2014-04-24

# CloudNotes: Annotation Management in Cloud-Based Platforms

Yue Lu
*Worcester Polytechnic Institute*

# CloudNotes: Annotation Management in Cloud-Based Platforms

by

Yue Lu

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

---

May 2014

APPROVED:

---

Professor Mohamed Eltabakh, Thesis Advisor

---

Professor Elke A. Rundensteiner, Thesis Reader

---

Professor Craig Wills, Department Head

**Abstract**

We present an annotation management system for cloud-based platforms, which is called CloudNotes. CloudNotes enables the annotation management feature in the scalable Hadoop and MapRedue platforms. In CloudNotes system, every piece of data may have one or more annotations associate with it, and these annotations will be propagated when the data is being transformed through the MapReduce jobs. Such an annotation management system is important for understanding the provenance and quality of data, especially in applications that deal with integration of scientific and biological data at unprecedented scale and complexity. We propose several extensions to the Hadoop platform that allow end-users to add and retrieve annotations seamlessly. Annotations in CloudNotes will be generated, propagated and managed in a distributed manner. We address several challenges that include attaching annotations to data at various granularities in Hadoop, annotating data in flat files with no known schema until query time, and creating and storing the annotations is a distributed fashion. We also present new storage mechanisms and novel indexing techniques that enable adding the annotations in small increments although Hadoops file system is optimized for large batch processing.

**Acknowledgements**

First and foremost, I wish to thank my advisor, Prof. Mohamed Eltabakh, a respectful, responsible and resourceful scholar, who gave me constant guidance, revised my paper with great care and offered me invaluable advices and informative suggestions.

Then I owe my gratitude to my thesis reader, Prof. Elke Rundensteiner, for her patient help and generous encouragement, which support me keeping improving this thesis.

I would also thank Mark Taylor, system administrator in WPI. He is always kind and supportive in helping me build the experiment environment, which is fundamental to my research.

My sincere thanks also goes to Dongqing Xiao and Yuguan Li, who also participated in this project and made great contribution to the project and my thesis as well.

I shall extend my thanks to all the DSRG members: Chuan Lei, Karim Ibrahim, Lei Cao, Qingyang Wang, Xiao Qin, Xika Lin, Ying Wang, Zhongfang Zhuang, for their inspirations and feedback on my research.

Last but not least, I would thank my parents: Junxin Lu and Shuming Chen for always being there for me during my research as well as my entire life.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Todays emerging applications in science are all generating and collecting data at unprecedented scale and complexity. Its not only the data is large and complex, but also the processing and analytics of data are complex. Thus, scientific applications are turning into cloud computing and scalable cloud-based platforms such as Hadoop [1, 10], which is the open source implementation of Google MapReduce [2]. Hadoop is a widely used cloud-based platform due to its superior properties, such as scalability to peta bytes of data over thousands of machines, flexibility in managing (un)structured data, elasticity in growing and shrinking resources, cost-effectiveness, and availability as an open source.

On the other hand, Annotating and curating the data plays a significant role in scientific experimentation and discovery process [3, 4, 5, 6]. Annotations can capture scientists notes and comments in all stages of the discovery process. They can also be unstructured free-text objects used for exchanging knowledge and Q/A messages among scientists, highlighting erroneous or conflicting values, and capturing users

understanding of data. Furthermore, annotations can be of specific type with well-defined structure and semantics to capture, for example, the provenance and lineage information for tracking the origin of the data.

With the advances in data management, there was a pressing need to capture and query the annotations in more systematic and efficient ways. Annotation management has been extensively studied in the context of relational database systems. Most of these techniques built generic frameworks for managing annotations, e.g., storage, indexing, and propagation at query time. Other systems create specialized systems for specific types of annotations, e.g., provenance tracking, and belief capturing.

We address the annotation management in the Hadoop platform. Although few systems have exploited specific types of annotations in Hadoop, i.e., the Ramp [7, 8] and PigLipstick [9] systems for tracking the data provenance and Stubby engine [11] for capturing the execution statistics and optimizing performance. We demonstrate that our proposed system (called CloudNotes) enables a general-purpose annotation management platform, and supports different types of annotation and meeting the diverse requirements of scientific applications. In this project, we propose to design and develop the CloudNotes system, an extensible annotation management system for cloud-based scientific applications.

## 1.2 Motivation Scenarios

### 1.2.1 Scenario 1: Data Verification and Revision

Scientific data are always subject to verification and possible corrections. With the current Hadoop-based technology, each step in this process must create a new dataset as presented in Figure 1.1. For example, a data verification tool may scan the

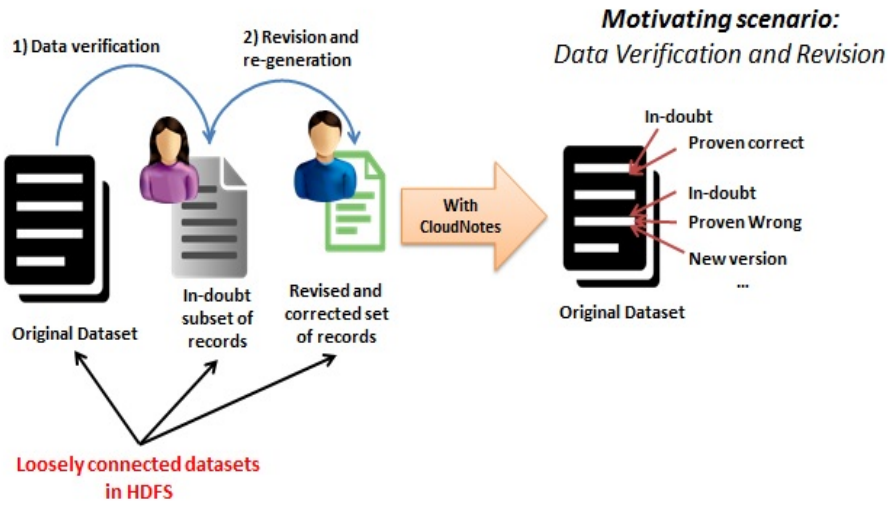


Figure 1.1: Motivation Scenario1: Data Verification and Revision

original dataset and create another dataset for the in-doubt records. Then, a revision phase, which may involve external activities, e.g., running wet-lab experiments, will create a third dataset for the proven-wrong and revised records. Now, the system has three datasets loosely connected and none of them is complete or directly query able, i.e., the original dataset still contains the wrong records, while the third dataset has only the revised records. The management of these datasets can easily go out of control, especially with multiple possible revisions. Note that the provenance tracking systems may help in propagating the provenance from the original dataset to the in doubt one. However there is no way to link the third dataset to the existing ones or to efficiently integrate them at query time. With CloudNotes (See Figure 1.1), scientists will be able to annotate the original dataset and highlight the in-doubt tuples, mark the ones proven wrong or correct, and even provide corrections and link them to the original records. Thus, there will be no need to fragment the data across multiple datasets, and the querying will be more efficient since these annotations will automatically propagate along with the original dataset whenever queried.

### 1.2.2 Scenario 2: Exchange of Auxiliary Information

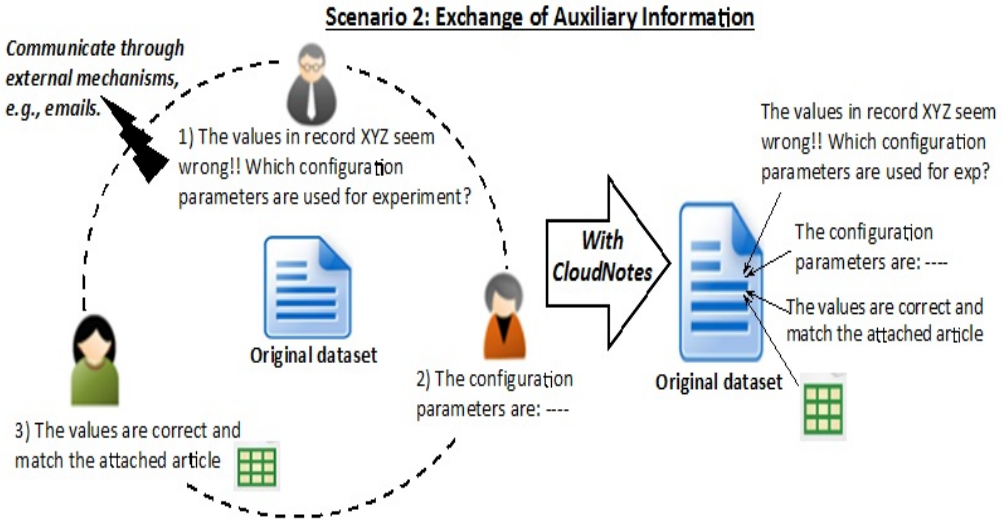


Figure 1.2: Motivation Scenario2: Exchange of Auxiliary Information

It is typical in scientific applications to share the data among multiple users as illustrated in Figure 1.2. Users may want to exchange information about the data stored in Hadoop, e.g., one user asks about the configuration parameters, a third user supplies an article matching and confirming the values in the dataset. These auxiliary and valuable information is typically exchanged outside the system, e.g., through emails, simply because there are no means inside Hadoop to share or capture this knowledge. This will, most probably, lead to the loss of the exchanged information will not be available. In contrast, CloudNotes will enable these annotations to be captured in a systematic way, persistently kept with the data, and be available to users at query time. Furthermore, CloudNotes can analyze the available annotations and infer, for example, that the record under investigation is of a high quality because its content is support by two users as well as an article.

### 1.2.3 Scenario 3: Fine Grained Data Authorization

Authorization in Hadoop is at the file level. Although useful, the records in a given file may contain data at different level of sensitivity, and hence warrant the need for a finer-grained access control (at the record level), which is not currently possible. With the current Hadoop-based technology, the original file may be broken into multiple fragments; each contains a set of records with the same access level. This approach is cumbersome for applications reading records across fragments. Moreover, it is problematic (almost infeasible) if the authorization levels change over time. With CloudNotes, the authorization levels can be modeled as annotations attached to each record according to its content. At query time, users jobs will be granted access to specific records based on their attached authorization levels. Note the unlike the original data that is read-only, annotations can be incrementally added to the data to reflect changes in the authorization levels over time.

### 1.2.4 Scenario 4: Assessment and Propagation of Data Quality

Scientific records may have different qualities based on their sources, the accuracy of their values, or even the observations and comments from users. Hence, scientists may want to run software tools over their data and assign quality scores to each record. And then, at query time, we may want to carry (or aggregate) these quality scores from the input records to the derived ones. With the current technology, this scenario is very challenging to handle. In contrast, in CloudNotes, the quality scores (can be many from different tool) will be modeled as annotations attached to each data record, and then users can define propagation strategies that allow automatic propagation (and aggregation) of these scores from the input records to the derived

ones.

In summary, these scenarios, which are very common in science domains, have demonstrated critical shortcomings in the current Hadoop-based technology, and have motivated the need for supporting annotation management in such scalable platforms to boost the progress and discoveries in scientific in scientific applications.

## 1.3 Challenges

Since the development of MapReduce computing model and its open-source implementation Hadoop, and they became a data magnet in various application domains due to their superior properties such as scalability to peta bytes of data over thousands of machines, flexibility, elasticity in growing and shrinking resources, cost-effectiveness, and availability as an open-source. This contributed to economic boost by making scalable computing more accessible for the common application developer opening the doors to building applications quickly that otherwise may not have been built. A flurry of research activities have been recently proposed around Hadoop platform ranging from high-level query languages, workflow management, indexing techniques and query optimization, and physical data layout optimizations, to online data processing, and provenance management. We envision a great potential in Hadoop as scalable, fault tolerant and open-source platform for large-scale data analytics. However, to flourish this potential in the context of scientific data the two features of annotation management and proactive recommendation of executing paths need to be coherently integrated inside Hadoop. Certainly, this integration involves many challenges and warrants fundamental changes and extensions to Hadoop infrastructure. These challenges include:

(1) Lazy Interpretation of Data: Data in Hadoop are stored in flat files with

no known structure or schema until query time. Thus, adding annotations that reference specific data pieces, especially at different granularities from record, task, to job level, is not straightforward. Nevertheless, the support for annotation data while it is begin generated (on-the-fly annotations).

(2) Lack of Incremental Updates: Annotations can be incrementally added in small batches over existing data. However, Hadoop file system is not optimized for handling small files and incremental updates. Therefore, new storage mechanisms are needed to handle annotations.

(3) Computing Model with Black-Box Operators: Unlike RDBMSs with well-defined, modularized operators, Hadoop has two black-box operators, namely map and reduce. Therefore, it is more challenging to optimize the annotation propagation at query time since the operators semantics are not explicit known to the system.

(4) Scalability and Distributed Processing: As data scales up, annotations also scales up and they get generated/stored in a distributed fashion, which are issues that have been overlooked by existing annotation management techniques, as they mostly focus only on centralized RDBMSs.

(5) Black-Box Execution Flow: In order for the system to be proactive in recommending execution paths and possible ways for exploring the data, it needs to gather as much information such as the data structure, the used functions or workflows, execution statistics, etc. Clearly, this against Hadoops nature of the black-box execution of users map and reduce tasks.

There has been a large body of work in "CloudNotes", including for general workflows. Although map and reduce functions as data transformations have become increasingly popular, we are unaware of any work that focuses specifically on building annotation manage system for Cloud-based platform. Also, we explore the overhead of annotation storage and cost of annotation propagation. Our goal is to enable

efficient and transparent annotation management in distributed platforms while keeping the overhead low. Overall, our contributions are: after describing the basic idea and the outline of the framework in Chapter 2, in Chapter 3 and Chapter 4, we introduce more implementation details. And we have already built a system called CloudNotes that implement the functionalities in the design phase. The experiment section reports the performance results using CloudNotes on the time and space overhead. Finally, we give the conclusion and discuss future work, including more functions and optimizing options for CloudNotes.

# Chapter 2

# System Overview

In this project, we proposed to design and develop the "CloudNotes" system, an extensible annotation management system for cloud-based scientific applications. In Figure 2.1, we extend Hadoop execution model to detect and track annotations. Storage layer in Hadoop file system includes data repository and annotation repository. We add a new component: Annotation Manager to coordinate annotations in "CloudNotes".

"CloudNotes" is driven by three fundamental and challenging research problems are: distributed fashion, lazily-interpreted data, and batch processing. In the following, I will describe each challenge in some more details.

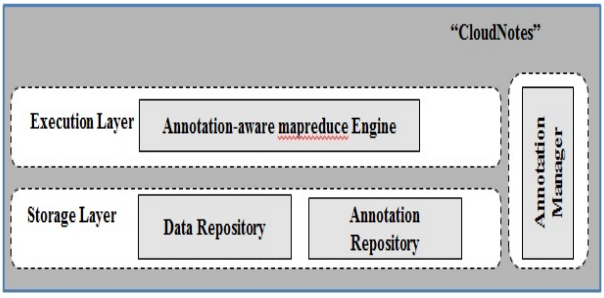


Figure 2.1: System Overview

## 2.1 Processing Annotations in a Distributed Fashion

Annotations in CloudNotes will be generated, stored, and processed in a distributed manner. These issues of scalability and distributed processing of annotations, although fundamental in cloud-based environment, are overlooked by existing techniques that focus mostly on centralized DBMSs.

Query Model: "CloudNotes" system is an extended query and data manipulation model that enables new functionalities such as annotating data and querying not only data but also the annotations associated with them by introducing high-level interfaces to seamlessly perform these functionalities.

Execution Infrastructure: The system has been built to capture and provide annotations base on the Map-Reduce working model. MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

Our system uses a wrapper-based approach, requiring little if any user intervention in most cases, and retaining Hadoop's parallel execution and fault tolerance.

## 2.2 Linking Annotations to Lazily-Interpreted Data

Data in Hadoop is stored in flat files with no known structure or schema until query time, and hence the notion of records is vague. Therefore, linking annotations to specific data segments is not straightforward, especially when considering annotations at various granularities, e.g., record-, file- levels. Therefore, an unique

identifier needed to identify the location of each record or each file in HDFS, which also used to connect with the row data and annotations. In our system, we define this structure to apply basic functionalities and assist the system to manipulate the annotations workflow.

## 2.3 Storing Annotations Into Batch-Optimized Storage

Annotations can be incrementally added in small batches. However, Hadoop's file system (HDFS) is not optimized for handling small files and incremental updates. Therefore, new storage mechanisms and novel indexing techniques are needed to efficiently store the annotations.

Besides considering the efficient storage structure of annotations, choosing proper annotation repository needs to be considered as well. Our system provide two-level storage structure: first level needs to support efficient incremental uploads of small annotation batches and also enable key-based retrieval at query time. Second-level storage can adaptively materialize not frequently used annotations in the local machines.

# Chapter 3

# Cloud-Enabled Annotation Management

In this chapter, we focus on adding the building blocks of annotation mechanism into CloudNotes. These building blocks will enable users' functions and processing tools to seamlessly and transparently annotate the data stored in Hadoop. As we illustrate in Figure 3.1, CloudNotes will create a wrapper around the Record Reader function in Hadoop that augments a new Object Identifier (OId) to each reported key-value pair. Assuming the data are read from HDFS files, the OIds will represent the beginning offsets of the data records inside the files. These OIds are of a newly introduced data type OID over which the abstract interfaces for manipulating the annotations will be created.

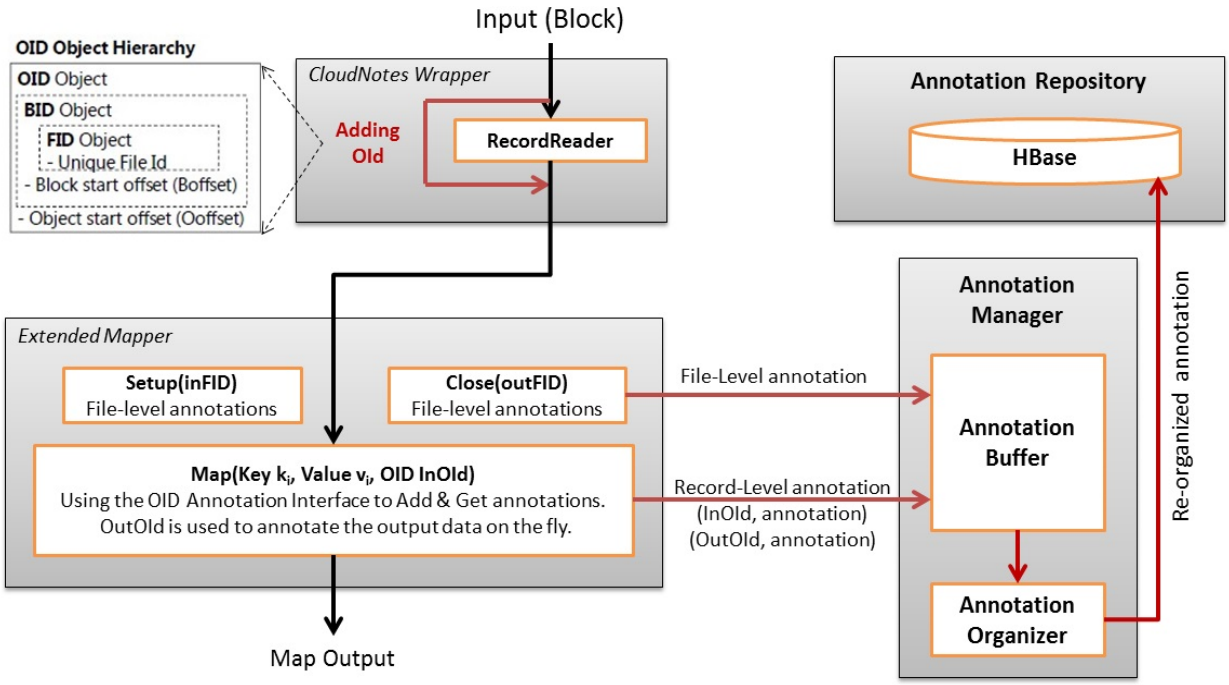


Figure 3.1: Flow of Adding Annotations in CloudNotes

## 3.1 Extending the MapReduce execution model

### 3.1.1 Object Identifier (OID)

The definition of OID (Object Identifier): We assume each input element has a Unique ID in Hadoop File System called OID. This OID identifies the location of each record in HDFS, and can be captured anytime during the MapReduce jobs. Assuming the data is read from HDFS files, the OIDs will represent the beginning offsets of the data record inside the files. The relationship between the original data and annotation attached on it is shown in Figure 3.2.

We will introduce new interfaces, AddAnnotation( ) and GetAnnotation( ), on top of OID for users code to manipulate the annotations corresponding to each record. However, the concept of OID is not physically exist in the Hadoop, so each object identifier is constructed just before run the map function on the certain key

value pair, and it will be deconstructed till the content of annotations are sent to the repository.

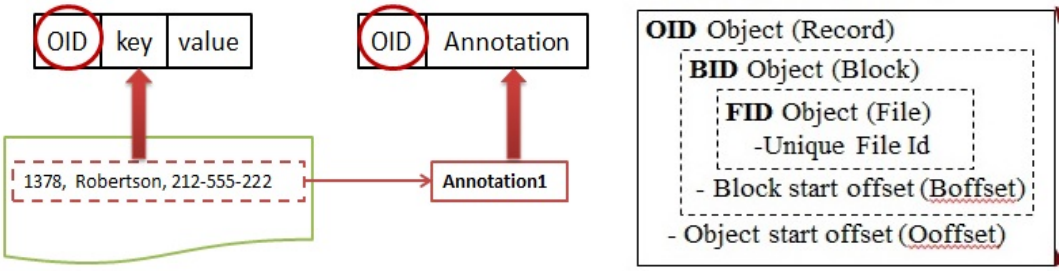


Figure 3.2: OID Concept and Hierarchy

### 3.1.2 Annotate Input Data: Map-Side Function

In Hadoop, all data elements are assumed to be key/value pairs. When running a MapReduce job consisting of a map function and a reduce function, the map output elements are grouped by their key before being processed to the reduce function. Otherwise, keys are simply part of the data.

Hadoop users supply the following ve components to dene a MapReduce job [1]:

*Record-reader*: Reads the input data and parses it into input key/value pairs for the mapper.

*Mapper*: Defines the map function.

*Combiner*: Defines partial aggregation by key (optional).

*Reducer*: Defines the reduce function.

*Record-writer*: Writes output key/value pairs from the reducer in a specied output format.

The mapper class of Hadoop will be extended to accept triplets of (Key, Value, OId) instead of the standard (Key, Value) pairs as shown in the Figure 3.1. Actually, Hadoop provides two Java MapReduce APIs. This new API, sometimes referred to

as Context Objects, was designed to make the API easier to evolve in the future. Our system extends the new interface only.

On the other side, when map task dealing with its data, numbers of annotations are generated, which are collected in the temporary annotation buffer in the map context. The map task needs to measure the size of annotations generated, and stream the batch of annotations to the repository, when it beyond the threshold.

Here is an example (See Figure 3.3) of AddAnnotation() function used to annotate input in a map only job:

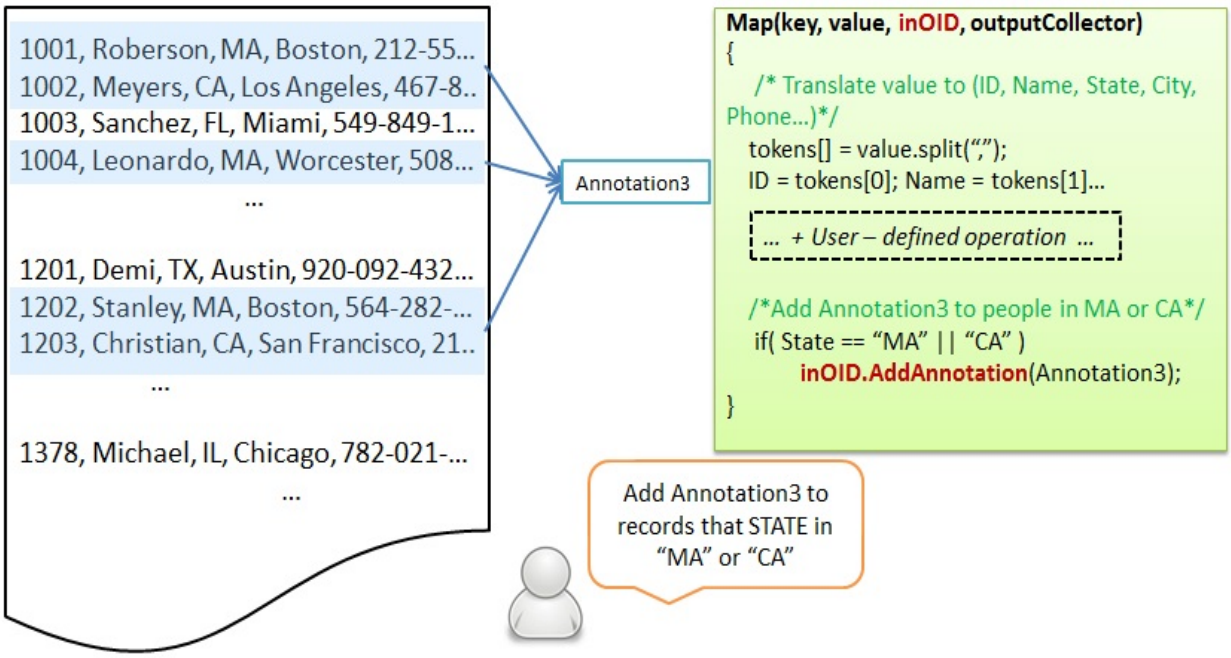


Figure 3.3: Annotate Input Data

In a single job, a user can define normal operations on data and annotation management as well. In this example, OIDs construct for input are part of arguments of map function. If user want to annotate records that State is "MA" or "CA", map function filters out particular records and the corresponded OID calls AddAnnotation( ) to annotate certain record.

### 3.1.3 Annotate Output Data: Map-Side Function

In addition to these extensions for annotating the input data, CloudNotes will enable annotating the output data while being generated, i.e., on-the-fly annotations instead of using another job to annotate output, which is expensive. To achieve this goal, we extend the reporting mechanism in Hadoop to keep track of and report back the position (offset) of each newly produced record within the output buffer (although it is not physically written to disk yet).

Actually, there is a byte counter in output context, to keep track of size of output data, on the other hand, the block size is defined when the HDFS starts, and the output directory is also given in the job configuration information. Therefore, combine these three components, we can construct an OId for each output data record, with its exact identical location in the HDFS, which is the OId return to Map function. Then Map function can use this OId to annotate their output records. Note that, the annotations in CloudNotes are always associated with the actual physical file (not the logical directories) and the OIds always refer to offsets relative to these files.

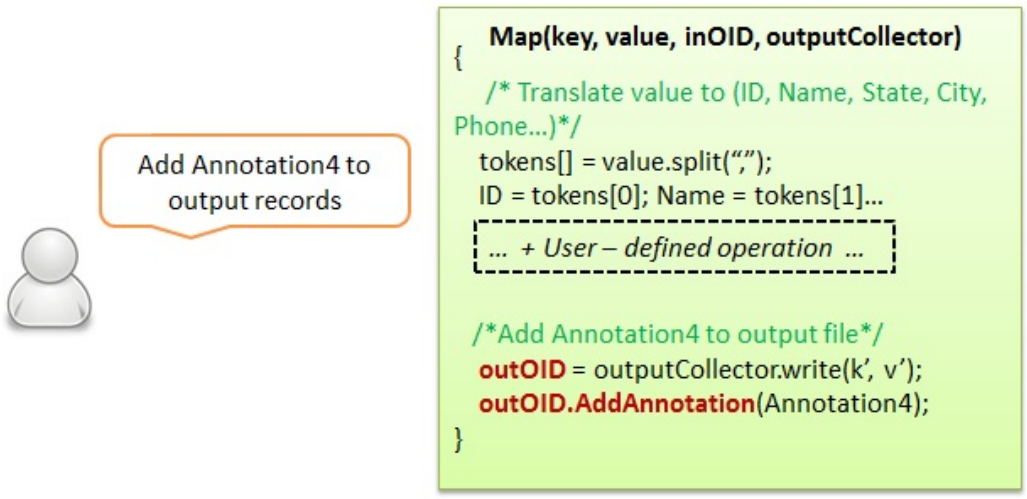


Figure 3.4: Annotate Output Data

Here is an example (See Figure 3.4) of AddAnnotation() function used to annotate output in a map only job:

In this job, user wants to annotate each record in output. When write( ) function is called to collect output record, corresponded output OID is constructed and returned to user. By using this new generated output OID, AddAnnotation( ) is provided to annotate particular output record as well.

## 3.2 Design Storage Scheme for Annotations

The output of a map task is of two types now, the regular key-value pair records (the original data records) that will be stored in HDFS, and the newly added annotations either on the input or output data. Storing the annotations directly into HDFS can be extremely inefficient because HDFS is optimized only for writing/reading big batches of data, files that are hundreds of megabytes, gigabytes, or terabytes in size. While annotations can be added in small batches from many jobs. Therefore, our design is to stream the annotations to a newly introduced Task-Level Annotation Manager that buffers the annotations till the completion of the map task, and then re-organizes the annotations for efficient storage in the Annotation Repository (See Figure 3.1).

### 3.2.1 Annotation Manager

The Annotation Manager is a distributed component that runs on each data node and communicates with the local Task Tracker for managing tasks. And this component is managed by the task tracker distributed on each data node. The Annotation Manager decides on the optimal timing for flushing its buffer to the Annotation Repository. And the statistics collected by Annotation Manager are stored in the

local file system.

However, The MapReduce model is to break jobs into tasks and run the tasks in parallel, that means, a classic failure mode to consider: failure of running task. There are various failure cases, like runtime exception, the task tracker marks the task attempt as failed or killed, then frees up a slot to run another task. However the annotation collected may have been sent to Annotation Manager, as the failed task will be executed again, then the annotations in Annotation Manager is duplicated. In order to avoid duplication, our idea is let Annotation Manager communicate with the running tasks, therefore, if the task failed, the Annotation Manager can notice status changed, and expire the annotation from certain task. Fortunately, the Annotation Manager is part of task tracker, which has built in task progress information collected mechanisms, which makes our idea easily implemented.

In addition, based on the annotations rate generated from the map tasks, the Annotation Manager may decide to combine the results from multiple mappers running on the same node before flushing them to the Annotation Repository. The fault tolerance mechanism of CloudNotes will be extended to take into account the possible failures of Annotation Manager, and to ensure that the generated annotations are permanently written to their repository before finalizing a job.

### 3.2.2 Annotation Pipeline in Hadoop

However, we need to consider that how does Annotation Manager, a sub-thread running as part of task tracker, to collect annotation from other map tasks, running as child processes. The communication approach among these processes is required to be efficient. Our implementation is extending the original protocol interface among all of them, which is used to report task status to task tracker basically. So we add our abstract sending annotation functions and other assisting functions

inside the protocol. Then, another problem raised, there is a limitation of this protocol, the size of data that can go through this protocol once is limited. In order to solve this bottleneck problem, we did test to estimate the threshold of annotation size that can be passed at one time, and we send annotations in batches. And to make the pipeline work smooth, we also need to serialize the OID object to String or bytes.

So, for this part of our annotations pipeline, the distributed component – Annotation Manager is working on each data node, and running independently to collect annotation from different tasks or jobs that on it own node. While Annotation Manager buffering annotation from running tasks, it also talks with these tasks to check the status, in order to give the proper status to certain collected block of annotations, and do the normalization and further steps.

## 3.3 Optimizing the Annotation Repository

Our idea as illustrated in Figure 3.1, is to store the annotations in Hbase system, a "Distributed Storage System for Structured Data"[12]. HBase system can efficiently supports incremental uploads of small annotation batches and also enables efficient key-based retrieval at query time. HBase is a distributed column-oriented database built on top of HDFS as well. And our annotations and OIds are structured data, but distributed in the cloud-based cluster which fit features in HBase. On the other hand, annotations are required as query-time retrieval, and high query speed. See, the HBase is the Hadoop application to use when you require real-time read/write random access to very large datasets[1], which satisfies all the requirements.

However, to efficiently store the annotations in Hbase, the Annotation Organizer which is part of the Annotation Manager needs to perform several crucial

optimizations including:

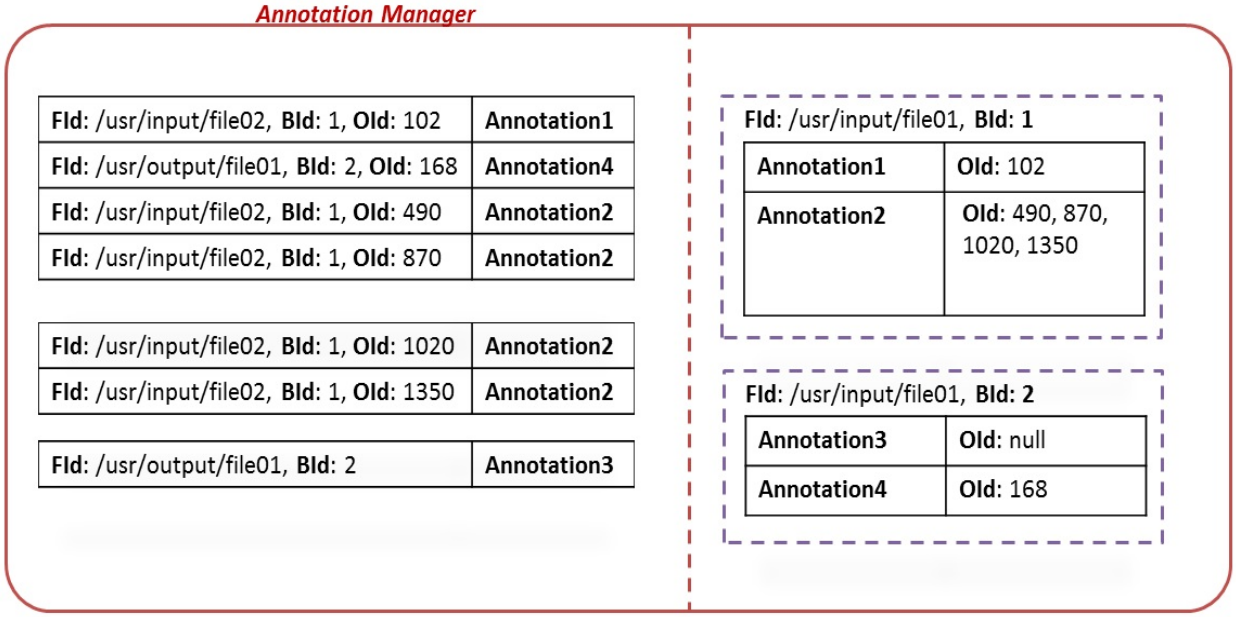


Figure 3.5: Promoting and Normalizing Annotation

### 3.3.1 Promoting Annotations to Block Level

CloudNotes typically reads the data in HDFS blocks, and hence it is more efficient to retrieve all annotations related to a given block at once. Therefore, CloudNotes promotes the annotations to the block-level and indexes them inside Hbase based on the Block Ids instead of the OIds (notice that Bid is part of OId).

Since the annotations sent from tasks are in several waves, different tasks can add annotation at same time, so we classify the annotation batches by their block id (See Figure 3.5). The advantage is, it's easier to mark the annotation status, especially, when a single task fails for some reason, we can roll back the annotation efficiently and clean the annotations from the certain task.

### 3.3.2 Normalization of the Annotation Schema

The Annotation Organizer needs to efficiently normalize the storage of annotations to avoid any unnecessary replication or storage overheads. In the distributed environment like Hadoop, the normalization process is not straightforward. Our idea is to investigate the map-, and node- normalizations, which normalize the annotations within the output of a single map task, the output of map tasks on a single data node, or the output from the entire job across all nodes, respectively.

That means, in our Annotation Manager, two level normalizations need to be implemented. The first level is the in-block level normalization: during the annotation de-serialized in the buffer, we hash the annotation text, and link the OIds which share the same piece of annotation together, so in general, a batch of original annotations will finally be normalized as a hash table for each block in memory (See Figure 3.5). The second level normalization, called cross blocks normalization, based on the first level normalization, with the continuous annotation coming into the buffer, these normalized hash tables generated, and they are merged one more time.

### 3.3.3 Compressing the OIds Using Bitmap

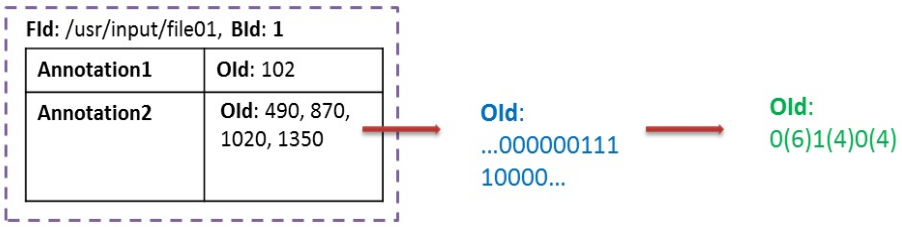


Figure 3.6: OIDs Compression

When promoting a record-level annotation to the block level, CloudNotes needs to keep track of the actual OIds of the records it is attached to. The Annotation

Organizer will make use of the fact that the list of OIds is fixed within given block. Hence, the list of OIds will be maintained once per block, and then each annotation will keep an array of bitmaps specifying the corresponding OIds it is attached to. The bitmap can be further compressed using Run-Length Encoding (RLE) [68] for compact representation.

This step is done while we de-serialize the annotations, to make the annotations stored efficiently in the Annotation Manager buffer and this compression phase also reduce the I/O consumption when Annotation Manager sends them into the Annotation repository (HBase).

## 3.4 Supporting Higher-Granularity and Reduce-Side Annotations

### 3.4.1 Define Fine-Granularity Annotation

We discussed how to add record level annotations inside map tasks. The next step is to support higher-granularity (record- and file-level) annotations. To enable this capability, we create the same set of annotation interfaces, e.g., AddAnnotation( ) and GetAnnotation( ), over the File (FID) data types (Refer to the Object Hierarchy in Figure 3.1).

To annotate a file, a corresponding FID object needs to be constructed based on the files unique name in HDFS, and then used to annotate the file. We extend Hadoops Setup( ) and Close( ) function that are called once per map task to receive the Bid of input of the input data block. Hence, users can annotate the input data file in Setup function by inFID, annotate output data file in Close function by outFID (See Figure 3.1).

To note that, to annotate the file level annotation is an optimization choice for user. Instead of calling AddAnnotation() function in the map function to execute adding operation for each single record, user can call file level adding annotation to the whole file directly, the system will treat it specifically to avoid executing same operation multiple times, which reduce the CPU consumption and I/Os.

The Annotation Organizer will ultimately re-organize the block level annotations and expand them to all records in the block (the storage is still efficient using the RLE-compressed bitmaps).

### 3.4.2 Reduce-Side Annotation

Reduce-side annotations are more straightforward that the map-side annotations because reducers can only annotate their output data (the input data are intermediate and get purged after the job completion). Reducers use the same proposed mechanisms to annotate their outputs. That means, user can call write( ) function in the reduce task, which returns the OID for output data. Same function is called the map task to annotate it output data. Add we extend Close( ) function in reduce side allows user annotate reduce side output file as well.

# Chapter 4

# Annotation Propagation and Query-Time Optimization

In this chapter, we focus on the efficient propagation of annotations at query time. How to retrieve, index, and possibly cache, the annotations from their repository? How to adaptively learn whether or not users jobs are interested in accessing the annotations?

## 4.1 Retrieval and Main-Memory Indexing of Annotations

### 4.1.1 Annotation Propagation from HBase

The main idea is to retrieve all the annotations attached to the input block at once, which show as Figure 4.1. More specifically, when a map task is scheduled to run over a given data block, the Annotation Manager will retrieve all annotations related to that block (based on its BId) from Hbase. Then, the Annotation Manager

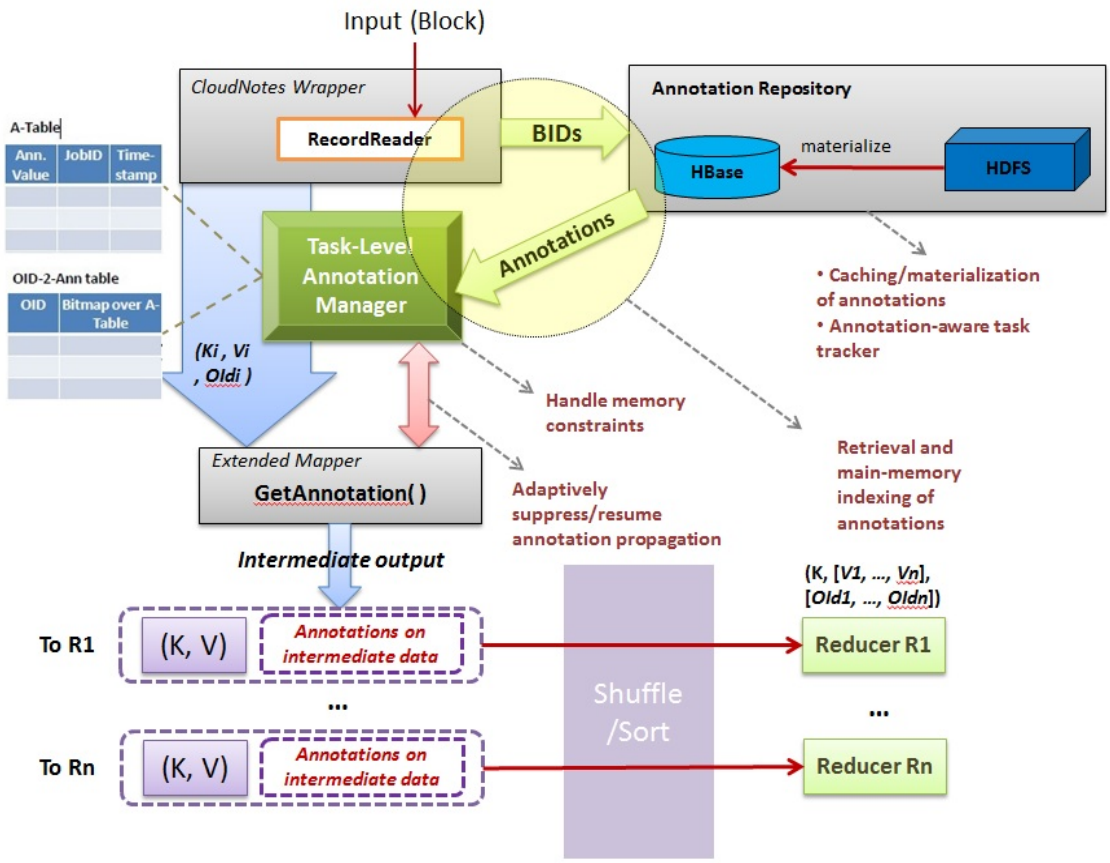


Figure 4.1: Annotation Propagation Pipeline

will build a two-level main-memory index structure (as depicted in Figure 4.2) that consists of an Annotation-Table (A-Table) storing the distinct annotations on the given block, and an inverted index (OID-2-Ann) mapping each record id (OId) in this block to the related entries in the A-Table.

This process will be performed before the start of each map task to enable efficient execution of GetAnnotation( ) function inside the map tasks. Thant's why we implement the fetching phase before running the map function. The steps to fetch annotation from HBase are simple as shown in the figure 4.1, which are: connecting the HBase, scan the data by the row key BlockID, then, the annotations returned from query are kept in the Task Level Annotation Manager buffer. Next, build

2-level-index showed in Figure 4.2, finally, when we construct input OId in map function, we extract the certain annotations from index.

### 4.1.2 Proactive Mechanisms

We also proposed proactive mechanisms, which can significantly enhance the performance, for predicating the future data blocks to be accessed on each data node, and pre-fetching /indexing the annotations for those blocks.

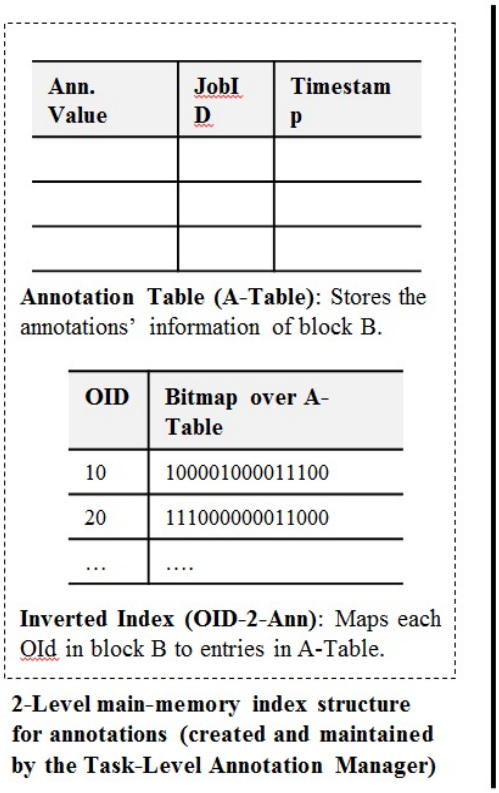


Figure 4.2: Annotation Index

The challenge here is to estimate which block is assigned to the local machine. Actually, the job tracker will split an input file for a new job into several map or reduce tasks, for each map task, it will deal with its own block of data, and be assigned to the proper task tracker in data node to execute. And the task tracker

will also communicate with job tracker by heartbeat, which contains its availability information [1]. By Analyzing these statistics, job tracker schedule the tasks to certain data nodes, and in the task tracker, there is a queue to store all the tasks assigned by the job tracker and are ready to be launched. Our proposed idea is to check this queue, and get the target block information in the job configuration in order to pre-fetch the annotations from HBase. The pre-fetched annotations are buffered in the Annotation Manager, then when the certain task starts, instead of sending query to HBase and wait for annotations, Annotation Manager can provide the annotations to the task. The function then can construct the two-level-index directly and execute the map function.

## 4.2 Adaptively Suppressing/Resuming Annotation Propagation

Users jobs in Hadoop-like environment can be black-box functions. Although this execution model is good for broader applicability, it poses more challenges in optimizing the systems performance, e.g., the system does not know in advance whether or not the users job is accessing the annotations. Therefore, CloudNotes deploys techniques to adaptively suppress/resume the propagation process.

The proposed idea is that if the first map task on each data node did not invoke GetAnnotation(), then the Annotation Manager will suppress the annotation propagation in the subsequent map tasks, i.e., it will not pay the cost of retrieving them from Hbase. The propagation will be resumed with the first explicit invocation of the GetAnnotation() function.

In order to implement this idea, we setup a flag for each job in the Annotation Manager running on particular nodes, to mark whether we need to fetch annotation

in this job or not. The default value is false although, the system fetch the annotation for the first map task of a job running on this data node. On the other hand, this flag can be invoked by the GetAnnoattion( ) function. If the GetAnnotation function is invoked, the flag will be set to true, that means system will keep fetching annotations at beginning of subsequent map tasks. However, if the GetAnnotation( ) function is not called in the whole task, the flag stays false, then the subsequent tasks check this flag, these tasks will not fetch annotations anymore.

Another case we also need to concern is that, the GetAnnotation function can be hidden in the if/else statement block, that means, although the first map task does not call the GetAnnotation() function and not update the flag to true, the subsequent tasks may still need the annotations. If the case happens, we force to fetch the annotation, then the tasks need annotations need to wait for the HBase's response. And the flag is modified to true means the job need annotations fetched. Our mechanism is, once, the GetAnnotation function is invoked, the flags are then never changed again.

These decisions will be made in a totally distributed fashion by each Annotation Manager on each data node. We plan to investigate different strategies for effective adaptability. For example, the strategy described resembles an eager adaptability strategy, but lazy adaptability is also applicable where the retrieval from Hbase is suppressed by default until the first explicit GetAnnotation() invocation is detected on each node.

## 4.3 Caching/Materialization of Annotations and Annotation-Aware Task Schedule

### 4.3.1 Caching and Materialization Mechanisms

As annotations move from Hbase to the compute nodes, the Annotation Manager may decide to materialize and store the annotations of a given block into HDFS and co-locate them with their data block (See the Annotation Repository in Figure 3.1). Colocation mechanisms can have significant impact on performance. However, several challenges need to be addressed including as follows:

(a) Avoiding creating small HDFS files, so annotations need to be grouped in large batches when materialized in HDFS.

(b) Selectively choosing which data blocks to materialize their annotations in HDFS  it does not have to be performed for all blocks.

(c) Tracking whether the materialized annotation files are up-to-date or new annotations over the data block have been added to Hbase, and how to combine the two sources when needed.

(d) Tracking where the annotation files are locates and whether or not they are replicated with all replicas of corresponding data block.

**Job Tracker**: The JobTracker is the service within Hadoop that farms out MapReduce tasks to specific nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack. Client applications submit jobs to the Job tracker. The JobTracker talks to the NameNode to determine the location of the data and locates TaskTracker nodes with available slots at or near the data. The JobTracker submits the work to the chosen TaskTracker nodes. The TaskTracker nodes are monitored. If they do not submit heartbeat signals often enough, they

are deemed to have failed and the work is scheduled on a different TaskTracker.

**Task Tracker**: A TaskTracker is a node in the cluster that accepts tasks - Map, Reduce and Shuffle operations - from a JobTracker. Every TaskTracker is configured with a set of slots, these indicate the number of tasks that it can accept. When the JobTracker tries to find somewhere to schedule a task within the MapReduce operations, it first looks for an empty slot on the same server that hosts the DataNode containing the data, and if not, it looks for an empty slot on a machine in the same rack.

The mechanism of caching and materialization: the Annotation Manager keeps the statistics that, each block of data (use BlockID as key) with annotation attached has its statistics to keep track of how many times the annotation is used in the user jobs. For considering the accuracy, the size of annotations and recent used timestamps etc. are part of statistics as well. Then the Annotation Manager can make decision that the annotations are frequently visited or annotations are in large size can be materialized in the local file system, and the directory (store the materialized annotations) information will also updates in the statistics correspondingly.

To note that, these statistics have replication in the local file system. However, we also need to consider that the data node may fail. If that happens, we need to protect the statistics that kept in the local file system. The job tracker has all the statistics from each data node collected, so when there is a data node fails or loses connection, we can copy the certain statistics from job tracker to the new setup node. Although they are not perfectly updated, but it can avoid loss of the whole statistics.

Therefore, instead of ask HBase for annotation immediately for each map task, Annotation Manager checks the statistics. If the annotation is materialized locally, the map task will cache the annotation from the directory stored in the Annotation

Manager and index the annotations for further use.

### 4.3.2 Annotation-Aware Scheduler

On the other hand this opens another issue of the annotation-aware scheduler where for users jobs accessing the annotations, the data blocks will no longer have the same cost/benefit  some blocks will have their annotations co-located with them while others do not. This warrants an interaction between the Job Tracker (responsible for scheduling tasks) and the Annotation Manager (responsible for tracking the location of annotations) to generate the best possible annotation-aware scheduling plan for tasks.

The original Hadoop scheduler is "Fair Scheduler". Fair scheduling is a method of assigning resources to jobs such that all jobs get, on average, an equal share of resources over time. When there is a single job running, that job uses the entire cluster. When other jobs are submitted, tasks slots that free up are assigned to the new jobs, so that each job gets roughly the same amount of CPU time. Unlike the default Hadoop scheduler, which forms a queue of jobs, this lets short jobs finish in reasonable time while not starving long jobs. It is also an easy way to share a cluster between multiple of users. Fair sharing can also work with job priorities - the priorities are used as weights to determine the fraction of total compute time that each job gets.

Our implementation is extending the heartbeat and the jobtracker and fair scheduler:

**heartbeat**: each data node sends periodic heartbeat messages to its name node. Besides the basic health status information included in the heartbeat, we add statistics updates and send to jobtracker to report the materialization status of each data node.

**jobtracker**: There is a map contains all the data nodes' availabilities, collected by the heartbeats from all the data nodes. Since task tracker name is the key of the map, we extend the value part, to contain the materialization statistics get from heartbeats for each certain data node. This information can be used for fair scheduling algorithm to decide tasks distribution.

**fair scheduler**: After fair scheduler calculate appropriate candidate nodes the do the task, if the number of candidates is not only one, usually this happens among the candidate nodes that all of them have input data replications and available at the same time, in such case, the scheduler will randomly choose one node to do the task. Our idea is to add another step here, to check the data node materialization statistics, to choose the node which has the annotation materialized to do the task. If none of them has the local annotation stored, we still randomly choose one node among the candidates.

## 4.4 Reduce-Side Annotation Propagation

Reduce-side annotation propagation will require further architectural extensions to the MapReduce engine as illustrated in Figure 4.1. The main idea of the proposed mechanism is to allow the map functions to annotate their intermediate results that will flow to the reduce functions.

### 4.4.1 Map-Side Output Collector Extension

Mappers can annotate its output data on the fly as described in Chapter 3.1. The only difference is that the Annotation Manager will keep the annotations on the local disk instead of shipping them to the Annotation Repository. In general, map task can freely annotate their output records, e.g., copy the annotations from the

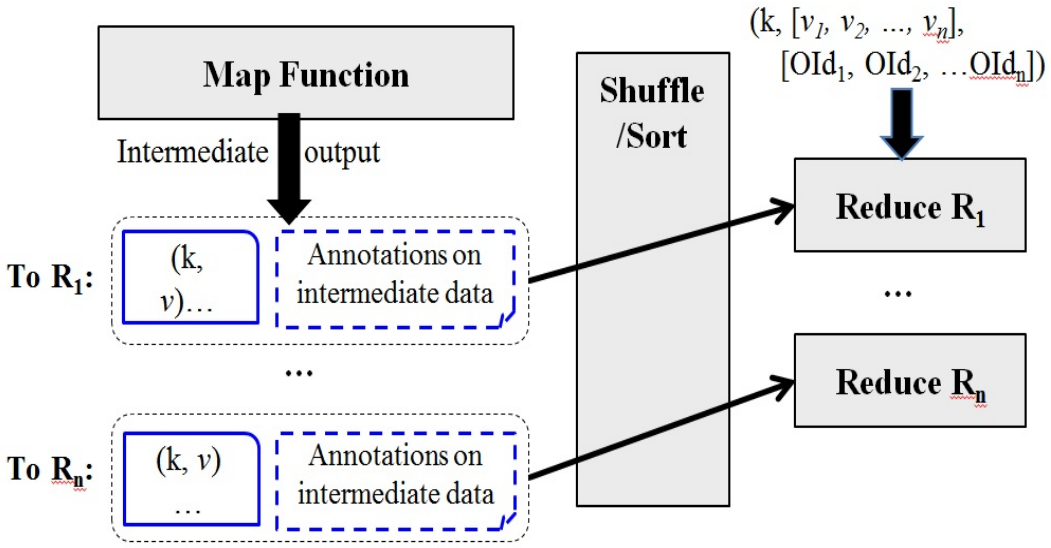


Figure 4.3: Reduce Side Annotation

inputs to outputs, or add new annotations.

The shuffling/sorting phase in Hadoop will be also extended to shuffle (but not sort) the annotations and ship them along with their data records to the corresponding reducers (See Figure 4.3). A challenging task is to carefully maintain the correct links between the annotations and their data records even after the sorting phase.

To make annotation bind with corresponding intermediate data record, while map function write output record with outOID, the system will serialize both value and outOID to map output buffer instead of sending outOIDs to Annotation Manager. And system uses global delimiter to distinguish original output value with outOIDs. After shuffle and sort phase, when values are de-serialized, the system needs to separate value and OID into two iterators, one is values, another is OIDs, which are part of arguments in new reduce function.

### 4.4.2 New Reduce-Side API

On the reduce side, two main modifications will take place:

First, the reduce functions will be extended to accept triplets of *(k, [v1, , vn], [OId1, , OIdn])* instead of the standard *(k, [v1, , vn])* pairs as depicted in the Figure 4.3 where each OIDi references the record corresponding to value vi and carries its propagated annotations. The user-defined reduce function can then manipulate the annotations as desired for each key group, e.g., summarize, aggregate, or consolidate the annotations.

Second, the Annotation Manager will index in main memory whenever possible the annotations received by the reducer based on their *<k, OId>* pairs before the reducers start consuming their input records.

# Chapter 5

# Experiments and Evaluation

## 5.1 Experiment Setup

Our experiments are focused on attaching annotation and annotation propagation. Here are the details of our experiments:

**Cluster Environment**

The cluster we used for our experiments consisted of 4 WPI compute cloud instances, each with 8GB instance memory, 3 Dual-Core AMD @2.2 GH, and 45 GB instance storage in HDFS. We launch all instances with 64-bit Linux and Java 1.6.0-30 and modified version of Hadoop 1.0.4. And the HBase we use is hbase-0.90.5.

**Hadoop Configuration**

One instance served as the master node and acted as both *name node* and *job tracker*; the other 3 instances served as *data nodes*. Each slave node is allowed to run two map tasks and two reduce tasks concurrently, and the number of reduce tasks was set to 100. The algorithm chosen for job tracker to scheduling tasks is *Fair Scheduler*. We configure Hadoop following the guidelines for real-world cluster

configurations. The changes from the default configuration included increasing the numbers of streams(100) merged at once while sorting files and higher memory-limit while sorting data (set *io.sort.mb* to 200). Finally, the replication factor for our output file was set to 1.

**Dataset**

We use 10 GB of input text generated randomly from 2000 thousands distinct words, and each line of record starts with ID randomly chosen from 1 to 100. And all data has same formats, we use $10^8$ random records as input (11 GB respectively).

Annotations are defined short context, so we randomly chosen a word from 5 distinct English words as annotation.

**Preliminary Jobs**

We present performance experiments conducted on two workflows, a Map-only job and a Map-Reduce job. The map only job in our experiment is to filter out input records. Another job performs aggregation function: group-by operation, a Map Reduce job.

## 5.2 Performance Results

Our performance results are summarized as follows:

### 5.2.1 Phase 1: Enable Add Annotation

We prepare our experiment for enable adding annotation phase. In the map only test job, it randomly chooses a number of input records and output records have annotation added, and this number is changing from 0 percent of original record number to 20 percent, as X-coordinate axis shows. With the percentage grows, the Y-coordinate axis represents the CPU time consuming for each run of job. Same

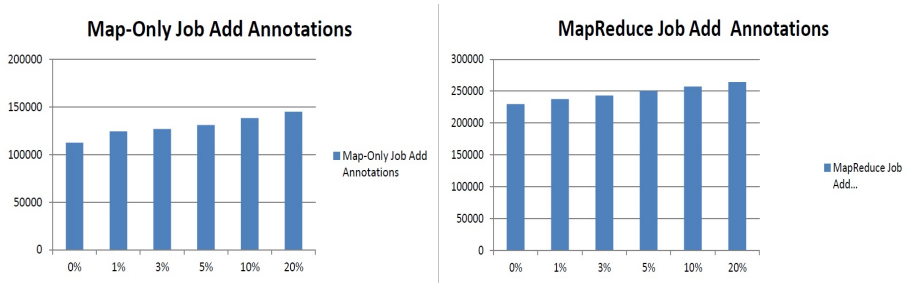


Figure 5.1: Adding Annotations

way to the map reduce job, the only difference is that instead of add annotations for the map output, we annotate the reduce side output data, however, the numbers of annotations does not change.

In the left Figure 5.1, for a map-only job, when we add 1% annotations, the time consumed increased 10%; when we add 10%, it increases linearly as 20%; and then 20% annotations added, CPU time grows to 28%.

In the right Figure 5.1, for a map-reduce job, which is more complex than the map only job. When we add 1% annotations, 4% extra CPU time consumed; when it's for 10% annotation target, 12% extra time consumed, and for 20%, CPU time increased 15%.

### 5.2.2 Phase 2: Annotation Propagation

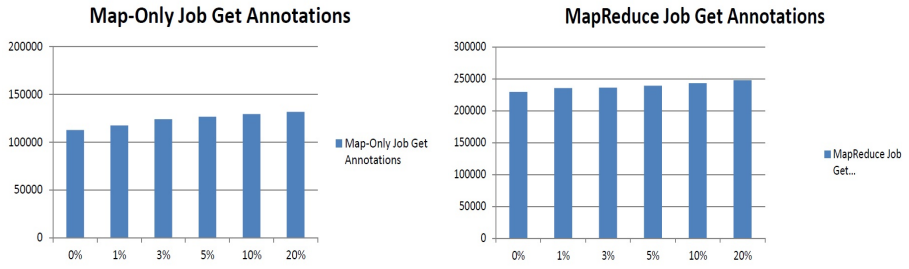


Figure 5.2: Adding Annotation to Data File

In the phase of Annotation propagation, two testing jobs we used are very similar,

since no matter it's a map only job or a map reduce job, for the normal propagation, we can only propagate annotations from HBase in map function. Our target is, with numbers of annotations stored in the repository increased, as numbers changed from 0% to 20% at X-coordinate axis, basically that's the annotation added in the phase 1, we keep records of CPU time consumed for each time we run the jobs to propagation different sizes of annotations as showed in Y-coordinate axis.

In the left Figure 5.2, for a map-only job, when we propagate 1% annotations, the time consumed increased 5%; when we get 10%, it increases linearly as 10%; and then 20% annotations got, CPU time grows to 17%.

In the right Figure 5.2, for a map-reduce job, which is more complex than the map only job. When we get 1% annotations, 3% extra CPU time consumed; when it's for 10% annotation target, 5% extra time consumed, and for 20%, CPU time increased 8%.

### 5.2.3 Phase 3: Reduce Side Annotation Transmission

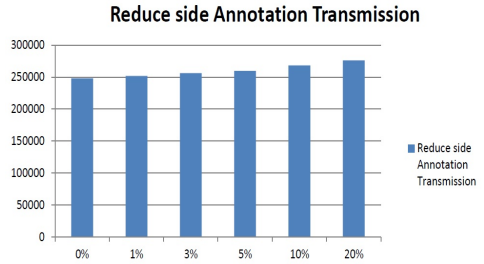


Figure 5.3: Reduce Side Annotation Transmission

In this phase, we only test the Map Reduce job, to pass annotations from map side as output to reduce side as a part of input, then annotations are propagated in the map side, and go through the intermediate part to reduce side, used in the reduce function. Our test case is that, each job need to propagate 20% annotations from

HBase in the map function, which is same as the phase 2, and call *sendtoReduce()* function to transmit OIDs to reduce function, and we call *getAnnotation()* then. However we need to test the overhead of intermediate annotation transmission, the range of annotation transmission size is ranging from 0% to 20%, which is showed as X-coordinate axis in the Figure 5.3. And CPU time consumed showed in Y-coordinate axis.

In the Figure 5.3, for this job, when we transmit 1% annotations, the time consumed increased 2%; when we pass 10%, it increases linearly as 8%; and then 20% annotations passed, CPU time grows to 11%.

## 5.3 Performance Evaluation

### Enable Add Annotation

We report the time and space overhead associated with adding annotations in our experiments. For each annotation percentage, we ran both jobs three times.

For adding annotations, the reasons that have extra time consuming mainly is normalization, especially for the second level normalization. As mentioned in chapter 3.3, we need to invert hash the annotation text to make it fit for storage.

We observe in Figure5.1, compare with map only job, map reduce job itself takes much more time. However, the time consumed for adding annotations operation is similar with it consumed in map only jobs. On the other hand, the ratio of extra time for add annotation operation takes, is decreasing, that means, when adding larger size of annotations, the overhead in total will not impact much on the execution time for the whole job. And the data size or the job complexity has little impact on the execution time, because the entire map output data set fit in the sort buffer.

*Observation 1*: when size of annotations added gets larger, it doesnt affect much

on the execution time for the whole job.

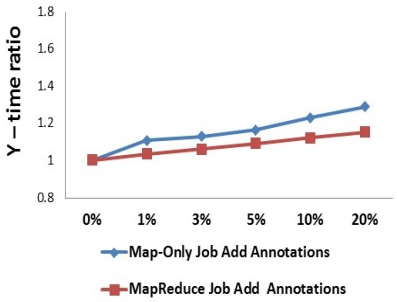


Figure 5.4: Comparison of Job complexity Effects on Adding Annotation

Meanwhile, base on same experiments, in Figure 5.4, Y axis means time ratio for each run of certain job. Red line is trend of map-reduce job, which is even lower than map-only job. That means time consumed on adding annotation compared with whole job is not significant.

*Observation 2*: the job complexity has little impact on efficiency of adding annotation.

### Annotation Propagation

We report the time and space overhead associated with annotations propagation in our experiments. For each annotation percentage, we ran both jobs three times.

When map task propagate annotation, we need to ask result from HBase and construct two-level-index and send annotations in map function. However building this index is expensive, that's why we have time overhead on annotation propagation.

We analyze the Figure 5.2, with the numbers of annotations we get from repository, (in our environment, HBase is installed in each instance), extra time increased, but it's not linearly increasing. That means, when Propagate larger size of annotations, the overhead in total will not impact much on the execution time for the whole job. The only thing need to mention is: the compare 1% propagation with

no annotation propagation, time increased significantly, because our resuming and suppressing checking here, when the first map task check no annotation needed, the sequential tasks won't ask annotation from HBase anymore. That's why propagation operation will not impact much on jobs without annotations.

*Observation 3*: when size of annotations propagated gets larger, it doesnt affect much on the execution time for the whole job.

### Reduce Side Annotation Transmission

We report the time and space overhead associated with annotations transmission in our experiments. For each annotation percentage, we ran both jobs three times.

The extra time consumed here basically is made of serialization/de-serialization and read/write disk. In particular, our annotation will be serialized as part of map output data written to local file system, and passed to the reduce task after de-serialized as OID as part of input argument in reduce function. From the Figure 5.3, the overhead of intermediate annotation transmission is showed, which is consistently small.

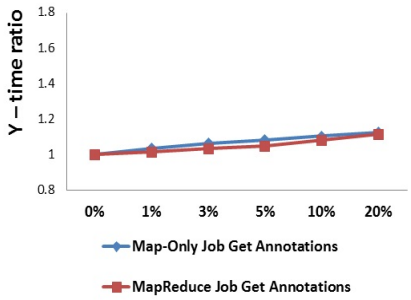


Figure 5.5: Comparison of Job complexity Effects on Annotation Propagation

Base on second and third experiments, in Figure 5.5, Y axis means time ratio for each run of certain job. Red line is trend of map-reduce job with annotation transimitted in between, and blue line is map-only job with annotation propagated

for map input data. These two lines are very close. That means time consumed on annotation propagation in both sides compared with whole job is tiny.

*Observation 4*: the job complexity has little impact on efficiency of annotation propagation.

# Chapter 6

# Related Work

In this project, we address the annotation management on the Hadoop platform, which is the open source implementation of Google MapReduce, is widely used cloud-based platform due to its superior properties. Few recent systems have exploited specific types of annotations in the Hadoop.

**Cloud and Hadoop-based analytics**

Cloud computing, e.g., Amazon S3 [22, 23], Elastic MR [24], and Google Cloud [25], is an emerging and widely spreading computing paradigm because of its unique desirable features. Hadoop [1] is a popular cloud-based computing platform for scalable data analytics. It is used by industrial sectors, e.g., Facebook, Yahoo, Amazon, and IBM use Hadoop for their own data as well as their products [26]. As open-source software, Hadoop is also a very attractive platform for research activities that have been recently proposed ranging from high-level query languages [9, 28, 29, 30, 31, 32, 40], workflow management [33, 27, 8, 35], and indexing techniques and query optimizations [36, 37], to physical data-layout optimizations [37, 38, 36], statistical and mining techniques [39, 40, 41, 42], and provenance management [7,

8, 9, 14, 15, 16]. However, annotation management in the context of Hadoop has not been addressed before. The closest related work are the Ramp [8] and Stubby [11, 34] systems. However, as we discussed in the chapter 1, these systems are not general-purpose annotation management systems, and they cannot support the wide range of functionalities offered by CloudNotes.

**Scientific databases**

Scientific databases and algorithms [3, 4, 5, 6, 43, 45, 47, 48, 49, 50] have been proposed to extend traditional database management system [3, 4, 5] with new features and functionalities. These extensions address the entire data management stack ranging from new data models [47, 48], physical data layouts, and access methods [45, 46], to provenance and annotation management mechanisms [44], sharing techniques [44], and workflow models and query languages [49, 50]. Most of these techniques and extensions have focused on relational database systems. Some systems have addressed other data models, e.g., SciDB for the array data model [51], and Pregel and GraphLab for the graph data model [52, 53]. However, these recent systems do not support annotation management, and part of our long-term research agenda (in chapter 1) is to address the annotation management over these systems and their complex data models.

**Annotation management**

Annotation management is widely applicable to a broad range of applications, yet it gained its significance from scientific applications [3, 4, 5]. Several generic frameworks for annotation management have been proposed, e.g., DAVID [54], Artemis and ACT [55], Pfam protein families database [56]. However, all of these techniques have focused on centralized relational databases, and hence none of them is appli-

cable to scalable and cloud-based platforms such as Hadoop. Moreover, none of these systems have addressed the extensibility feature to instantiate different types of annotations within a single system, nor they apply mining techniques to extract hidden knowledge from the annotations.

**Provenance techniques**

Data provenance has two main approaches: inversion-based, e.g., [57], in which the system maintains the inverse of the processing operations to re-generate the origin of the data, and annotation-based, e.g., [7, 8, 9, 16], in which the lineage information are stored as annotations attached to the generated data. Provenance is extensive studied in the context of relational databases [3, 4, 5], scientific workflows [58], and update/exchange and heterogeneous systems [59]. Recently, provenance tracking in Hadoop has been addressed in Ramp [8], PigLipstick [17], and Hadoop/Kepler [23, 22]. The most similar system is Ramp. In the Ramp, data provenance can be captured for map and reduce functions transparently. And it supports backward tracing and forward tracing, by using a wrapper-based approach. However, as we presented in the section1, CloudNotes has broader applicability and solves problems beyond what provenance techniques can handle.

**Extensible systems**

Extensibility has shown to be very effective in systems' design and has been addressed in different contexts. It has been addressed in prototype databases, e.g., ProsgreSQL [60] and GENESIS [61], and in commercial products, e.g., Oracle [62], IBM Starburst [63], and Sybase [65]. Extensibility ranges from extending new data types [64], new indexes and access methods [66], to new query optimization rules [67]. The Proposed project is novel in addressing the extensibility in the context of

annotation management.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

This project designs annotation management framework on the top of Hadoop, with two directional pipelines: Annotating original data and Annotation propagation. And "CloudNotes" also provides different properties based on annotation in MapReduce workflow, such as design OID to trace and track the data workflow via data unique location as its identity in the Hadoop file system. We have built a prototype system as an extension to Hadoop that supports various functions for annotations. On the other hand we defines several optimization choices for users. Our system supports fine-grained annotations, basically, record-level annotation and file-level annotation. And annotate output data on the fly is implemented in our system. In the end, the performance numbers and evaluations based on experiments are reported as well.

## 7.2 Future Work

Our future work mainly contain three parts, optimization for current "CloudNotes" system, annotation type extension, and data mining techniques on annotations in distributed system.

**System implementation optimizations:**

As described in the chapter4.2, we provide a basic method to make decision on suppressing and resuming annotations for map reduce tasks. We also plan to investigate different strategies for effective adaptability. For example, the strategy described in the section 4.2 resembles an eager adaptive strategy, but lazy adaptability is also applicable where the retrieval from HBase is suppressed by default until the first explicit GetAnnotation() function invocation is detected on each node.

**Extensibility for typed and action-enabled annotations**

We plan to focus on the extensibility feature of CloudNotes for rapid instantiation of a wide variety of annotation types having different semantics and different propagation strategies. The main idea is that on the top of the CloudNotes's core, we provide pluggable modules where users plugin their methods to instantiate different types of annotations. Few examples of these types include:

(a) Provenance for keeping track of the origin of the data as they evolve.

(b) Fine-Grained Authorization for providing record level access control based on the records' content.

(c) Versioning for associating corrections and chaining newer versions to existing records.

The challenges include:

(1) Extending CloudNotes execution model to support extensibility: We plan to extend the MapReduce framework with pluggable modules that execute automatically at specific stages of the execution plan. These modules will define how each annotation type behaves at query time. More specifically, we plan to add pre- and post- modules for the main stages of MapReduce, i.e., map, reduce and combine stages.

(2) Language extensions and instantiation of typed annotations: We will extend the CloudNotes's language to allow defining and instantiating new types of annotations. Each type will have: unique name, set of user-defined functions for the pluggable modules, set of properties that will enable customized and optimized execution as will be described in sequel.

(3) Optimized execution for typed annotations: CloudNotes will enable various optimizations based on the annotations' properties including: Bypassing module execution: that Annotation Manager can keep track of the records that have typed annotations; Suppressing annotations: it is possible that an annotation type does not access (retrieve) the existing annotations; Checking prerequisites and halting propagation: An annotation type may mandate certain properties of users' job to hold in order to guarantee correct propagation semantics.

(4) Data versioning as annotation: Data versioning has been widely adopted in database systems. Versioning is desirable feature especially in scientific applications. The key challenges addressed by these systems are how to chain the versioned records, and how to access them efficiently. We will study two approaches for supporting data versioning as annotations like Versioning as regular annotations or versioning as typed annotations.

**Data mining techniques on annotations:**

With the annotation size increasing, instead of retrieving all the annotations attached to the target data, summary is required. Or annotation may need to support key word search and semantic search to reducing annotation propagation time overhead and return concise results to user. The challenge for this work is very obvious: the annotations are stored in cloud-based platform, and treated in distributed behaviors. Not like centralized annotation repository, each data node only has its own annotation and meta-data. So the normal data mining strategies are hard to apply in the CloudNotes.

# Chapter 8

# Reference

[1] The Apache Software Foundation. Hadoop. http://hadoop.apache.org.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, pages 137150, 2004.

[3] D. Bhagwat, L. Chiticariu, and W. Tan. An annotation management system for relational databases. In VLDB, pages 900911, 2004.

[4] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In Proceedings of the 16th International Conference on Database Theory, ICDT 13, pages 177188, 2013.

[5] M. Y. Eltabakh,W. G. Aref, A. K. Elmagarmid, M. Ouzzani, and Y. N. Silva. Supporting annotations on relations. In EDBT, pages 379390, 2009.

[6] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. SIGMOD Record, 34(4):3441, 2005.

[7] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In CIDR, pages 273283, 2011.

[8] H. Park, R. Ikeda, and J.Widom. Ramp: A system for capturing and tracing

provenance in mapreduce workflows. In VLDB. Stanford InfoLab, August 2011.

[9] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. PVLDB, pages 346357, 2011.

[10] The Apache Software Foundation. HDFS architecture guide.http://hadoop.apache.org/hdfs/docs/current/hdfs-design.html.

[11] H. Lim, H. Herodotou, and S. Babu. Stubby: A Transformation-based Optimizer for MapReduce Workflows. PVLDB, 5(11):11961207, 2012.

[12] The Apache Software Foundation. HBase. http://hbase.apache.org/.

[13] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In PODS, 2011.

[14] O. Biton, S. C. Boulakia, and S. B. Davidson. Zoom*UserViews: Querying relevant provenance in workow systems. In VLDB, 2007.

[15] S. Bowers, T. M. McPhillips, and B. Ludascher. Provenance in collection-oriented scientic workows. Concurrency and Computation: Practice and Experience, 20(5), 2008.

[16] S. B. Davidson et al. Provenance in scientic workow systems. IEEE Data Eng. Bull., 30(4), 2007.

[17] Amsterdamer Y, Davidson S B, Deutch D, et al. Putting lipstick on pig: enabling database-style workflow provenance[J]. Proceedings of the VLDB Endowment, 2011, 5(4): 346-357.

[18] F. GEERTS and J. V. D. BUSSCHE. Relational completeness of query languages for annotated databases. DBPL, 4797:127137, 2007.

[19] S. B. Davidson et al. Provenance in scientic workow systems. IEEE Data Eng. Bull., 30(4), 2007.

[20] P. Missier, N. Paton, and K. Belhajjame. Fine-grained and ecient lineage

querying of collection-based workow provenance. In EDBT, 2010.

[21] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In PODS, 2011.

[22] Wang J, Crawl D, Altintas I. A framework for distributed data-parallel execution in the Kepler scientific workflow system[J]. Procedia Computer Science, 2012, 9: 1620-1629.

[23] Wang J, Crawl D, Altintas I. Kepler+ Hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems[C]//Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science. ACM, 2009: 12.

[24] Stafford C M, Harrison C, Beers K L, et al. A buckling-based metrology for measuring the elastic moduli of polymeric thin films[J]. Nature materials, 2004, 3(8): 545-550.

[25] Jian S, Xiaojing J. A Study of the Influence of Technical Architecture on the Total Cost of Google Cloud Computing Platform[J]. Telecommunications Science, 2010, 1(1): 38-44.

[26] Lohr S. Google and IBM join in cloud computingresearch[J]. New York Times, 2007, 8.

[27] Yang H, Dasdan A, Hsiao R L, et al. Map-reduce-merge: simplified relational data processing on large clusters[C]//Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007: 1029-1040.

[28] https://hive.apache.org/

[29] George L. HBase: the definitive guide[M]. O'Reilly Media, Inc., 2011.

[30] Khetrapal A, Ganesh V. HBase and Hypertable for large scale distributed storage systems[J]. Dept. of Computer Science, Purdue University, 2006.

[31] Thusoo A, Sarma J S, Jain N, et al. Hive-a petabyte scale data warehouse

using hadoop[C]//Data Engineering (ICDE), 2010 IEEE 26th International Conference on. IEEE, 2010: 996-1005.

[32] Olston C, Reed B, Srivastava U, et al. Pig latin: a not-so-foreign language for data processing[C]//Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008: 1099-1110.

[33] Nguyen P, Halem M. A mapreduce workflow system for architecting scientific data intensive applications[C]//Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing. ACM, 2011: 57-63.

[34] Lim H, Herodotou H, Babu S. Stubby: A transformation-based optimizer for mapreduce workflows[J]. Proceedings of the VLDB Endowment, 2012, 5(11): 1196-1207.

[35] Fei X, Lu S, Lin C. A mapreduce-enabled scientific workflow composition framework[C]//Web Services, 2009. ICWS 2009. IEEE International Conference on. IEEE, 2009: 663-670.

[36] Jahani E, Cafarella M J, R C. Automatic optimization for MapReduce programs[J]. Proceedings of the VLDB Endowment, 2011, 4(6): 385-396.

[37] Afrati F N, Ullman J D. Optimizing multiway joins in a map-reduce environment[J]. Knowledge and Data Engineering, IEEE Transactions on, 2011, 23(9): 1282-1298.

[38] Herodotou H, Lim H, Luo G, et al. Starfish: A Self-tuning System for Big Data Analytics[C]//CIDR. 2011, 11: 261-272.

[39] Chu C T, Kim S K, Lin Y A, et al. Map-reduce for machine learning on multicore[C]//NIPS. 2006, 6: 281-288.

[40] Thusoo A, Sarma J S, Jain N, et al. Hive: a warehousing solution over a map-reduce framework[J]. Proceedings of the VLDB Endowment, 2009, 2(2): 1626-1629.

[41] Ganapathi A, Chen Y, Fox A, et al. Statistics-driven workload modeling for the cloud[C]//Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on. IEEE, 2010: 87-92.

[42] Dean J. Experiences with MapReduce, an abstraction for large-scale computation[C]//PACT. 2006, 6: 1-1.

[43] Papadomanolakis S, Ailamaki A. Autopart: Automating schema design for large scientific databases using data partitioning[C]//Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on. IEEE, 2004: 383-392.

[44] Camon E, Magrane M, Barrell D, et al. The Gene Ontology annotation (GOA) database: sharing knowledge in Uniprot with Gene Ontology[J]. Nucleic acids research, 2004, 32(suppl 1): D262-D266.

[45] Gray J, Liu D T, Nieto-Santisteban M, et al. Scientific data management in the coming decade[J]. ACM SIGMOD Record, 2005, 34(4): 34-41.

[46] Gollub J, Ball C A, Binkley G, et al. The Stanford Microarray Database: data access and quality assessment tools[J]. Nucleic Acids Research, 2003, 31(1): 94-96.

[47] Codd E F. Extending the database relational model to capture more meaning[J]. ACM Transactions on Database Systems (TODS), 1979, 4(4): 397-434.

[48] Haskin R L, Lorie R A. On extending the functions of a relational database system[C]//Proceedings of the 1982 ACM SIGMOD international conference on Management of data. ACM, 1982: 207-212.

[49] Han J, Fu Y, Wang W, et al. DMQL: A data mining query language for relational databases[C]//Proc. 1996 SiGMOD. 1996, 96: 27-34.

[50] Zhang C, Naughton J, DeWitt D, et al. On supporting containment queries in relational database management systems[C]//ACM SIGMOD Record. ACM,

2001, 30(2): 425-436.

[51] Brown P G. Overview of SciDB: large scale array storage, processing and analysis[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 963-968.

[52] Low Y, Bickson D, Gonzalez J, et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud[J]. Proceedings of the VLDB Endowment, 2012, 5(8): 716-727.

[53] Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 135-146.

[54] Dennis Jr G, Sherman B T, Hosack D A, et al. DAVID: database for annotation, visualization, and integrated discovery[J]. Genome biol, 2003, 4(5): P3.

[55] Carver T, Berriman M, Tivey A, et al. Artemis and ACT: viewing, annotating and comparing sequences stored in a relational database[J]. Bioinformatics, 2008, 24(23): 2672-2676.

[56] Bateman A, Birney E, Cerruti L, et al. The Pfam protein families database[J]. Nucleic acids research, 2002, 30(1): 276-280.

[57] Zhang J, Jagadish H V. Lost source provenance[C]//Proceedings of the 13th International Conference on Extending Database Technology. ACM, 2010: 311-322.

[58] Davidson S B, Boulakia S C, Eyal A, et al. Provenance in Scientific Workflow Systems[J]. IEEE Data Eng. Bull., 2007, 30(4): 44-50.

[59] Buck J T, Ha S, Lee E A, et al. Ptolemy: A framework for simulating and prototyping heterogeneous systems[J]. 1994.

[60] Krishnamurthy S, Chandrasekaran S, Cooper O, et al. TelegraphCQ: An architectural status report[J]. IEEE Data Eng. Bull., 2003, 26(1): 11-18.

[61] Bhalla U S, Bower J M. Genesis: a neuronal simulation system[M]//Neural

Systems: Analysis and Modeling. Springer US, 1993: 95-102.

[62] Connolly T M, Begg C E. Database systems: a practical approach to design, implementation, and management[M]. Pearson Education, 2005.

[63] Widom J. The Starburst rule system: Language design, implementation, and applications[J]. IEEE Data Engineering Bulletin, 1992.

[64] Florescu D, Kossmann D. Storing and Querying XML Data using an RDMBS[J]. IEEE Data Eng. Bull., 1999, 22(3): 27-34.

[65] McGoveran D. Guide to SYBASE and SQL Server[M]. Addison-Wesley Longman Publishing Co., Inc., 1999.

[66] Fagin R, Nievergelt J, Pippenger N, et al. Extendible hashinga fast access method for dynamic files[J]. ACM Transactions on Database Systems (TODS), 1979, 4(3): 315-344.

[67] Chaudhuri S. An overview of query optimization in relational systems[C]//Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. ACM, 1998: 34-43.

[68] http://$en.wikipedia.org/wiki/Run - length_encoding$