Toward Personalized and Scalable Voice-Enabled Services Powered by Big Data

Jong Hoon Ahnn

Cloud Research Lab, Samsung Research America - Silicon Valley
75 W Plumeria Drive, San Jose, CA 95134, USA
Email: {j.ahnn}@samsung.com

Abstract—Recently advances in ASR and big data technologies drive more personalized services in many areas of services. A speaker adaptation is one good example which requires huge computation cost in creating a personalized acoustic model and corresponding language model over 100s millions of Samsung product users. We propose a personalized and scalable ASR system powered by the big data infrastructure which brings data-driven personalized opportunities to voice-enabled services such as voice-to-text transcriber, voice-enabled web search in a peta bytes scale. We verify the feasibility of speaker adaptation based on 107 testers’ recordings and obtain about 10% of recognition accuracy. We study an optimal set of execution environments by executing jobs running either on Hadoop 1 (59 machines) or Hadoop 2 (15 machines) cluster, and move forward performance optimization strategies: workflow compaction, file compression, best file system selection among several distributed file systems. We devise a metric for the cost of personalized model creation to compare the efficiency of one cluster with the other cluster, and it provides the estimated total execution time for the given number of machines. We finally introduce our in-house object storage and data storage design, and their high performance compared to state-of-the-art systems, optimized for voice-enabled services to effectively support small and large files (1KB-100KB for speech files, 10MB for a language model, 30MB for an acoustic model).

I. INTRODUCTION

With the rising attention on the voice-enabled service, the personalization of such service is preferred. One direction to provide more personalized nature to ASR is known as speaker adaptive recognition or speaker adaptation where an acoustic model is adapted to an individual speaker to improve the quality of speech recognition. The other direction is to personalize a language model in ASR where a statistical language model assigns a probability to a sequence of words according to individual speaker’s way of speaking reflecting direct, tone, intonation, and other linguistic properties that may be uniquely identifiable among multi-user ASR environment. This work focuses more on the former while exploring both of directions to provide better personalized ASR experience. One research question is whether or not the speaker adaptation really works for mobile services in case where each utterance is relatively shorter than the other scenario such as continuous transcriber, web search. An example would be call mom, search restaurant nearby. This shorter utterance pattern may not guarantee the improvement of a personalized acoustic and language model at all times. Another research question is whether or not the small amount of user’s speech data with no large computation requirement (significantly less than typical model training) can actually contribute to making the model better.

Figure 1 presents the frequency of the top 100 utterances from Samsung S-Voice logs which reveals a well-known long tail effect on user’s speech behavior. The top most frequently spoken words are call with 82,056,990 times, and following words are i, is, are galaxy, and hi following the power law. This means that we may have lots of opportunities to personalize voice-enabled services where individual user’s speech can be highly customized in the long tail area.

The idea of the personalizing ASR has been extensive discussed in the literature ([25], [22], [24], [23]). We however realize that implementing such system is a big challenge due to a lack of scalable, distributed computing and storage systems. Often, researchers in the area of ASR assume that a scalable system that supports large-scale ASR system is readily available when needed. Furthermore, a trend toward personalizing acoustic model and language model boosts this problem even more difficult in terms of parallel computation and distributed storage. Parallel computing must support personalized model generation of individual users to meet
SLAs in voice-enabled services. Distributed storage must store user history data (e.g., speech files, transcribed text, other user history logs, acoustic models, language models) in a virtually infinite capacity. For instance, CMU Sphinx [8] requires 10 mega bytes (MB) of an acoustic model and 30MB of a language model per person in our personalized ASR scenario. In order to personalize at least 100 million users in Samsung’s S-Voice service, the required storage capacity only for acoustic models can easily exceed over 4 peta bytes (PB). In fact, each user requires 62MB in total, becoming at least 6.2 PB in the same scenario that cannot be dealt with small-scale computing and storage systems. Thus, the simple term, personalization of the ASR service challenges in several ways: large-scale computation, distributed storage, and reliable and seamless data sharing between compute nodes and storage’s data nodes.

To this end, personalizing voice-enabled service becomes a system problem for given well-known speech libraries. This trend behooves us to consider the big data platform as a candidate for the proposed ASR backbone that has personalization capability in a scalable manner for from millions to billions of users. Hadoop [15] is on the top list since it has been deployed in several large-scale services, (e.g., Yahoo reported 10s of thousands machines in their Hadoop cluster, Facebook reported 600 machines in their cluster [29]). In our experiments, our in-house Hadoop-like cluster (59 machines) is able to create one million users’ personalized acoustic models (PAMs) within one hour and 100 million PAMs within 3.5 days without applying any optimization techniques. This means the voice-enabled system periodically can refresh speaker-adapted acoustic models in an every hour for a one million active user scenario where inactive user’s PAM does not require to be renewed frequently. Furthermore, the integration with the big data infrastructure for such voice-enabled services allows to provide many more potential opportunities in computation (scalable computing environment and storage with reliability), analysis with eco-system tools (e.g, elasticsearch, hive, pig, etc).

The contribution of this work is follows,

- To our best knowledge, this work is the first exploration of personalized voice-enabled services in implementation and practice built on top of the big data infrastructure.
- We implement a voice-enabled service system toward commercialization considering reliability and scalability, considering 100s of million users scenario.
- We devise a PAM computation metric to compare the efficiency of one cluster with the other cluster. This metric becomes really useful when the cluster size is different and also provides the estimated number of nodes to meet SLAs for a given scenario.
- We propose performance optimization techniques in a practical level: workflow compaction, file compression, small and large file support strategy.
- We introduce our in-house object storage and data storage design, and their high performance compared to state-of-the-art systems, optimized for voice-enabled services to effectively support small and large files.

II. RELATED WORK

Sphinx is a well-known speech library written in C and Java and developed by CMU. One drawback to this is no neural network support. Kaldi [12] implements advanced features such as subspace GMM and FST-based speech recognizer. Unlike Sphinx [8] and HTK [11], it is written in C++ instead of C. Another interesting part of Kaldi is that it is using weighted finite state transducer (WFST) as the unifying knowledge source representation. However, the composed decoding WFST would naturally outgrow the system memory as the vocabulary size goes large and knowledge source gets more complicated. HTK is another speech library that is mainly based on MPE and MMIE. Julius is a high-speed speech recognizer that can decode a 60k vocabulary. One speed-up techniques of Sphinx 3.X was context-independent phone Gaussian mixture model selection (CIGMMS), however, Julius [13] comes with a set of Japanese models, not English, and this might be one of the reasons why it is not as popular as HTK, Sphinx, Kaldi. In the consideration of large-scale speech-enabled services on top of the big data infrastructure such as Hadoop, all four speech libraries aforementioned require lots of efforts on deployment in service by even requiring source code modification. In order to implement scalable system and advanced ASR system, seamless integration with the data infra is unavoidable.

Recently, the study on distributed speech recognition system has been extensively explored in the literature [7]. You et al. [6] proposes an OpenMP-based speech recognizer for continuous speech streams. The drawback of their proposal is that the system keeps all the streaming data in memory, and potential out-of-memory problem exists for large data stream. Chong et al. [5] proposes a method to parallelize speech recognizer for parsing given voice streams. No proposal is made for scalable system as data go big where scalable storage and computing need to be in place. Chang et al. [3] presents a Cloud-assisted speech recognition service for personal mobile device. Even though the scheme is highly optimized for mobile devices, backend data storage and computation nodes are not scalable, specially when highly personalized services are needed such as personalized acoustic model and language model generation/retrieval in a real time manner.

III. PERSONALIZAION OF ASR: SPEAKER ADAPTATION

The feasibility of speaker adaptation has been well known [4], [3]. Most speech libraries such as Sphinx, Kaldi, HTK [11] expects to have about 10 percent of speaker adaptation
improvement with the decent amount of speech files. There are however several dependent variables which are not known due to variations of training set, testing set, language model, acoustic model, and adaptation set. This section verifies such claims from the literature since the goal of our work is to implement a fully working system toward commercialization. Sphinx supports two different methods in this direction: maximum a-posterior (MAP) re-estimation of the parameters and maximum likelihood linear regression (MLLR). For details, refer to [8]. From our experiments, MLLR does not contribute the speech recognition accuracy at all data set we used. Therefore, the rest of discussion in this section is relying on MAP in Sphinx.

We first start with three male in-house testers: one native American and two testers who have lived in U.S. more than 10 years. Each tester recorded one script twice so that the first recording is used for adapting and the second for testing where one script makes around 10-30 utterances using Sphinx. Figure 2 presents the accuracy on PAM in 1-(word error rate) based on scripts from five different children’s books (yellow-coded). The barplot plots min, 25 percentile, median, 75 percentile, max from bottom to top. The dot within the bar indicates mean of the distribution. The blue-colored baseline BL (with general acoustic model) on the first column is presented to compare with other PAMs. The last two columns present the accuracy of all script combined: all1 denotes the first recording is used for adaptation; all2 denotes the second recording is used for adaptation. With the small adaptation set of 10-30 utterances (B, 0, A, B, AA), it is difficult to observe the improvement due to larger variations in accuracy. We however clearly see the benefit of PAM in all1 and all2 dataset, which consist of 89 utterances each. The results say that we can have well-known accuracy improvement (10%) with around 90 utterances in children’s books, compared to the baseline (BL).

In order to better verify the speaker adaptation, we reflect smartphone user’s use cases to scripts such as call mom, search for best gifts for thanks giving, directions to home, best Japanese restaurant near me. Scripts consisting of 200 utterances are sampled from Samsung’s S-Voice logs and
We describe five design principles in the big data infrastructure and design necessary components for the personalized voice-enabled services by following each principle. On the left bottom corner in Figure 5, we have given unique stacks: offline storage, online storage, execution engine, data science, and services. The offline storage stores big data in a fault-tolerant and distributed manner. Since the offline storage is mostly for batch processing, it is not designed for real-time data fetching scenarios. On the other hand, online storage is designed to write and read files and tables from run-time processes in the execution engine. The execution engine may require either offline or online storage depending upon application scenarios. In most of time in this paper, the execution engine is referred by batch processing such as map/reduce jobs. On top, the data science layer performs business-specific algorithms. For instance, ASR runs a continuous transcriber, which will require acoustic model and language model to provide speech-to-text transcription real-time. On the highest layer, the design has services where Web-based Restful APIs (e.g., Jetty [9]) are one of the commonly referred example.

Followed by the aforementioned design principles, we design personalized voice-enabled services in a following way. A user speaks to the devices where a speech file is transferred through the Web Restful APIs on the services layer. The Web service called ASR engine at run time translates the user’s speech to text on the data science layer. Initially, the ASR engine loads a general acoustic model (AM) and general language model (LM), which are not personalized yet. As the user’s speech files pile up on the offline storage corresponding to distributed file systems such as Hadoop DFS (HDFS), it is possible to create personalized models adapted to user’s intonation, dialect, context, or ambient noise, basically reflecting everything in his recordings. This back-end processing is performed in the execution engine layer; we call it a speaker adaptation engine where individual user’s ASR-related history logs are ready to use from the online storage for fast fetching. For the personalized model creation, we target a scalable computing environment setup ranges from 1M to 100M user scenarios to achieve. The online storage consists of two major software systems: object storage and data storage. The object storage is used to store files larger than several hundreds KB, while the data storage stores small data in a structured way such as key-value based tables. The choice of both online storages naturally becomes a system problem in terms of performance and scalability and needs to be dealt with scenario by scenario on applications. Our goal is to provide an optimal strategy for distributed file systems to effectively support voice-enabled services.

Toward personalizing ASR services, speaker adapted models must be loaded upon requests by the ASR engine at run time which will require fast retrieval. It is norm that the online storage is faster than the offline storage by design. We will discuss which storage system on which scenario makes the best performance as a primary goal in Section VII.

V. Implementation

This section describes the implementation of the personalized ASR service powered by the big infrastructure. Hadoop [15] is our choice as a base system to implement all whole system components: online storage, execution engine, data science, and services layer.

A. Online Storage

The online storage is known to be faster than the offline storage system in nature. HBase [16] is one of the best key-value data storage when we develop the system. From the personalized ASR system, we require 1KB-100KB files including logs and speech files, 10MB for personalized language model, and 30MB for personalized acoustic model. Due to the nature of personalization, the system must fetch files fast in real time. In order to support such various files, the online storage such as HBase has a fundamental drawback when the size of file goes big since the online storage is designed not for large files but for small data or small files. To resolve this issue in a systematic way, there are several recently proposed systems such as Facebook’s Haystack [14], Twitter’s Blobstore [17], and Hbase large object (LOB) storage [18]. They are very similar to our in-house object storage and data storage in that all three systems aims to store small and large binary large objects, and fastly fetch files at low cost. Our in-house object storage
supports list, create and delete buckets, and put and get objects via the Restful APIs implemented by Jetty [9].

The data storage is a key-value based data store where HBase can be accessed by Restful APIs, and its supporting operations are create tables, store/delete/update values in tables, get a rows in table. The Restful APIs allow the ASR system to easily store and fetch data from tables. On top of the data storage, we implement the object storage for storing large objects such as PLM and PAM files. The data storage is used both in real-time (speech-to-text conversion) and batch processing (individual user’s PAM generation) from the execution engine. The design principle of the object storage is quite similar to Facebook’s Haystack where the data storage is a major storage for file’s metadata and larger files are stored in the HDFS part of the object storage. If a fetching file is less than a threshold, say 100KB, it reads a pointer to HDFS from the metadata in the data storage that leads to the real file in HDFS. We note that the threshold is somewhere we need to investigate further in the evaluation section.

The current implementation of in-house object storage and data storage does not support to specify file locations as objectstorage://targetdir/targetfile or datastorage://targetdir/targetfile in the Hadoop streaming interface. This type of file access is already supported by the native HDFS APIs such as hdfs://targetdir/targetfile. We implement this functionality since it is better off to support such an interface to smoothly integrate Hadoop ecosystems as supplemental storage schemes.

B. Execution Engine

The execution engine roles as a batch processor for generating personalized acoustic models and language models for individual speakers. In 100 million (M) user scenario, the size of data can easily pile up to 6.2PB. For agile and better personalized voice-enable services, we may need more frequent re-computation to keep up-to-date models. A typical approach in large data processing is the use of map/reduce programing model. We however note that none of speech libraries directly support the map/reduce, requiring lots of library modifications, which are obviously inefficient.

The basic goal of the execution engine support is to have speech library run without any modification.

To this end, a Hadoop streaming interface meets our requirement. It is a utility that comes with the Hadoop distribution. The utility allows you to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. Some of usage examples are detailed in [19]. When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized. As the mapper task runs, it converts its inputs into lines and feed the lines to the stdin of the process. In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper. In our execution, we do not require reducers to execute. This leads to a potential problem of resource under-utilization in Hadoop 1 since pre-assigned mapper slots cannot be shared with reducer slots leaving them idle. In the worst case, 944 mappers can run when no reducers among 590 slots in our Hadoop1 cluster are at work. This is a well-known problem extensively studied in the literature [20]. Fortunately, Hadoop 2 resolves this issue by centralizing resource management of mappers and reducers in one place.

B. Execution Engine

The execution engine roles as a batch processor for generating personalized acoustic models and language models for individual speakers. In 100 million (M) user scenario, the size of data can easily pile up to 6.2PB. For agile and better personalized voice-enable services, we may need more frequent re-computation to keep up-to-date models. A typical approach in large data processing is the use of map/reduce programing model. We however note that none of speech libraries directly support the map/reduce, requiring lots of library modifications, which are obviously inefficient.
an output of prior results is fed to the next program in order. The last four programs are originated from SphinxTrain 1.0.8. The first four programs generate a personalized acoustic model by applying the MLLR method while the last two programs generate another personalized acoustic model by applying the MAP method. Details of methods are out of scope and refer to [8]. We start with a bundle acoustic model in PocketSphinx called hub4wsj_sc_6k for our result to be re-playable by any other researchers. An automated script creates two different PAMs from MAP and MLLR method. From the result, we take a model that performs better speech recognition.

C. Data Science and Services

Figure 6 is a screenshot for Web page implementation of ASR services while Figure 7 is a voice-enable mobile application screenshot. Both applications access to Apache Web server where the personalized ASR is loaded and transcribes voice to text. In the Web application, when a user’s speech file is sent to the ASR service, it responds three different scripts: original script, transcribed text from general acoustic model, and transcribed text from personalized acoustic model. The corresponding speech recognition accuracy is presented in both transcribed results: 75% and 83.33%, respectively. It is common that the error rate is calculated in word-by-word comparison. This means the user obtained 8.33% of improvement in ASR accuracy by the personalization technique. The mobile screenshot also shows both before and after results in a similar fashion.

CMU Sphinx provides two versions of speech libraries: PocketSphinx and Sphinx 4. PocketSphinx is a light-weight implementation based on SphinxBase implemented in C while Sphinx 4 is fully implemented in Java which can lead to a potential performance problem. From our test, Sphinx 4 (several seconds to response) is way slower than PocketSphinx (less than 50ms) in a bundle transcriber application on top of Apache Web server. So, our choice is to have PocketSphinx implementation in our front-end service layer for better performance.

VI. EXPERIMENT SETTINGS

We briefly describe experiment settings for the PAM generation jobs on Hadoop. Figure 8 presents a whole flow from a job submission client to the compute node that performs actual Sphinx jobs. A one PAM creation requires to call six different Sphinx programs (each program corresponds to a job) in three different speech libraries: SphinxBase, PocketSphinx, and SphinxTrain. In the optimization discussion, we perform a workflow compaction from six to one job to reduce the total execution time. Our description follows the order of numbers we put in the figure. Once the job client requests a workflow execution via a Hadoop streaming interface (1), the interface sends this request to a Hadoop job manager (2). Note that Hadoop 1 calls it a job tracker while Hadoop 2 calls it an application manager. The job manager assigns empty slots for map jobs to each of requested Sphinx jobs (3). After the job assignment, the rest of unassigned jobs are left in the job queue to wait for other empty slots. The job manager is also responsible for shipping all the required program binaries, specified files and directories to a temporary directory on the assigned compute node. Note that this shipping can be a bottleneck if large files are transferred. Usually Hadoop streaming supports a distributed cache up to 10GB to avoid file system access, leading to the performance degrade. One challenge is the large number of PAMs generation requires PB scale file read from somewhere. It is obvious that the 10GB distributed cache cannot fit to our scenario of execution. This situation leads for each compute node to read required 62MB uncompressed (4-1) or 35MB compressed files (4-2) from HDFS. Another options are reading files from object storage (4-3), data storage (4-4), or external distributed file systems such as Google object storage (4-5). Such five considerations are motivated from a performance improvement perspective which we will deeply discuss in Section VII. Finally, output files including a PAM (30MB uncompressed, 17MB compressed) need to be shipped to user-specified file systems (can be any of five options aforementioned) after the job completion. The last mission of the compute node also incurs the large write file operation time.
VII. RESULT AND OPTIMIZATION

We run the proposed system in two different clusters: 59 nodes running Hadoop 1 (Hadoop1 cluster) and 15 nodes running Hadoop 2 (Hadoop2 cluster). Hadoop1 cluster has the capability of 944 slots for mappers and 590 slots for reducers. Both clusters hold homogeneous machines with Intel Xeon 2.00GHz 12 cores, 64GB memory. The later section describes four different optimization techniques to enhance the performance in computation assuming that the speaker adaptation capability is bounded by a specific speech library, in this case CMU Sphinx [8]. This optimization goal is reasonable since we aim to build a scalable voice-enabled library, in this case CMU Sphinx [8]. This optimization goal is reasonable since we aim to build a scalable voice-enabled service by the help of big data platform.

PAM Computation Metric. Terasort is a well-known tool for measuring the sorting efficiency of the Hadoop cluster based on 1 terabytes of data [21]. The sorting efficiency is defined in  \( \frac{MB}{sec \times core} \). Inspired by the metric of terasort, we define the efficiency of PAM computation metric \( E \) as

\[
E = \frac{P}{T \times C},
\]

where \( P \) denotes the total number of PAMs generated, \( T \) denotes the total elapsed time to generate PAMs, and \( C \) denotes the total number of CPU cores used in the PAM generation. This metric is useful to compare one Hadoop cluster with the other, yet the different number of cores.

A. Hadoop1 cluster vs Hadoop2 cluster

As a preliminary execution, we run 40K PAMs jobs on a single machine-based Hadoop1 cluster, and it took 7.8 days to complete. As a parallelism test, we also run the same jobs under the condition that 25 job submission clients concurrently submit each job to the 59 nodes Hadoop1 cluster, and this run took 3 days to complete. In the later experiment, the concurrent number of mappers does not go up 30, meaning that parallelism is not fully taken into account. Given that amount of computation, we would investigate the performance of Hadoop1 and Hadoop2 cluster, with no optimization technique applied. Figure 9 and Figure 10 present PAM generation time in hour as the number of PAMs grows on Hadoop1 and Hadoop2 cluster, respectively. For the 40K PAMs scenario, Hadoop1 finishes jobs in 1.25 hr and Hadoop2 does them in 2.25 hr. Note that Hadoop1 cluster has 3 times more nodes than Hadoop2 cluster. Given that node difference, such result is not shocking. Figure 11 provides an effective way to compare the efficiency metric defined in Eq. 1. For given scenarios from 10K to 50K PAMs, Hadoop2 is more efficient about 15% to 25% depending on scenarios.

The reason why we stopped our experiments on 80K for Hadoop1 and 50K for Hadoop2 is that we got a Hadoop history sever crash over that number. We find that excessive syslog (10s of GB) from Sphinx library causes an out-of-heap memory problem although we run stable versions of Hadoop1 and Hadoop2.

PAM Efficiency Metric. As we define an efficiency metric in Eq. 1, we here verify how good the metric provides estimated time for the given number of PAMs to be generated in both Hadoop1 (H1) and Hadoop2 (H2) as shown in Figure 12. \( E_{H1} \) and \( E_{H2} \) denote the efficiency of PAM computation for H11 and H2, respectively. \( E_{H1} \) and \( E_{H2} \) are calculated at 80K and 50K PAMs scenarios, respectively. The x-axis presents the number of PAMs under either H1 or H2. Except for the case of H2.20K, all the other cases sit within 95% of confidence interval. This verifies the effectiveness of the metric Eq. 1 experimentally.

The following experiment in Figure 13 goes on to say how both clusters can be scalable as the number of PAMs grows, and how many nodes are required to meet the real world service scenario. For instance, when we may need to update 100M PAMs every week, the number of nodes requires to meet the requirement is around 700 nodes for...
Hadoop1 and 600 nodes for Hadoop2. This estimation is very important to consider when designing the scalable voice-enabled system for commercialization. The trend in the estimated time stays in log scale as we put more nodes in clusters. In a cost effective aspect, 700 nodes and 1000 nodes do not make much difference in performance for one week cycle in PAM update. Given that typical price of 4K-5K dollars per machine, 300 machines can affect a huge memory loss. The PAM update period is one another factor to consider in this direction. If we set the update period as 2 weeks, 400 and 300 nodes are enough for Hadoop1 and Hadoop2 respectively, saying a half of investment on cluster purchase. It estimates 100M PAMs generation time in 4.4 days on 1000 nodes H1 and H2 cluster. Also both H1 and H2 can finish 1M PAMs generation within 1 day on the same hardware settings.

B. Hadoop Overhead

As an optimization effort, we study the overhead caused by Hadoop including job coordination, file operations, cluster management overhead, etc. Figure 14 compares the total job execution time for 10K to 50K PAMs scenarios with the hadoop overhead in time. A way we measure the overhead is that we eliminate all the Sphinx programs while keeping all the other operations including file read and write. We obtain around 25% of time difference between the two if we normalize the figure. In other words, Sphinx computation running on Hadoop only takes 25% of the total execution time, and this gives us to consider more optimization opportunities such as file compression, workflow compaction. In fact, Sphinx requires 6 different jobs to run, and we optimize 6 jobs to 1 job for each PAM running on a Hadoop compute node. By such workflow compaction, we save 62.5% of the total execution time since this save is mainly due to the Hadoop overhead.

C. File Compression

A file compression technique is another interesting area to explore for the execution time optimization. Each PAM generation requires 62MB of files shipping to a temporal directory on an assigned Hadoop compute node. The file compression reduces it to 35MB meaning that we can save a half of file reading time. We however need to consider decompression time for the compressed file after downloading. Figure 15 consider such scenario to study the feasibility of file compression benefit in time. We set up a script to read 50 iterations of PAM file from HDFS and zipped (=compressed) PAMs to compare with the former. The red-colored bottom dots present reading time for compressed PAM while the baseline for uncompressed PAM reading time is on the top dots. After considering the decompression time after downloading we obtain blue dots in the middle that has 70% of file reading time saving compared to the baseline. This means HDFS file operation has lots of overhead in time, and file compression may effectively reduce the operation time. To prove such consideration we set up 10K PAM execution on both uncompressed PAM and compressed PAM scenario in Figure 16. If we normalize two bars in the figure, we obtain 18% of difference in between the two. This proves the effectiveness of the file compression.

D. Small File Support

Other than PAM generation, the ASR creates lots of small files including speech files (1KB-100KB typically) and text logs (1KB-50KB). To support such transactions for file write and read operations, we compare various existing options we can consider. Figure 17 presents the measure of file read time ranging from 10KB to 120KB considering four different technologies available: the method starting with SDP denotes in-house developed systems. In fact, most of in-house developed systems outperform well-known Google object storage. Among SDP systems, data storage and object
storage are the one to pursue. The data storage is a key-
value based system that stores (key,value) pairs in tables
while the object storage is a hybrid file system that utilizes
HBase and HDFS in one place to make it better performed
for various file size. Figure 18 presents write time of SDP
systems. Although the number of read is much higher than
the number of write in a typical ASR system, we want to
make sure that system does not be overkilled by the write
operation. It seems both the data storage and object storage
performs fairly good enough to consider within the small
file size range.

E. Large File Support

As aforementioned, large files up to 30MB must be
effectively supported for the PAM generation. We observe
that the data storage needs to be carefull when considering
due to the nature of database system, where large files
are less likely to be effectively supported in terms of fast
read and write. The rest of options we have is that HDFS
native APIs and SDP object storage. The measurement is
performed in the early morning of weekdays to make sure
that the cluster does not have other large file operations.
We only present the performance of reads since the number
of reads is much larger than the number writes for the
large files. Figure 19 presents read time for various file
size under no traffic injected to H1 cluster, and the object
storage outperforms over the HDFS native in most of file
sizes. In order to observe the situation where lots of file
operation traffic injected, we generate 944 mappers, each of
which writes 30MB file to HDFS and reads it five times
accordingly. We consider this dummy traffic as very large
traffic since the number of mappers is bounded by 944 in H1
cluster. We measure read time of various files with different
size while the underlying file operations exist. In Figure 20,
we observe the shift of the crossing point (15MB to 5MB)
compared to Figure 19. This means that the SDP object
storage can provide better performance under 5MB of files
for given large traffic. For the given workload in the cluster, it is recommended to adapt the file system in a hybrid way where the SDP HDFS native is preferred over 5MB file size.

VIII. Conclusion

We first verified the feasibility of speaker adaptation based on 107 testers’ recordings and obtained about 10% of recognition accuracy. We proposed a scalable and personalized ASR system powered by the big data infrastructure. We reported Hadoop2 is 15%-25% more efficient than Hadoop for given PAM scenarios, and Hadoop overhead is about 75% of the total execution time. With workflow compaction, we saved 65.2% of the total execution time, and file compression also contributed to save 70% of file operation time and 18% of the total PAM creation time. For the best file system support, we explored several options: most of small files less than 120KB can be effectively supported by either the in-house object storage and the in-house data storage; large files less than 30MB can be effectively supported in a hybrid approach. Based on the PAM computation metric with 95% of confidence interval, we obtained the estimated time: less than 1 day for 1M PAMs and less than 4.4 days for 100M PAMs under 1000 nodes cluster in Hadoop1 and Hadoop2.

REFERENCES