20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013) IOP Publishing
Journal of Physics: Conference Series **513** (2014) 032080 doi:10.1088/1742-6596/513/3/032080

Running a typical ROOT HEP analysis on Hadoop MapReduce

S A Russo[1] , M Pinamonti[2] and M Cobal[3]

[1] CERN, IT Department, Genève 1211, CH
[2] ICTP, High Energy Physics Department, Trieste 34127, IT
[3] University of Udine, Department of Chemistry, Physics and Environment, Udine 33100, IT

E-mail: `stefano.alberto.russo@cern.ch`

Abstract. We evaluate how a typical ROOT High Energy Physics (HEP) analysis can be executed on Hadoop MapReduce. We take into account several aspects and we propose a method to perform the analysis in a completely transparent way to ROOT, the data and the user: the goal is to let ROOT run without any modifications and to store the data in its original format. The solutions which has been found to solve the encountered problems can be easily ported to any HEP code, and in general to any code working on binary data relying on independent sub-problems like HEP particle collision events. We tested the method by running an analysis code for the top quark pair production cross section measurement with the ATLAS experiment at the Large Hadron Collider in the CERN laboratory.

1. Introduction

In HEP as well as in many other fields, the classic and most common approach to distributed computing is to consider the storage and the computational resources as two independent entities (Figure 1). In this scenario applications which are data-intensive, like those taking place at the LHC [1] experiments [2], can easily saturate the communication channels.

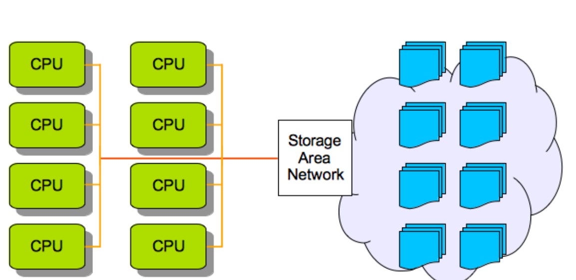

Figure 1. Classic computing model and communication channels.

Figure 2. Hadoop MapReduce computing model.

Hadoop MapReduce [3] is a widely used open source software framework designed to support data-intensive distributed applications, developed by the Apache Software Foundation and

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

inspired by Google's MapReduce [4] and the Google File System [5]. Hadoop MapReduce is made of two components: the *Hadoop Distributed File System (HDFS)* and the *MapReduce computational model and framework*, which also provides job scheduling and tracking facilities. Its main feature is to let the storage and computing resources to overlap, allowing each node which performs the analysis to keep a share of the data on its local storage (Figure 2). If the nodes can operate relatively independently on their share of data and if the aggregation of the partial results is a light task, this layout is particularly efficient and allows to avoid the risk of saturating the communication channels, since most of the I/O is local. This feature is referred to as *data locality*.

2. Related work

In the context of HEP analyses, several efforts have been spent over the past decades to optimise the computation, including data locality. On the Grid infrastructures, and in particular on the WLCG [2], data locality is implemented in macro zones to reduce data transfers across the geographically distributed Grid sites. HEP analyses are usually performed using ROOT [6], and to achieve data locality within a specific Grid site (at cluster level) is possible using PROOF [7]. This is a ROOT extension that allows to parallelise the analyses and that can optimise work distribution for data locality. It is anyway a HEP-specific solution and as of today a HEP-specific file system (XRootD) is required on the cluster to achieve data locality.

Hadoop MapReduce has already been partially tested for HEP analyses. In particular, J. Ekanayake, et al [8] tested the MapReduce approach using wrapper functions to execute the ROOT analysis code. The data was placed in its original ROOT binary format on a network file system so that it could be accessed without reading it via the HDFS (that, as will be exhaustively covered afterwards, is very problematic), therefore not providing data locality at all.

As of today, there is no known way to run binary code accessing a binary data set (like ROOT with its native data format) on Hadoop MapReduce while taking advantage of data locality, and the present paper provides a method to make it possible.

A first attempt of a performance comparison/benchmarking of ROOT on Hadoop MapReduce using the method proposed has ben carried out by S. Lehrack, et al. [9], for an evaluation vs. PROOF: on a cluster with a standard and unoptimised Hadoop installation without full data locality, it proved to be able of running ROOT only $\sim 20\%$ slower than PROOF.

3. HEP analyses and Hadoop MapReduce

The concept behind the Hadoop MapReduce model is a data-driven computation, which is started by specifying the data files to analyse (instead of specifying how many jobs to submit, or how many cores to reserve). Files stored on the HDFS are sliced in chunks, which correspond to the file system blocks (typical values are 64, 128 MB, or even 1 GB). Chunks are placed across the Hadoop cluster in a configured number of replicas (usually three) for data redundancy and workload distribution. The MapReduce computational model consists in two logical steeps, the *Map* and the *Reduce*. In the Map stage every chunk of a file is processed by a single Map task, which processes it by *records*. A record is the smallest piece of information that the MapReduce framework can process in one go. In the Reduce stage, the Reduce task collects the outputs from the Map tasks, aggregating them and producing the final output (Figure 3). This operation has to be light: Hadoop MapReduce does not even implement data locality for the Reduce tasks, as their I/O weight is indeed supposed to be negligible in comparison to the overall analysis.

In the HEP field, in several cases the analysis can be executed independently on large sets of data files (Map) and is finalised by aggregating a relative small number of summary objects, like numbers and histograms (Reduce). In this case, the computational and I/O weight of the aggregation operations compared to the analysis step are negligible, and the analysis perfectly fits in the MapReduce model. Moreover, very often the same data have to be analysed over and

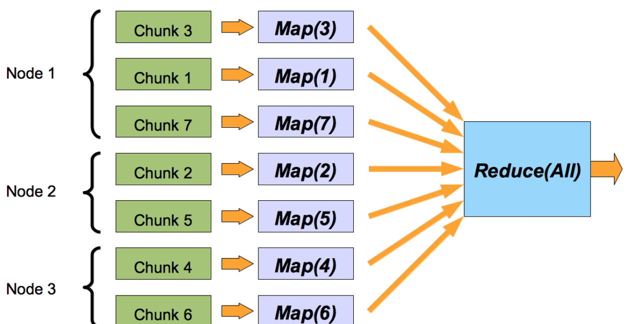

Figure 3. The Hadoop MapReduce computing schema. In a text-oriented analysis, a file is usually analysed by dividing each of its chunks in records on the new line character. Each Map task processes therefore one line at time of the assigned chunk, and the Reduce aggregates the partial results.

over again to finalise physics results. There is therefore a huge potential advantage by a data locality approach within this class of analyses. In case of an aggregation step not negligible in comparison to the analysis, a MapReduce approach may bringing only partial benefits and the Reduce stage should be treated differently than the the method proposed in this paper.

The natural way to port a HEP analysis to the Hadoop MapReduce model would be to match particle collision events to records in a one-to-one correspondence. Map tasks would therefore process a chunk of the data set analysing it event by event, and the Reduce would then collect every Map's partial result merging them. This approach has anyway two main drawbacks:

(i) Using standard Hadoop records to represent single events (i.e. using a CSV format) would lead to a huge amount of unnecessary I/O reads, as a typical HEP analysis requires only a few out of the many variables available. To select only the relevant ones without reading all the data, a column-based approach would be more suitable.

(ii) The software for HEP analyses rely nowadays on large and complex frameworks which are developed, maintained and used by thousands of persons over several years. The Map stage could indeed be potentially very complex, and porting a code developed for one of these frameworks could be very challenging and time consuming. Moreover, non-optimised MapReduce code code can easily lead to waste consistent amounts of CPU power [10].

Other approaches should therefore be considered, and in our evaluation we take as reference the ROOT framework, base block de-facto standard for higher level HEP analysis frameworks, which provides a column-based storage using branches.

ROOT relies on a binary data format, and dealing with binary data sets within the Hadoop MapReduce framework is not a simple task, as they cannot be handled in terms of chunks and records but only in terms of set of files. Binary files cannot indeed be trivially sliced in chunks on a size-basis, since the chunks would result in corrupted data[1]. Moreover, a standard (i.e. new line-based) record delimiter, not aware of what an event is and how to read it from the binary data, clearly does not work. A possible solution could be to use Hadoop Sequence Files. Sequence Files are made up by merging a set of files (which can be both plain text or binary), and their records correspond to the single files of the set. Map tasks can therefore access binary files as the records of a Sequence File, but this approach would require an additional step of conversion, and most importantly significant problems in terms of data locality would arise when

[1] Chunking a textual file on a size basis also results in "corrupted" data, as a record (line, or set of lines) can be truncated at any point. But in this case the Map tasks are still able to read the chunk, and they can ask Hadoop to give them the (few) missing bytes from the next chunk to reconstruct the corrupted record - this is how Hadoop MapReduce works. In case of a binary file, the Map tasks just can't read only a chunk of the original file, and therefore Hadoop's procedure to deal with truncated records would fail at the first step.

working with HEP data sets. Sequence Files are indeed not designed to work with big files[2], as the typical HEP data files which contain a great number of events. To preserve data locality events should be re-encoded as single binary files, and only then merged into a Sequence File.

Another important aspect to take into account is that ROOT binary data has to be accessed not by the Java Map tasks, but from ROOT. Running a third party code on Hadoop MapReduce without any modifications is possible via Hadoop Streaming, which allows the data to be processed using the standard input. This approach works fine with plain text (Hadoop was developed with text-based data formats in mind) but is not suitable for binary data. Even Hadoop Sequence Files would therefore not be suitable. Assuming to accept modifying the ROOT code, the data could be accessed using Java (Hadoop MapReduce's native API) or using C++ via Hadoop PIPES, but the problems outlined in the previous paragraph about how to store binary data on the HDFS would be still open.

4. Method proposed

The method proposed in this paper solves all these problems. The idea is to store the ROOT data in its original format on the HDFS, and make the Map tasks process not chunks, but *entire files*. Map tasks have then to perform no actions but spawning a ROOT instance, which takes over the analysis on the file that the Map task was originally in charge of processing. Analysing just one file would mean having no parallelisation, but executing the MapReduce job on a set of files will result in starting a Map task for every file of the set, running in parallel. The parallelizable unit has then been raised from the chunk to an entire file, and the parallelisation moved from a single file to a set of files. HEP data sets usually consist in several files grouped by some criteria, so that they perfectly fit in this schema. As an example, in the ATLAS experiment [11] computing model [12], data is stored by LHC run ($\sim 10^5 - 10^6$ events), luminosity blocks ($\sim 10^4$ events), and only then by ROOT files, each containing a set of $\sim 10^2 - 10^4$ events [13].

The implementation of this solution first requires to avoid chunking of binary data files. This can be achieved by setting the HDFS chunk size equal or greater than the file size, for each of them. Secondly, a custom record has to be defined to allow processing a chunk (which now corresponds to an entire file) in one go. The working schema is therefore:

$$\textit{one Map task} = \textit{one chunk} = \textit{one file} = \textit{one record.}$$

Map tasks must then let the analysis be performed by ROOT. For this purpose a Java Map task acting as a wrapper is used, which spawns a ROOT instance. Finally, ROOT has to access the files (chunks) to analyse from the HDFS. To allow *every* ROOT code to work out of the box, the only way is to access these files from a standard file system. The only existing methods for accessing HDFS files in this mode are to use the *FUSE-DFS module*, or to obtain the file in the Map's local sandbox before running the ROOT instance using the *Hadoop command line tools*. They both work and preserve data locality[3], but they have important drawbacks: FUSE introduces a non-negligible latency and a heavy overhead (reads are $\sim 20-30\%$ slower [14]), and the command line tools perform an unnecessary disk-to-disk copy if the file is accessed locally.

When looking for a way of improving this situation, it has to be taken into account that Hadoop's *Delayed Fair Scheduler* can schedule the Map tasks to achieve data locality for nearly 100% of the tasks, in almost all the use-cases [15]. Therefore, how to handle the case in which data has to be accessed over the network does not really matter, given the very limited impact, and both the Hadoop tools and the FUSE-DFS module are suitable[4].

[2] In a Sequence File there is no mapping between the contained files (records) and its chunks on the HDFS.
[3] If the access method is invoked from a node which has a copy of the data, then the access is done locally.
[4] If the case of accessing data over the network would become not negligible, there is a third access method to

The new data access method devised in case of data locality is to bypass the Hadoop framework, pointing ROOT to the file on the local file system corresponding to the HDFS file to be analysed. This is possible since the working hypotheses assure that every file to be analysed is not chunked[5], and therefore stored by HDFS into single files on the nodes' local file systems. The information about the availability of a local replica of the file to be analysed and its location on the node's local file system can be obtained from simple Hadoop tools. The entire procedure is schematised below.

(i) The MapReduce job is started on a data set of binary ROOT data (a set of files). Since each of these files is stored on the HDFS in only one chunk, every Map task will be in charge of analysing one entire file of the data set.

(ii) The job scheduler takes into account the location of the files on the HDFS and assigns the Map tasks to the computing resources maximising data locality.

(iii) Every Map task checks if on the assigned node there is a local replica available for the file to analyse, and if so it obtains its path; otherwise it sets the path to the location of the file on FUSE-DFS, or it uses Hadoop command line tools to copy it in the local sandbox and sets the path accordingly. Then it starts the ROOT instance with the path just set.

(iv) ROOT starts on the given file, accessing it from a standard file system in any case, and performs the analysis.

(v) The output is then collected by the Reduce task and merged to the final, aggregated result.

With this approach the ROOT code has to be made available to the Map tasks, by both storing it on the HDFS as a complete self-contained package or on a network file system accessible from every Hadoop node. If the size of the ROOT code becomes comparable with the size of the files to analyse, since it has to be accessed by every Map task the consequent data transfer cannot be neglected, and some kind of caching mechanism is required. A solution is for example to make the first Map task on every node to download the code in a local shared location, where it will be available for all the next Map tasks. Network file systems with strong caching policy would be even better.

Map tasks' ROOT output data can be transferred to the Reduce task by temporally storing it on the HDFS. The locations of these temporary files are just plain text lists of HDFS paths, which can be submitted to the Reduce task via the standard MapReduce framework. To perform the final aggregation of the partial results, the Reduce task will act again as a wrapper for a custom code (ROOT, a script, etc.). The latter will receive the HDFS paths of the temporary files via standard input, one per line, as the Map tasks end. The temporary files can be accessed by the Reduce task via one of the access methods discussed for the Map task, which in this case, since the negligible I/O weight by hypothesis, are all suitable.

By putting the pieces together, a MapReduce job acting as a wrapper for a generic user's ROOT code can be easily written. Users can use it to run their own analyses by specifying the input data set, the location of the Map code, the location of the Reduce code, and the location where to store the output. User's Map and Reduce code has to be prepared following just a few guidelines: the Map must receive as first argument the file to process, its output must follow a naming convention to be uploaded to the HDFS and to be accessed from the Reduce afterwards, which must read from the standard input, one per line, the locations of the files to merge in the final result.

be evaluated, the *ROOT HDFS module*. This module requires a custom ROOT build and to modify the analysis code, but if it would prove to be efficient could be an alternative to consider.

[5] The method can work even if the working hypothesis have not been respected, since in case of a chunked file it can switch back on the access method for non local files, delegating Hadoop how to correctly access it.

5. A real case: a top quark analysis in the ATLAS experiment

Object	Order of magnitude	Type	On Hadoop MapReduce
Event	1	ROOT data structure	unknown (binary)
File	$10^2 - 10^4$	ROOT file, set of events	chunk, record
Lum. block	10^4	Dir. (set of Files)	Dir.
Run	$10^5 - 10^6$	Dir. (set of Lum. blocks)	Dir.
Data set	$10^5 - 10^9$	Dir. (set of Runs)	Dir. (input data set)

Table 1. Mapping between logical units of the ATLAS data acquisition model, their order of magnitude, their data types and the corresponding objects on Hadoop MapReduce.

The method presented above for running ROOT on Hadoop with a MapReduce approach has been tested on a real case, the top quark pair production cross section measurement analysis performed by the ATLAS Collaboration. This analysis, implemented using a code named *ICToP2*, follows a "cut-and-count" workflow where every event undergoes a series of selection criteria and in the end is accepted or not. The cross section is then obtained by comparing the number of selected events with the luminosity, the efficiency in the selection of signal events, and the expected background events. The details of the analysis can be found in [16].

The data used for the test case was related to the top quark pair final state with a high momentum isolated electron and has been taken with all the subsystems of the ATLAS detector in fully operational mode, with the LHC producing proton-proton collisions corresponding to a centre of mass energy of 7 TeV with stable beams condition during the 2011 run up to August.

Given the huge amount of data produced by the detector, ATLAS relies on a lightweight format for final user specific analyses, named D3PD [17]. This format, which consists in flat ROOT n-tuples, is the most common format used for physics analyses, since it keeps only the interesting events and information for a particular analysis (so reducing noticeably the size).

The data set on which the ICToP2 code operates is therefore a set of D3PD n-tuples of "filtered" (interesting) events for the top quark pair production cross section analysis. The data taking conditions described above resulted in a data set corresponding to an integrated luminosity of 2.05 fb^{-1}, with a size of 338.6 GB when considering only electron channel D3PDs. According to the ATLAS data acquisition model, this data set is composed of 8830 files organised in a three-level hierarchy. The hypotheses assumed in Section 3 are therefore fully confirmed. The mapping between the objects involved in the ATLAS data acquisition model, their order of magnitude, their data types, and the corresponding objects on Hadoop MapReduce are listed in Table 1.

The Hadoop cluster used for the test case is a ten node cluster with eight CPUs per node, with no RAID between the disks and a HDFS replication factor of two. The ICToP2 code has been compiled without any modifications and the data set copied straightforward from its original location at CERN Tier-0, thanks to a maximum size of the data set files of $\sim$ 48 MB which fits in a default HDFS block size of 64 MB.

The analysis has been performed using a Java MapReduce wrapper for the ICToP2 code as previously described, and worked as expected, leading to a total of 8830 Map tasks (one per file) and an average of 883 data files analysed per node. The aggregation of the partial results in the Reduce task was done by a simple Python script, which was in charge of summing the number of relevant events observed in every file of the data set by the Map tasks. This sum is computed as the Map tasks progressively end and so partial results are made available. Table 2 shows the status report from the Hadoop job tracker while running the analysis. Data locality was observed to be achieved in all the Map tasks, which confirms the expected behaviour.

The network overhead[6] by the Hadoop MapReduce infrastructure for handling this MapReduce job has been measured to be $\sim$ 1.17 GB. The ICToP2 code size is $\sim$ 12 MB and, as already discussed, each node performing the analysis has to transfer it only once. Since the entire Hadoop test cluster have been used for the tests, the consequent total data transfer has been of $\sim$ 0.12 GB. Given that data locality has been achieved for all the Map tasks, and that the data transferred from the Map to the Reduce tasks has been considered as negligible and indirectly included in the network overhead, these values lead to a total data transfer of $\sim$ 1.29 GB across the cluster for analysing the 338.6 GB data set. This significative result is summarised in Table 3. For completeness, the physics results obtained have been compared with the official ones [16], which were obtained using the same ICToP2 code, and agreement has been found.

Kind	% Complete	Num Tasks	Pending	Running	Complete
map	48.33%	8830	4462	100	4268
reduce	16.07%	1	0	1	0

Table 2. The Hadoop Jobtracker status report while running the analysis. The number of 100 concurrent Map tasks is because the cluster was (arbitrary) configured to run ten Map tasks per node.

	Data transfers
Code	0.12 GB
Overhead	1.17 GB
Input data set	0
Total:	1.29 GB

Table 3. Data transfers needed to perform the test case analysis on the 338.6 GB data set across the ten node cluster, using Hadoop MapReduce with the proposed method.

6. Conclusions

We evaluated how a typical ROOT High Energy Physics (HEP) analysis can be executed on Hadoop MapReduce, and we presented a method which allows to run ROOT without any modifications and to store the data in its original format. A user can therefore easily run its own ROOT code, as thanks to the method presented no specific knowledge about Hadoop MapReduce is required, and changes in the computing model are minimal. The benefits brought by Hadoop MapReduce are fully exploited. They include data locality, which removes network bottlenecks for I/O bound applications, a robust distributed file system, a fault tolerant job scheduler and an excellent scalability by design. The method has been tested on a real case, an analysis to measure the top quark pair production cross section with the ATLAS experiment, and worked as expected: once stored, the data set has been analysed by being entirely read from the local drives of the cluster nodes, completely avoiding data transfers over the network.

The method has already been successfully applied in another context [9], and on a cluster with a standard and unoptimised Hadoop installation without full data locality it proved to be able of running ROOT only $\sim$ 20% slower than PROOF.

Possible future work includes evaluating the scalability of the method depending on the size of the input and its comparison with other existing technologies, using optimised test setups to obtain accurate results.

References

[1] L. R. Evans, The Large Hadron Collider, Proceedings of the 1995 Particle Accelerator Conference, Dallas, TX, Volume 1, Pages 40-44, 1-5 May 1995.

[2] I. Bird , et al., LHC Computing Grid Technical Design Report, CERN-LHCC-2005-024, LCG-TDR-001, ISBN 92-9083-253-3, 20 June 2005 (also available on http://cdsweb.cern.ch/record/840543).

[6] Defined as the total network traffic on the cluster's private network while performing the analysis, in conditions of data locality achieved for all the Map tasks and no transfer of the code. It therefore represents only Hadoop MapReduce internal communications. Measured by first storing the code on the local nodes and then monitoring the network usage while performing the analysis (with data locality for all the Map tasks).

[3] Apache Hadoop MapReduce, http://hadoop.apache.org [viewed 18/10/2013].
[4] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Communications of the ACM - 50th anniversary issue: 1958 - 2008, Volume 51 Issue 1, January 2008, Pages 107-113, ACM New York, NY.
[5] S. Ghemawat, et al, The Google File System, ACM SIGOPS Operating Systems Review - SOSP '03, Volume 37 Issue 5, December 2003.
[6] I. Antcheva, et al, ROOT A C++ framework for petabyte data storage, statistical analysis and visualization, Computer Physics Communications, Volume 182 Issue 6, June 2011, Pages 1384-1385.
[7] G. Ganis, J. Iwaszkiewicz and F. Rademakers, Data Analysis with PROOF, PoS(ACAT08) 007, Volume 1, Proceedings of XII Advanced Computing and Analysis Techniques in Physics Research, Erice, IT, 3-7 November 2008.
[8] J. Ekanayake, S. Pallickara and G.Fox, Mapreduce for data intensive scientific analyses, IEEE Fourth International Conference on eScience, 7-12 December 2008, Indianapolis, IN.
[9] S. Lehrack, J. Ebke and G. Duckeck, Evaluation of Apache Hadoop for parallel data analysis with ROOT, to be published in Journal of Physics: Conference Series, Proceedings of Computing in High Energy and Nuclear Physics 2013, Amsterdam, NL, 14 - 18 October 2013.
[10] Z. Baranowski, et al, Sequential Data access with Oracle and Hadoop: a performance comparison, to be published in Journal of Physics: Conference Series, Proceedings of Computing in High Energy and Nuclear Physics 2013, Amsterdam, NL, 14 - 18 October 2013.
[11] The ATLAS Collaboration, et al., The ATLAS Experiment at the CERN Large Hadron Collider, Journal of Instrumentation, Volume 3, Issue 08, 14 August 2008.
[12] G. Duckek (ed.), et al., ATLAS Computing Technical Design Report, CERN-LHCC-2005-002, ISBN 92-9083-250-9, 20 June 2005 (also available on http://cdsweb.cern.ch/record/837738).
[13] D. Costanzo, et al., Metadata for ATLAS, ATL-GEN-PUB-2007-01, 05 April 2007.
[14] Mounting HDFS, http://wiki.apache.org/hadoop/MountableHDFS [viewed 18/10/2013].
[15] Matei Zaharia, et al., Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling, Proceedings of the EuroSys '10 5th European conference on Computer systems, Pages 265-278, ACM New York, NY.
[16] The ATLAS Collaboration, et al., Measurement of the top quark pair cross-section with ATLAS in pp collisions at $\sqrt{s} = 7$ TeV in the single-lepton channel using b-tagging, ATLAS-CONF-2011-035, 21 March 2011 (available on http://cdsweb.cern.ch/record/1337785).
[17] W. Bhimji, et al., The ATLAS ROOT-based data formats: recent improvements and performance measurements, Journal of Physics: Conference Series, Volume 396, Proceedings of Computing in High Energy and Nuclear Physics 2012, New York, NY, 21 - 25 May 2012 (also available as ATL-SOFT-PROC-2012-020, 14 May 2012, on http://cdsweb.cern.ch/record/1448601).