2012 IEEE 28th International Conference on Data Engineering Workshops

Stream As You Go: The Case for Incremental Data Access and Processing in the Cloud

Romeo Kienzler [1], Rémy Bruggmann [2], Anand Ranganathan [3], Nesime Tatbul [1]

[1]*Department of Computer Science, ETH Zurich, Switzerland*
romeok@student.ethz.ch, tatbul@inf.ethz.ch

[2]*Department of Biology, University of Bern, Switzerland*
remy.bruggmann@biology.unibe.ch

[3]*IBM T.J. Watson Research Center, NY, USA*
arangana@us.ibm.com

***Abstract*— Cloud infrastructures promise to provide high-performance and cost-effective solutions to large-scale data processing problems. In this paper, we identify a common class of data-intensive applications for which data transfer latency for uploading data into the cloud in advance of its processing may hinder the linear scalability advantage of the cloud. For such applications, we propose a "stream-as-you-go" approach for incrementally accessing and processing data based on a stream data management architecture. We describe our approach in the context of a DNA sequence analysis use case and compare it against the state of the art in MapReduce-based DNA sequence analysis and incremental MapReduce frameworks. We provide experimental results over an implementation of our approach based on the IBM InfoSphere Streams computing platform deployed on Amazon EC2, showing an order of magnitude improvement in total processing time over the state of the art.**

I. Introduction

Cloud infrastructures provide high-performance and cost-effective solutions to large-scale data processing problems. For computational tasks where linear scale-out is feasible through parallel processing over partitioned data, task completion time can be roughly halved by doubling the cluster size. Depending on the degree of scale-out as well as the pricing model of the cloud provider, it is possible to improve performance essentially at no additional cost, when cost of the additional cluster nodes can be amortized by a cutdown in the overall run time at each node.

This theory has a catch. It assumes that the complete input data is already stored in the cloud (has either been generated in the cloud or been transferred into the cloud in advance of processing) so that data transfer latencies can be ignored. However, in reality, there are many applications where this assumption does not hold. For example, in scientific applications such as DNA sequence analysis, large amounts of data are generated by special devices outside the cloud. While this data needs to be shipped into the cloud for scalable analysis, it is not a critical requirement to store the complete input and output data in the cloud on a long-term basis. Furthermore, in some use cases, data may involve sensitive information (e.g., use of DNA data in personalized medicine [1]), where it may even be undesirable to store the complete data in the cloud. On the other hand, transferring large data sets into the cloud can introduce significant latencies and may even become a bottleneck that hinders the scalability advantage of the cloud (not to mention the additional storage costs). Therefore, we believe that data transfer latency should be minimized unless in-advance/long-term storage of complete data sets is an explicit requirement of the application.

In this paper, we propose an incremental data access and processing approach for data-intensive cloud applications that can hide data transfer latencies while maintaining linear scalability. Similar in spirit to pipelined query evaluation in traditional database systems [2], data is accessed and processed in small increments, thereby propagating data chunks from one stage of the data analysis task to another as soon as they are available instead of waiting until the whole dataset becomes available. This way we can process data mostly in memory (hence, reduce time-consuming I/O to local disk and cloud storage, and avoid storage costs) as well as achieving pipelined parallelism (in addition to the existing partitioned parallelism), leading to a reduction in overall task completion time. In our approach, data is accessed in a "stream-as-you-go" fashion instead of in whole batches, making a stream-based data management architecture a good base for implementation.

We have designed our "stream-as-you-go" approach for a certain class of cloud applications characterized by the following. First of all, the data analysis algorithms involved in the application should be suitable for incremental processing (like only non-blocking operators can be evaluated in a pipelined fashion in traditional databases). Furthermore, workload intensity in terms of data size and computational complexity are also important factors. In particular, for applications that involve small datasets, data transfer latency would not play a major role in total processing time, and therefore, incremental processing would not make much of a difference in performance (the first column in Figure 1). On the other hand, for applications that involve simple computations, incremental processing would only be effective at very high uplink bandwidths, which may not be feasible in practice (the first row in Figure 1). However, complex computations over large datasets are pre-destined for incremental processing,

978-0-7695-4748-0/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDEW.2012.69

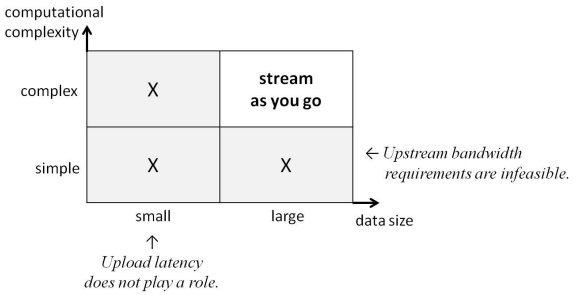

Fig. 1. Stream-as-you-go for data- and compute-intensive cloud applications.

since in this case, both the bandwidth requirements are feasible and the transfer latencies constitute a significant fraction of the total processing time (the "stream-as-you-go" quadrant in Figure 1). As also evidenced by our real-life use case described in the next section, this class of data- and computation-intensive workloads is very common in practice.

The work we describe in this paper has been inspired by a real use case from bioinformatics (DNA sequence analysis). In a recent publication, we have described this use case in detail together with an initial stream-based implementation and some preliminary results [3]. In this paper, we provide a more thorough investigation of the problem, covering the whole data analysis pipeline (which involves a challenging sorting stage) and comparing our approach to the state of the art, including an experimental study over larger datasets. More specifically, in this paper we use new DNA analysis software in our use case implementation (Bowtie [4] and SOAPsnp [5] instead of SHRiMP [6]), so that we can directly compare our implementation against the state of the art (Crossbow [7]). Furthermore, we present a parallel sorting algorithm based on data partitioning, distributed insertion sort, and red-black trees. Last but not least, we show experimental results that provide an order of magnitude improvement in total processing time over the state of the art.

The rest of this paper is organized as follows: In Section II, we provide a brief summary of our bioinformatics use case. Then in Section III, we give an overview of the state of the art. Section IV describes our stream-based, incremental processing solution. In Section V, we present experimental results over an implementation of our solution based on the IBM InfoSphere Streams computing platform [8] deployed on Amazon EC2 [9]. Finally, we conclude with a discussion of future work in Section VI.

II. Large-scale DNA Sequence Analysis

Determining the order of the nucleotide bases in DNA molecules and analyzing the resulting sequences have become very essential in biological research and applications. With the invention of the Next Generation Sequencing (NGS) methods in 2004 [10], higher amounts of genetic data can be read in much less time and at lower cost [3], which have led to the generation of very large datasets.

NGS is used to sequence DNA in an automated and high-throughput process. DNA molecules are fragmented into pieces of 100 to 800 *base pairs (bps)*, and digital versions of DNA fragments are generated. These fragments, called *reads*, originate from random positions of DNA molecules. Therefore, the reads must first be aligned into a sequence by mapping them back to a reference genome [11]. While doing this, polymorphisms between analyzed DNA and the reference genome (e.g., T replaced by C, or A replaced by G) can be observed. A polymorphism of a single bp is called *Single Nucleotide Polymorphism (SNP)* and is recognized as the main cause of human genetic variability [12].

Aligning NGS reads to genomes is computationally intensive. Li et al give an overview of algorithms and tools currently in use [11]. To align reads containing SNPs, probabilistic algorithms have to be used, since finding an exact match between reads and a given reference is not sufficient because of polymorphisms and sequencing errors. For example, on a small cluster consisting of 25 nodes with a total of 232 CPU compute cores and 800 GB main memory, a single genome alignment process can take up to 10 hours.

It is predicted that the NGS technology will eventually become available on a clinical level, making it a part of the standard healthcare process to check patients' SNPs before medical treatment (a.k.a., "personalized medicine"). Widespread use of this technology in research as well as in everyday applications will make the efficient analysis of the generated datasets an important requirement. Thus, this is an application area which can greatly benefit from cloud-based infrastructures. Although existing solutions show nearly linear scalability, they pose significant limitations in terms of data transfer latencies and cloud storage costs, which we explore in this paper.

III. State of the Art

In this section, we summarize the state of the art in large-scale DNA sequence analysis as well as incremental data processing in the cloud, focusing on recent works that are the closest to ours.

A. MapReduce-based DNA Sequence Analysis

We have found two MapReduce-based DNA sequence analysis software in the literature: Cloudburst [13] and Crossbow [7]. Both of these have been built on Hadoop to be deployed in the cloud and they make heavy use of the sort function provided by Hadoop.

Cloudburst. Cloudburst is a simple NGS read aligner like Bowtie [4]. It generates *k-mers* (i.e., small subsequences of DNA with k around 4), which are emitted by the Map function. This is done for reads as well as for the reference genome. Since Hadoop sorts and partitions data between Map and Reduce, the reducers work on a subset of data only containing reads and parts of the reference genome with at least k common characters. Therefore, the search space is drastically reduced.

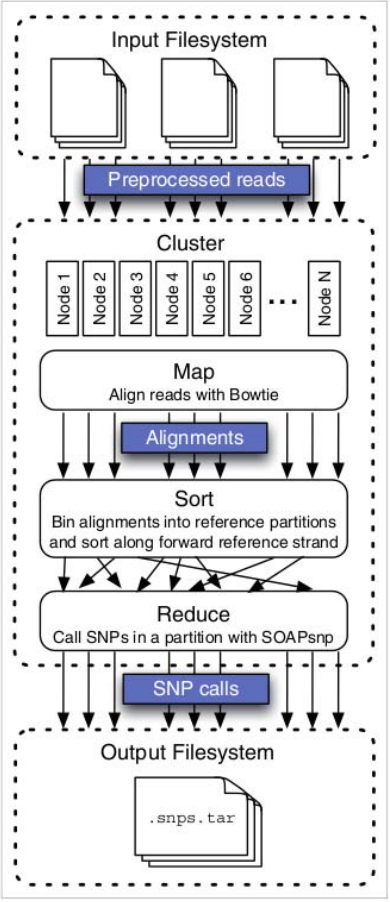

Fig. 2. Crossbow workflow [7].

Crossbow. Crossbow provides a Hadoop-based implementation of the SNP detection process using Bowtie [4] and SOAPsnp [5], where reads aligned by Bowtie are sorted and fed into SOAPsnp for *SNP calling* (another term for SNP detection). Figure 2 shows the complete Crossbow workflow. Bowtie is implemented as part of a Map function and SOAPsnp is implemented as part of a Reduce function. The sorting is handled by Hadoop's shuffling and sorting stage between Map and Reduce. The workflow also involves Preprocessing and Postprocessing steps. Preprocessing is done in order to convert from a four-lines-per-read format (FASTQ) [14] to an internal one-line-per-read format. The reason for this format conversion is the underlying HDFS file system [15]. HDFS automatically splits files into 64 MB (by default) blocks, and with a four-lines-per-read format, the split boundaries may possibly lie within reads and break them. It is important to note that the Preprocessing step is not parallelized. Finally, the Postprocessing step is included to concatenate the distributed results residing in HDFS to an output file residing on the local file system.

We experimentally compare our stream-based approach to the MapReduce-based approach of Crossbow in Section V.

B. Incremental MapReduce

Incremental data processing in the cloud is not a new idea. In particular, there have been many proposals recently for turning MapReduce from a batch processing framework into a more incremental one. We have found two categories of related work in this space:

Incremental Processing. Significant effort has been made recently to extend the traditional MapReduce paradigm to break the barrier between the Map and the Reduce phases, allowing reducers to run on partial results from mappers (e.g., [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27]). One of the key differences across these is the way the sorting phase is handled: Some apply it on partitions of the post-mapper or the pre-reducer stages, while others do so in chunks using a spill file. Like in our approach, these approaches are also capable of processing data in small increments and returning early results (for various purposes including delta processing or online aggregation). However, unlike in our approach, the complete input dataset has to be materialized in HDFS before the Map phase can start, which requires in-advance transfer of data into the cloud.

MapReduce over a Streaming Engine. A smaller number of works have applied the MapReduce programming model to streaming engines (e.g., [28], [29]). Like in our approach, these approaches use a stream-based data processing system underneath, and therefore, can both provide an incremental and a stream-as-you-go processing model. The fundamental difference to our work is their use of MapReduce as the programming model.

IV. The Stream-as-you-go Approach

As discussed in the previous section, MapReduce-based Crossbow provides a state-of-the-art, cloud-enabled software package for DNA sequence analysis with nearly linear scalability. Unfortunately, it does not avoid the data upload latency problem. Although the incremental processing extensions can help solve the barrier problem within the MapReduce framework, they also suffer from the upload latency. In this paper, we propose a "stream-as-you-go" approach that solves this problem by incremental data access and processing over a stream-based data management architecture.

In this section, we describe the "stream-as-you-go" approach in the context of our DNA sequence analysis use case and using the IBM InfoSphere Streams (or Streams for short) computing platform, noting that the approach is general enough to be applied to other similar use cases or using other stream processing engines.

Figure 3 shows our complete data processing pipeline consisting of Streams operators and Figure 4 illustrates the different data formats we are using for the streams flowing between these operators. While we do not use the MapReduce programming model, Map, Sort, and Reduce steps of the Crossbow workflow (Figure 2) correspond to Bowtie, PartitionByGenomePosition + In-memory Insertion Sort, and SOAPsnp operators in our Streams operator graph (Figure 3), respectively. The rest of this section will explain the major

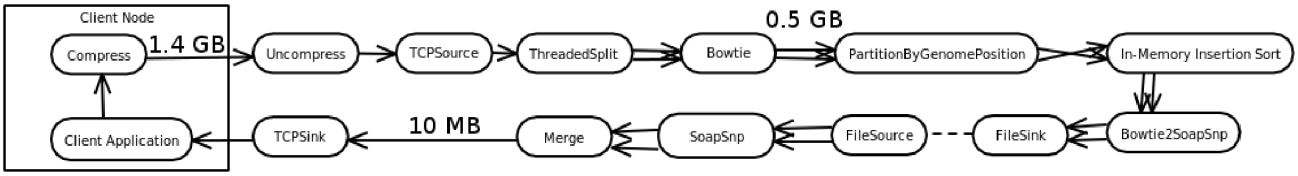

Fig. 3. Streams-based data processing pipeline: Client sends compressed read data to the cloud. After being uncompressed, data gets submitted to a Streams application which first splits it across the available cluster nodes. Split reads are then aligned in parallel using Bowtie. The output from each Bowtie instance is further partitioned by genome position to be then sorted using a distributed in-memory insertion sort algorithm. After some data conversion steps, the sorted data is fed into SOAPsnp for SNP calling. Results which are incrementally generated on each processing node in parallel are finally merged and sent back to the client over a TCP connection.

```
(1) Read file in FASTQ - Format
@SRR014475.1 :1:1:108:111 length=36
GAGTTTTACGTCGTCCTAAAACAGTACATAAAAATA
+SRR014475.1 :1:1:108:111 length=36
I3IIIII+I(%BH43%III7I(5IIIIIII*<&II+

(2) Bowtie output
SRR014475.3 :1:1:101:937 length=36 + gi|110640213|ref|NC_008253.1| 3393863 GAAGATCCGGTACAACAAAACCTGATGTAAATGGTA IIIIIIIIIIIIIIIIIAIIIIIIAII%I<IIII0G  0 7:T>C,27:G>T

(3) Converted soapSNP input
SRR014475.6885 TTTTCATTCTGACTGCAACGGGCACTTTGTCTCTTT IIIIIIIIIII6IIII%II7I33&"/%9,3&0$')4 33 a 36 + gi|110640213|ref|NC_008253.1| 3 3 A->24C30 A->26T30 G->34T30

(4) soapSNP output
gi|110640213|ref|NC_008253.1|   28480   C       G       0       A       0       0       11      C       0       0       0       16      1.00000       30.3125 0
```

Fig. 4. Different intermediate data formats: (1) FASTQ read file, single read with read quality; (2) Aligned read, 3393863 indicates position on the reference genome, where read GAAGATCCGGTACAACAAAACCTGATGTAAATGGTA has been found; (3) SOAPsnp needs a slightly different order of fields and a converted cigar string; (4) At position 28480, C has been mutated to G.

steps of our Streams-based data processing pipeline in more detail.

A. Client and Input Data Entry

A client application running on a node outside the cloud reads a file from its local hard disk and streams it into the cloud through a TCP connection. The UNIX command "netcat" could be used for this purpose. We have implemented a Java version of this command, extending it with a progress bar and some built-in performance measurement functions. Furthermore, in order to reduce the amount of data to be transferred, we have transparently added Compress and Uncompress steps to the pipeline using a TCP tunnel (Zebedee [30]). Zebedee uses the "zlib" compression library, providing a factor of 4 compression on NGS read data.

TCPSource, a built-in Streams operator, provides the single data entry point for input data by listening to a specific TCP port. Besides receiving data, conversion to an internal data format and type checking are handled as part of this stage.

B. Data Split

ThreadedSplit, a multi-threaded Streams operator, splits data into disjoint partitions based on the number of processing nodes in a given cluster. This provides that each queue to the succeeding operators on each node is served by its own thread, ensuring that all queues are always filled to their maximum capacity before the ThreadedSplit operator blocks. Blocking means not accepting any further tuples, causing the whole upstream pipeline to also block. This blocking determines the upstream bandwidth requirement.

C. Read Alignment using Bowtie

The Bowtie operator implements the read alignment stage based on a Bowtie process and standard UNIX pipes. More specifically, it consists of three threads: (i) one for accepting tuples from ThreadedSplit and writing them to the input pipe of the Bowtie process; (ii) another one for encapsulating the Bowtie UNIX process; and (iii) the last one for reading results from the pipe of Bowtie process and forwarding them to the succeeding operator.

D. Partitioning the Data by Genome Position

Horizontal partitioning divides a dataset into disjoint subsets based on a predicate. Each subset can then be streamed to a separate cluster node. In order to maintain load balance across these nodes, subsets should roughly be of equal size, which might be challenging in case of data skew.

In our use case, we partition the mapped reads generated by Bowtie based on their position on the reference genome. As can be seen in Figure 5, the mapped reads follow a uniform distribution across genome positions, leading to equal-sized subsets. Therefore, range partitioning can be used without facing the data skew problem. As a result, each subset contains reads from one particular region of the genome, providing continuous ranges to downstream operators.

In general, there are three main use cases for genome sequencing: whole genome re-sequencing [31], whole exome sequencing [32], and RNA-seq gene expression analysis [33]. Our use case falls under whole genome re-sequencing, where the complete genomic DNA of an organism is fragmented to be sequenced and aligned to a reference genome. While the

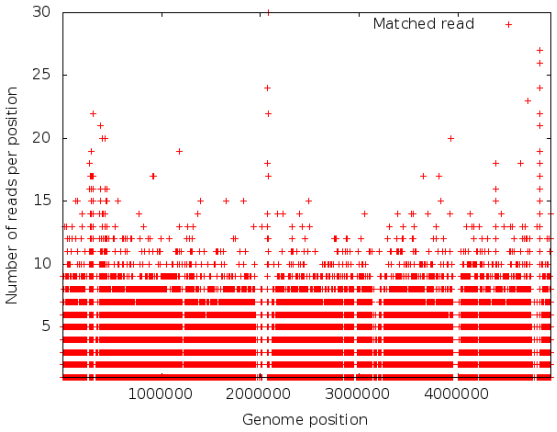

Fig. 5. Reads across genome positions are uniformly distributed.

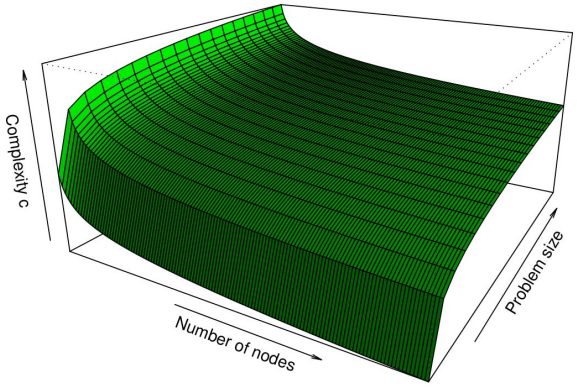

Fig. 6. Already at small cluster sizes (16 nodes), sorting complexity converges to a small number (requiring 10 memory accesses per insertion).

two other use cases sequence genes slightly differently, in all of these use cases, the read distribution among the reference genome is either known or can be derived in advance based on gene annotations. Therefore, data can be easily divided into even-size partitions based on genome positions.

E. In-memory Insertion Sort

After aligned read data is partitioned by genome positions, each partition needs to be sorted in parallel. To achieve this, we use a distributed in-memory insertion sort, where we keep a list in heap memory of each node and insert tuples at their correct positions in this list as they arrive. The list itself is organized as a red-black tree [34] so that insertions and re-balancing thereafter can be handled in logarithmic time.

We now analyze our sorting algorithm more closely in terms of its complexity and memory requirements. For red-black trees, insertion time is within $O(log(n))$ and tree re-balancing after insertions is within $O(log(1))$. As $O(log(1)) \in O(log(n))$, the sorted insertion of each tuple into a list of size n is $O(log(n))$. Considering that our parallel sort algorithm operates over data (of size n) partitioned across a cluster (of size s), insertion time of each tuple is $O(log(\frac{n}{s}))$. In Figure 6, we plot complexity for $n \in 0..5500000$ and $s \in 1..128$. Starting from 16 nodes, complexity converges to 10. This means that for each sorted insertion, 10 accesses to the heap memory segment are done. Every EC2 instance of type m1.large comes with about 7 GB free main memory. Assuming a cluster of 128 nodes, this would give us nearly 1 TB of available memory for sorting. The sorted dataset in our use case is about 1 GB; but even for much larger datasets, we do not consider main memory usage as a potential limitation for NGS read data processing.

F. SNP Detection using SOAPsnp

Next, Bowtie's sorted output partitions must be processed in parallel for SNP detection. However, since Bowtie's output format is not compatible with the input format of SOAPsnp (illustrated in Figure 4), a conversion step is needed (Bowtie2SOAPsnp in Figure 3). The conversion includes changing the positions of certain fields as well as expressing differently the information on how many and which letters are different between the reference and the read.

Since SOAPsnp is not capable of reading from standard UNIX pipes, input has to be provided as a file. For this, we have used extended versions of the built-in FileSink and FileSource operators in Streams.

G. Output Data Delivery

All output streams from the parallel SOAPsnp processes are merged at a single node to be sent back to the client over a TCP connection. At this stage, application-specific compression techniques (e.g., returning back only the positions of matched reads instead of the reads themselves) could be used to reduce the amount of output data transferred between the cloud and the client. In our use case, this was not a critical need, since the size of the output data was not that large.

V. Experiments

A. Experimental Setup

Our experiments compare three alternative solutions to cloud-based DNA sequence analysis, all of which use Bowtie for read alignment and SOAPsnp for SNP detection:

- a trivial (standalone) solution that is based on standard UNIX commands,
- the Hadoop-based Crossbow solution, and
- our stream-as-you-go solution that is based on the idea of incremental processing using a stream processing engine (IBM InfoSphere Streams).

In all of the experiments, we measure the total processing time in minutes, as the number of Amazon EC2 nodes (m1.large) is exponentially varied between 1 and 16. Each EC2 instance provides 7.5 GB of main memory and 2 virtual CPU cores equivalent to four 1.2 GHz Xeon processors.

As our dataset, we have used the "E. Coli Small Example" dataset provided at the Crossbow website [35]. The read file in this dataset is taken from an E. Coli experiment and

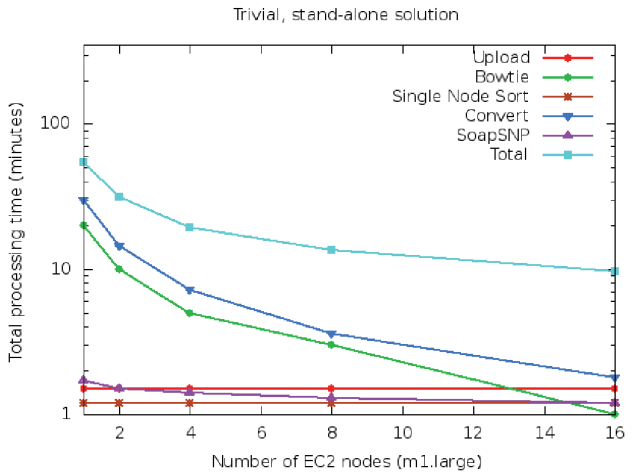

Fig. 7. The trivial solution.

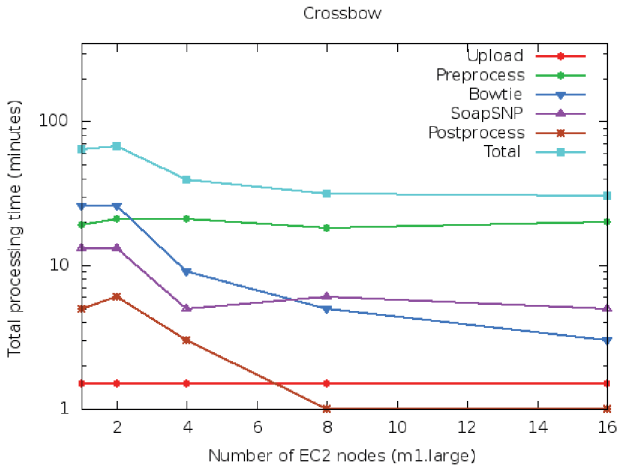

Fig. 8. Crossbow.

contains 8922730 reads with a total size of 1.4 GB. The process aligns these reads against the E. Coli reference genome (NC_008253.1) containing 5594158 base pairs with a total size of 5.4 MB.

B. Experimental Results

The Trivial Solution. This solution is implemented based on standard UNIX commands *rsync, scp, split, cat, sort* as well as command line tools *bowtie* and *soapsnp*. Assuming the input read file is located on a local system, *rsync* is used to copy the data to a single cloud node. *rsync* provides online data compression, which helps us reduce the data transfer time. After the data transfer, the read file is split into n partitions using the UNIX command *split*, where n indicates cluster size. After the split, data partitions are transferred to the n cluster nodes using *scp*. In this case, *rsync* is not used, since within the cluster, compression is not necessary due to sufficient bandwidth (more than 200 MBit/s). Using the partitioned read data, *bowtie* is invoked in parallel on each cluster node. Bowtie results are then transferred to a single cloud node to be merged using *cat* and sorted using *sort*. After the sort, partitions are re-built using *split* which are then transferred to the cluster nodes. Since Bowtie output and SOAPsnp input are not completely compatible, we have used a custom-written Java program in order to convert between these two data formats in a parallel fashion. Subsequently, *soapsnp* has been invoked in parallel as well. After transferring the results of SOAPsnp to a single cloud node, we again use the *cat* command to merge all results to a single file, which is finally transferred back to the local client machine using *rsync*.

Figure 7 shows the total processing time for different stages of the trivial implementation as the number of cluster nodes is increased. In all cases, the times for initial data upload, the single-node sorting stage, and the SOAPsnp stage stay constant, whereas the times for Bowtie and the data format conversion step get reduced with increasing number of nodes, converging towards the upload time. Overall, the total processing time asymptotically converges towards the upload time.

Crossbow. This solution represents state of the art in MapReduce-based DNA sequence analysis in the cloud. In Crossbow, Bowtie is implemented as a mapper and SOAPsnp is implemented as a reducer using Hadoop streaming [36] allowing the usage of UNIX binaries for data processing. Furthermore, Crossbow makes heavy use of Hadoop's sorting facility, where Bowtie simply emits tuples which get sorted by the Hadoop framework. Sorted partitions of Bowtie output are fed into SOAPsnp reducers. Finally, data is merged into a single file via a postprocessing step.

Figure 8 shows the total processing time for different stages of the Crossbow workflow as the number of cluster nodes is increased. Like in the trivial solution, there is a constant upload latency. In addition, there is a constant latency for the preprocessing step, which is higher. This non-parallelized preprocessing step dominates the total processing time as the cluster grows, constituting a significant bottleneck. Otherwise, processing times for the rest of the stages decrease as more nodes are added. It is interesting to observe that SOAPsnp in Crossbow needs significantly more time than that of the trivial solution, which we believe is because the reduce time in Hadoop also includes the times spent for hashing, partitioning, and sorting.

The Stream-as-you-go Solution. The main difference of this approach over the previous two is that data is transferred, accessed, and processed in small increments as it streams through the processing pipeline. We first discuss if the upstream bandwidth requirements of our experimental use case are feasible (i.e., are we in the "stream-as-you-go" quadrant of Figure 1?). Upstream bandwidth requirements can be estimated with the following function: $f(n,t) = \frac{n*t}{4}$, where n is the number of parallel processing nodes and t is the throughput of the slowest operator in the parallel pipeline. The factor of 4

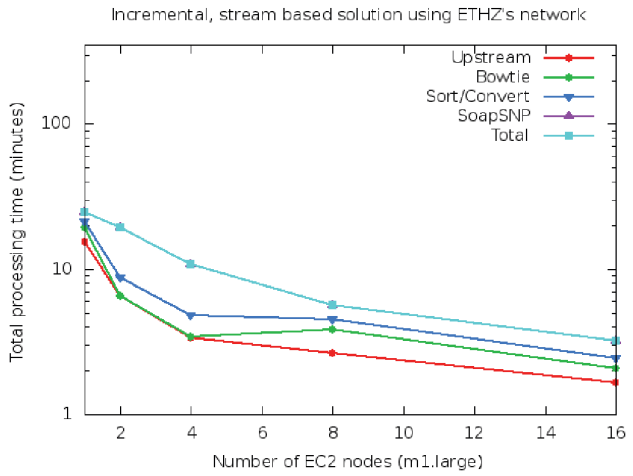

Fig. 9. The stream-as-you-go solution.

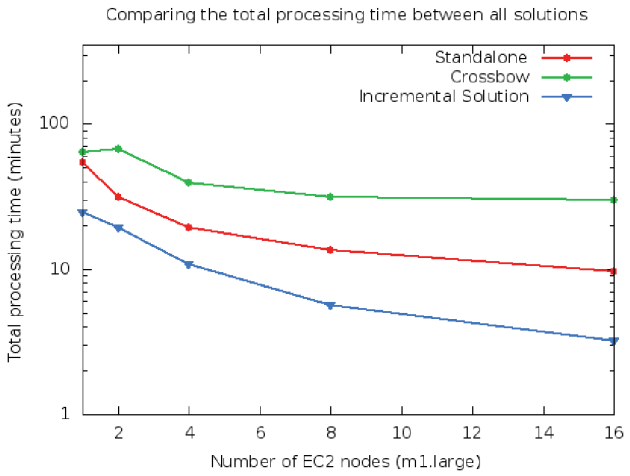

Fig. 10. Summary of experimental results.

in the formula is due to the data compression achieved using Zebedee (Section IV). In our use case, Bowtie came out to be the slowest operator with 9.3 MBit/s throughput measured on a single m1.large EC2 instance, whereas SOAPsnp has a throughput of 41 MBit/s. Therefore, for a 16-node cluster, the upstream bandwidth requirement can be estimated as 37.2 MBit/s (i.e., 9.3 MBit/s divided by the compression factor 4 and multiplied by the number of nodes 16). As a result, although our university network has more than 80 MBit/s uplink capacity, we have limited it to 37.2 MBit/s in our experiments.

Figure 9 shows the total processing time for different stages of our stream-as-you-go implementation as the number of cluster nodes is increased. However, the graphs of this experiment should be interpreted differently than the previous graphs, since, due to the incremental processing model, different stages of the processing pipeline overlap in time. More specifically, each graph shows the time difference between the upload start time and the termination time of the process that belong to that stage of the pipeline (i.e., the graphs from bottom to top are cumulative). We see that the total processing time for all stages of the processing pipeline scales linearly with the size of the cluster. Although we do not suffer from the data transfer latency any more, the lower bound for total processing time is still determined by the total data transfer time in the network.

Summary. Figure 10 summarizes the total processing time graphs of all three approaches on the same plot. As it can be clearly seen, our incremental stream-as-you-go approach achieves almost an order of magnitude reduction in processing time compared to the state of the art.

VI. Conclusions and Future Work

In this paper, we have proposed a stream-as-you-go approach to data- and compute-intensive cloud applications which can hide data transfer latencies while maintaining linear scalability. We have shown that re-implementing a MapReduce-based DNA sequence analysis application on a stream processing engine to provide incremental data processing capabilities can reduce the total processing time by almost an order of magnitude when deployed in a cloud-based environment. Furthermore, our incremental distributed sort implementation shows that even a stage of the data processing pipeline which require access to the whole data set can benefit from incrementality.

Future work includes adding fault tolerance support to our stream-based approach, an important feature that is also supported by the MapReduce-based alternatives. Moreover, we would like to further explore the stream-as-you-go idea in a way to provide a more generally applicable incremental processing framework than an application-specific one.

Acknowledgments. We would like to thank Tamer Özsu for his valuable input. This work has been supported in part by an IBM faculty award.

References

[1] G. H. Fernald, E. Capriotti, R. Daneshjou, K. J. Karczewski, and R. B. Altman, "Bioinformatics Challenges for Personalized Medicine," *Bioinformatics*, vol. 27, no. 13, 2011.

[2] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, vol. 25, no. 2, 1993.

[3] R. Kienzler, R. Bruggmann, A. Ranganathan, and N. Tatbul, "Large-scale DNA Sequence Analysis in the Cloud: A Stream-based Approach," in *Euro-Par 6th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'11)*, August 2011.

[4] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and Memory-efficient Alignment of Short DNA Sequences to the Human Genome," *Genome Biology*, vol. 10, no. 3, 2009.

[5] R. Li, Y. Li, X. Fang, H. Yang, J. Wang, K. Kristiansen, and J. Wang, "SNP Detection for Massively Parallel Whole-genome Resequencing," *Genome Research*, vol. 19, no. 6, 2009.

[6] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno, "SHRiMP: Accurate Mapping of Short Color-space Reads," *PLoS Computational Biology*, vol. 5, no. 5, 2009.

[7] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg, "Searching for SNPs with Cloud Computing," *Genome Biology*, vol. 10, no. 11, 2009.

[8] "IBM InfoSphere Streams," http://www.ibm.com/software/data/infosphere/streams/.

[9] "Amazon Elastic Compute Cloud," http://aws.amazon.com/ec2/.
[10] K. V. Voelkerding, S. A. Dames, and J. D. Durtschi, "Next Generation Sequencing: From Basic Research to Diagnostics," *Clinical Chemistry*, vol. 55, no. 4, 2009.
[11] H. Li and N. Homer, "A Survey of Sequence Alignment Algorithms for Next Generation Sequencing," *Briefings in Bioinformatics*, vol. 11, no. 5, 2010.
[12] F. S. Collins, M. Guyer, and A. Chakravarti, "Variations on a Theme: Cataloging Human DNA Sequence Variation," *Science*, vol. 278, no. 5343, 1997.
[13] M. C. Schatz, "CloudBurst: Highly Sensitive Read Mapping with MapReduce," *Bioinformatics*, vol. 25, no. 11, 2009.
[14] "FASTQ Format Specification," http://maq.sourceforge.net/fastq.shtml.
[15] "The Hadoop Distributed File System: Architecture and Design," http://hadoop.apache.org/common/docs/r0.18.3/hdfs_design.pdf.
[16] J.-H. Boese, A. Andrzejak, and M. Hoegqvist, "Beyond Online Aggregation: Parallel and Incremental Data Mining with Online Map-Reduce," in *WWW Workshop on Massive Data Analytics over the Cloud (MDAC'10)*, April 2010.
[17] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears, "Online Aggregation and Continuous Query Support in MapReduce," in *ACM SIGMOD Conference*, June 2010.
[18] J. Dittrich, J. A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)," *PVLDB*, vol. 3, no. 1, 2010.
[19] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling," in *International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*, September 2010.
[20] A. Verma, N. Zea, B. Cho, I. Gupta, and R. Campbell, "Breaking the MapReduce Stage Barrier," in *IEEE International Conference on Cluster Computing (CLUSTER'10)*, October 2010.
[21] M. Elteir, H. Lin, and W. chun Feng, "Enhancing MapReduce via Asynchronous Data Processing," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS'10)*, December 2010.
[22] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapReduce: A Distributed Computing Framework for Iterative Computation," in *IPDPS DataCloud Workshop*, May 2011.
[23] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang, "Nova: Continuous Pig/Hadoop Workflows," in *ACM SIGMOD Conference*, June 2011.
[24] N. Pansare, V. Borkar, C. Jermaine, and T. Condie, "Online Aggregation for Large MapReduce Jobs," *PVLDB*, vol. 4, no. 11, 2011.
[25] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A Platform for Scalable One-pass Analytics using MapReduce," in *ACM SIGMOD Conference*, June 2011.
[26] R. Lammel and D. Saile, "MapReduce with Deltas," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, July 2011.
[27] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini, "Incoop: MapReduce for Incremental Computations," in *ACM Symposium on Cloud Computing (SOCC'11)*, October 2011.
[28] N. Backman, K. Pattabiraman, and U. Cetintemel, "C-MR: A Continuous MapReduce Processing Model for Low-Latency Stream Processing on Multi-Core Architectures," Brown University, Tech. Rep. CS-10-01, 2010.
[29] D. Logothetis, C. Trezzo, K. Webb, and K. Yocum, "In-situ MapReduce for Log Processing," in *USENIX Annual Technical Conference*, June 2011.
[30] "Zebedee Secure Tunnel," http://sourceforge.net/projects/zebedee.
[31] "Whole Genome Sequencing," http://en.wikipedia.org/wiki/Full_genome_sequencing.
[32] "Whole Exome Sequencing," http://en.wikipedia.org/wiki/Exome_sequencing.
[33] "RNA-Seq," http://en.wikipedia.org/wiki/RNA-Seq.
[34] "Red-Black Trees," http://en.wikipedia.org/wiki/Red-blacktree.
[35] "Crossbow," http://bowtie-bio.sourceforge.net/crossbow/.
[36] "Hadoop Streaming," http://hadoop.apache.org/common/docs/r0.15.2/streaming.html.