Noname manuscript No.
(will be inserted by the editor)

Decorating the Cloud: Enabling Annotation Management in MapReduce

Yue Lu, Yuguan Li, Mohamed Y. Eltabakh
Worcester Polytechnic Institute (WPI), Computer Science Department, MA, USA
{**ylu, yli, meltabakh**}@cs.wpi.edu

the date of receipt and acceptance should be inserted later

Abstract Data curation and annotation are indispensable mechanisms to a wide range of applications for capturing various types of metadata information. This metadata not only increases the data's credibility and merit, and allows end-users and applications to make more informed decisions, but also enables advanced processing over the data that is not feasible otherwise. That is why annotation management has be extensively studied in the context of scientific repositories, web documents, and relational database systems. In this paper, we make the case that cloud-based applications that rely on the emerging Hadoop infrastructure are also in need for data curation and annotation, and that the presence of such mechanisms in Hadoop would bring value-added capabilities to these applications.

We propose the *"CloudNotes"* system, *a full-fledged MapReduce-based annotation management engine*. CloudNotes addresses several new challenges to annotation management including: (1) Scalable and distributed processing of annotations over large clusters, (2) Propagation of annotations under the MapReduce's blackbox execution model, and (3) Annotation-driven optimizations ranging from proactive prefetching and colocation of annotations, annotation-aware task scheduling, novel shared execution strategies among the annotation jobs, and concurrency control mechanisms for annotation management. These challenges have not beed addressed or explored before by the state-of-art technologies. CloudNotes is built on top of the open-source Hadoop/HDFS infrastructure, and experimentally evaluated to demonstrate the practicality and scalability of its features, and the effectiveness of its optimizations under large workloads.

Authors' address:
Worcester Polytechnic Institute
100 Institute Rd., Worcester, MA, USA, 01609.
Tel.: +1-508-831-6421
Fax.:+1-508-831-5000

1 Introduction

Data curation and annotation is the process of attaching auxiliary metadata information—usually referred to as *"annotations"*—to the base data. Annotations can have a broad range of usage and diverse semantics, e.g., highlighting erroneous or conflicting values, attaching related articles or documents, encoding the provenance information, and linking statistics and quality measures to the data. That is why annotation management has been extensively studied in the context of relational database systems [10,13,15,22, 26,27,50,52]. Most of these techniques build generic frameworks for managing annotations, e.g., storage, indexing, and propagation (reporting) at query time.

In parallel, the infrastructure development has evolved to the cloud-based scalable, and highly distributed systems such as the MapReduce and Hadoop infrastructures [19,42]. Hadoop/HDFS is currently a backbone system for many emerging applications, e.g., log processing [35], business enterprises [1,8,45], clickstream analysis, and scientific applications [28,53]. In this paper, we will first make the case that these Hadoop-based applications are also in need for data curation and annotation, and that the presence of such annotation management mechanisms in Hadoop would bring value-added capabilities to these applications. And then, we will introduce the proposed *"CloudNotes"* system, *the first MapReduce-based annotation management engine*.

1.1 Motivation Scenario: *Management and Handling of Data Qualities*

Most Hadoop-based applications, e.g., log processing, clickstream analysis, and scientific applications, collect and manage data coming from multiple sources. Each source may have different credibility and trustiness degrees that affect the records' qualities. For example, in log processing [35] the data can be periodically collected from hundreds of sites and thousands of servers worldwide. The collected log records may thus have different qualities based on their sources, the accuracy of their values, the absence or presence of certain fields, how and when they are transmitted or obtained, etc. Similarly, scientific applications are increasingly leveraging Hadoop infrastructure because of its desirable properties to science domains, e.g., its flexible and unstructured data model, and the NoSQL computing paradigm [28, 53]. In these applications, data can be collected from many labs, generated using different instruments and experimental setups, and have different levels of curation. Based on these factors, the data records within a given dataset may have different degrees of qualities. Therefore, a very crucial and value-added workflow to all these applications would consist of the following four steps:

(1) **Assessment of Qualities:** In which an automated tool, e.g., a map-reduce quality assessment job(s), executes over the data, considers all the factors mentioned above, and estimates a quality score for each record. It is even practical in some applications to have several of these quality assessment tools, each is producing its own quality estimates. This step is the only easy and feasible step to perform in plain Hadoop—assuming that the needed quality-related factors are already available within the data.

(2) **Attachment to Base Data:** In which the metadata information (e.g., the estimated quality scores) need to be linked back and attached to its data records. This step is not feasible in Hadoop since the data files are "read only", and there is no way to attach auxiliary information to their records. Even the ugly solution of storing the annotations manually in separate files, which reference the data file has serious drawbacks. This is because the annotations may be incrementally added in small batched over time (which degrades the retrieval performance), the data records may not have unique identifiers, e.g., primary keys, to facilitate their reference across files—which is very common in Hadoop-based applications, and more seriously, developers have to manually modify (and complicate) each query on the data in order to retrieve back the metadata. Therefore, there is no easy and efficient way to link annotations to the data in plain Hadoop.

(3) **Automated Propagation:** In which the records' annotations (the quality scores) should automatically and seamlessly propagate along with their data records as inputs to users' tasks. If achieved, then the annotations can be incorporated into the processing cycle as fits each application's semantics, e.g., one application may discard the records below a certain quality threshold from its processing, while another application may output only the low-quality records for further investigation. Clearly, the propagation of the metadata into the users' tasks (map-reduce jobs) in not possible with the current technology in Hadoop.

(4) **Reflection on Outputs:** In which the metadata on the input data may be reflected on the output data produced from a given processing. For example, a map-reduce job processing the data, e.g., an aggregation or model-building tasks, may not only produce new output results, but also estimates the qualities of the produced results based on the inputs' qualities, and attach these estimates (as metadata) to the new records. As a result, end-users and higher-level applications will have better insights about the results, and can make more informed decisions. Unfortunately, the reflection of the inputs' metadata on the outputs is not a viable operation without building an end-to-end annotation management engine in Hadoop infrastructure.

This scenario is by no means exclusive, and many other applications would benefit from building an annotation management engine in Hadoop. For example, a scientific application, e.g., in biology, may periodically need to link newly published articles to subsets of its data, e.g., a new scientific article is published on the genes belonging to the *Cytokine Receptor* families. In this case, the application may need to run a map-reduce job over the data to identify the qualifying genes, and then attach a link of the new article over each of them. Another example from log processing in which a complex workflow is computing aggregations and statistics over the data. The application wants to keep track of the provenance of the output records, i.e., attaching the provenance information as annotations to each output record. As will be discussed in detail in Sections 6 and 7, a general-purpose annotation management engine—such as CloudNotes—can be used for lineage tracking, whereas provenance-specific systems, e.g., [6,7,9,38], cannot be used for general-purpose data annotation and curation.

1.2 Design Guidelines and Driving Principles

The landscape of Big Data infrastructures is very diverse ranging from fully unstructured no data model with black-box query paradigm (Hadoop/HDFS) to fully structured data model with declarative query constructs (Impala, Presto, Hive/Hadoop), from disk-based processing (Hadoop/HDFS, Hive/Hadoop) to memory-based processing (Spark), and from batch-oriented processing (Hadoop/HDFS) to record-oriented processing (HBase). Because of their fundamental differences, the design methodology and objectives of a corresponding annotation management engine would be also fundamentally different (more details are in Section 6).

In general, three key aspects in annotation management are greatly influenced by the underlying infrastructure, which are: (1) The interaction with the annotation management engine, i.e., the mechanisms by which the annotations are added and/or retrieved, (2) The granularities at which annotations are supported, and (3) The propagation and possible transformations that can be automatically supported on top of the raw annotations.

In CloudNotes, we opt for Hadoop/HDFS due to its popularity, flexible data model, and the diverse applications for which Hadoop/HDFS is the ideal infrastructure (as those highlighted in Section 1.1). This is evident from the countless projects and applications proposed in literature on top of Hadoop/HDFS. CloudNotes's objective is thus to bring the annotation management capabilities to these applications without forcing them to migrate their data to some other infrastructures. As a result, applications would get the benefits of annotation management while retaining the advantages of having a flexible data model with no structure known in advance, and the efficient execution of complex workloads involving full scans, aggregations, and joins.

The inherent characteristics of Hadoop/HDFS would influence CloudNotes's design as follows:

• **Automated Creation and Consumption of Annotations:** In RDBMSs, end-users may manually investigate and annotate their data. However, in Hadoop-based applications, such manual investigation and curation is not practical. Instead, the assumption is that the annotations will be produced by automated processes (map-reduce jobs), and also consumed and leveraged by other automated processes (map-reduce jobs). And it can be the case that the same job acts as both a producer and a consumer of the annotations.

• **Single-Granularity Annotations:** Annotation management engines typically support annotating the data at the finest granularity provided by the underlying data model. For example, in RDBMSs, annotations can be at the granularity of table cells, rows, columns, etc. [10,26], and in array-based systems, annotations are defined at the granularity of array cells [49]. Supporting annotations at a smaller granularity, e.g., a sub-value within a table cell, becomes an application-specific task and encoded by the application.

CloudNotes inherits the same principle. Since HDFS has a single unit of granularity, which is the object formed from the InputFormat layer and passed to the mapper layer, CloudNotes supports annotating the data at this granularity. Annotating a smaller granularity, e.g., assigning a quality score for a specific sub-field within an object, is still feasible, but the field references has to be encoded and manipulated by the application within the annotation itself.

• **Blackbox Annotation Propagation:** Hadoop uses a blackbox map-reduce computing paradigm, where the actual computations and transformations applied to the data are unknown. As a result CloudNotes's objective is not to blindly apply transformations on the annotations, but instead to seamless bring the annotation and curation capabilities handy to the Hadoop-based application developers to integrate them into the processing cycle as fits each application's semantics. For example, the annotations should propagate along with their data whenever accessed without having the developer to code that in each query, or worry about how and where the annotations are stored, how they propagate, or any system-level optimizations to make that happen.

1.3 Challenges

As we target building an annotation management engine in a new context and infrastructure (compared to the well-studied traditional DBMSs), several unique challenges arise including:

(1) Managing Jobs of Different Behaviors with Possible Interactions and Conflicts: In plain Hadoop, users' jobs have one behavior, which is reading some existing datasets and generating new datasets. In CloudNotes that is not the case. Submitted jobs can be purely accessing the datasets (neither adding nor retrieving annotations), only adding annotations to exiting (or new) datasets, only retrieving annotations, or a mixture of these behaviors, e.g., one job can retrieve exiting annotations, add new annotations to its input, create new dataset, and annotate this dataset. Ensuring correct execution among these job types is a new challenge to be addressed.

(2) Expensive Annotation Jobs: Unlike RDBMSs in which annotating few records can be efficiently performed, e.g., using indexes, annotating the data in CloudNotes is expensive, i.e., it requires running expensive Hadoop jobs over large data files to probably annotate few qualifying records. Therefore, categorizing these annotation jobs, and optimizing their execution is new to annotation management techniques. Even exiting Hadoop-based provenance systems, e.g., [6,7,38], do not face such challenge because their metadata information are system-generated during the regular execution.

(3) Non-Trivial Sharing and Concurrency Among Annotation Jobs: Sharing execution among Hadoop jobs (which are read-only jobs) has been previously explored as one type of query optimization [36]. However, annotation jobs have the unique characteristic of being read/write jobs in the sense that one job can be annotating a given dataset, while another job is reading the same dataset and retrieving its annotations. This opens new challenges of concurrency control among the jobs, which is new to Hadoop, as well as investigating mechanisms for creating possible sharing opportunities while guaranteeing correct execution.

(4) Annotation Propagation in Distributed Environment: Annotations in CloudNotes will be generated, stored, and propagated in a fully distributed manner. These issues of scalability and distributed processing of annotations have not been addressed by existing techniques that focus mostly on centralized DBMSs. On the other hand, Hadoop-based provenance techniques only generate metadata, but never address the challenges of propagating them back whenever the data is accessed.

1.4 Summary of Contributions

The key contributions of this paper are summarized as follows:

• Extending the MapReduce infrastructure (architecture- and functionality-wise) by embedding a fully distributed annotation management engine into its execution model. We introduce high-level abstract interfaces that enable applications to annotate their data. We also investigate several storage schemes for efficient organization of annotations.

• Developing automated annotation propagation mechanisms for transparently carrying the annotations along with their data to users' map-reduce jobs. We propose optimizations for efficient propagation including annotation-aware task scheduling, proactive prefetching of annotations, and annotation-to-data colocation.

• Proposing a new approach for shared execution among multiple annotation jobs to minimize their total overhead. We propose a categorization of the annotation jobs into different types, and model their relationships using a dependency graph that guides the sharing strategy while guaranteeing correct execution. We also introduce a concurrency control mechanism among the data-related and annotation-related jobs for preventing inconsistent states and dirty-reads over the annotations.

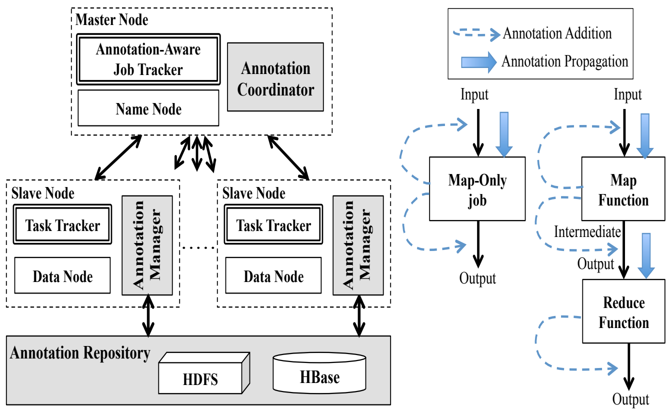
• Building the CloudNotes prototype engine on top of the Hadoop/HDFS infrastructure and experimentally evaluating its features and functionalities. We compare CloudNotes with other related work and systems, e.g., plain Hadoop, Ramp [38], and HadoopProv [7], to illustrate the broader applicability of CloudNotes and the effectiveness of the proposed optimizations.

• CloudNotes opens the new research direction of building annotation management engines over Big Data infrastructures. As will be highlighted in Section 6, the diversity of these infrastructures introduces numerous challenges to annotation management that have not been explored before.

The rest of the paper is organized as follows. In Section 2, we present an overview on CloudNotes's architecture. In Sections 3 and 4, we introduce the extended annotation-aware MapReduce engine, and the annotation propagation mechanisms, respectively. In Section 5, we present several optimization techniques and design issues in CloudNotes. The related work and experimental analysis are presented in Sections 6, and 7, respectively. Finally the conclusion remarks are included in Section 8.

2 CloudNotes Overview

The architecture of CloudNotes is presented in Figure 1(a). CloudNotes extends the MapReduce infrastructure by intro-



(a) CloudNotes Components (Extended MapReduce Infrastructure) (b) Annotation Flow in CloudNotes

Fig. 1 CloudNotes Architecture.

ducing new components (solid-gray boxes), i.e., the *Annotation Coordinator*, *Annotation Manager*, and *Annotation Repository* components, as well as extending other existing components (double-lined boxes), i.e., the *Annotation-Aware Job Tracker*, and the *Annotation-Aware Task Tracker*. The *Annotation Repository* is a distributed storage layer for storing the annotations. It is a hybrid system consisting of both the HBase and HDFS storage engines. The *Annotation Manager* is a distributed single-threaded component running on each slave node in the cluster (similar to a task tracker in Hadoop). The Annotation Manager performs several functionalities including: (1) The communication with the local map and reduce tasks to either accept and organize their newly added annotations (annotation addition), or retrieve the existing annotations from the Annotation Repository and deliver them to users' tasks (annotation propagation), and (2) Interacting with the Annotation Repository component for storing or retrieving annotations in a distributed fashion. The Annotation Manager is the heart of CloudNotes as it performs most of the reading and writing tasks of annotations in a totally distributed fashion to ensure scalable and bottleneck-free execution.

The *Annotation Coordinator* is a centralized component running on the master node, and it performs several functionalities including: (1) Managing the metadata information related to annotations, e.g., whether a specific set of annotations is stored in HBase or HDFS, and whether or not a given job requires annotation propagation, and (2) Communicating with the distributed Annotation Managers and passing instructions to them for optimizing the users' tasks that read the annotations from the Annotation Repository. Finally, the *Annotation-Aware Job Tracker* and *Task Trackers* are extended versions of the standard components in Hadoop. They communicate with their counterpart components, i.e., Annotation Coordinator, and Annotation Managers, respectively, to provide better scheduling for annotation-related jobs and tasks.

CloudNotes provides an optimized infrastructure for two basic functionalities, which are: (1) *Annotation Addition* (the ability to annotate data), and (2) *Annotation Propagation* (the ability to retrieve the annotations associated with a given data object). Figure 1(b) illustrates these two functionalities in the context of map-reduce jobs. Conceptually, map functions should be able to annotate both their input and output data. This is true even under the case of map-reduce jobs where the map's output is intermediate. In contrast, reduce functions should be allowed to annotate only their output data since their inputs will be purged after the job's completion. With respect to annotation propagation, map functions should be able to access the annotations associated with their input data, e.g., the HDFS file under processing. Moreover, reduce functions should be able to access the annotations passed to them from the map functions.

3 Annotation-Aware MapReduce Execution Model

In this section, we focus on developing the building blocks of the annotation management mechanism in CloudNotes.

3.1 Annotation Addition

In order to annotate the data in a transparent way, we introduce a *unique object identifier (OId)*—similar to the tuple Ids or primary keys in relational databases. These OIds will enable high-level applications to reference specific data objects, add new annotations, or retrieve existing annotations, without being exposed to the underlying storage and representation complexities.

Annotating Input Data−(Map-Side): As presented in Figure 2, we propose extending the InputFormat class in Hadoop by creating a wrapper around the RecordReader function. The wrapper augments a unique *Object Identifier (OId)* to each reported key-value pair. The OId objects consist of the file's Id (unique path within HDFS), the block Id (the block's start offset within the file), and the record's relative order *RelOrder* (the relative order within the block starting from 1 in each block).

The OIds provide an abstraction and referencing mechanism to the objects formed from the InputFormat layer (which is the processing unit in Hadoop/HDFS). This abstraction is similar to that proposed in other systems [6,7,23, 38]. However, the main difference is that the OIds in CloudNotes are visible to the map and reduce functions because they are the developers' gateways for manipulating and retrieving the annotations. Throughout the paper and for the ease of presentation, we assume that a given dataset is always accessed by a single InputFormat, and hence the objects to be processed and/or annotated always have the same offsets. Existing techniques that reference the HDFS objects either for indexing or provenance tracking, e.g., [6,7,23,38], have the same assumption as it very rare for a dataset to be read by different InputForms each forming different record structures and boundaries. [1]

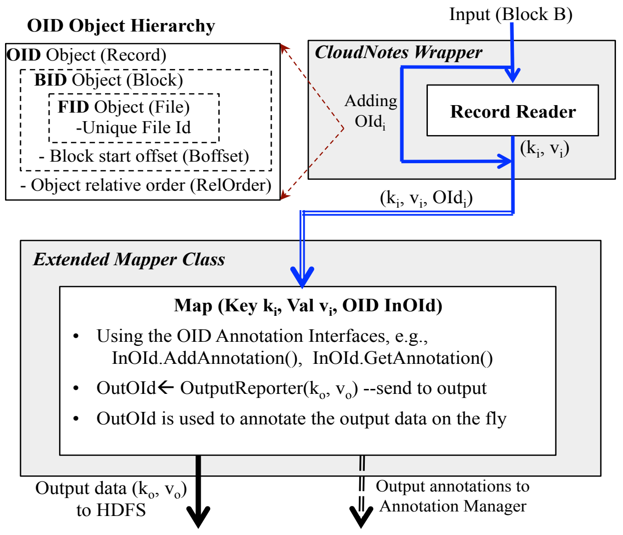

OID Object Hierarchy
OID Object (Record)
BID Object (Block)
FID Object (File)
-Unique File Id
- Block start offset (Boffset)
- Object relative order (RelOrder)
Input (Block B)
CloudNotes Wrapper
Adding OId$_i$
Record Reader
(k$_i$, v$_i$)
(k$_i$, v$_i$, OId$_i$)
Extended Mapper Class
Map (Key k$_i$, Val v$_i$, OID InOId)
Using the OID Annotation Interfaces, e.g., InOId.AddAnnotation(), InOId.GetAnnotation()
OutOId← OutputReporter(k$_o$, v$_o$) --send to output
OutOId is used to annotate the output data on the fly
Output data (k$_o$, v$_o$) to HDFS
Output annotations to Annotation Manager

Fig. 2 Annotation-Enabled Mapper Class.

The mapper class of Hadoop will be also extended such that the *Map()* function will accept triplets in the form of `(Key, Value, InOId)` instead of the standard `(Key, Value)` pairs as depicted in Figure 2. On top of the OID data type, we introduce several annotation-based manipulation interfaces. One of these interfaces is the `InOId.AddAnnotation(AnnValue)` for adding a new annotation to the data object corresponding to the *InOId* identifier. For example, continuing with our motivation scenarios, the Map() function illustrated in Figure 3 may attach links of scientific articles to the *Cytokine Receptor* gene records as indicated in Lines 4-5. Also, the function may estimate the quality of each record based on its content (Line 7), and then attach this estimate to its corresponding record (Line 8). It is worth highlighting that the added annotations will not be written in the regular way to HDFS, instead they will be collected by the local Annotation Manager for optimized storage as will be discussed in Section 3.2.

Annotating Final Output Data−(Map and Reduce Sides): Although the *InOId* input parameter provides a mechanism for annotating the input data of a map function, annotating the output data is still a challenge. For example, the Map() function in Figure 3 may need to copy the quality score(s) of its input records to the produced output, or even carry the provenance along with the output. We will

[1] The assumption of a single InputFormat for a given dataset can be easily relaxed in CloudNotes by extending the OID object to maintain both the *start* and *end* offsets of the formed records. As such, annotations will be attached to *"byte segments"* within the file. Annotations can then propagate along with these byte segments independent from the used InputFormat.

```
Map (Key $k_i$, Val $v_i$, OID InOId)
1.  // Parsing and other processing
2.      …..
3.  //Attach a scientific article's link to the Cytokine Receptor genes
4.  If (geneFamily = 'Cytokine Receptor') Then
5.      InOId.addAnnotation("Link to article: <URL>");
6.  //Attach quality-related annotations from tool Assessment1
7.  score1 = Assessment1($v_i$) ;      //Returns a score based on the content
8.  InOId.addAnnotation("Assessment1 estimated quality: " + score1);
9.  //Derive and produce an output record <Out.key, Out.value>
10. OutOId = OutputReporter(Out.key, Out.value);
11. //Attach the input's quality score to the output
12. OutOId.addAnnotation ("Passed quality from input:" + score1);
13. //Carry the provenance information
14. OutOId.addAnnotation ("Provenance: " + InOId);
```

Fig. 3 Example of the Annotation-Based Interfaces (Annotating the input and output of Map-Only Job).

now present the mechanism for annotating a job's final output, i.e., the map output in a map-only job, or the reduce output in a map-reduce job. Passing the annotations over the intermediate data is presented in Section 4.2.

In order to facilitate annotating the output data on-the-fly, i.e., while being produced and even before writing them to local disk or HDFS, we extend the reporting mechanism in Hadoop. The extended reporting mechanism keeps track of and returns the position of each newly produced record within the output buffer (See the *OutputReporter()* inside the map function in Figure 3, Line 10). This returned *OutOId* can now be used within the same Map() function to annotate the desired output records. The pseudocode in Figure 3 illustrates that after the Map() function produces an output record (Line 10), it annotates that record with input's quality score (Line 12), and also carry its provenance information along with it (Lin 14). In general, an application's semantics can be more complex, e.g., have multiple inputs contributing to a single output.

Reduce-side annotations are even more straightforward because reducers are allowed to annotate only their output data. In this case, reducers will use the same mechanism as that of annotating the map-output data, i.e., the reporting mechanism in reducers will return back an *OutOId* for each produced output record, and then the same Reduce() function can add annotations to this record as desired. It is important to emphasize that the proposed extensions and interfaces enable applications to embed the process of annotating the data—either inputs or outputs—with the actual and regular processing of the data, i.e., it is not mandated to have special data scans or customized jobs dedicated for the annotation process.

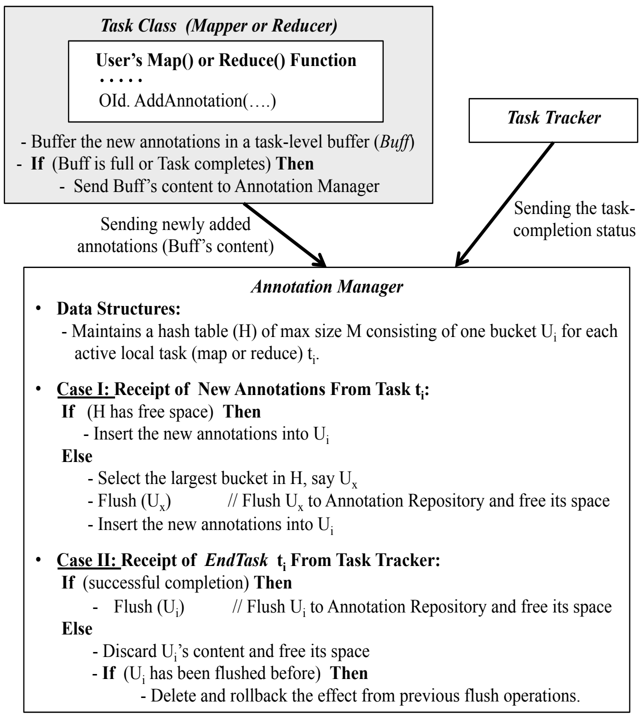
3.2 Annotation-Based Storage Scheme

In CloudNotes, the output of a map or reduce task can be categorized into two types: (1) The regular key-value pair records (the data records) that are stored directly into HDFS, and (2) The newly added annotations having a 4-ary schema consisting of: $\{OId, Value, Curator, Timestamp\}$, where *OId* is the object's identifier being annotated, which references either an already existing data object (e.g., map-input annotation), or a new data object being created (e.g., reduce-output annotation), *Value* is the annotation's content, *Curator* is the job's Id, and *Timestamp* is the job's starting timestamp. In this section, we investigate different storage schemes for efficient storage of annotations, namely *"Naïve"*, *"Enhanced"*, and *"Key-Based"*.

The three schemes share a common mechanism for passing the annotations from the task side (map or reduce) to the Annotation Manager. They differ only on how the Annotation Manager organizes and stores the annotations in the Annotation Repository. The common mechanism is illustrated in Figure 4. When a user's Map() or Reduce() function issues an `AddAnnotation()` command, the new annotation will be buffered in a local buffer within the task. When the buffer is full or the task completes, the buffer's content is sent out to the Annotation Manager (See the top-left box in Figure 4). This task-level buffer is fully transparent from end-users, and it is maintained by the mapper and reducer classes in CloudNotes. The buffer does not have to be large, e.g., it may hold only 10s of annotations. Despite that, its presence is crucial for reducing the communication traffic between users' tasks—which can be many executing concurrently—and the Annotation Manager.

On the Annotation Manager side, it maintains a dynamic main-memory hash table H that holds a number of buckets equals to the number of tasks running simultaneously on its node, i.e., each task (map or reduce) t_i has a corresponding bucket U_i. The number of buckets can be either pre-defined at the job starting time, or dynamic and get increased as more tasks start on the same node. Each bucket can grow dynamically as needed as long as the total size of H is below a pre-defined upper bound M. When the Annotation Manager receives new annotations from task t_i (Case I in Figure 4), the annotations will be added to the corresponding bucket U_i if space permits. Otherwise, the Annotation Manager selects the largest bucket in H and flushes its content to the Annotation Repository (The `Flush()` function) to free space for the new annotations. The Annotation Manager also receives periodic messages from the local Task Tracker regarding the completion of tasks (Case II in Figure 4). If task t_i completes successfully, then its corresponding bucket U_i is flushed to the Annotation Repository. Otherwise, U_i is discarded,

The key differences among the three storage schemes are in the `Flush()` function as discussed in sequel and sum-


Task Class (Mapper or Reducer)
Task Tracker
Sending newly added annotations (Buff's content)
Sending the task-completion status
Annotation Manager

```
User's Map() or Reduce() Function
.....
OId. AddAnnotation(….)

- Buffer the new annotations in a task-level buffer (Buff)
- If (Buff is full or Task completes) Then
      - Send Buff's content to Annotation Manager
```

```
• Data Structures:
  - Maintains a hash table (H) of max size M consisting of one bucket $U_i$ for each
    active local task (map or reduce) $t_i$.

• Case I: Receipt of New Annotations From Task $t_i$:
  If (H has free space) Then
      - Insert the new annotations into $U_i$
  Else
      - Select the largest bucket in H, say $U_x$
      - Flush ($U_x$)      // Flush $U_x$ to Annotation Repository and free its space
      - Insert the new annotations into $U_i$

• Case II: Receipt of EndTask $t_i$ From Task Tracker:
  If (successful completion) Then
      - Flush ($U_i$)      // Flush $U_i$ to Annotation Repository and free its space
  Else
      - Discard $U_i$'s content and free its space
      - If ($U_i$ has been flushed before) Then
            - Delete and rollback the effect from previous flush operations.
```

Fig. 4 Functionalities of the Annotation Manager.

	Naïve	Enhanced	Key-Based
Storage engine	HDFS	HDFS	HBase
Load on HDFS file system	Frequent creation of small anno. files	Frequent updates (addition & deletion) of annotation files	Prefetching files during execution
Need for a periodic daemon process	No	Yes, for merging and deleting small annotation files	No
Freezing State	No	Yes, for the daemon process	No
Optimized annotation representation	No	Yes, normalization and compression	Yes, normalization and compression
Annotation-to-data locality	No	Yes. Needs locality-aware placement policy	Yes. Needs prefetching from HBase
Performance	Slowest	Competitive to Key-Based	Competitive to Enhanced

Fig. 5 Summary of Annotations' Storage Schemes.

marized in Figure 5. In this section, we focus only on the storage of annotations on permanent data, i.e., outputs from mappers in a map-only job or from reducers.

Naïve Storage Scheme: This is the baseline scheme in which the `Flush()` function writes the new annotations directly into HDFS files without special pre-processing or optimizations. For simplicity, lets consider first a map-only job (with a unique Id $JobId$) processing an input data file F_{in} and producing an output data file F_{out}. All newly generated annotations from mapper m_i processing HDFS block B_i in F_{in} will be stored in a corresponding HDFS file $F_{in-B_i-JobId}$. An annotation record within the file will be in the form of: <*OId.RelOrder, Value, Curator, Timestamp*>. For better organization, all annotation files related to F_{in}—across all jobs—will be stored under one directory called F_{in}-$Annotations$. This organization and naming convention is important as it will later simplify the retrieval of annotations at propagation time at the block level as discussed in Section 4. Notice that in CloudNotes annotations are always linked to physical files in HDFS not directories. In the case of processing a directory as input to a job—which is a valid operation in Hadoop— then F_{in} will refer to each physical file under this directory independently.

As summarized in Figure 5, the key advantage of the Naïve scheme is its simplicity. However, it has major disadvantages including: (1) the Naïve scheme is blind to the future retrieval pattern of the annotations, where annotations over a data block B_i are always retrieved with that block. This leads to an un-guided random placement of the annotations in HDFS, and as studied in [24], intelligent placement and colocation or related data has significant positive impact on retrieval performance. And (2) the Naïve scheme is also blind to the creation pattern of the annotations, where the annotations can be added over time by many jobs and each may add few annotations, and also a single annotation can be attached to many data records. This leads to the creation of many small annotation files which is extremely inefficient because HDFS is optimized only for writing/reading big batches of data [44]. Moreover, it introduces un-necessary redundancy that increases the I/O overhead at the creation and retrieval times. The Enhanced and Key-Based schemes will address these limitations in different ways.

Enhanced Storage Scheme: The Enhanced scheme offers two main improvements over the Naïve scheme. First, it deploys a periodic daemon process that processes the annotation files under each annotation directory and merges the small files into larger ones. In the merge process, the annotation files are merged at the block level, i.e., files belonging to different blocks will not be merged. The reason is to preserve the efficiency of annotation retrieval—at the block level—during propagation time (Section 4). Therefore, under a given annotation directory, all annotation files of block B_i, i.e., having a name like "F_{in-B_i-*}", will be merged together to a new file "$F_{in-B_i-merged}$".

During the execution of the daemon process, the Annotation Repository enters a *freezing state* in which no map-reduce jobs can access the annotations. The daemon process is triggered by the Annotation Coordinator on all of the cluster nodes at once. The freezing state during the merge process is important to guarantee the consistency and correctness of the annotations retrieved by users' jobs. More specifically, the merge process will need to delete a set of small files (say *n* files), and add a new file. These *n+1* operations are not atomic and they may take few seconds depending on the number of files and their sizes. During this period, a concurrent user job (if allowed) may read inconsistent and incorrect annotations. However, to minimize the freezing duration, the merge step of the small files into the big one is performed without any freezing, and the big file is created in a temporary directory hidden from users' jobs. Only before deleting the old files and copying the new file to its final location, the Annotation Coordinator starts the freezing state.

The second enhancement is providing better organization and optimized representation before writing the annotations to HDFS, i.e., *compaction* and *locality-aware* storage. This functionality is provided by the `Flush()` function as depicted in Figure 6. The function receives a set S of annotation records to store (These annotations are for a single HDFS block B_i). In Step 1, the records will be grouped based on the annotation's content, and thus an annotation that appears multiple times on many records will form one group. The reason behind this reorganization is to create better chances for compression (Steps 2 and 3). For example, the annotations on a single object are usually distinct and no compression can be performed, whereas a single annotation can be attached to many objects, and hence Step 1 avoids such redundancy by reorganizing the annotations.

Steps 2 and 3 will then transform all the OIds within each group, more specifically the *OId.RelOrder* values, to a bitmap representation, and then compress it using RLE compression technique as illustrated in Figure 6. This representation is very compact even if a single annotation is attached to many data records or to an entire data block. As the experimental evaluation will confirm (Figure 17), the compression introduces minimal overhead when adding the annotations, while achieving big savings when retrieving and propagating them.

After compressing the representation of the annotations (set *G"*), CloudNotes will try to colocate the storage of *G"* along with the corresponding data block B_i in HDFS. The reason is that the annotation files will be always read in conjunction with their data files, and hence colocating them on the same data nodes should yield faster data access, less network congestion, and an overall better performance as reported in [24]. Therefore, the Enhanced scheme uses a *locality-aware* placement policy for storing the annotation files on the same set of nodes holding B_i's data (Steps 4, 5).

It is worth highlighting that both optimizations are *best effort* approaches. That is, the compaction operation is performed only within the input set S—which can be a subset of the annotations on a given data block— and hence the same annotation may still appear multiple times per block. Moreover, the locality-aware placement policy does not guarantee 100% colocation, e.g., if the data nodes storing B_i are near-full or down, then the annotation files will be stored on other data nodes. The best effort approach ensures

```
Enhanced.Flush() Function
• Input      // Set of K annotation records on block $B_i$ in file $F_{in}$
    $S = \{ < OId_i, Value_i, Curator_i, Timestamp_i > \}, \quad 1 \le i \le K$
• Compaction
  1. Group S based on the annotations' content $A_i = \{Value_i, Curator_i, Timestamp_i\}$
     $G = \{< A_i, [OId_1.RelOrder, OId_2.RelOrder, ....] >\}$
  2. For each entry in G, replace the list of relative orders by a bitmap
     $G' = \{< A_i, 000011100000000010.... >\}$
  3. Compress the bitmap using Run-Length Encoding (RLE)
     $G'' = \{< A_i, 0(4)1(3).... >\}$
• Locality-Aware Placement Strategy
  4. Identify the cluster nodes on which the replicas of $B_i$ are stored
  5. Store the entries of G" in HDFS files on these nodes (Best Effort)
```

Fig. 6 Flush Function under the Enhanced Storage Scheme.

that CloudNotes is elastic enough and resilient to failures under the different circumstances. The key characteristics of the Enhanced storage scheme are summarized in Figure 5.

Key-Based Storage Scheme: Unlike the other two schemes, the Key-Based scheme leverages HBase as the primary storage for storing the annotations. This is because HBase suits more the annotations' workload, which involves possibly frequent incremental additions of small batches as well as key-based retrieval based on the block Id at propagation time. In the Key-Based scheme, the `Flush()` function will perform the same compaction steps as in the Enhanced scheme (Step 1-3 in Figure 6), and then set *G"* will be stored in the *key-value* HBase store. The *key* will be the data block Id (B_i) while the *value* will be set *G"*. As summarized in Figure 5, the Key-Based scheme has several advantages over the Enhanced scheme including: (1) The annotations related to a given data block are automatically grouped together (for free) since they are stored in HBase based on the block Id, (2) No need for a periodic daemon process, and (3) There is no freezing state.

Unfortunately, these desirable features do not come for free. The limitation of the Key-Based scheme is that colocating the annotations along with their related data blocks becomes a tricky operation since they are stored in different storage systems. To overcome this limitation, we propose in Section 5.2 an extension to the Key-Based scheme based on a new proactive and prefetching mechanism. This mechanism predicts the assignment patterns between a job's tasks and the slave nodes, and prefetches the corresponding annotations even before they are requested.

4 Annotation Propagation

In this section, we present the annotation propagation mechanisms in CloudNotes, i.e., automatically carrying the existing annotations to the map-reduce jobs.

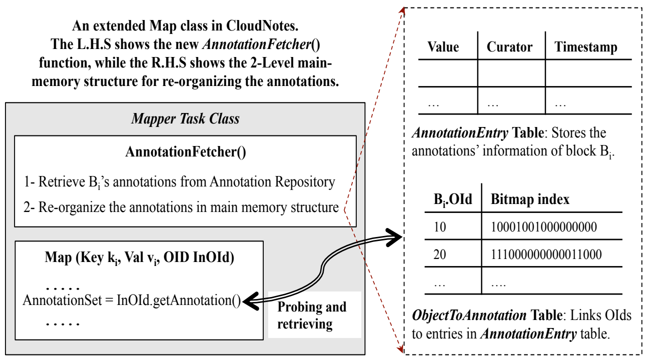

An extended Map class in CloudNotes.
The L.H.S shows the new AnnotationFetcher() function, while the R.H.S shows the 2-Level main-memory structure for re-organizing the annotations.
Mapper Task Class
AnnotationFetcher()
1- Retrieve B_i's annotations from Annotation Repository
2- Re-organize the annotations in main memory structure
Map (Key k_i, Val v_i, OID InOId)
.....
AnnotationSet = InOId.getAnnotation()
.....
Probing and retrieving
Value
Curator
Timestamp
AnnotationEntry Table: Stores the annotations' information of block B_i.
B_i.OId
Bitmap index
10
10001001000000000
20
11100000000011000
ObjectToAnnotation Table: Links OIds to entries in AnnotationEntry table.

Fig. 7 Extended Map Class for Annotation Propagation.

```
MemoryConstraint_AnnotationFetcher()
1.  Set TurncatedFlag = False, LastOId = null
2.  For each B_i's annotation Loop
3.      - If ( space permits & TurncatedFlag = False ) Then
4.          - Update the AnnotationEntry and ObjectToAnnotation tables
5.      - Else If ( space permits & TurncatedFlag = True ) Then
6.          - Update the AnnotationEntry and ObjectToAnnotation tables only
7.              Up to LastOId.
8.      - Else      // No space and we need to free up some memory
9.          - Discard the lower half (L) of the ObjectToAnnotation table
10.         - Discard annotations in AnnotationEntry referenced only by entries in L.
11.         - Set  TurncatedFlag = True, LastOId = the Last OId in ObjectToAnnotation
12.         - Go to Line 3.
```

Fig. 8 Annotation Propagation Under Memory Constraints.

4.1 Map-Side Annotation Propagation

Abstract Propagation Interface: For easy retrieval of annotations, we introduce a new annotation-based interface, called *getAnnotation()*, defined on top of the OID identifier class. Therefore, as depicted in Figure 7, within a user's Map() function, the new interface can be executed on any of the input records, i.e., `InOId.getAnnotation()`, to retrieve the annotations attached to this record. The interface returns an iterator over a set of annotations having the schema of $\{<$*value*, *curator*, *timestamp*$>\}$. Then, user's code can manipulate and process these annotations as fits the application's semantics. In the following, we present the underlying propagation mechanism of the `getAnnotation()` interface. We first introduce the basic idea, and then extend it to work under given memory constraints.

Basic Propagation Mechanism: Assume a user's job processing an input HDFS file F_{in}, and that each map task m_i is assigned a data block B_i in F_{in}. Since the basic unit of processing in Hadoop is a *data block*, then CloudNotes also retrieves the annotations at the data-block level. In order to perform this functionality, the mapper class in CloudNotes has been extended by introducing a new function, called `AnnotationFetcher()`, as illustrated in Figure 7. `AnnotationFetcher()` is a hidden function from end-users, and it automatically executes at the beginning of each map task and before the start of the user-defined Map() function.

The `AnnotationFetcher()` of a map task m_i performs two main functionalities (See Figure 7): (1) Retrieving the annotations attached to data block B_i from the Annotation Repository, and (2) Re-organizing these annotations for faster access by the user-defined Map() function. In the case of the Naïve and Enhanced storage schemes, `AnnotationFetcher()` will retrieve B_i's annotations by reading all annotation files having a name like "F_{in-B_i-*}" under directory F_{in}-$Annotations$. In contrast, in the case of the Key-Based scheme, the annotations will be retrieved from HBase based on block Id B_i.

While retrieving B_i's annotations from the Annotation Repository, `AnnotationFetcher()` will also re-organize them in memory such that a propagation request from the user-defined Map() function, i.e., `OId.getAnnotation()`, can be efficiently answered. For this purpose, `AnnotationFetcher()` builds a two-level main-memory data structure as depicted in Figure 7. The 1st level is the *AnnotationEntry* table that stores a single entry for each distinct annotation on B_i ($<$*value*, *curator*, *timestamp*$>$), while the 2nd level is the *ObjectToAnnotation* table that stores a record's Id in B_i, i.e., OId, and a bitmap indicating the entries in *AnnotationEntry* that are linked to this OId. The *ObjectToAnnotation* table is sorted on the OId values. For example, referring to Figure 7, the first entry in *ObjectToAnnotation* indicates that the 10th record in B_i has three annotations attached to it stored in the entries number 1, 5, and 8 in *AnnotationEntry*.

Incorporation of Memory Constraints: Annotation propagation should be a light-weight process that does not consume much memory, otherwise users' tasks may suffer from a memory shortage. Hence, the `AnnotationFetcher()` function always operates under memory constraints. In CloudNotes, a small percentage of the task's assigned memory, e.g., 10% or 15%, is allocated to the annotation propagation process. Consequently, if B_i's annotations and their data structures presented in Figure 7 cannot fit in the allocated memory, then `AnnotationFetcher()` needs to retrieve the annotations in an iterative manner using the procedure presented in Figure 8.

In Line 1, the *TruncatedFlag* is initialized to False indicating that the function will try to put as many annotations as they can fit in the allocated memory. For each retrieved annotation, if space permits, it will be added to the data structures *AnnotationEntry* and *ObjectToAnnotation* (Lines 3-4). Otherwise, `AnnotationFetcher()` will limit its focus to only the top half of the *ObjectToAnnotation* table, i.e., it will try to keep in memory only the annotations related to the leading subset of B_i's records while dropping the others (Lines 9-12). In this case all annotations in *AnnotationEntry* that are attached only to the lower half will be eliminated from this iteration (Line 10), and the *TruncatedFlag* will be set to True (Line 11). For the subsequent retrieved annotations, if space permits, the annotation will only update the data structures up to the *LastOId*, and no new OIds larger than *LastOId* will be considered (Lines 5-7). If the function runs out of space again, the same algorithm repeats in an iterative manner.

In this iterative version, the user-defined Map() function will be called by the mapper class until the *LastOId* record tracked by the `AnnotationFetcher()` function is reached. And then, the mapper class will interrupt the calling of the Map() function, and re-executes `AnnotationFetcher()` to prepare the annotations on a new set of coming data records. In these subsequent executions, B_i's annotations will be read from the local file system—Not HBase or HDFS—since they are already retrieved in the first iteration and stored locally.

4.2 Reduce-Side Annotation Propagation

In map-reduce jobs, the map function may annotate the data records passing to the reduce function. These annotations should propagate along with their <*key*, *value*> pairs, and be seamlessly accessible within the user-defined Reduce() function. CloudNotes does not put any prior assumptions or restrictions on the annotations passing to the reduce function. The assumption is that the output records from a map function are, by default, un-annotated. Map() functions can freely annotate their outputs as suits the semantics of a given application, e.g., copying the input annotations (or subset of them) to the output, adding new annotations, etc. The main reasons for this assumption are that: (1) CloudNotes is a generic annotation-based engine that does not put prior restrictions or assumptions on how annotations should be generated, (2) Map-reduce jobs may involve blackbox functions, and thus the semantics of a given job may be unknown to CloudNotes, and (3) Given the developed annotation-based interfaces, it would require very minimal effort from developers—two or three lines of code— to add new annotations to the map's output, or even copy the annotations from the inputs to the outputs.

Managing Annotations on Intermediate Data: From the end-users' perspective, there is no difference between annotating the map's output in a map-only job or in a map-reduce job. The mechanism introduced in Section 3.1 for annotating map-side outputs is the same in both cases. However, the Annotation Manager will manage the intermediate annotations in a different way as follows. Assume an output

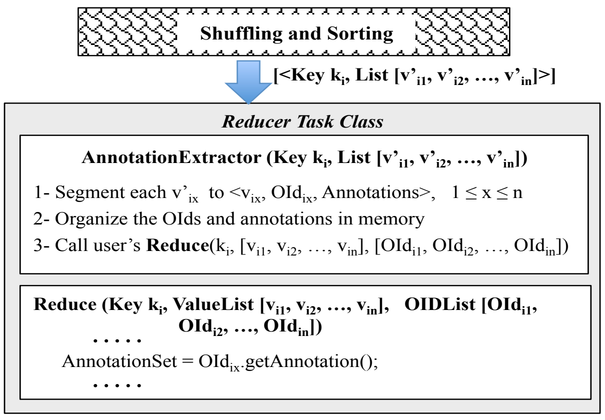

Shuffling and Sorting
[<Key k_i, List [v'_i1, v'_i2, …, v'_in]>]
Reducer Task Class
AnnotationExtractor (Key k_i, List [v'_i1, v'_i2, …, v'_in])
1- Segment each v'_ix to <v_ix, OId_ix, Annotations>, 1 ≤ x ≤ n
2- Organize the OIds and annotations in memory
3- Call user's Reduce(k_i, [v_i1, v_i2, …, v_in], [OId_i1, OId_i2, …, OId_in])
Reduce (Key k_i, ValueList [v_i1, v_i2, …, v_in], OIDList [OId_i1, OId_i2, …, OId_in])
. . . . .
AnnotationSet = OId_ix.getAnnotation();
. . . . .

Fig. 9 Annotation Propagation in Reduce-Side Functions.

record from the map function $< k_i,\ v_i >$, which will be assigned a unique object identifier *OutOId*$_i$. This is the OId returned from the reporter interface within the Map() function as presented in Section 3.1. The only internal difference is that the *OutOId*$_i$ will now reference an intermediate local file instead of a permanent HDFS file in the case of map-only job. Conceptually, the *OutOId*$_i$ is just a placeholder to carry the annotations to the Annotation Manager, which will be then concatenated into the value field, and the intermediate local file will not be referenced again. More specifically, the added annotations on *OutOId*$_i$ will be collected by the Annotation Manager, and then a new corresponding output record $< k_i,\ v'_i >$ will be formed such that:

$$v'_i = v_i \ || \ \textit{separator} \ || \ \textit{OutOId}_i \ || \ \textit{annotations}$$

That is, the key k_i of the output record will remain unchanged, while the value v_i will be extended to carry the *OutOId*$_i$ and its attached annotations. After forming the modified output record $< k_i,\ v'_i >$, all subsequent steps including writing the intermediate data to local disks, and the shuffling and sorting phases will execute in the same standard way as in Hadoop without any change. This is because the shuffling and sorting phases depend only on the k_i component, which has not been changed.

Reduce-Side Extensions: At the reduce side, reducers will now receive pairs in the form of $< k_i,\ [v'_{i1}, v'_{i2}, ..., v'_{in}] >$, where k_i is an output key, and v'_{i1}, ..., v'_{in} are all *modified* output values corresponding to k_i. To process such input, we introduce two main extensions to the reducer class in CloudNotes as depicted in Figure 9. First, we introduce a new function, called `AnnotationExtractor()`, which will automatically execute before the user-defined Reduce() function. `AnnotationExtractor()` is a hidden function from end-users and it is responsible for segmenting the modified values $v'_{ix}, \forall\ 1 \leq x \leq n$, and separating the original map-generated values v_{ix} from their corresponding identifiers *OutOId*$_{ix}$ and annotation values. The function will organize the OIds and their annotations in main memory, and then call the user-defined Reduce() function.

The second extension involves extending the signature of the Reduce() function as depicted in Figure 9. The new signature accepts triplets in the form of: $< k_i,\ [v_{i1}, v_{i2}, ..., v_{in}],\ [\mathit{OutOId}_{i1}, \mathit{OutOId}_{i2}, ..., \mathit{OutOId}_{in}] >$, where k_i is a single input key, v_{i1}, ..., v_{in} are the data values associated with k_i, and OutOId_{i1}, ..., OutOId_{in} are the OIds corresponding to each value. With the new input signature, it becomes feasible for user's code to retrieve the annotations associated with any of the input values—Using the interface `OId.getAnnotation()` as illustrated in Figure 9. Recall that the input OIds to a Reduce() function are used only for the annotation propagation purpose. [2]

4.3 Special Annotation Propagation Cases

In some cases, a user's map-reduce job—which can be a blackbox job—may not be interested in propagating the existing annotations, i.e., the Map() or Reduce() functions do not involve the execution of the *getAnnotation()* method. In these cases, it is an unnecessary overhead and waste of resources for CloudNotes to retrieve the annotations from the Annotation Repository, and then make no use of them. To optimize these cases, we introduce two new Boolean configuration parameters at the job level, i.e., *"MapSideAnnotationPropagation"*, and *"ReduceSideAnnotationPropagation"*, that inform the system whether or not the Map(), and Reduce() functions require annotation propagation, respectively. These two parameters will be sent by the Annotation-Aware Job Tracker to all tasks of the job. When the *MapSideAnnotationPropagation* is set to False, the execution of the `AnnotationFetcher()` function within the mapper class (Figure 7) will be by-passed, and hence no annotations will be retrieved from the Annotation Repository.

Similarly, when the *ReduceSideAnnotationPropagation* is set to False, any annotations produced from the Map() functions on the intermediate data will be automatically discarded. Furthermore, the `AnnotationExtractor()` function within the reducer class (Figure 9) will be by-passed. As a result, the shuffling and sorting phase as well as the reduce function will encounter no extra overhead if no annotation propagation is required.

5 Optimizations & Design Issues

In this section, we introduce several advanced features of CloudNotes including optimizing the annotation addition across different jobs (Section 5.1), optimizing the annotation propagation under the Key-Based storage scheme (Section 5.2), and the consideration of concurrency control and fault tolerance (Section 5.3).

5.1 Lazy and Shared Annotation Addition

Traditional Hadoop jobs are *read-only data-centric* jobs, i.e., reading exiting datasets and generating new datasets. In contrast, CloudNotes jobs have diverse types and behaviors ranging from *annotation-only* jobs, *data-only* jobs, to *data-annotation* jobs. In this section, we propose a categorization of these jobs, and model their relationships using a dependency graph. Based on this modeling, we introduce a *sharing and lazy evaluation strategy* that reduces execution overheads while ensuring correct execution.

In CloudNotes, a map-only job may scan an input file, and annotate each record not matching a specific format as *"corrupted record and should be skipped"*. We refer to this type of jobs as *"annotation-only"* jobs, and we propose a *lazy and shared execution strategy* that combines multiple annotation-only jobs in a single plan.

Since users' jobs may involve blackbox Map() and Reduce() functions, CloudNotes depends on few newly introduced configuration parameters to capture some important annotation-related properties. In order to know whether or not a given job is an annotation-only job, we introduce a new configuration parameter, called *"AnnotationOnlyJob"*, which is set (True or False) during the job's configuration phase. It is worth highlighting that only map-only jobs can have this parameter set to True. In contrast, jobs including of a reduce phase cannot be annotation-only jobs because reducers must produce output data, otherwise their execution becomes meaningless. The second Boolean configuration parameter is the *"MapSideAnnotationPropagation"* that informs the system whether or not the user's job (more specifically the map-side) involves annotation propagation. These two configuration parameters will be taken into account by the *Annotation-Aware Job Tracker* when scheduling jobs as presented in Figure 10.

The flow chart in Figure 10(a) illustrates the execution procedure of the Annotation-Aware Job Tracker when receiving a new job J_x, while Figure 10(b) illustrates an example. The Annotation-Aware Job Tracker maintains a dependency graph G for tracking the dependencies among a set of annotation-only jobs annotating the same input HDFS file. G will be divided into levels as illustrated in Figure 10(b), where the jobs in the same level do not depend on each other, and hence they can share their execution.

When a new job J_x is submitted, the Job Tracker checks if J_x is not an annotation-only job, then J_x will be directly scheduled for execution in the standard way. Otherwise, J_x will be added to the dependency graph G. If

[2] The OIds passed to a reduce task are implemented using the same *Iterator* mechanism currently used for passing the values. Therefore, if the inputs' size is too large to fit in memory, they are streamed from disk as needed in the same standard way.

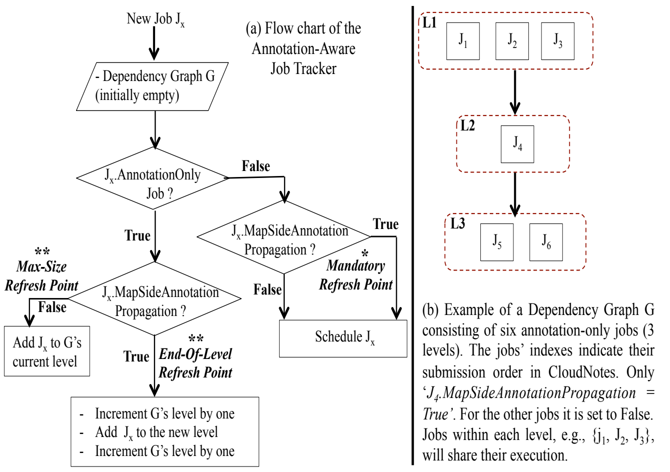

New Job J_x
(a) Flow chart of the Annotation-Aware Job Tracker
- Dependency Graph G (initially empty)
J_x.AnnotationOnly Job ?
False
True
J_x.MapSideAnnotation Propagation ?
True
* Mandatory Refresh Point
False
Schedule J_x
** Max-Size Refresh Point
False
Add J_x to G's current level
** End-Of-Level Refresh Point
- Increment G's level by one
- Add J_x to the new level
- Increment G's level by one
L1
L2
L3
(b) Example of a Dependency Graph G consisting of six annotation-only jobs (3 levels). The jobs' indexes indicate their submission order in CloudNotes. Only 'J_4.MapSideAnnotationPropagation = True'. For the other jobs it is set to False. Jobs within each level, e.g., {j1, J2, J3}, will share their execution.

Fig. 10 Lazy & Shared Execution of Annotation-Only Jobs.

$J_x.MapSideAnnotationPropagation$ is set to False, then J_x does not depend on the existing annotations and it will be added to G's current level. For example, assume Jobs J_1, J_2, and J_3 are all annotating an input file F_{in}, and they are not accessing the existing annotations. Then, The three jobs will be added to G's 1^{st} level as illustrated in Figure 10(b). In contrast, if $J_x.MapSideAnnotationPropagation$ is set to True, then J_x reads the annotations of previous jobs, and hence J_x will be added in a separate level in G to ensure that it will execute after the previous ones (See J_4 Job in G). In addition, the subsequent jobs of J_x cannot share the same level with J_x, and they have to start from a new level. Otherwise J_x may read some of their annotations, which leads to incorrect execution. For example, Jobs J_5 and J_6 in Figure 10(b) will be added in a separate level after J_4's level.

This lazy and shared execution strategy enables CloudNotes to group several annotation-only jobs together. These jobs do not need to be submitted in a single batch, e.g., they may be submitted to CloudNotes in isolation from each other over a period of time. Yet, the Annotation-Aware Job Tracker will be able to buffer these jobs and combine them for later execution only when needed.

Processing the pending jobs in G has to be performed level-by-level starting from the top level $L1$. For each level, CloudNotes creates a single *jumbo* map-only job that executes all of the jobs' Map() functions in that level in any sequential order, e.g., J_1, J_2, and J_3, will all share the same input scan, and a jumbo map function will call the jobs' Map() functions in any order. As indicated in Figure 10(a), there are three *Refresh Points* at which the Annotation-Aware Job Tracker executes the pending jobs in G and updates the annotations. These refresh points provide a tradeoff mechanism between being *more eager or lazy* in evaluating the annotation jobs.

The first point is a *mandatory refresh point* at which a none annotation-only job is submitted and this job is accessing the existing annotations, and thus any pending annotations must be refreshed (marked with *). In the case where the graph is large, the user's job may experience a delay as it has to wait for a longer period of time until the annotation jobs are executed. To limit such effect, CloudNotes considers the second and third refresh points (marked with **). In the second refresh point, *End-Of-Level Refresh Point*, a new level will be added to G, and thus the previous level will not be enlarged any more and no further sharing opportunities can be added. In this case, the jobs of this previous level can execute together and be removed from the graph. The third refresh point, *Max-Size Refresh Point*, ensures that even if no new levels will be added, there is still a limit (though a system-defined parameter) on the number of the annotation jobs that can be grouped in one level. If this limit is reached, G will be refreshed and the pending jobs are eliminated.

In general, our experimental analysis confirms that increasing the sharing opportunities minimizes the overall execution overhead (Section 7.4). However, a user's job may be penalized if it has to wait for other annotation jobs to be performed (the *Mandatory Refresh Point*). As such, the other two refresh points should be triggered frequently enough to reduce the triggering of the mandatory refresh point, but not too frequently to the extent of having no sharing, which consumes way more system resources and may also delay other users' jobs. As a rule of thumb, the *End-Of-Level Refresh Point* should be triggered whenever possible. In contrast, the max limit for the *Max-Size Refresh Point* should be set by the system's admins depending on the frequency of submitting users' jobs, which will be discussed in more detail in the experiment section.

5.2 Prefetching and Annotation-Aware Task Scheduling

One of the advantages of the Enhanced storage scheme over the Key-Based scheme is that the former can colocate the annotation files with the data files, and hence enhances the propagation performance. To bring this benefit to the Key-Based storage scheme, we propose a *prefetching mechanism* that Annotation Managers on the different slave nodes will adopt. The basic idea is that Annotation Managers will proactively prefetch annotations for future map tasks from HBase to their local file systems, and hence when a map task starts, it can read the prefetched annotations from its local storage (if available). Our objective is to design mechanism that is: **(1) Predictive**, where the Annotation Manager tries to anticipate which data blocks will be processed on which nodes as this information is not known for certain in advance, **(2) Lightweight** and does not consume much resources, **(3) Asynchronous** and does not require synchronization between nodes, and **(4) Best effort** and does not have to provide hard guarantees—Recall that colocation in the Enhanced scheme is already a best effort approach.

```
Task Tracker & Annotation Manager on Cluster Node N
- Input variable RepFactor          // $F_{in}$'s replication factor
- Input variable MapsInParallel     // Number of J's map tasks that can run in parallel
- Local variable AggressiveFactor   // [0, 1] controls how aggressively to prefetch
1.  // Launch task $m_i$ from job J that reads data block $F_{in}.B_i$
2.  If ( J.MapSideAnnotationPropagation = False ) Then
3.      - Launch $m_i$ . J needs no access to annotations.
4.  Else If ( $m_i$ is the 1st task of J on Node N) Then
5.      - Launch $m_i$. AnnotationFetcher() will retrieve $B_i$'s annotations from HBase
6.      // Prefetch for future expected tasks
7.      - Prepare a list of $F_{in}$'s data blocks stored in N succeeding $B_i$ ➔ Set $F_{in-N}$
8.      - PrefetchCnt = ( $|F_{in-N}|$ / RepFactor ) x AggressiveFactor
9.      - Randomly select PrefetchCnt blocks from $F_{in-N}$ ➔ $PrefetchList_{Fin-N}$
10.     - Order the block Ids in $PrefetchList_{Fin-N}$ and prefetch the first
            MapsInParallel entries from the list
11.     - Send $PrefetchList_{Fin-N}$ to the Annotation-Aware Job Tracker in the heart beat
12. Else
13.     - If ($B_i$'s annotations are prefetched) Then
14.         - Launch $m_i$.
15.         - AnnotationFetcher() will retrieve $B_i$'s annotations from local disk
16.         - Advance in $PrefetchList_{Fin-N}$ and prefetch for the next block
17.     - Else
18.         - Launch $m_i$.
19.         - AnnotationFetcher() to retrieve annotations from HBase
20.         - If (a previous prefetch was missed) Then
21.             - Advance in $PrefetchList_{Fin-N}$ until the smallest Id after $B_i$
22.     - End If
23. End If
```

Fig. 11 Prefetching Mechanism for Key-Based Storage Scheme (Executing task m_i from job J on Node N).

The main steps of the prefetching mechanism are presented in Figure 11. Assume a user's job J is reading an input HDFS file F_{in}, and map task m_i processing block B_i will run on a cluster node N. If J does not involve annotation propagation, then no prefetching will take place (Lines 2-3). Otherwise, if m_i is the 1^{st} task in J to run on node N, then m_i will retrieve its annotations remotely from HBase (Line 5), and then the Annotation Manager on N will try to prefetch the annotations for the future anticipated map tasks (Lines 6-11).

The anticipation and prediction made by the Annotation Manager rely on the *data locality* property that the scheduler usually tries to maintain, i.e., a map task processing data block B_i will run, with a high probability, on a node storing B_i. Previous studies have shown that the standard Hadoop scheduler achieves more 92% locality success under replication factors of 2 or 3 [24]. Therefore, CloudNotes relies on this property to perform the prefetching mechanism. In addition, as will be described later in this section, the prefetching information will be passed to the Annotation-Aware Job Tracker to further increase the chances of making use of the prefetched annotation data on specific nodes.

In order to specify the source from which the annotations should be retrieved, the Annotation Manager will pass such instructions to the `AnnotationFetcher()` function within the mapper class, e.g., Lines 5, 15, and 19. To start prefetching, the Annotation Manager needs to predict which map tasks within job J will run on its node N. In order to do this, it will first identify all F_{in}'s data blocks stored in node N (called F_{in-N}), and then estimates how many of J's map tasks will run on N (called *PrefetchCnt*) (Lines 7-8). For example, if the number of blocks stored in N is $|F_{in-N}|$, and F_{in} has a replication factor *RepFactor*, then with a high probability the number of map tasks that will be assigned to node N is $|F_{in-N}|$ / *RepFactor* (Line 8). The Annotation Manager will then randomly select a number of entries equals to *PrefetchCnt* from F_{in-N}, and send these entries to the *Annotation-Aware Job Tracker* in the next heartbeat (Line 9-10).

As indicated in Line 8, we incorporate an aggressiveness factor (*AggressivenessFactor*) that ensures the Annotation Manager will not get overloaded with the prefetching task. This parameter is adaptively set by the Annotation Manager for each job independently, i.e., if the load on node N is not high, then for a newly coming job, the Annotation Manager may set *AggressivenessFactor* to 1. In this case, it will try to prefetch the annotations for the entire predicted list. In contrast, if the load becomes higher, then for the next job the *AggressivenessFactor* may be set closer to 0 indicating that the Annotation Manager will put less effort in the prefetching task. As indicated in Lines 11, 16, and 21, the Annotation Manager will not prefetch its entire list at once, instead the prefetching is performed in a progressive manner as the job advances, i.e., as some of the prefetched blocks are either processed, or missed (their tasks are assigned to other nodes), then more prefetching operations will take place (Lines 16, and 21, respectively).

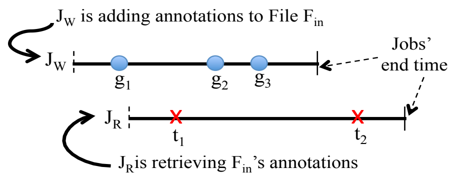
Lines 13-22 in Figure 11 handle the processing of subsequent tasks assigned to node N. If the annotations of the data block B_i are already prefetched, then the `AnnotationFetcher()` function will read them from the local file system (Line 15), otherwise it will query the HBase engine to retrieve them (Line 19). Notice that it is possible that B_i's related annotations may not be prefetched even though B_i is in the prefetch list *PrefetchList*$_{F-in-N}$, e.g., the Annotation Manager did not have enough time to prefetch B_i's annotations before having the task assigned to node N. Yet, since the prefetching mechanism is best-effort, CloudNotes will operate normally and the task will retrieve the annotations from HBase.

Finally, to further increase the chances that the prefetched annotation information will be used, the Annotation-Aware Job Tracker will collect the prefetch lists (*PrefetchList*$_{F-in-N}$) sent from the different nodes, and incorporate this information into the task scheduling policy, i.e., when scheduling a map task, a higher priority will be given to data nodes that can achieve both the *data* and *annotation locality*. If not possible, then the default scheduling policy of Hadoop takes place.

It is worth highlighting that we opt for designing a fully-distributed asynchronous prefetching mechanism instead of having the Annotation Coordinator controls this process because of the following reasons: (1) The Annotation Coordinator would still rely on the data locality property to predict where tasks would run, and this information is already known to each cluster node in isolation, (2) The proposed prefetching mechanism allows for the same annotation information to be prefetched on multiple nodes. While this may add some unnecessary overhead, it is a desirable feature that should be kept even under centralized decision as it gives more choices and flexibility to the task scheduler, and (3) It is not desirable to overload the Annotation Coordinator with information such as the load on each node, which ones are busy or free, etc. In a Hadoop-like environment, it is preferred to take such decisions independently and accordingly adjust the prefetching effort of each node (refer to the *AggressivenessFactor* in Figure 11). Because of these factors, we opt for the proposed mechanism over a centralized alternative.

5.3 Concurrency Control and Fault Tolerance

Concurrency Control: Concurrent execution in CloudNotes requires a special consideration since multiple concurrent jobs can be adding and/or retrieving annotations over the same data file at the same time. For example, the following figure illustrates two concurrent jobs J_W and J_R accessing the same data file F_{in}, where J_W is adding annotations while J_R is retrieving annotations.


J_W is adding annotations to File F_{in}
J_W
g_1
g_2
g_3
Jobs' end time
J_R
t_1
t_2
$J_R$is retrieving F_{in}'s annotations

If J_W's added annotations become instantaneously visible after each task to J_R, then inconsistent execution may occur, e.g., task $J_R.t_1$ will read the annotations generated from task $J_W.g_1$ but not the annotations from the rest of J_W's tasks because they did not execute yet. In addition, J_W may fail at any time, and thus J_R should not access J_W's annotations before its successful completion. Moreover, if J_W's annotations become visible to other jobs only when J_W completes, then inconsistent execution may still occur, e.g., task $J_R.t_2$ will be able to read J_W's annotations while the previous tasks in J_R did not have access to J_W's annotations.

To solve these concurrency issues, CloudNotes deploys a simple concurrency control mechanism that is based on enforcing two rules: (1) A job's output annotations are not visible to other jobs until the job's successful completion, and (2) A job starting at time t reads the annotations generated from only the jobs finished before t. These rules ensure that no job will read *dirty* or *partial* annotations from other jobs. Enforcing these two rules is in fact straightforward under the different storage schemes. For example, under the Naïve and Enhanced schemes, the names of the newly created annotation files in HDFS from a running job J_W will be prefixed with a special prefix *"_temp"* that makes these files invisible to other jobs. At J_W's completion, the Annotation Coordinator will rename these files to their final names and set their *last modification time* to J_W's completion timestamp. Meanwhile, when a job J_R is propagating the annotations, the `AnnotationFetcher()` function within the mapper class (Refer to Figure 7) will be responsible for reading only the related annotation files whose *last modification time* is smaller than J_R's start timestamp.

In the case of the Key-Based scheme, the annotations generated from a running job J_W will be written to a *temporarily* table in HBase. The Annotation Coordinator will move the content of this table to the main table only when J_W completes successfully. It will also ensure that the unique *version numbers* assigned to the written records is larger than or equal to J_W's completion timestamp. For a job J_R propagating the annotations, the `AnnotationFetcher()` function will ensure that the retrieved annotation records have a version number smaller than J_R's start timestamp. Notice that the timestamp checks performed by the `AnnotationFetcher()` function involve minimal overheads because they are not applied per annotation. Instead, they are applied at a larger granularity, i.e., an HDFS file level (in the cases of Naïve and Enhanced schemes), and an entire task's output level, which is written as a single record in HBase (in the case of the Key-Based scheme).

Fault Tolerance: Fault tolerance and recovery management is an inevitable component in all modern infrastructures. Some techniques manage failures by either providing partial results [31], re-doing the entire transaction [47], or re-doing the sub-tasks that have failed [48]. Hadoop fault tolerance belongs to the last category, and it is known for its simplicity and efficiency.

CloudNotes fully retains the desirable fault tolerance feature of plain Hadoop. For example, at the *Job Failure* level, the job's output annotations (if any) have not been visible to the outside world until the failure as discussed above, and thus they will be directly purged by the Annotation Coordinator. At the *Task Failure* level, the Annotation Manager for each task is keeping track of whether the failed task has flushed any of its output annotations (if any) to the Annotation Repository, and in this case such flush operations will be rolled back (See Case II in Figure 4). Handling a Reduce task failure in CloudNotes is of no difference to that in plain Hadoop because the annotations on the intermediate data are embedded within the data. And thus, the data and their annotations will be automatically re-retrieved for the new reducer. A failed mapper task will be re-scheduled by the Annotation-Aware Job Tracker using the same scheduling policy that targets achieving both the *data locality* and *annotation locality* (whenever possible). For that purpose, the prefetching information received by the Job Tracker from the different nodes (Section 5.2) remain unchanged as long as there is no node failure.

A *Node Failure* in CloudNotes includes the failure of either the node's local file system (Data Node), its Task Tracker, or its Annotation Manager. In this case, the Job Tracker will exclude this node from its list and discard its prefetching information. Moreover, the Job Tracker will inform the Annotation Coordinator to delete any annotation outputs produced from tasks that will be re-executed. The Annotation Coordinator will identify those to-be-deleted outputs by the job's unique Id along with the block Ids processed by these tasks. Notice that the storage of permanent annotations is either handled by HDFS and replicated on different nodes (in the case of the Naïve and Enhanced storage schemes), or handled by HBase (in the cases of the Key-Based scheme). In both cases, a node failure will not cause a loss of annotation information. The highest level of failure is a *Cluster Failure* in which one of the master node's components, i.e., Name Node, Job Tracker, or Annotation Coordinator, fails. In this case the entire cluster needs to be re-started, and all outputs of incomplete jobs will be purged.

6 Related Work

Query and Workflow Optimizations in MapReduce. As an open-source infrastructure, Hadoop/HDFS has been a very attractive platform for several research activities, e.g., high-level query languages [29,39], indexing techniques and query optimization [11,21,32], physical data-layout optimizations [4,24], workflow management techniques [9,18,20,37], among many others. Several of these techniques use the concept of *"annotations"* in specific context to carry special types of metadata information. For example, Nova [37] is a workflow management system on top of Pig/Hadoop. It uses special type of system-defined annotations to annotate the workflow graph with execution hints, e.g., the transfer mode of the data between processes, and the output format and schema of each task. Another system is the Stubby engine [33], which leverages the annotations for capturing the processing flow, collecting execution statistics, and profiling jobs' performance.

Nova and Stubby systems are not annotation management systems, instead they use *process-* and *workflow-centric* annotations that are system generated to optimize execution. CloudNotes is fundamentally different from these systems because it is a *data-centric* general-purpose annotation management engine, where the annotations are tied to the data not to processes or workflows, added by the applications to carry various types of metadata ranging from quality measures to auxiliary related documents, and also get automatically propagated whenever the data is accessed.

Provenance Management in MapReduce. Lineage tracing in Hadoop has been studied in literature, e.g., [6,7,9,38]. The Ramp system [38] focuses on tracking the data provenance within map-reduce jobs, i.e., it assigns artificial OIDs for each input data record, and then Ramp produces each output record along with its provenance information (the contributing input OIDs). The work proposed in [18] tracks the provenance in the *Kepler+Hadoop* system designed for scientific workflows. It extends the building blocks of Kepler, e.g., the actor nodes, to capture and propagate the provenance in the distributed Hadoop system. These techniques have shown to introduce significant processing overhead from provenance tracking, e.g., around 75% to 150% [7]. The HadoopProv [7] and MrLazy [6] systems addresses this issue of high overhead by separating the provenance tracking of the map and reduce phases (each phase writes its provenance information to disk separately), and then joining their results (only when needed) through another query to construct the final output-to-input lineage information.

The above mentioned techniques are all coarse-grained as the logic of the map and reduce functions is assumed to be blackbox, e.g., each output from a reduce function is linked to all its inputs (a concrete example is given in Section 7, Figure 19(c)). The Lipstick technique in [9] overcomes this limitation in the context of Pig workflows. Since Pig is a high-level declarative query engine, the Lipstick technique can track fine-grained record-level provenance information. Newt [34] is another fine-grained lineage tracking system, which uses the lineage information for debugging and tracking the errors in execution workflows. Newt further enables data mining and deeper analytics on top of the extracted lineage information for data cleaning. The techniques proposed in [3,5] also enable provenance analytics and visualization through a multi-layered architecture system.

All of the aforementioned techniques are focusing on a specific problem, which is *provenance tracking*. Although of a great importance to diverse applications, provenance tracking systems are distinct from general-purpose annotation management and curation engines. That is, (1) They cannot serve the diverse application scenarios presented in

Sections 1 and 7, (2) Their metadata information—which is the lineage information—carry only execution-driven and system-generated information, whereas modern applications usually have a much broader range of metadata information to be systematically captured and maintained, and (3) The tracked provenance information is isolated from the data at query time, i.e., if an application queries a data file F, there is no mechanism by which F's provenance information is automatically propagated with the data. Developers have to manually code (in each query) a retrieval mechanism of the provenance information—which is a tedious, error-prone, and complicated process.

CloudNotes is complementary to these techniques, and it overcomes the above mentioned limitations. On one hand, CloudNotes's annotations: (1) Are generated and created by the applications (not by the execution workflows), (2) Can carry different types of metadata information ranging from quality measures, provenance information, error highlights, auxiliary related articles or docs, etc., and (3) Propagate automatically as the data is queried, and thus application developers can directly leverage them in the data analytics cycle without the need to complicate the code of each query. On the other hand, CloudNotes addresses new challenges not present in the provenance-tracking systems, e.g., categorization and optimization of the annotation jobs via dependency graphs, annotation propagation supported by underlying prefetching, colocation, and compression techniques, and addressing the concurrency control among the data- and annotation-based jobs.

Annotation and Provenance Management in Relational Databases. Annotation management has been extensively studied in the context of relational DBMSs [10,26,27,41]. Several of these systems focus on extending the relational algebra and query semantics for propagating the annotations along with the queries' answers [10,26]. Other systems support special types of annotations, e.g., treating annotations as data and annotating them [16], and capturing users' beliefs as annotations [25]. In spite of their evident contributions, all of these techniques have focused on the centralized relational databases, and hence none of them is applicable to scalable, distributed, and cloud-based platforms such as Hadoop. Compared to these systems, CloudNotes is the first to bring the annotation and curation capabilities to Hadoop-based applications. Since relational DBMSs and the Hadoop/HDFS infrastructure are fundamentally distinct, CloudNotes is also profoundly distinct in its design, features, and capabilities from the traditional annotation management engines.

In relational database, since the data's structure is known in advance, the annotation management techniques, e.g., [10,14,26,41], as well as the provenance tracking techniques, e.g., [12,15], have addressed the challenges of multi-granular annotations, e.g., on specific table cells, rows, columns, etc. Some techniques as in the SubZero system [49] address the provenance at the array-cell granularity within an array data model. Unlike these systems, CloudNotes is built on top of the HDFS file system where no knowledge about the data organization is known (except the objects formed by the InputFormat layer). Therefore, in CloudNotes, the annotations are systematically managed only at that object-level granularity.

Another key difference is that in RDBMSs, the SQL query language is declarative and has the underlying relational algebra foundation. Therefore, the semantics of each operator in the query pipeline is known to the query engine. This has enabled complex transformations and processing on top of the annotations to be systematically applied at query time, e.g., transforming the annotations as they go though the projection, join, and grouping operators [10,26], tracking forward and backward the annotations on views [14], and studying relationships such as query containment [41] and provenance semirings [30]. Similar operations have been studied in the context of the Lipstick system over Pig/Hadoop [9] since Lipstick assumes known data schema and query constructs.

Unlike these techniques, CloudNotes is designed on top of a *blackbox* execution paradigm, where the logic inside the mappers and reducers is unknown. Therefore, CloudNotes's objective is to build an efficient infrastructure support, e.g., storage, propagation mechanisms, and optimizations, that brings the annotation and curation capabilities handy to the Hadoop-based application developers to utilize them in diverse applications (In the experiment section, we demonstrate five different application scenarios). An interesting future work is to extend CloudNotes to less-generic and more-expressive systems, e.g., Hive/Hadoop or Pig/Hadoop, where both the structure of the data and the query logic and constructs are known, and thus several of the aforementioned techniques can be investigated.

Other Emerging BigData Infrastructures. Several other computing paradigms over Big Data have been proposed including SQL-Based engines, e.g., Cloudera Impala [40], and Facebook Presto [46], in-memory processing engines, e.g., Apache Spark [2,51], and column-family systems, e.g., HBase [43]. Designing and building annotation management engines on top of these systems is an interesting and challenging open research question. For these systems, we envision fundamentally different annotation management engines from CloudNotes.

For SQL-Based engines, e.g., [40,46], the semantics of the annotation propagation will be based on the well-defined semantics of the SQL operators, e.g., select, project, join, grouping and aggregation. Therefore, similar to the state-of-art techniques in RDBMSs [10,26,27,41], each SQL operator need to be extended—at the semantic and algebraic level—to understand, manipulate, and propagate the annotations according to the operator's semantics. Since the data is assumed to have a known structure, then the issues of multi-granular annotations (on table cells, rows, columns, and arbitrary combinations of them) will need to be addressed as well. Moreover, the pipelined nature of processing in these SQL-Based engines and their ability to retrieve specific data tuple(s) without the need for complete scans will certainly require new distributed mechanisms for annotation addition and propagation.

On the other hand, Spark infrastructure [2,51] is an in-memory highly-distributed engine on top of which different types of processing can be supported, e.g., Spark-SQL, Spark-Streaming, and GraphX. Therefore, the design of an annotation management engine in Spark mandates a totally different design from CloudNotes, e.g., it would require an in-memory annotation management engine—otherwise it will become a bottleneck, and probably a multi-layered design that consists of a core generic component on top of which each type-specific engine, e.g., Spark-SQL and Spark-Streaming, can build customized components for annotation management.

HBase is another popular Hadoop-based storage engine. It is better suited for column-oriented structured data with frequent record-level lookup and retrieval. HBase has fundamental differences compared to HDFS that raises different challenges from those addressed by CloudNotes, and would require re-thinking the design methodology of an efficient annotation management support on top of HBase. The following table highlights these key differences.

Metric	HBase	HDFS
Storage model	Column Family & relatively structured data	No model & unknown structure
Access mechanism	Indexed record-level	Full scan
Write mode	Incremental record-level inserts & multi-version record-level updates	Batch upload & Read Only
Query mechanism	Set of APIs, e.g., get(), and put()	Blackbox map & reduce functions
Suitable Workload	Record-level lookups	Full scans, joins, & aggregations

Based on these differences, an annotation management engine for HBase would address different challenges including: (1) Multi-granular annotations, where annotations can be linked to subsets of table cells, columns, column families, or even an entire table, (2) Multi-version annotations, where different annotations may be attached to different versions of the same tuple, (3) ColumnFamily-driven storage schemes, where annotations may be best stored and colocated within the column families, (4) Annotation-aware HBase APIs, where the standard HBase APIs need to be extended to automatically manipulate and propagate the annotations (along with their related data), e.g., when the get() function selects specific column family(s), only the related annotations to those families should propagate, and (5) multi-version annotation propagation, where semantics and mechanisms need to be investigated on how annotations should (or should not) propagate across versions of the same tuple.

The proposed CloudNotes system is the first step towards supporting annotation management over Big Data infrastructures, and it opens this interesting research direction over other platforms, where each has its own inherent characteristics and challenges.

7 Experiments

In this section, we experimentally evaluate the CloudNotes system and compare it with plain Hadoop, Ramp, and HadoopProv systems. We consider: (1) Evaluating the performance of CloudNotes's newly added features, (2) Comparing the various design alternatives, and (3) Studying the effectiveness of the proposed optimizations.

7.1 Experimental Setup and Workloads

Cluster Setup: CloudNotes is developed on top of the Hadoop infrastructure (version 1.1.2). All experiments are conducted on a dedicated local shared-nothing cluster consisting of 20 compute nodes. Each node consists of 32-core AMD 3.0GHz CPUs, 128GB of memory, and 2TBs of disk storage, and they are interconnected with 1Gbps Ethernet. Each node runs CentOS Linux (kernel version 2.6.32), and Java 1.6. We used one server as the Hadoop's master node, while the other 19 servers are slave nodes. Each slave node is configured to run up to 20 mappers and 12 reducers concurrently. The following Hadoop's configuration parameters are used: sort buffer size was set to 512MB, JVM's are reused, speculative execution is turned off, and a maximum of 4GB JVM heap space is used per task. The HDFS block size for either of the data or annotation files is set to 64MB with a replication factor of 3. Each experiment is executed 3 times and the average values are presented in the figures.

ClickStream Datasets: We use real-world datasets and workloads collected from our collaboration with a click-stream company. The company collects the clickstreams from millions of users over thousands of websites, and analyze the data to understands users' behaviors and predict the ads to display on different sites for the different users. The data we experiment with is approximately 1.2TBs, and each record consists of the several information on each click-stream activity, e.g., the visited website, timestamp, the duration between clicks, the type of the OS and the Internet explorer used, the location of the ads within the page, the conversion ratio, etc. The record sizes range from several hundred bytes to few KBs of log information.

Annotation Workload: The company executes different types of jobs and analytical tasks in which annotation management can be of a real benefit. Among these jobs, we identified the following ones:

(1) *Data Verification and Cleansing:* in which data verification jobs—usually map-only jobs—execute over the data, verify the correctness of each record, e.g., whether or not it is well formatted, identify missing fields, detect and eliminate out-of-order records, etc., and produce a cleaner dataset. Currently, they execute around 10 variations of this job using different algorithms, tuning parameters, and quality thresholds, and each variation produces a separate dataset. Clearly, this involves significant storage overhead due to the possible overlapping among these datasets.

(2) *Model Building, and Sessionization Jobs:* in which they execute business-oriented jobs—usually map-reduce jobs— that are used for understanding and predicting users' buying behavior. For example, the sessionization job is to aggregate users' clickstreams based on the user's Id (IP address), sort the activities, apply some filtering and cleaning, and then divide them into logical sessions. Their objective is to carry the provenance information along with the output, e.g., for an identified user's session they want to track which records participated in that session. Moreover, for a constructed model, they want to assess the quality of its different components based on the quality of the input records participated to this component.

(3) *User-Based Profile Generation:* in which several log datasets, e.g., previous recommendations and displayed ads, the positions in which the ads have been displayed in the web page, the time of the day, and the user's action, will be joined and aggregated to build a profile for that user. Since these expensive jobs will execute frequently and will touch large subsets of data, it is of the company's interest to even piggyback annotation-generation tasks within the profile generation jobs—to avoid having dedicated scans for annotating the data.

CloudNotes can be leveraged to realize and optimize this workload as summarized in Figure 12. For the data verification jobs, CloudNotes can fully eliminate the need for creating different variations of the data by directly annotating the input records with the output of each verification task (Referred to as Job_1: *In-Situ Verification* in Figure 12). Each annotation will include the details of the underlying verification tool, e.g., the used algorithm, the configuration parameters, and any input functional dependencies expected to hold, as well as the tool's result such as *"Passed"* or *"Failed"*, the set of violated functional dependencies, or an assigned quality score between [0, 1]. In some cases, if a significant percentage of the input records is expected to fail the verification tasks, then the company prefers to create a single output dataset containing the union of records passing any of the verifications, and annotating them with the verification information and scores (Job_2: *Ex-Situ Verification* in Figure 12).

Job ID	Type	Description	Annotation-Only Job
Job_1 (In-Situ Verification)	Map-only	Annotating map-input data	Yes
Job_2 (Ex-Situ Verification)	Map-only	Annotating map-output data	No
Job_3 (Sessionization)	Map-Reduce (Aggregation)	Reduce-side propagation + Annotating reduce-output	No
Job_4 (Model Building)	Map-Reduce (Join & Aggregation)	Map-side propagation + Reduce-side propagation + Annotating reduce-output	No
Job_5 (User-Based Profiles)	Map-Reduce (Join & Aggregation)	Annotating map-input and/or reduce output	No

Fig. 12 Summary of the Annotation Workload.

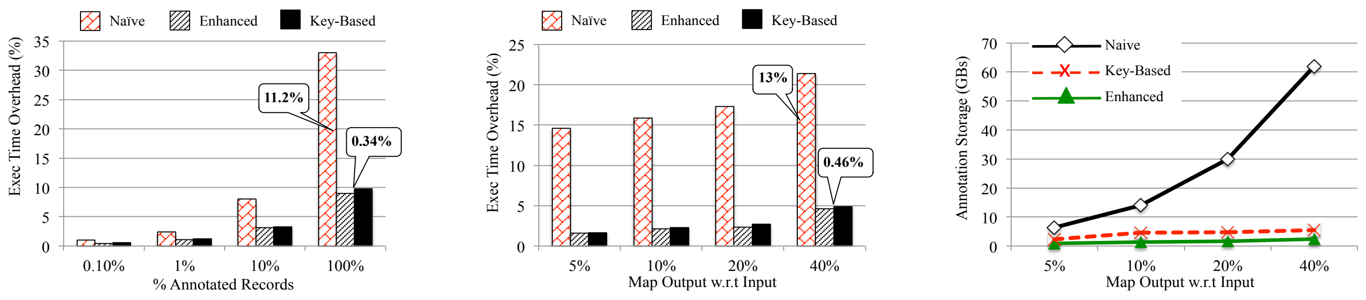
	Job_1	Job_2	Job_3	Job_4	Job_5
# Annotation-Related Lines	1	3	1 (in map) + 5 (in reduce)	4 (in map) + 5 (in reduce)	----

Fig. 13 Num of Additional Code Lines for Annotation Management.

For the sessionization aggregation job, CloudNotes will enable propagating the provenance information to the reduce side, and annotating the final output with such information (Job_3: *Sessionization* in Figure 12). In contrast, for the model building task, the system needs to retrieve the existing annotations, e.g., the quality scores, propagate them from the map side to the reduce side, and then the scores will be aggregated (e.g., average-based), and get attached to the final output (Job_4: *Model Building* in Figure 12). Finally, general-purpose jobs can be leveraged to annotate the inputs or outputs without the need for dedicated annotation-based jobs (Job_5: *Profile Generation* in Figure 12).

As mentioned before, in the workload we assume the annotations are attached at the object-level, i.e., the objects formed from the underlying input or output formats, as it is the unit of processing known to the system. In the case where an application needs to annotate specific fields, e.g., associating different quality measures to specific fields, then the application developer would need to encode the field's reference, e.g., position or name, within the annotation value.

In Figure 13, we show the usability of CloudNotes from the developer's point of view. The presented values indicate the number of code lines that need to be added for annotation management, i.e., adding or retrieving annotations (other than any application-related code). For example, in Job_1, after assessing the quality of each input record (according the verification tool semantics), only one additional code line is augmented to add the annotation (Similar to Line 5 in Figure 3). For Job_5, the number of additional code lines to be added depend on which annotation job among the other ones



- Avg. $T_{plain} \approx$ 388 sec.
- Indicated storage overhead w.r.t input size. It decreases proportional to the % of annotated records.

(a) Job_1: In-Situ Verification: Time Overhead

- Avg. $T_{plain} \approx$ 457 sec (5% output), 521 sec (40% output)
- Indicated storage overhead w.r.t. output size. The overhead % is the same for the different output sizes.

(b) Job_2: Ex-Situ Verification: Time Overhead

- Original dataset size is 1.2 TBs
- This storage overhead is from the least-selective verification task.

(c) Job_2: Ex-Situ Verification: Storage Overhead

Fig. 14 Annotation Addition and Propagation of Job_1 and Job_2.

is piggybacked within Job_5. The presented numbers demonstrate how easy developers can perform the basic operations on the annotations without the need to worry about the underlying infrastructure-related details, e.g., how the annotations are stored, how they propagate, when the annotation jobs are actually executed and optimized, the concurrency among the jobs, etc.

Default Parameter Settings: In the experiments, we will vary and study the effect of several parameters. However, unless otherwise is stated in a specific experiment, we assume the following default settings: (1) The Key-Based storage scheme is the default scheme, (2) The Enhanced and Key-Based schemes when used are full-fledged, i.e., they include bitmap compression, locality-based placement, and prefetching with aggressiveness factor set to 1, (3) The average annotation size is approximately 200 bytes, and the number of distinct annotations is 20, (4) The map and reduce tasks use a task-level buffer of size 20 to accumulate the newly added annotations before sending them to the Annotation Manager (Refer to Section 3.2), and (5) Map tasks do not require disk-swapping while propagating the annotations (Refer to Section 4.1).

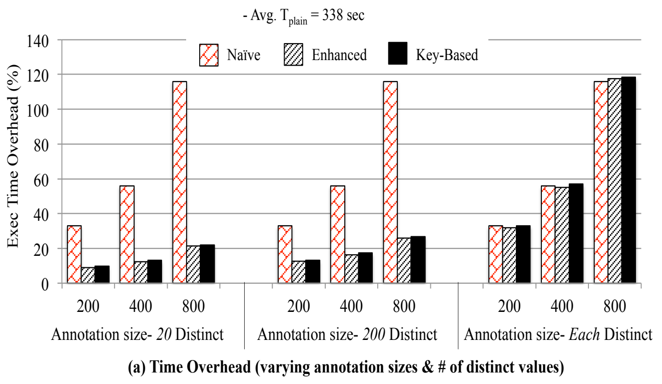
7.2 Performance Evaluation: Annotation Addition

In Figure 14(a), we evaluate the performance of Job_1, i.e., annotating the input data without producing any outputs. We execute each of the verification tasks to generate its annotations, and we report the averaged results over the 10 verification tasks. The annotation sizes range between 100 to 200 bytes. The number of distinct annotations generated by a tool is around 20 values, e.g., the algorithm name and parameters are fixed for a given execution, but the quality scores and the names of missing fields may differ. The tools typically annotate the entire dataset, i.e., 100% on the x-axis. However, to study the scalability pattern of CloudNotes, we vary the percentage of annotated records between 0.1% to 100% as indicated in the figure. The execution time overhead (y-axis) illustrate the overhead percentages on the job's completion time calculated as: $100 * (T_{annotation} - T_{plain})/T_{plain}$, where $T_{annotation}$, and T_{plain} are the job's completion time with annotation addition, and in plain Hadoop, respectively. For references, we include the absolute T_{plain} time in most figures.

The results show that when the underlying storage scheme is either the Enhanced or Key-Based, then the time overhead is relatively very small (at most 10% slowdown). In contrast, the Naïve scheme encounters around 33% overhead. This is mostly due to the space overhead occupied by the annotations. In the former two storage schemes, the overhead w.r.t the input data size is tiny (around 0.34%) because of the re-organization and compression before storing the annotations, while the Naïve method encounters around 11.2% storage overhead as indicated in Figure 14(a).

In Figure 14(b), we evaluate the performance of Job_2, i.e., producing map output for records under a certain quality level and annotating it on the fly. In this experiment, it is not straightforward to generate a union (without duplicates) from the datasets produced from all verification tasks. Therefore, the experts identify the least-selective tool along with its tuning parameters, and this will be the tool to run first to create and annotate a superset of tuples. Then, subsequent verification tasks can add more annotations as the execute over this superset data. The results in Figure 14(b) illustrate the performance of this least-selective tool. The x-axis shows the different sizes of the produced dataset relative to the original input, while the y-axis indicates the slowdown percent in execution time. The same trend as in Figure 14(a) is observed, where the Enhanced and Key-Based schemes achieve around 6x speedup compared to the Naïve storage scheme. This is due to the effective compression achieved by the first two schemes. This is evident from Figure 14(c) that illustrates the additional storage (for the annotations) created by the verification tool.

In Figure 15, we wanted to have a better understanding of how the size and the number of distinct values of annotations can affect the performance. This is clearly related



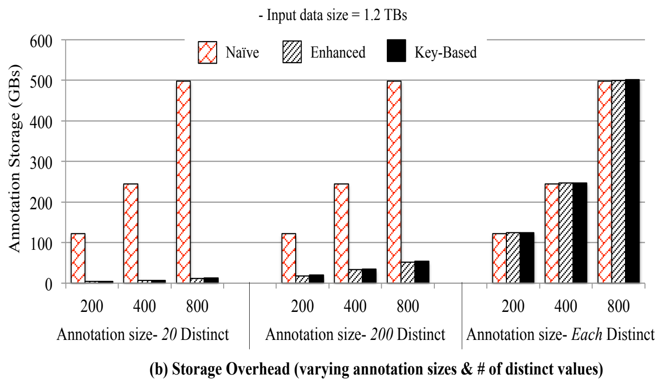
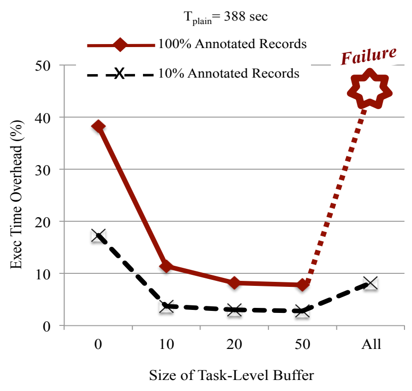


Fig. 15 Effect of Annotation Sizes and # of Distinct Values on Performance (Job_1).



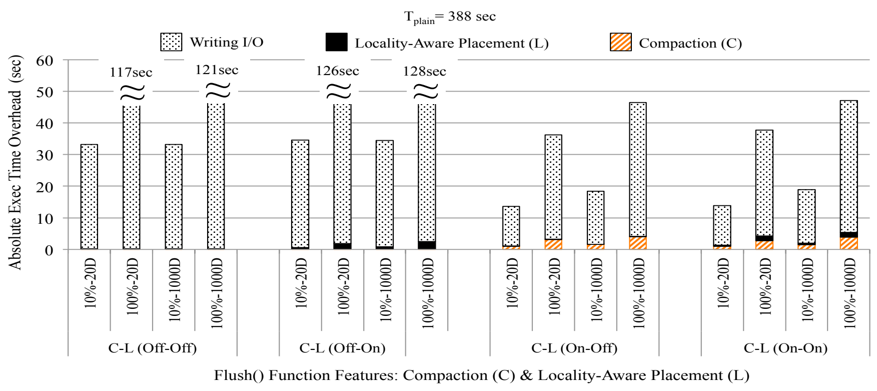


Fig. 16 Effect of Task-Level Buffer (Job_1). **Fig. 17** Effect of Enabling/Disabling Flush() Function Features (Job_1).

to the storage schemes and how effective the compression can be. In the experiment, we consider Job_1 (*In-Situ Verification*) for annotating the entire input dataset, i.e., approximately 1.2TBs. We consider three different values of annotation sizes in bytes {200, 400, 800}, and vary the number of distinct values over {20, 200, *Each*}, where *Each* means each record will get a different annotation. As the figures show, as long as the annotation values are not entirely distinct, the Enhanced and Key-Based schemes can perform orders-of-magnitudes better than the Naïve scheme. The execution time overhead (Figure 15(a)) is a direct reflection to the storage overhead (Figure 15(b)).

From Figures 14 and 15, we observe that the Key-Based scheme encounters a slightly higher overhead (usually negligible w.r.t. the entire job's time) compared to the Enhanced scheme. We contribute this difference to the following: (1) HBase is maintaining and storing additional columns, e.g, version numbers and timestamps, for each annotation record. These fields add between 2% to 5% of storage overhead on top of the annotation sizes. And (2) The network communication overhead between the HDFS nodes and the HBase storage nodes. This overhead in not present in the Enhanced scheme since the annotations and the data are *mostly* colocated, i.e., the annotations are read from the local file system. In subsequent experiments, we will further study the storage scheme with and without colocation and pre-fetching (Figure 20).

In Figures 16 and 17, we further analyze the effect of several optimizations on the annotation-addition process. More specifically, we study the effect of the task-level buffer located between a map or reduce task and the local Annotation Manager (Refer to Figure 4), and the effect of the bitmap compaction and the locality-aware annotation placement in the Flush() function (Refer to Figure 6). In these experiments, we use the Enhanced storage scheme, and re-execute the same experiment presented in Figure 14(a), i.e., Job_1*: In-Situ Verification*.

In Figure 16, we vary the task-level buffer on the x-axis from 0 (disabled) to *All* (holding all annotations until task completes) under two scenarios, which are annotating 10% and 100% of the input records. As the results show, the buffer has significant effect on the performance. If it is disabled, then the map tasks (20 running concurrently on a single node) are all sending each annotation to the Annotation Manager. Since the communication is performed thought inter-process-communication (IPC)—similar to the communication with the Task Tracker, the Annotation Manager becomes a bottleneck and it slows down the performance. Notice that with each message, the task needs to wait for the acknowledgment to resume its work. With a

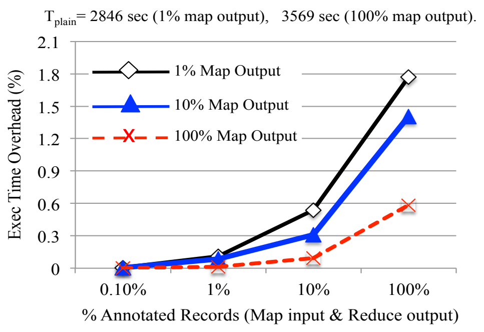


Fig. 18 Piggybacking Annotation Addition to Other Tasks (Job_5).

relatively small-size buffer, e.g., 10 to 50 annotations, the communication overhead becomes less severe and overhead from the annotation addition drops significantly.

On the other hand, when the buffer is set to infinity, i.e., a task sends its annotations only when it completes, tasks have failed because they ran out of memory. In this experiment all tasks have failed because each task annotates all its input records. However, in general different tasks may generate different number of annotations, and thus only subset of tasks may fail, which ultimately leads to a job failure. In general, we observed that there is no magical size that works the best for different scenarios. However, setting the buffer to a small size, e.g., 10 or 20 messages, is enough to make this step of the process a bottleneck-free.

In Figure 17, we study the effect of the two key steps within the Flush() function, i.e., the bitmap compression (denoted as 'C') and the locality-aware placement (denoted as 'L'). The x-axis is divided into four segments, where the C-L combination is set to either *On* or *Off*. And then, under each segment we consider 10% or 100% annotation of the map input, and the annotations' domain is either 20 or 1000 distinct annotations (denoted as 20D, and 1000D, respectively). The y-axis measures the absolute time added by the various components, which include writing the annotations to HDFS, execution the locality-aware placement policy, and applying the bitmap compression.

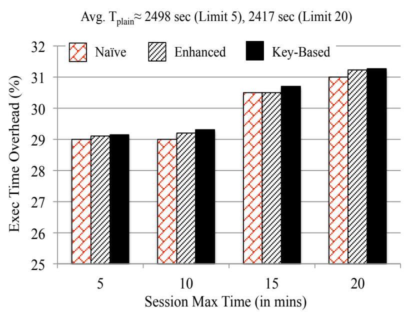
As expected, the dominant factor in all cases is the I/O of the writing phase compared to a minor CPU overhead for the over two components. However, by comparing the last two segments of Figure 17 to the first two segments, we conclude that the compaction step plays a clear role in reducing the I/O overhead (and hence the overall overhead). This is because the size of the annotations to be written becomes much smaller (as reported from other experiments in Figure 14(f-g)). When the locality-aware placement feature is enabled, i.e., the 2^{nd} and 4^{th} segments, the added overhead is minor and unnoticeable, but it has a positive impact on the annotation propagation as discussed next.

In Figure 18, we study piggybacking the annotation addition tasks on other jobs, e.g., since some user tasks will scan the data anyway, they may annotate this data for free without the need to run annotation-specific jobs. In this experiment, we use Job_5 (Refer to Figure 12), the average annotation size is set to 200 bytes, the number of distinct annotations is 200 values, and the storage scheme is Key-Based. The job will annotate a percentage of the map-input records and reduce-output records (the x-axis in Figure 18). We study the performance overheads under the cases where the map output (going to the reduce side) is either 1%, 10%, or 100% of the map's input. As the results show, since Job_5 is an expensive map-reduce job, then annotating the data can be piggybacked almost for free. As the figure illustrates, the overheads are mostly below 1% of the job's total execution time. This is a desirable feature of CloudNotes , especially in the cloud-based *pay-as-you-go* model [17] in which clients are keen to optimize their workloads and pay less.

7.3 Performance Evaluation: Annotation Propagation

We start by studying the sessionization query define in Job_3 in Figure 12, i.e., grouping users' transactions, sorting them by time, dividing them into logical sessions, and then reporting statistics about each session. The goal is to keep track of the provenance information of the records contributing to each session. In this job, the map-side functions are not accessing any existing annotations, instead they are generating their own annotations, i.e., annotating their output records with unique OIDs and passing them to the reduce side.

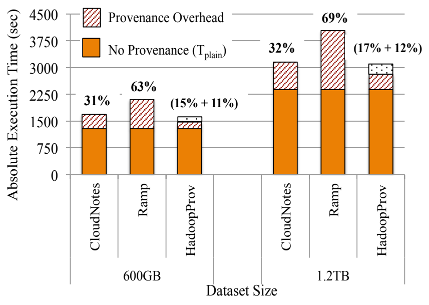
In Figure 19(a), we vary the session limit (max allowed time for a session) over the x-axis, and as this limit gets larger, then number of sessions gets smaller. This corresponds to a smaller reduce-output size. The job without any annotation propagation takes around T_{plain} = 2498 sec (for Limit=5) as indicated in the figure. The results in this experiment show a different trend compared to the previous experiments, which is that the three storage schemes have relatively the same performance. This is expected because the tasks of generating the annotations (OIDs) from the map-side (the OIDs), and going through the shuffling and sorting phase until reaches the reduce side are identical in the three schemes. Moreover, since the annotations (provenance) produced for each reduce-output record are distinct, then the compression effectiveness is almost zero. And thus, the three schemes become relatively similar in performance. In Figure 19(b), we report a more detailed analysis of the sessionization job under the Key-Based storage scheme, and a max session time of 10 mins. The presented table illustrates the overheads introduced at each processing stage due to the provenance propagation. Clearly, the major overheads lie in the shuffling/sorting phase, and the output generation phase. In both phases, the increase in time is due to the increase of shuffled/sorted data, and the produced output.



(a) Provenance Propagation in Job_3 (Sessionization)

Measure / *Job*	Avg. Map time (sec)	Shuffle/Sort time (sec)	Avg. Reduce time (sec)	Intermediate data size (GB)	Output Size (GB)
Plain	17.2	674.5	137	458	144
Annotation Propagation	19.3	1090	153.6	553	223

(b) Detailed Analysis of Job_3:
{Session time = 10mins, Storage Scheme = Key-Based}



(c) Comparison with Ramp & HadoopProv:
(Modified Sessionization Job)

Fig. 19 Annotation Propagation (Reduce-Side) and Comparison with Ramp & HadoopProv Systems (Job_3).

Although Job_3 focuses on provenance propagation, we could not directly compare CloudNotes with the other provenance systems, e.g., Ramp [38], and Hadoop-Prov [7]—which are designed specifically for MapReduce-based provenance tracking. The reason is that user sessions are logical units that reducers compute internally, and thus Ramp and HadoopProv cannot capture the provenance at such logical level because they treat the reduce function as a blackbox. For example, if one user has 100 click events (which is an input to one reduce execution), and the reducer divides them into 5 sessions (according to their time constraints), e.g., Session 1 has 30 events, Section 2 has 15 events, etc., then to Ramp and HadoopProv, each output of these five sessions is linked to all the 100 input events, which is clearly not the correct application semantics. This is an example of many-to-one inputs-to-outputs that black-box provenance tracking systems cannot handle.

To have a fair comparison with these systems, we modified Job_3 such that instead of reporting each session, the job will compute statistics over all of the user's sessions, e.g., the average session time across all of the users' sessions. Hence, the each reducer input produces one output record to which all of the inputs have contributed. In this case Cloud-Notes , Ramp, and HadoopProv are semantically equivalent.

The results from this experiment are presented in Figure 19(c). We experiment with two dataset sizes (the x-axis), and measured the absolute execution time to follow the same representation in [38]. The CloudNotes and Ramp systems are almost identical on how mappers generate the provenance information, and how this information is carried within the shuffling and sorting phases until reaches the reducers. However, the key difference is that Ramp is less effective in storing the provenance information, i.e., it introduces a mapping layer where the provenance information is linked to the data though a primary-key foreign-key relationship, and the foreign key gets replicated with each record Id contributing to the output. This adds significant storage overhead, which translates to the illustrated execution time overheads. The results illustrate that Ramp's overhead is double the overhead encountered by CloudNotes.

On the other hand, HadoopProv does not fully compute the output-to-input provenance during the user's job—with the objective of minimizing the provenance tracking overhead that user's jobs experience. Instead HadoopProv generates partial isolated information from the map and reduce phases. That is why the the execution overheads encountered during the user's job is relatively smaller as depicted in Figure 19(c), i.e., 15% and 17% for the 600GBs and 1.2TBs datasets, respectively. To compute the final provenance graph, HadoopProv needs a separate job to join the isolated information and generate the final output, which is equivalent to CloudNotes and Ramp outputs (See Figure 19(c)).

This experiment shows that CloudNotes not only can serve a broader range of applications compared to the provenance-tracking systems, but also has comparable performance to HadoopProv. This is due to CloudNotes's effective design and storage including compression, colocation, pre-fetching, and annotation-aware task scheduling, which collectively save significant I/O overhead as well as shuffling/sorting overheads.

The performance of Job_4, which is a map-reduce job of model building, is presented in Figure 20. Typically, not a single model is generated from the data. Instead, different models are generated for different types of users, different product types, or different regions in the world. The goal is to estimate the quality of a given model based on the qualities of its contributing records. Job_4 is distinct from the other jobs in that it involves map-side propagation, i.e., the map functions read the annotations attached to the input data (which are the quality scores) and pass them to the reduce functions. On the x-axis we vary the number of annotations attached to each input record, e.g., 1 means one available annotation from one verification tool, while 10 means 10 available annotations from the 10 verification tools. Since pre-fetching and annotation-to-data locality are important in map-side propagation, we study the performance under five storage schemes, i.e., *Naïve*, *Enhanced*—with colocation

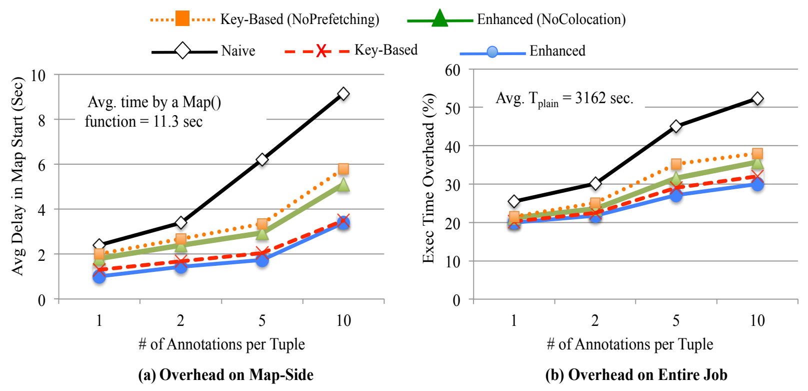


Fig. 20 Annotation Propagation (Map & Reduce Sides) of Job_4.

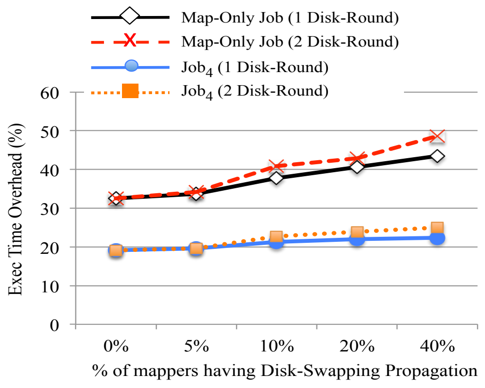


Fig. 21 Memory-Constraint Annotation Propagation.

enabled, *Enhanced (NC)*—meaning No Colocation, *Key-Based*—with pre-fetching enabled, and *Key-Based (NP)*—meaning No Pre-fetching.

In Figure 20(a), we present the average delay time (in sec) encountered by the Map() functions before they start executing. This delay is due to the retrieval of the annotations and organizing them in memory by the Annotation Manager. The figure illustrates that the execution delay depends on the underlying storage scheme. The Naïve scheme is the worst as it requires reading many small files from HDFS, while the Enhanced scheme is much faster because of its compression on disk, and consolidation of the small files into a single large file. When the colocation property and the pre-fetching mechanism are disabled, the two schemes encounter additional overhead (approximately 20% in the worst case).

The overall overhead percentage for the entire Job_4 is presented in Figure 20(b). Since Job_4 is a map-reduce job, then the big gap between the various storage schemes (from the map-side) narrows down when we consider the entire job. Nevertheless, still the Enhanced (with colocation) and the Key-Based (with pre-fetching) are the dominant schemes.

Based on our analysis w.r.t the different storage schemes, we conclude that: (1) Both Key-Based and Enhanced schemes outperform the Naïve scheme (the baseline) almost in all scenarios, (2) Key-Based encounters a slight overhead compared to the Enhanced scheme due to the network communication and the additional storage for maintaining auxiliary columns in HBase, and (3) In the case where HBase is already deployed on the cluster, then it is better to use the Key-Based scheme since it neither requires a daemon process nor has a freezing state (Refer to Figure 5). Otherwise, CloudNotes can efficiently leverage the HDFS using the Enhanced scheme without mandating the deployment of HBase.

In Figure 21, we study the case where the main-memory data structures built by the Annotation Manager for organizing the annotations (Refer to Figure 7) do not fit in the allocated memory space. In this case, part of the annotations will be stored to and read from the local disk. In the experiment, we study the slowdown in performance on Job_4, which is a map-reduce job, as well as a synthetic map-only. In the synthetic job, the map side will perform the same functionality as in Job_4, i.e., reading the log data and the attached quality-related annotations, except that there is no reduce-side. Since the map-only job is cheaper than the map-reduce job, we expect the disk-based propagation to be more influential in the former case. In this experiment, we assume a Key-Based storage scheme, and that each data tuple has a single quality-related annotation attached to it.

Referring to Figure 21, we vary, over the x-axis, the percentage of data blocks (map tasks) for which a read from disk is needed. We evaluate two cases: (1) The need to write and read from the disk only once (labeled `1 Disk-Round`), and (2) The need to write and read from the disk twice (labeled `2 Disk-Round`). The experiment confirms that CloudNotes is resilient to failures even under memory constraints since both jobs completed successfully. However, as expected, the more map tasks requiring disk-swapping during the propagation, the higher the overhead on the job's execution. Although the disk accesses are all local I/Os, i.e., no HDFS access, there is a clear slowdown, especially for the map-only job. This is because it is a less-expensive job (compared to Job_4), and hence it is more sensitive to this overhead.

7.4 Performance Evaluation: Advanced Features

In Figure 22(a), we evaluate the shared and lazy execution strategy among a set of annotation-only jobs, e.g., the different variations of the in-situ verification tool of Job_1. On the x-axis, we vary the number of annotation-only jobs that can be shared from 1 to 16 (all in one level in the dependency graph), and measure the required total execution time under the sharing case and the case where the sharing is disabled. Clearly, the sharing strategy of CloudNotes can be very effective in saving system's resources. The ultimate savings

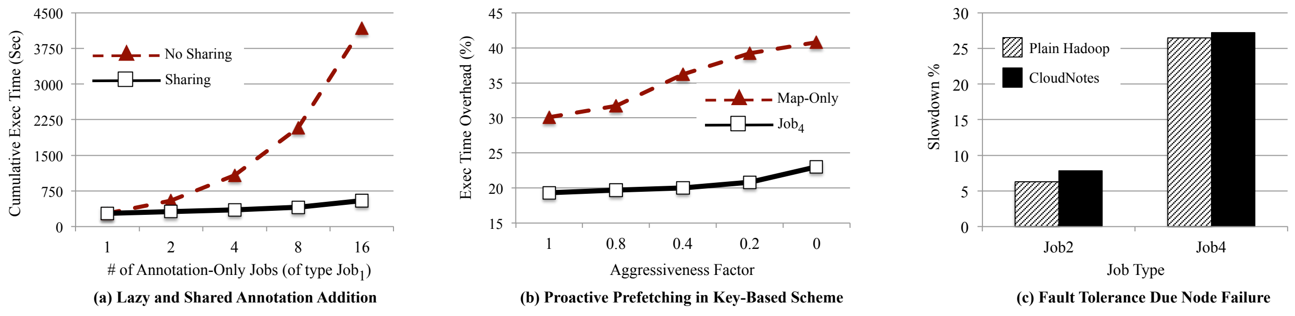


Fig. 22 Evaluation of CloudNotes's Advanced Features.

will depend on the ratio of the annotation-only jobs to the other types of jobs as well as the their execution order. But, in general, even a sharing of a small number of jobs, e.g., 4 or 8 which can be the max limit set for the *Max-Size Refresh Point*, can save significant time and resources. As such, the overheads encountered by the individual jobs, which may reach 20% or 30% in some cases, would be substantially reduced as they share their execution, and most of the I/O and computations will be shared.

To assess the importance of the *End-Of-Level Refresh Point*, we performed an experiment to measure the wait time of a user's job (in the *Mandatory Refresh Point*) under various heights of the annotation's dependency graph (from 1 to 4 levels). The results are included in the following table.

	Graph height & Num jobs in each level			
	1 ("4")	**2 ("4-1")**	**3 ("4-1-4")**	**4 ("4-1-4-1")**
Wait Time	352 Sec	630.6 Sec	1013 Sec	1405 Sec

As indicated in the table, the increase in the graph's height translates to big jumps in the job's wait time. That is why CloudNotes eagerly triggers the *End-Of-Level Refresh Point* to avoid increasing the graph's height, especially that there is no sharing across levels.

The results in Figure 22(b) show the sensitivity of annotation propagation (under the Key-Based scheme) to the *aggressiveness factor* that controls the prefetching of annotations. This factor is used for jobs involving map-side propagation. Therefore, we evaluate the performance for the two job types studied in Figure 21, i.e., Job_4, and its map-only variation. We assume the number of annotations per tuple is one, and we vary the aggressiveness factor over the x-axis between 0 (no prefetching) to 1 (very aggressive in prefetching). The results show that indeed the prefetching has a positive effect of the job's completion time. The map-only job is more sensitive to the prefetching because its execution time is small compared to the other two jobs. And hence, the overhead from communicating with HBase becomes more significant for the map-only job. This results can be used to tune the Annotation Manager to be more aggressive in prefetching in the cases of map-only jobs, while being less aggressive in the map-reduce jobs.

In Figure 22(c), we study the performance of CloudNotes under a node failure. In the experiment, we take one slave node down at the 50% of the job's completion time. And hence, its data will be lost, and its previous tasks need to be re-executed. We experiment with the two jobs, Job_2 and Job_4, and in each case, we measure the slowdown percentage relative to the job's completion time without a failure, i.e., the plain Hadoop job will be compared relative to the T_{plain} time, while the other jobs will be compared to the $T_{annotation}$ time. The underlying storage scheme in the experiment is set to Key-Based. As studied in [4,24], it is typical that the recovery from a failure in a map-reduce job be more expensive than that in a map-only job. Nevertheless, with respect to the annotation management, CloudNotes retains almost the same performance as plain Hadoop when recovering from a failure. The annotation addition job Job_2 encounters slightly higher overhead since the Annotation Coordinator needs to delete the annotations written by the failed tasks before starting their execution again.

8 Conclusion

We proposed the CloudNotes system, the first MapReduce-based annotation management engine. CloudNotes makes it feasible for cloud-based applications relying on Hadoop/HDFS to seamlessly integrate both the data and the annotations into the same processing cycle. Compared to the state-of-art techniques, CloudNotes addressed several unique characteristics to annotation management including: (1) The management and processing of annotations, e.g., generation, storage, indexing, and automatic propagation, in a large-scale and distributed environment, (2) Novel categorization of CloudNotes's jobs based on their behaviors, and proposing new sharing strategies that take their dependencies into account, and (3) New design issues and optimizations, e.g., lazy evaluation, prefetching mechanisms, annotation-aware task scheduling, and concurrency control mechanisms for the concurrent read/write annotation jobs. Promising future work and extensions to CloudNotes include investigating the annotation management problem over other Big Data infrastructures.

References

1. IBM InfoSphere BigInsights. http://www-01.ibm.com/software/data/infosphere/biginsights.
2. Spark: Lightning-fast cluster computing. https://spark.apache.org.
3. *Milieu: Lightweight and Configurable Big Data Provenance for Science*, Santa Clara, CA, 2013. IEEE.
4. A. Abouzeid and et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB*, pages 922–933, 2009.
5. R. Agrawal, A. Imran, C. Seay, and J. Walker. A layer based architecture for provenance in big data. In *Big Data*, pages 1–7, 2014.
6. S. Akoush, L. Carata, R. Sohan, and A. Hopper. MrLazy: Lazy Runtime Label Propagation for MapReduce. In *HotCloud*, 2014.
7. S. Akoush, R. Sohan, and A. Hopper. HadoopProv: Towards Provenance as a First Class Citizen in MapReduce. In *USENIX Workshop on the Theory and Practice of Provenance*, 2013.
8. Amazon Elastic MapReduce. Developer Guide, API Version 2009-03-31, 2009.
9. Y. Amsterdamer, S. B. Davidson, D. Deutch, and et.al. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, pages 346–357, 2011.
10. D. Bhagwat, L. Chiticariu, and W. Tan. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.
11. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, 2010.
12. P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, pages 539–550, 2006.
13. P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren. Curated databases. In *Proceedings of the 27th ACM symposium on Principles of database systems (PODS)*, pages 1–12, 2008.
14. P. Buneman and et. al. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.
15. P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. *Lec. Notes in Comp. Sci.*, 1973:316–333, 2001.
16. 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.
17. R. Buyya. Market-Oriented Cloud Computing: Vision, Hype, and Reality of Delivering Computing As the 5th Utility. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 1–15, 2009.
18. D. Crawl, J. Wang, and I. Altintas. Provenance for MapReduce-based data-intensive workflows. In *WORKS workshop*, pages 21–30, 2011.
19. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.
20. J. Dias, E. Ogasawara, and et.al. Algebraic dataflows for big data analysis. In *International Conference on Big Data*, pages 150–155, 2013.
21. J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11):1591–1602, 2012.
22. M. Y. Eltabakh, W. G. Aref, A. K. Elmagarmid, M. Ouzzani, and Y. N. Silva. Supporting annotations on relations. In *EDBT*, pages 379–390, 2009.
23. M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. Haas, H. Pirahesh, and J. Vondrak. Eagle-Eyed Elephant: Split-Oriented Indexing in Hadoop. In *EDBT*, pages 89–100, 2013.
24. M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.
25. 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.
26. F. Geerts and et al. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, page 82, 2006.
27. F. Geerts and J. Van Den Bussche. Relational completeness of query languages for annotated databases. In *DBPL*, pages 127–137, 2007.
28. T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. MapReduce in the Clouds for Science. In *CloudCom Conference*, pages 565–572, 2010.
29. *Hive*. http://hadoop.apache.org/hive.
30. G. Karvounarakis and T. J. Green. Semiring-annotated Data: Queries and Provenance. *SIGMOD Rec.*, 41(3):5–14, 2012.
31. W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial Results in Database Systems. In *SIGMOD*, pages 1275–1286, 2014.
32. B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, pages 985–996, 2011.
33. H. Lim, H. Herodotou, and S. Babu. Stubby: A Transformation-based Optimizer for MapReduce Workflows. *PVLDB*, 5(11):1196–1207, 2012.
34. D. Logothetis, S. De, and K. Yocum. Scalable Lineage Capture for Debugging DISC Analytics. In *SOCC*, pages 17:1–17:15, 2013.
35. D. Logothetis, C. Trezzo, and K. C. Webb. In-situ mapreduce for log processing. In *USENIXATC,*, pages 9–9, 2011.
36. T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, pages 494–505, 2010.
37. C. Olston and et.al. Nova: continuous pig/hadoop workflows. In *SIGMOD Conference*, pages 1081–1090, 2011.
38. H. Park, R. Ikeda, and J. Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. In *VLDB*. Stanford InfoLab, August 2011.
39. *Pig*. http://hadoop.apache.org/pig.
40. J. Russell. Couldera-Impala. *O'Reilly Media*, 2013.
41. W.-C. Tan. Containment of relational queries with annotation propagation. In *DBPL*, 2003.
42. The Apache Foundation. Hadoop. http://hadoop.apache.org.
43. The Apache Foundation. Hbase. http://hbase.apache.org/.
44. The Apache Software Foundation. HDFS architecture guide. http://hadoop.apache.org/hdfs/docs/current/hdfs-design.html.
45. A. Thusoo, R. Murthy, J. S. Sarma, Z. Shao, N. Jain, P. Chakka, S. Anthony, H. Liu, and N. Zhang. Hive - a petabyte scale data warehousing using hadoop. In *ICDE*, 2010.
46. M. Traverso. Presto: Interacting with petabytes of data at Facebook. 2013.
47. J. Ullman. Principles of database and knowledge-base systems. volume 1, 1988.
48. T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 3rd edition, 2012.
49. E. Wu, S. Madden, and M. Stonebraker. SubZero: A Fine-grained Lineage System for Scientific Databases. In *ICDE*, pages 865–876, 2013.
50. D. Xiao and M. Y. Eltabakh. InsightNotes: summary-based annotation management in relational databases. In *SIGMOD Conference*, pages 661–672, 2014.
51. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.
52. C. Zhang, J. Naughton, D. Dewitt, and Q. Luo. On supporting containment queries in relational database management systems. In *In SIGMOD*, pages 425–436, 2001.
53. C. Zhang, H. D. Sterck, A. Aboulnaga, H. Djambazian, and R. Sladek. Case Study of Scientific Data Processing on a Cloud Using Hadoop. In *HPCS*, pages 400–415, 2009.