Preparing Data For The Data Lake

Alekh Jindal, Samuel Madden

CSAIL, MIT

Abstract

Data preparation is increasingly becoming one of the biggest challenges in processing big data. While recent tools such as Tamer and Trifacta address the problem of integrating and cleaning the datasets as they come in, preparing these datasets for efficient processing over a variety of query workloads is still challenging. In this talk, I will discuss these challenges and describe our tool which allows for fine-grained data preparation, via a data preparation plan, and efficiently runs this plan while uploading the data to HDFS.

1. INTRODUCTION

Data preparation is increasingly becoming one of the biggest challenges in processing big data and good preparation is critical to making good use of the data, i.e. quickly moving from data to insights. This preparation includes (i) getting data in from a variety of sources, incurring tasks such as data integration and cleaning to make the data usable, and (ii) using this data in a variety of ways (application workload), incurring tasks such as logical and physical data design to deliver good performance. A number of recent tools, such as Tamer [8] and Trifacta [9], have addressed the problem of integrating and cleaning datasets. However, preparing datasets in order to efficiently process them over different workloads still remains a challenge. This is because end users increasingly face heterogenous and ad-hoc query workloads for which they want to prepare their data in an ad-hoc manner. Such data preparation could involve building samples or other statistics, creating layouts, indexes, partitions, and other physical structures, and applying compression and other encoding schemes. Existing approaches to workload-based data preparation have three major limitations:

(1.) Ease-of-use. The current practice, typically, is to write ad-hoc scripts, e.g., using Python, Perl, or Bash, which is tedious and time consuming. Furthermore, since different workloads require different data preparation, analysts often end up duplicating much of their effort. On the other extreme, languages such as RodentStore [3] and WWhow! [7] allow users to express their data designs at a high level. However, it is not clear how these translate to the actual underlying data preparation tasks. Therefore, we need a more principled way of allowing data analysts and scientist to specify their workload-based data preparation steps.

(2.) Flexibility. Several existing tools are limited to specific data preparation steps, such as indexing or partitioning the entire datasets. Given that workloads are often heterogenous and changing, end users want greater flexibility in preparing their datasets. This is important because they may prepare data differently in different scenarios, prepare data heterogeneously to guard against the worst case scenarios, or selectively prepare only the relevant portions of the data in order to avoid high preparation costs. All this requires the end users to have full control over their dataset and the flexibility to arbitrarily prepare them.

(3.) Efficiency. Data preparation is typically a time consuming process and the analysts need to wait for a significant amount of time while the data is being prepared. This is counter productive in the modern setting where the analyst would want to start using the data as soon as it is available, perform his analysis, and move on to the next dataset. This means that data should be prepared as soon as the it gets stored in a distributed storage system and the preparation should not take longer than processing the raw data itself. This calls for preparing the data in an efficient manner.

In this paper, we describe building a system for preparing data in a scalable manner. Our goal is to allow end-users to easily and efficiently prepare their datasets for a variety of workload configurations. In the following, we describe our approach in Section 2, and discuss several use-cases in Section 3.

2. OUR APPROACH

We propose a new tool which eases much of the end user pain for preparing their datasets over a given workload. Below we describe how our tool address the three challenges described in Section 1, namely the ease-of-use, flexibility, and efficiency.

Data Preparation Plan. Our tool allows developers to specify the data preparation at a logical level, i.e., as a set of operators describing the data preparation task. The system takes care of running these operators in a distributed fashion, allowing developers to focus on actual preparation logic rather than the messy execution details. To do this, our tool provides a *logical dataset abstraction*, wherein developers reason about the logical datasets, e.g., sets of tuples or lists of log records. This is more natural to interact as developers do not have to worry about how the data is physically stored in the underlying file system. Internally, a logical dataset can be physically represented in multiple ways (e.g, as one file or many). As a result of this abstraction, developers can express the data preparation logic in a file-system independent manner, thereby making it easier to configure, reuse, and extend the preparation. All without hacking into the file system or the query engine.

Fine-grained Data Flow. Data preparation will vary depending on the application workload. Therefore, instead of hard-coding a fixed set of preparation into the underlying file system [6, 5, 4], our tool allows developers to specify custom data preparation. Developers can also specify different preparation for different datasets (in the same file system) as well as change them later on. To do this, our tools allows for data preparation at different data granularity, e.g., the entire dataset, or a given replica, block, or tuple. Each data item has a list of one or more labels (IDs) associated with it, denoting the preparation that have been applied to it. For example, a replication operator assigns a replica ID, a placement operator assigns a location ID, and a serializer operator assigns a serialization ID. These labels (or IDs) could then be used to conditionally or selectively prepare different portions of the data differently. For example, the users can compose multiple operators by chaining, pipelining, or branching them based on the data item labels. As a result, users have fine-grained control to prepare their data for their workloads.

Prepare-As-You-Upload. Given that data preparation is needed

for almost all modern applications, our tool allows developers to prepare their dataset while it is being uploaded into a distributed file system. This means that the prepared data is available as soon as it is uploaded. Our tool interacts and tightly integrates with the HDFS, and masks much of the preparation costs within more expensive operations, such as disk and network I/O, during upload. Furthermore, it runs the preparation jobs on a cluster of machines, utilizing the multi-core CPUs on each machine and efficiently moving the data during the preparation process. As a result, there is very little overhead of data preparation and it does not block the resources for the actual query processing. Finally, our tool does not change the underlying HDFS. Rather, it works as a software layer that reads and write data on the application or users' behalf. As a result, it can work out-of-the-box with existing HDFS installations.

3. USE CASES

We now show four different use cases where our tool makes it easy to prepare datasets for different workloads.

(1.) Violation Detection. Our tool allows users to perform violation detection while uploading their datasets, thereby speeding up the data cleaning process. Users can detect which portions of the data violate their business rules, often the most expensive step in the data cleaning process [2], and apply simple repairs. Though data repair may be an iterative process, our tool allows users to apply one-pass repairs on their datasets. For example, consider the TPC-H lineitem table, which includes one entry per item per order in a hypothetical business analytics application. This table includes a shipdate field (the date the item shipped) and a linestatus field (whether the item has shipped or not). Suppose we may want to enforce a *functional dependency* (FD) that *shipdate determines linestatus*, i.e., that any product shipped on a particular date has the same linestatus. Our tool allows users to detect all data items which violate this FD, i.e., that have the same shipdate but different linestatus, using the following plan of two *blocks* $b_1$and b_2:

$$b_1 = p_{\text{parse, rangePartition, 64mbPartition, store}}$$
$$b_2 = p_{\text{shuffleParse, detectViolations}}$$

In this preparation plan, b_1 is a pipeline of operators that range partitions the data on shipdate and collects each partition in a different physical file, which are then stored on HDFS. We then iterate over every pair of data items in each range partition and check whether or not there is a violation using a detect operator (block b_2). These violated records could subsequently be output to a violations file (for correction) or simply discarded.

(2.) Data Sampling. Sampling is a common technique to gather quick insights from very large datasets [1]. However, a key problem in using samples is the process of generating samples themselves: producing a sample requires an entire pass over all of the data. To help with sampling, our tool allows users to collect samples as the data is being uploaded, with very minimal overhead. For example, we can create Bernoulli samples by probabilistically replicating some of the tuples in the dataset. The replicated data items form the samples and are collected into a separate physical file. The preparation plan for sampling consists of a block b of three operators: $b = p_{\text{parse[store,sampleStore]}}$. The braces after parse operator indicate that the output is fed to both store and sampleStore. This means that the parsed tuples are stored in full as well as sampled. Likewise, we can also perform reservoir sampling by adding each data item into a reservoir (and removing items from it with a given probability) and then producing the sampled data items in the end, i.e., in the *finalize* method of the preparation operator.

(3.) Data Analytics. Our tool also allows developers to easily create physical designs, without making deep changes to HDFS or software installations. For example, while HDFS only considers the physical data blocks for data placement, our tool also enables placement based on the logical content of data blocks. A preparation plan with content-based data placement is as follows:

$$b_1 = p_{\text{parse, rangePartition, 64mbPartition}}$$
$$b_2 = p_{\text{dummy [hotLocate, coldLocate]}}$$
$$b_3 = p_{\text{store}}$$

In this preparation plan, we have two Locator operators, one for hot data blocks and another for the cold data blocks. Such content-based data placement allows users to control how the data is distributed logically (e.g., based on the value of some attribute) and where the query processing takes place. This has several interesting applications, including: (i) utilizing only a portion of the cluster to save energy or multiplex resources, (ii) improving data locality, and (iii) isolating concurrent queries to different nodes.

(4.) Storage Space Optimizations. Despite the plummeting price of disks, storage space still remains a concern in replicated storage systems with large datasets. Our tool offers several methods to deal with this. Consider replication for instance. A distributed file system typically constraints how data is replicated. For example, HDFS employs file-based replication that causes all parts of the file to be replicated in an identical manner. Users can only control how many times a file is replicated. Our tool allows users to control both what parts of the data are replicated and how many times. This control becomes crucial when different parts of the data have different relative importance. For example, a user storing weblogs in HDFS might replicate the most recent logs (hot data) more frequently than the older logs (cold data). To do this, user would simply apply a range partitioner (based on date) to cause data in the hot range to be replicated more frequently than other data. Such a preparation plan is as follows:

$$b = p_{\text{parse, rangePartition [10xReplicate, 2xReplicate], store}}$$

Such a preparation plan saves storage space for the large fraction of data that is cold and achieves better data locality over hot data, resulting in better overall performance on skewed workloads.

4. CONCLUSION

Data preparation is crucial to efficient data processing. However, existing tools to prepare data for a given workload are monolithic and limited. In this paper, we described a fine-grained approach to data preparation which allows developers full control over how to preprocess and prepare their datasets, using a logical data preparation plan. As a result, developers can quickly prepare their datasets for a variety of use-cases, including violation detection, data sampling, data analytics, and storage space optimizations.

5. REFERENCES

[1] S. Agarwal and al. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013.

[2] X. Chu, I. F. Ilyas, and P. Papotti. Holistic Data Cleaning: Putting Violations into Context. In *ICDE*, 2013.

[3] P. Cudré-Mauroux, E. Wu, and S. Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.

[4] J. Dittrich and al. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11):1591–1602, 2012.

[5] M. Y. Eltabakh and al. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *PVLDB*, 4(9):575–585, 2011.

[6] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: right shoes for a running elephant. In *SoCC*, page 21, 2011.

[7] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. In *CIDR*, 2013.

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

[9] Trifacta, http://www.trifacta.com.