathcal{O}$ and selection $\mathcal{S}$ operators on top of a traditional join operator $\bowtie$. In this case the current optimizers cannot apply any of the known transformation rules to create equivalent query plans. In this section, we introduce several important equivalence rules involving the summary-based operators, and extend the query optimizer to leverage them and create a larger pool of possible query plans.

5.1 Extended Equivalence Rules

• **Rules for Summary-Based Selection ($\mathcal{S}_p(R)$):** Few important rules involving the $\mathcal{S}$ operator include:

$$\mathcal{S}_p(\sigma_c(R)) \;=\; \sigma_c(\mathcal{S}_p(R)) \tag{1}$$

$$\mathcal{S}_p(R \bowtie_c S) \;=\; \mathcal{S}_p(R) \bowtie_c S, \text{ iff } p \text{ is on instances in } R \text{ not in } S. \tag{2}$$

Proof: Rule 1 is correct since neither the σ operator changes the summaries' content nor the $\mathcal{S}$ operator changes the data's content. And thus, the commutativity property between the two operators apply. This rule enables the system to switch the order of predicates and use the available indexes—either on the data or the summaries—as needed. Rule 2 enables pushing the summary-based selection operator before the join operator. Rule 2 is correct since predicates p are on instances linked only to one of the two relations, say R. Therefore, when the $\bowtie$ operator merges the summary objects attached to the joined tuples, the summary objects related to p are guaranteed not to change since they have no counterparts on S. And hence, the rule applies.

• **Rules for Summary-Based Sort ($\mathcal{O}_{f[,direction]}(R)$):** We focus on an important case where an existing Summary-BTree index can provide R's tuples in an *interesting order* to the query, and hence the sort operator can be eliminated. The following rules state that the order of R's tuples is preserved under certain transformations. We use notation $\overline{R}^L$ to indicate that R has an *interesting order* w.r.t a classifier instance L.

$$\sigma_c(\overline{R}^L) \;=\; \overline{\sigma_c(R)}^L \tag{3}$$

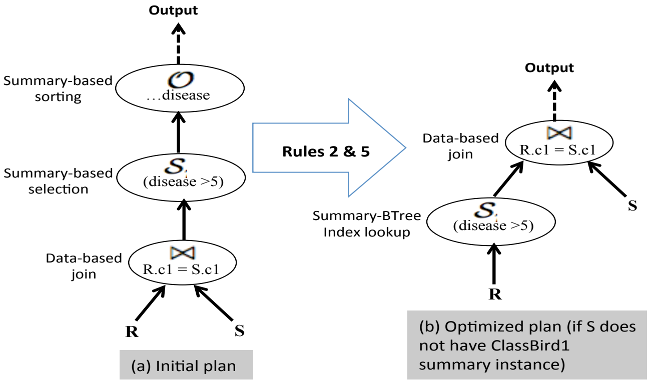
$$\mathcal{S}_p(\overline{R}^L) \;=\; \overline{\mathcal{S}_p(R)}^L \tag{4}$$

$$\overline{R}^L \bowtie S \;=\; \overline{R \bowtie S}^L, \text{ iff } \bowtie \text{ preserves } R\text{'s order, and } \mathtt{L} \text{ is not on } S. \tag{5}$$

$$\mathcal{J}(\overline{R}^L, S) \;=\; \overline{\mathcal{J}(R,S)}^L, \text{ iff } \mathcal{J} \text{ preserves } R\text{'s order, and } \mathtt{L} \text{ is not defined on } S. \tag{6}$$

Proof: Rules 3 and 4 indicate that the selection operators (σ and $\mathcal{S}$) do not change the interesting order of R and preserve it in the output. This is guaranteed since these operators do not change the content of their summaries. For the join operators (Rules 5 and 6), the order w.r.t L is preserved only if two conditions are met: (1) The join algorithm preserves R's order, e.g., R is the outer relation of the join, and (2) Relation S does not have the summary instance L defined on it. If the 2^{nd} condition is not met, then the join operators ($\bowtie$ or $\mathcal{J}$) would merge the summary objects of L, and thus the order may not be preserved. Otherwise, Rules 5 and 6 also applies.

Example 4: *Assume a query Q that joins Relation R depicted in Figure 4 with another relation $S(c1, c2)$ based on data attributes $R.c_1 = S.c_1$. Then, Q selects only the tuples with more than five disease annotations, i.e.,* `ClassBird1.disease > 5`*, and produces the output sorted by the count of these disease annotations. An initial query plan based on the sequence presented above is illustrated in the following figure (Figure 5(a)). Then, consider the following two cases:*
Case I*: Relation S has the* ClassBird1 *summary instance defined on it. In this case, the summary-based selection operator cannot be pushed below the join operator, and the system will use the initial plan in Figure 5(a).*
Case II*: Relation S does not have the* ClassBird1 *summary instance defined on it. In this case, the system will use Rule 2 to push the summary-based selection operator before the join. And assuming that* ClassBird1 *summary instance on R is indexed, then the index can be used to retrieve the tuples with more than five disease annotations (in a sorted order). Then, based on Rule 5, the join operator preserves the order of the tuples, and hence the summary-based sort operator can be removed as illustrated in Figure 5(b).*



(a) Initial plan

(b) Optimized plan (if S does not have ClassBird1 summary instance)

Figure 5: Rule-Based Equivalent Plans in InsightNotes.

• **Rules for Summary-Based Filter ($\mathcal{F}_p(R)$):** Few important rules involving the $\mathcal{F}$ operator include:

$$\mathcal{F}_p(R \bowtie_c S) \;=\; \mathcal{F}_p(R) \bowtie_c S, \text{ iff } p \text{ is on instances in } R \text{ not in } S. \tag{7}$$

$$\mathcal{F}_p(R \bowtie_c S) \;=\; \mathcal{F}_p(R) \bowtie_c \mathcal{F}_p(S), \text{ iff } p \text{ is structural predicate.} \tag{8}$$

Proof: Rules 7 and 8 address pushing the filter operator before the join. Both rules aim for eliminating unnecessary summary objects—and hence their processing cost in the query pipeline—as early as possible. Rule 7 can be proved in similar way to Rule 2, i.e., the $\bowtie$ operator is guaranteed not to alter the summary objects related to predicate p since they are attached to only relation R. Similarly, the $\mathcal{F}$ operator does not change the data's content, and hence the join predicates c are not affected. Therefore Rule 7 applies.

Rule 8 indicates that if the predicate is *structural*—A *structural* predicate is defined as a predicate on the *InstanceID* or the *SummaryType* of the summary object—then p can be pushed to both sides before the join operation. For example, referring to Figure 4, if a query is interested only in the summary objects of instance *ClassBird1*, then all other summaries of instances *ClassBird2* and *TextSummary1* can be dropped as early as possible. Rule 8 can be proved in the same way as Rule 7.

• **Rules for Summary-Based Join ($\mathcal{J}_p(R, S)$):** Few important rules involving the $\mathcal{J}$ operator include:

$$\sigma_c(\mathcal{J}_p(R,S)) \;=\; \mathcal{J}_p(\sigma_c(R), S), \text{ iff } c \text{ is on } R\text{'s attributes.} \tag{9}$$

$$\mathcal{S}_{p1}(\mathcal{J}_{p2}(R,S)) \;=\; \mathcal{J}_{p2}(\mathcal{S}_{p1}(R), S), \text{ iff } p1 \text{ is on instances in } R \text{ not in } S. \tag{10}$$

$$T \bowtie_c \mathcal{J}_p(R,S) \;=\; \mathcal{J}_p((T \bowtie_c R), S), \text{ iff } p \text{ is on instances not in } T \text{ and } c \text{ does not involve } S\text{'s attributes.} \tag{11}$$

Proof: Rules 9 and 10 address pushing the selection operators (σ or $\mathcal{S}$) before the summary-based join operator whenever possible. It is always a valid transformation in the case of the σ operator as long as the predicates c are on one of the two relations (Rule 9). This is because the σ and the $\mathcal{J}$ operate on disjoint pieces of the tuple, i.e., the data values, and the summaries, respectively. Rule 10 is correct since the summary objects related to $p1$ are only attached to relation R. And thus, the $\mathcal{J}$ is guaranteed not to alter these objects after the join. Rule 11 states the conditions for switching the order between summary- and data-based join operators. The order can be switched iff the summary-based join predicates p involve instances not defined on T. And thus, joining early with T ($T \bowtie_c R$) is guaranteed not to affect the evaluation of p.

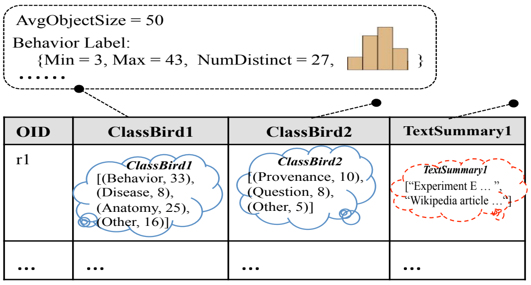


Figure 6: Example of Classifier-Type Maintained Statistics.

5.2 Cost Model and Cardinality Estimation

Statistics Collection: The equivalence rules presented in Section 5.1 enable the query optimizer to generate a larger pool of equivalent query plans. The next step is to estimate the cost of the new summary-based operators in order to select the cheapest plan. Towards this goal, InsightNotes maintains several statistics over the summary objects attached to a given relation R. These statistics are similar to those maintained by traditional DBMSs except that they capture the internal semantics of the summary objects.

Demonstrating over an example, assume relation R has three summary instances linked to it as illustrated in Figure 6. Then, for each summary instance (one column in Figure 6), InsightNotes maintains the average object size (`AvgObjectSize`). In the case of Classifier-type objects, e.g., *ClassBird1* and *ClassBird2*, the size is fixed for all objects within one instance. In contrast, for the Snippet-type and Cluster-type objects, the size may differ from one object to another. Moreover, the system maintains several statistics for each classifier label within the Classifier-type objects. For example, for *ClassBird1*, four data structures are maintained—one for each class label. Each data structure holds some statistics on the *count* field associated with that label, which include `{Min, Max, NumDistinct, Equi-Width Histogram}` as depicted in Figure 6. These statistics are maintained whenever a summary object is updated.

Cardinality and Cost Estimation: To avoid re-inventing the wheel, the new summary-based operators leverage the same heuristics that the standard SQL operators use to estimate their cardinalities and costs. For example, the filter operator $\mathcal{F}$ uses the same heuristics as the standard projection operator π, e.g., based on the *AvgObjectSize* statistics, the $\mathcal{F}$ operator estimates the size of the new tuples and the number of needed disk blocks. Similarly, the summary-based selection operator $\mathcal{S}$ uses the same heuristics as the standard selection operator σ, e.g., referring to the $\mathcal{S}$ operator in Example 4, the system uses the maintained statistics (`{Min, Max, NumDistinct, Equi-Width Histogram}`) over the *ClassBird1.Disease* label to estimate the number of output tuples having more than 5 disease-related annotations. Moreover, if a Summary-BTree index is used to answer this predicate, then the number of performed I/Os can be estimated based on the index's theoretical bounds.

The summary-based join operator $\mathcal{J}$ also follows the same heuristics as the standard join operator $\bowtie$, e.g., the size of joining two relations R and S based on an equality join on `ClassBird2.Provenance` can be estimated by multiplying the size of both relations, and then dividing by the largest value between the `NumDistinct` statistics on `ClassBird2.Provenance` from both sides. Currently, InsightNotes supports only two implementation choices for the $\mathcal{J}$ operator, which are either a block nested-loop join, or an index-based join.

6. EXPERIMENTS

The proposed extensions are implemented within the InsightNotes prototype engine [22], which is based on the open-source PostgreSQL DBMS. The experiments are conducted using an AMD Opteron Quadputer compute server with two 16-core AMD CPUs, 128GB memory, and 2 TBs SATA hard drive.

Application Datasets: We use annotated database that stores information related to 10s of thousands of birds worldwide. The largest annotated table in the database is the *Birds* table that stores the birds' basic information. The table consists of 45,000 tuples, each consisting of 12 attributes, e.g., scientific name, Ids across different systems, description, genus, family, and habit. The table size in the database is approximately 450MBs. The collected number of annotations is approximately 9x10^6 annotations describing 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. The size of each annotation varies between 150 and 8,000 characters. The total size of the raw annotations table (the 9x10^6 annotations) is around 5GBs.

Summarization Techniques: InsightNotes has several integrated data mining techniques for annotation summarization, e.g., the Naive Bayes [10] technique for annotation classification, the CluStream technique [2] for incremental clustering of annotations, and the LSA (Latent Semantic Analysis) technique [18] for text summarization and snippet creation. For the purpose of our experiments, we link the *Birds* table with two summary instances: (1) A Classifier summary instance *ClassBird1* that classifies each annotation to one of the labels: {*'Disease', 'Anatomy', 'Behavior', 'Other'*}, and (2) A Snippet summary instance *TextSummary1* that summarizes each annotation larger than 1,000 characters and creates a snippet that has a maximum of 400 characters. We then create a Summary-BTree index over *ClassBird1*.

Index Creation Overhead: The first set of experiments study the overheads associated with the creation of the summary-based index (Figures 7, 8, and 9). In the experiments, we vary the number of annotations (over the x-axis) between 450x10^3 (corresponding to 10 annotations per tuple on average), to 9x10^6 (corresponding to 200 annotations per tuple on average). Figure 7 illustrates the storage overhead of both the *Baseline* and *Summary-BTree* schemes discussed in Section 4.1. In the former scheme, the summary objects are replicated and stored in a normalized form, and then a standard B-Tree index is created over them. In contrast, in the latter scheme, a Summary-BTree index is created over the de-normalized representation of the summary objects. As the results show, the index size in both cases is almost the same. However, the proposed *Summary-BTree* scheme saves up to 65% of the storage overhead as it requires no replication of the data. The results also show that the storage overhead is almost fixed under the different sizes of the raw annotations. The reason is that once each data tuple has an attached classifier summary object, then the number and size of the summary objects becomes fixed and will not change. The increase in the number of annotations only changes the integer value assigned to the class labels, which does not affect the size.

In Figures 8 and 9, we measure the time overhead of creating the indexes in *bulk* and *incremental* modes, respectively. In the *bulk* mode (Figure 8), the raw annotations and the summary objects will be first created, and then the indexes will be built. This is the recommended mode for initial uploading of large datasets into the database. We measured, over the y-axis, the relative time of creating the index to the time of uploading the raw annotations and creating the summary objects. The indexing time under the *Summary-BTree* scheme involves the time for itemization, insertion

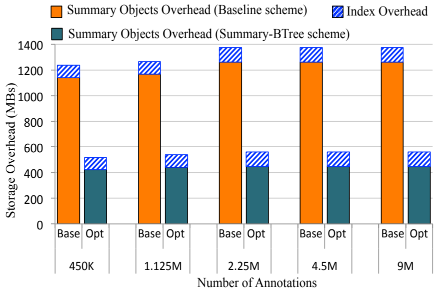


Figure 7: Storage Overhead.

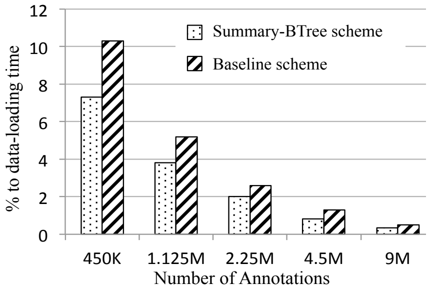


Figure 8: Bulk Index Creation.

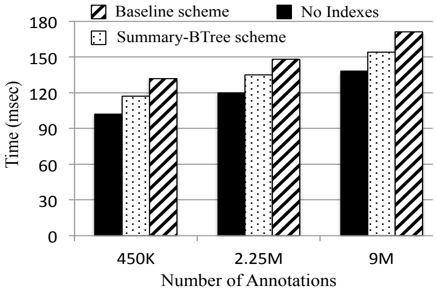


Figure 9: Incremental Indexing.

of indexed keys, and computing the backward references. For the *Baseline* scheme, the indexing time includes the de-normalization and storage in other tables, and the insertion of the indexed keys. The figure illustrates that the creation of the Summary-BTree index is more efficient than the baseline index by up to 35%.

The performance of the incremental indexing is studied in Figure 9. We considered the cases of inserting annotations with: (1) No indexes, (2) A Summary-BTree index, and (3) A Baseline B-Tree index. For each data point in the figure, we insert 100 annotations, measure the insertion time of each annotation under the three cases, and then report the average over the 100 insertions. As the figure shows, the indexing overhead using the Summary-BTree index is approximately 10% to 15% of the insertion time, while the baseline indexing scheme has around 20% to 37% overhead due to the de-normalization step.

Query Performance: The next set of experiments study the effect of utilizing the Summary-BTree index to speedup queries involving summary-based predicates (Figures 10, 11, 12, and 13). The results in Figure 10 illustrate the performance gain from the Summary-BTree index using a Select-Project (SP) query, where the selection predicate is in the form of: `"r.$.getSummaryObject('ClassBird1').getLabelValue('Disease') = constant"`. The query's response time is presented in the y-axis (in Log scale) under three cases: (1) using no indexes, (2) using the *Baseline* standard B-Tree index, and (3) using the Summary-BTree index. We experimented with different query selectivities, i.e., 0.1%, 1%, and 5%, and the differences were minor in each case. Therefore, in Figure 10 we report the results of only the 1% selectivity (around 450 data tuples). The figure illustrates that the Summary-BTree index has approximately 3x speedup over the baseline index. This is because the latter index involves more levels of indirection, and hence requires more join operations to reach the desired data tuples. As expected both indexes achieve around two orders of magnitude speedup compared to the *NoIndex* case.

The experiment in Figure 11 studies the performance of a Select-Project (SP) query involving two conjunctive predicates: (1) A range predicate selecting the tuples having a number of anatomy-related annotations within a given range, i.e., `"r.$.getSummaryObject('ClassBird1').getLabelValue('Anatomy') in [x,y]"`, and (2) A keyword search predicate over the text summarization instance, i.e., `"r.$.getSummaryObject('TextSummary1').containsUnion(kw1, ...)"`. When the index scan over *ClassBird1* is disabled (the *NoIndex* case), InsightNotes uses a table scan followed by a summary-based selection operator $\mathcal{S}$ to apply both predicates. In contrast, when the index scan is enabled, InsightNotes uses the index to evaluate the range predicate and on top of that a $\mathcal{S}$ operator to apply the keyword search predicate. The results illustrate that the *Summary-BTree* index is around 2x faster than the baseline index.

It is worth noting that in the previous experiments, the *Baseline* indexing scheme is used only to evaluate the selection predicates involved in the query. Yet, the for the summary propagation purpose to end-users, InsightNotes still reads the summary objects from its de-normalized storage, i.e., *R_SummaryStorage* (Refer to Figure 4). And hence, both indexes do not pay the cost of building the summary objects from their primitive components. To confirm that depending only on the *Baseline* scheme (the normalized storage of summary objects) can significantly slowdown the summary propagation, we performed the experiment in Figure 12. In the experiment, we used the same query as in Figure 11 and compared between the two indexing schemes. The only difference is that the *Baseline* scheme in this experiment will not only evaluate the predicates, but also form the summary objects for propagation. In this case, the *Baseline* scheme showed around 7x slower performance compared to the Summary-BTree indexing scheme.

In Figure 13, we study the effectiveness of augmenting the Summary-BTree index with backward pointers that point directly to the annotated data tuples instead of the conventional pointers that point to the indexed objects. We use the same SP query used in Figures 10. In the experiments, we consider four cases. The first case is that the index uses the backward pointers, and the annotation summaries are propagated along with the query's result (labeled `Backward-Propagation`). The second case is that the index uses the backward pointers, and the annotation summaries are not propagated along with the query's result (labeled `Backward-NoPropagation`). The other two cases are the same of the above except that the index uses the conventional pointers instead of the backward pointers, i.e., the Summary-BTree index pointers will point to the *ClassBird1* summary objects. The results in Figure 13 show that propagating the annotation summaries has almost the same cost under both the backward and conventional pointers. The reason is that the join operation between the data table and its SummaryStorage table has a 1-1 cardinality, and hence the performance is very similar regardless of which table is used as the outer table in the join. In contrast, if the summary propagation is not required, then the backward pointers will save unnecessary join with the SummaryStorage table, which achieves up to 4x speedup in query execution.

Effectiveness of Query Optimization and Transformation Rules: In Figures 14 and 15, we study the effect of some of the new transformation rules and query optimizations proposed in Section 5. The first experiment (Figure 14) measures the performance of the query demonstrated in Example 4 in Section 5. Relations R and S in the rules correspond to the *Birds* and *Synonyms* tables, respectively. The *Synonyms* table consists of approximately 225,000 tuples and linked to the *Birds* table in a many-to-one relationship.

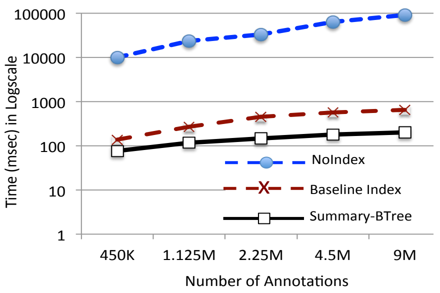


Figure 10: Index vs. No Index (SP Query)

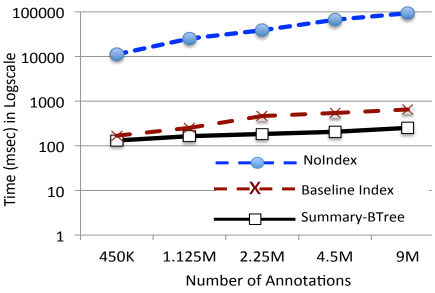


Figure 11: Two-Predicates SP Query

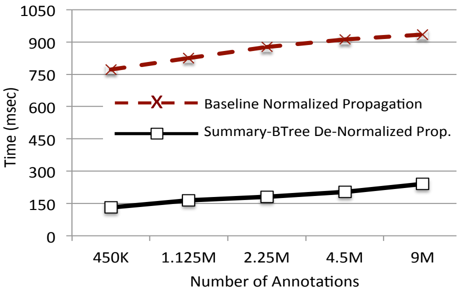


Figure 12: De-Normalized Propagation.

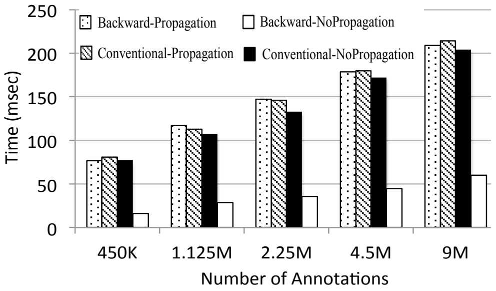


Figure 13: Effectiveness of Backward Ptrs.

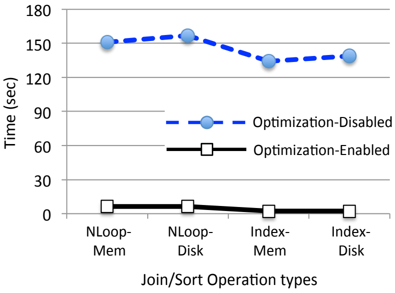


Figure 14: Optimization Rules {2, 5}.

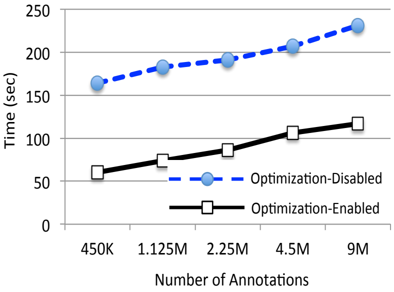


Figure 15: Optimization Rule {11}.

Only the *TextSummary1* instance is linked to the *Synonyms* table, and hence Optimization Rules 2 and 5 can be applied. The experiment compares the response time of the default query plan (Figure 5(a)) against that of the optimized query plan (Figure 5(b)). We set the dataset size to 9x10^6 annotations, and we consider two cases for each of the join and sort operators as illustrated in the x-axis. The join operator either uses an index-based algorithm with an index on the join column in S (labeled `Index`), or a block nested-loop join algorithm (labeled `NLoop`), and the sort operator either uses a memory-based (labeled `Mem`) or disk-based (labeled `Disk`) sort algorithms. The figure illustrates the effectiveness of the transformation and optimization rules in all of the four cases to speedup the query's response time by a factor of 15x.

In Figure 15, we study the effectiveness of Optimization Rule 11, where the order between data- and summary-based join operators can be switched. Relations R and S correspond to the same tables as in the previous experiment, and the summary-based join between them involves a summary-based keyword search on their combined *TextSummary1* summary objects—No summary-based index can be used in this case. Relation T is a replica to relation R, and hence they have a 1-1 relationship through an indexed column for the birds' unique identifiers. With no optimizations, the default plan performs the $\mathcal{J}(R, S)$ operation first using a block nested-loop join, and then performs the data-based join ($\bowtie$) with T. In contrast, the optimized plan switches the join order to make use of the available index on the birds' identifiers in T. Thus, the join operation $R \bowtie T$ is performed first using an index-based join, and then the results is summary-based joined ($\mathcal{J}$) with S. The performance results in Figure 15 indicate that the optimized plan achieves around 3.5x speedup compared to the default plan.

Usability Case Study: Similar to the motivating example presented in Section 1.1, we performed a usability case study to show direct impact of the newly added features on users' experience. We formed a team of 20 students divided into two groups, where one group uses the basic InsightNotes engine while the other group uses the extended system (called *InsightNotes+*). Each student will answer each of the three queries highlighted in Figure 16. In the figure, we report the average time taken by each group (including the time of writing the query), and the results' accuracy.

Query Semantics	# Qualifying data tuples	InsightNotes Group	InsightNotes+ Group
Q1: Report the data tuples sorted based on the number of attached disease-related annotations	100	Time: 5.2 min Accuracy: 100%	Time: 40 sec Accuracy: 100%
Q2: Join version 1 of the data (V1) with version (V2) and report the same objects, i.e., V1.ID = V2.ID, having different number of provenance-related annotations	5	Time: 8.1 min Accuracy: 100% ***(Reports 450 Tuples)***	Time: 54 sec Accuracy: 100%
Q3: Select the birds' records having more than 3 question-related annotations	10	--- ***(Reports 45K Tuples)***	Time: 52 sec Accuracy: 100%

Figure 16: Usability Case Study.

As the results show, both groups are able to answer $Q1$ and $Q2$ queries with 100% accuracy. However, the *InsightNotes* group took significantly longer time to produce the results—which may not be acceptable in many applications. The reason is that InsightNotes cannot fully answer any of these queries, and thus a manual effort is needed to post-process the answer produced from InsightNotes. For example, in $Q1$ the students need to manually sort the 100 data tuples based on the number of their disease-related annotations (summary-based sorting), while in $Q2$, they needed to go over the joined tuples (based on the ID data columns)—which are 450 tuples—and manually check the second join predicate (based on the number of provenance-related annotations) and report the 5 qualifying tuples. For $Q3$, since InsightNotes cannot apply a summary-based selection operation, all the data tuples (45,000) will be reported, and it is impractical to manually select the desired tuples from them. On the other hand, the *InsightNotes+* group is able to answer the three queries in few seconds.

7. RELATED WORK

Annotation management has been extensively studied in the context of relational DBMSs [4, 9, 14, 15, 21]. Several of these systems focus on extending the relational algebra and query semantics for propagating the annotations along with the queries' answers [4, 9, 14, 21]. The Mondrian system [14] has proposed extensions to treat the annotations as first-class citizens, where users can query and manipulate the annotations through newly defined operators. Other systems address the annotation propagation in the context of containment queries [21], logical views [7], or automated copying to newly inserted data [11, 17]. The systems proposed in [8, 13] support special types of annotations, e.g., treating annotations as data and annotating them [8], and capturing users' beliefs as annotations [13]. All of these systems share a common limitation, which is that they all manipulate the raw annotations. Therefore, they do not provide any support for summarizing, extracting useful knowledge, or applying analytics over the raw annotations. The InsightNotes system and its extensions proposed in this paper address such limitations, and enable end-users to query the annotation summaries in novel ways, which otherwise were not possible.

In the domains of e-commerce, social networks, and entertainment systems, e.g., [12, 19], 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. They use such extracted knowledge to take actions, e.g., providing recommendations and targeted advertisements. However, unlike relational DBs, the retrieval mechanisms in these systems are typically straightforward and do not involve complex processing or transformations, i.e., no advanced query processing is required over the annotations summaries once created. Therefore, these techniques do not address the complex query processing and optimization challenges prevalent to scientific relational DBs that are addressed in this paper.

Scientific systems and workflows have also leveraged the concept of semantic and ontology-based annotations, e.g., [3, 6]. These systems use semantic annotations to either summarize complex workflows [3], or help in building and verifying workflows [6]. These systems are based on 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. Nevertheless, the proposed summary-based query operators, access methods, and optimizations are all new and have not beed addressed in current systems.

8. CONCLUSION

The large volume, increasing complexity, and hidden semantics of the emerging annotation repositories in modern applications create unprecedented challenges to annotation management techniques. In this paper, we proposed extensions to the *InsightNotes* system for elevating the annotation summaries from being *"propagate-only"* objects to be *"first-class"* citizens. Hence, it becomes feasible for applications to express complex queries over both the data and their attached annotation summaries, which otherwise is not possible. The key contributions include: (1) Proposing manipulation functions and query operators to seamlessly operate on the summary objects at query time, (2) Developing specialized summary-based indexing scheme and access methods for efficient predicate evaluation and retrieval of the summary objects, and (3) Introducing an extended query optimizer that enables advanced optimizations for queries involving both the summary-based and the standard query operators. The extensions are implemented within the InsightNotes prototype engine, and the results have demonstrated the practicality and efficiency of the proposed extensions and techniques w.r.t both the system's performance, and users' experience.

As part of future work, we plan to enrich the system with more implementation choices for the summary-based operators, enable multi-level (hierarchical) summarization, and extend the querying mechanisms over the multi-level model.

9. 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] C. Bogdanschi and S. Santini. An annotation database for multimodal scientific data. In *Proc. SPIE 7255, Multimedia Content Access: Algorithms and Systems III*, pages 307–314, 2009.

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] K. Ibrahim, D. Xiao, and M. Eltabakh. InsightNotes+: Advanced Query Processing in Summary-Based Annotation Management. Technical Report WPI-TR-14-05: http://web.cs.wpi.edu/∼meltabakh/TR-14-05.pdf.

[17] 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.

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

[19] A. Rae, B. Sigurbjörnsson, and R. van Zwol. Improving tag recommendation using social networks. In *Adaptivity, Personalization and Fusion of Heterogeneous Information*, RIAO, pages 92–99, 2010.

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

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

[22] D. Xiao and M. Y. Eltabakh. InsightNotes: Summary-Based Annotation Management in Relational Databases. In *SIGMOD Conference*, pages 661–672, 2014.