CedCom: A High-Performance Architecture for Big Data Applications

Tanguy Raynaud, Rafiqul Haque, and Hassan Aït-kaci
Laboratoire d'InfoRmatique en Image
et Systèmes d'information (LIRIS)
Claude Bernard University Lyon 1
Lyon, France 69100
Email:{tanguy.raynaud-gallonet, rafiqul.haque, hassan.ait-kaci}@univ-lyon1.fr

***Abstract*—Distributed architecture is widely used for storing and processing Big Data. Operations on Big Data need first, locating the required data blocks and then, reading them. Data can be located in different types of memories in particular, *cache memory*, *main memory*, and *secondary memory*. Reading data from secondary memory to process Big Data jobs is not an ideal approach especially for high performance applications because, accessing data in secondary devices can be slow for processors. In addition, fetching data from main memory is time consuming due to limited I/O bandwidth. These system level issues are barriers for optimizing performance of Big Data applications. Simply put, for optimizing the application performance, it is not sufficient to have efficient algorithms only, an efficient architecture is needed to provide faster data access by the processors. The need for such an architecture has been documented in the literature, however, the state of the art is still missing an efficient architecture. This paper develops a promising architecture which caches data in main memory. It essentially transforms a main memory into a *attraction memory* which enables high-speed data access. Also, it enables automatic migration of data blocks and computations across the nodes contained in the clusters. It offers an exchange protocol for fast transfer of data blocks between the different physical nodes and speeds up job processing. The proposed architecture combines the power of Cache-Only Memory Architecture (COMA) and the structural principle of Hadoop.**

I. Introduction

Data is becoming bigger everyday. Today, its size ranges from Gigabytes (GBs) to Petabytes (PBs). It has been predicted by many such as, Zikopoulos *et. al* that the data size will reach to *Yottabytes* in future [1]. [1]The rapid increase of data has given the rise to several problems related to computation (*e.g.* processing a job), predominantly, *efficient access to mammoth size datasets*. *Time to access data* is a critical attribute since it is one of the determinators of the efficiency in terms of processing jobs on Big Data. In most of the literature related to Big Data, importance has been given to optimization algorithms such as *query processing algorithms*. These have heavily been investigated, and improved, for querying Big Data efficiently. System level issues such as high-speed data access and also fetching them to CPU caches in case of *cache miss* are important too; yet, surprisingly overlooked in existing technologies. A cache miss refers to an unsuccessful attempt by a CPU to read or write a data block in its cache.

[1]Yottabyte: http://en.wikipedia.org/wiki/Yottabyte

In distributed environment, efficient management of memory is of critical importance. In this, different time variants are vital performance metrics and as such determine the performance of applications. For instance, *access time* to a desired data block is an important metric and determine the performance of query processing. It is composed of *time to locate data blocks* and *time to load data into main memory from the hard disk*. These two in addition to others such as *query execution time* determine the efficiency of Big Data applications. Thus, the delay to locate data blocks or loading data can degrade the performance significantly.

Furthermore, accessing data stored in secondary devices is time-consuming. In fact, even the disks with the fastest RPM (rotation per minute) consume a significant amount of time to read write data from to secondary devices. Therefore, it is highly unlikely that a high-performance application would be able to perform jobs efficiently using disk-based system architectures. The architecture such as Hadoop [2], GFS [3] are disk-based, and thus do not guarantee high performance.

Instead of disk-based, conventional *in-memory* based architecture could be an option to optimize processing time. However, that is not a suitable option for the applications which process real-time queries on Big Data sets. The key reason is limited *I/O bandwidth*. In addition, the size of main memory available on the market is an obvious limitation. Although, the technologies such as Virtual Shared Memory [4] are available to deal with the size problem, communication overhead and complexity of cache coherence can not be ignored. Taking all these facts into account, we conclude that *an efficient system architecture for supporting the Big Data applications is yet to be defined.*

Our main interest in this research paper lies at the system layer. Our objective is to *develop a high-performance architecture called 'CedCoM' (CEDAR Cache Only Memory), which will enable efficient data processing by making data access faster for the processors by increasing the cache hit ratio.* The proposed architecture combines the power of Cache-Only Memory Architecture (COMA) and the structural principle of Hadoop. It aims to solve two major problems. First, avoiding systematic access to secondary storage when the processors are to execute a job. This is possible by finding an effective and efficient way to provide access all necessary data blocks to a machine. Second, enabling dynamic migration of data blocks between nodes. It is worth noting that, the ultimate goal of the architecture is to support high-performance query processing

on Big Data.

The rest of the paper is organized as follows. In Section II we describe the motivation of this research. A high-level overview of the CedCom architecture is presented in Section III. Section IV describes the development of the architecture. Experiments and results are presented in Section V. Section VI presents the related work. The final section provides a conclusion and the outlook of this research.

II. Motivation

There are many efficient query processing algorithms. In the area of distributed databases, efficient algorithms play a key role to process queries efficiently. However, reducing time to access physical data is not within the scope of such algorithms. Also, they are not concerned with *time required to fetch data* from secondary storage or a main memory to CPU cache. However, as mentioned in Section I, the temporal attributes heavily influence the overall performance of processing time of a job in particular, querying Big Data. Since query processing applications are typically I/O-bound and processors are faster than memory access, guaranteeing high-speed access to data is of paramount importance. If *access time* can be reduced, the applications will be able to process jobs on Big Data with high-performance. This will essentially solve the problem related to processing real-time queries efficiently on Big Data.

III. The Design of CedCoM Architecture

This section presents our CedCoM architecture which combines the features of COMA and Hadoop. Its novelty is that, it adopts the structure of so called "Flat COMA (COMA-F)" [5]. We adopt COMA-F because unlike conventional hierarchical COMA (*e.g.*, the HORN DDM [**?**]), in this the data nodes are interconnected and can potentially communicate with each other directly using a point-to-point network [6]. Additionally, like Hadoop, our architecture is capable of distributing data across the nodes which comprise clusters. However, unlike Hadoop, data is stored in attraction memories. These memories are physically located in main memory and logically implemented as cache memory. In CedCoM, there is no notion of main memory because it is transformed into attraction memory. Figure 1 shows the CedCom architecture.

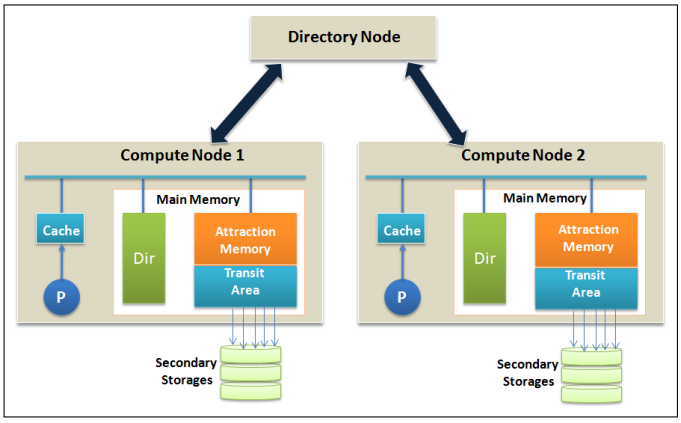

Fig. 1. The Architecture of CedCom

The architecture comprises *directory node* and *compute nodes* (Note that, we use the terms compute node and data node interchangeably in this paper). A compute node consists of *processors*, *conventional cache memory*, and *main memory*. Figure 1 shows that the larger portion of the main memory of the compute nodes is transformed into *attraction memory*. The size of attraction memory is predefined by users. In addition, the remaining portion is called *transit area*. Additionally, the CedCom architecture allocates a certain amount of memory for storing a *directory* (*Dir*, shown in the above figure). It is worth noting that the CedCom architecture provides flexible space management for transit areas and directories.

The data blocks that are needed to execute a job by a compute node are not necessarily stored on the same node. Since the CedCoM architecture relies on COMA, data blocks do not need a particular *home node*. Any compute node can contain any data block. If a block is required by the node but stored in another node, the target node contacts source node and then the block is migrated from the source to target node. However, in case of large size data blocks, the proposed architecture enables transferring computations to the target node, in lieu of data blocks. This is a substantial feature of the proposed architecture. The key idea is to reduce the *delay* which can be the result of high traffic which are created due to an exhaustive number of interactions between the nodes. Transferring computation to the data hosts would avoid communication between the nodes and eventually would ensure faster job processing.

The architecture contains a *local directory* which indexes the data blocks stored in the local attraction memory. The attraction memory contains the data blocks. The transit area contains the *least recently used (LRU)* data blocks that are transferred from the attraction memory. Also, it temporarily stores the data blocks that are about to be migrated or copied to other compute nodes.

In addition to main memory, the CedCom architecture enables storing data blocks onto *secondary storage* (*e.g.*, Hard Disk) on special conditions. For instance, there is no space in the attraction memory of any of the nodes which belong to a cluster. The CedCom architecture allocates secondary storage automatically when needed. However, the architecture would load the data blocks automatically in attraction memories or transit areas from the secondary device as soon as the required spaces are found available.

The *directory node* is essentially a metadata server which provides information such as locations and state of the data blocks. Like Hadoop, the compute nodes are strongly linked with the directory nodes whereas the connection between compute nodes are kept open for a temporary session. The closing of the session should terminate all connections between nodes.

IV. Development of CedCoM Architecture

This section describes the development of CedCoM architecture. We used C++ programming language for the development. It is a suitable language for developing high-performance computing (HPC) applications with a real control of the memory used for computation. The solution developed in this paper is platform-neutral, which means, it is independent

of any specific operating system. The following sub-sections describe the development of CedCom architecture.

We describe the construction of CedCom in three steps. In the first step, we describe the the building blocks of CedCom. Then, we describe how we develop the core components: *the compute node* and *the directory node*. Finally, we explain the details of advanced operations.

A. Basic Concepts

The following subsections provide a detail of the basic building clocks of CedCom architecture.

1) Data Structure: We start the development of CedCom by defining the structure of data blocks. Since the data blocks will frequently be migrated within clusters, it is important to define the size of data blocks.

To create a generic and portable structure, CedCom partitions data according to its underlying file system. The size of the data blocks are predefined. Each block has a unique identifier which is a reference key to the blocks and used in storing blocks in compute nodes. The unique key of a data block must not be altered. It must be the same for all processing nodes storing that data block. This identifier is assigned by the directory node when a node registers a new incoming data. A node can register blocks based on its storage capability and thus, can obtain a range of identifiers.

Data is stored in the 'data' part of the nodes. It is the smallest unit of a large file. The size of data stored in one slot may vary, but cannot exceed the threshold (maximum size) which was predefined. To identify the source of data, a field name called 'filename' is added. It enables a node to find different parts of a file by using their 'filenames'. A directory located in the directory node enables finding the data blocks by their filename.

2) Network communications: The CedCom architecture is specially designed for Big Data applications and therefore, it handles a large quantity of data that is distributed within and across clusters. The data can be stored and migrated over the network if needed. Thus, an efficient communication protocol is required to enable faster communication between the nodes.

Since C++ provides a basic network implementation, we decided to use a high-level tool provided by the boost suite called *'Boost Asio'*. This tool was selected because it provides pre-built functions that are easy to use in the network layer. Initially, we selected *'OPEN MPI'*, however, since it facilitates only parallelization of the task processing but not data transferring, eventually the tool was not used.

Boost Asio provides a function to open *synchronous* or *asynchronous* connections between servers. It simplifies network utilisation by overloading the basic C++ functions. Combining 'Boost Asio' with basic functions of C++, CedCom architecture provides functions to open *sockets* between clients and servers. In addition, it provides functions to write or read binary streams to the socket. The main advantage of using 'Boost Asio' is the portability.

3) Connection Management: The CedCom architecture handles distributed data that are supplied by a client. Currently, the architecture does not have any native client for loading data. It relies on external client more specifically, the application client. The connection between client and compute nodes is established in two steps. First, the client requires the meta-information about the data nodes included in the cluster. To do so, it contacts the directory node and fetches information such as the IP address, the TCP port, the storage space available, and the utilization ratio of memory. Based on the collected information, the client produces a data distribution plan. Then, it opens connections with different compute nodes and sends data blocks to those nodes.

The size of the data packets is critical. Typically, it depends on the network bandwidth. The CedCom architecture enables receiving data with small blocks or a large blocks (e.g., 500 MB or more). For the small size blocks, the data received by the compute nodes are aggregated immediately in the registration queue. For the large size blocks, the nodes use a specific *input buffer* which temporarily stores the incoming data blocks. In order to confirm the correct order of arriving data blocks, a unique number is assigned with each block which can be deemed as an *identifier* or a *tag number*. Upon receiving a block, the compute node verifies the tag number to ensure the correctness. If the verification is not successful, the compute node sends an 'error' message which contains the block's tag number, to the client. The client uses it as a reference number and can find easily the invalid block and resend it to the node.

The compute node automatically detects when the client leaves or closes connection and therefore closes the socket, clears the input buffer, and starts waiting for the next data blocks. The CedCom architecture does not use any 'close' protocol. The client opens a connection with a node, sends data blocks, and then disconnects itself.

The compute node uses a 'registration queue' to add data blocks received by the node. The queue temporarily stores the blocks before adding them in the attraction memory of the nodes. The new blocks are stored in registration queue, because no unique identifier was assigned when blocks were created. The directory node assign unique identifiers to the blocks, which are then added to the attraction memory.

4) Replication Management: Like Hadoop, the CedCom architecture uses the notion of data replication. However, the architecture would not allow creating replicas in attraction memories. Since data can be migrated dynamically from node 'X' to node 'Y', we argue that replication would not be necessary. However, in special cases such as two or more nodes need the same data blocks for processing tasks, CedCom creates replicas in attraction memories dynamically.

Additionally, CedCom enables creating (maximum) one replica in secondary storage of nodes. In this case, it differs the replication policy of Hadoop. The notion of replication has been adopted in this architecture to make the system fault tolerant. For instance, if a compute node does not send a signal (we call it *heartbeat*) during a predefined timeout period, the directory node automatically considers it an *inactive node*, and issues a command to a free the node to load the replicas of the inactive node into its attraction memory. The compute node is allowed to store only one replica of a data block. When restoration of data blocks are completed, the directory node automatically updates its own global index. If an update

occurs, the block replicas must be updated as well. This update employs the principle of the *write-update* protocol.

5) Serialization: Serialization is a critical concept in CedCom. The reasons are two-fold. Firstly, it enables saving data in a known format. Secondly, it enables sending data easily from one node to another by merely sending binary streams in the sockets.

To simplify the serialization process, we use a technology called *'Boost Serialization'*. It enables transforming a C++ class in a binary format or parse a binary stream in a specified class. More importantly, it enables choosing the variables of a class to be serialized or not. This 'Boost Serialization' enables serializing a part of the *std* container, like *vector* or *map*. Another advantage of this library is that it recursively serializes the classes. This means, if a variable is an instance of another serializable class, it will easily be serialized like a *primitive type* in the source code.

6) Configuration: The CedCom architecture is complex. Therefore, initializing parameters in a command line interface is non-trivial task. To simplify, a parameter file has been integrated with CedCom. The parameter file contains all the information required at the initialization phase. It will greatly help users (e.g., *administrators*) to initialize the compute nodes and the directory node. The parameters stored in those files are given in the following:

Parameters for compute Nodes

- Data storage path
- IP address of the directory node
- Communication port of the directory node
- Heartbeat port of the directory node
- Transfer port use to establish a connection with an other compute node
- Port the client has to use to establish a connection
- Heartbeat interval delay
- Replication port

Parameters for Directory-node:

- Data storage path
- Communication port
- Heartbeat port
- Port used for setting up a connection

7) Client Connection: The users will use the client applications to distribute data to compute nodes. Since no client component has been implemented in this paper, the users need to use application's client for data distribution. The application client contacts the directory node to obtain information required to establish connections with compute nodes. The directory node has an asynchronous server that is used to communicate with the client on a predefined port. This client receives various information from the directory node. Below a list of information is given.

- IP Address and port to open the communication
- Free space storage
- Size of the block
- Memory utilization ratio

These help the client to partition files into blocks of specific size expected by the compute nodes. When the partition is completed, the data blocks are sent to compute nodes.

B. Core Components

The subsections below provides a detail how the core components were implemented.

1) The compute nodes: The compute nodes store data and execute jobs. Figure 2 shows the structure of a compute node.

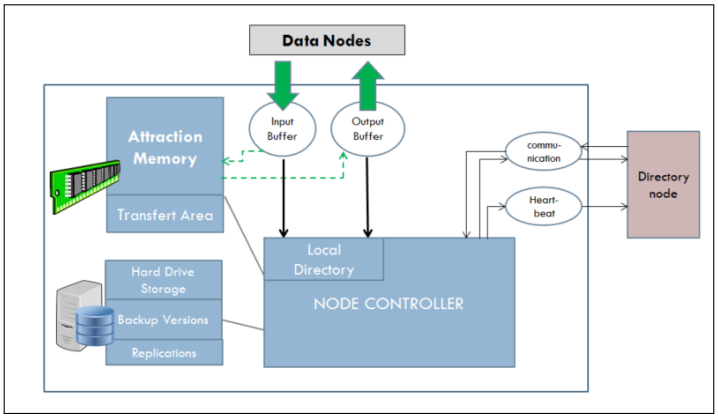

Fig. 2. Structure of a compute node

The CedCom architecture organizes memory in an unique style. In the sub-section below, we describe the memory organization of the CedCom architecture.

A. The memory organization: Each node has local memory. In this architecture, data is stored exclusively in main memory (RAM) to improve the performance by reducing time needed to access data. The CedCom architecture comprises three types of memories specifically, *Attraction Memory*, *Transit Area*, and *Secondary Storage* as a single unit. A new data block will first, attempt to be stored in attraction memory. However, as we mentioned in Section III it will be stored in secondary storage if and only if the attraction memories of all the nodes in the cluster are filled with data blocks. The transit area will store the *least recently used* and *ready to be migrated* blocks. This area can be considered as the transit point for data migration. In fact this is the reason we named it *Transit Area*. The implementation of these storage in CedTMart is discussed in the following subsections:

I. Attraction Memory: The main purpose of attraction memory is to store the data blocks that have higher *cache hit* ratio. Cache hit refers to a successful attempt made by CPU to read or write a data block in the CPU cache. These data blocks are required for processing tasks such as a *query*. Attraction memory enables faster data access by processors.

a. Associative-cache: In order to implement the attraction memory, we used a technique called *multi-way associative cache*. This technique divides memory slots into subsets and uses the *hash keys* to store and find the data. The term 'way' is used to define the maximum number of items each set can contain. This technique is also called *n-way associative cache* which can be written as follows, 2^N slots of the memory into

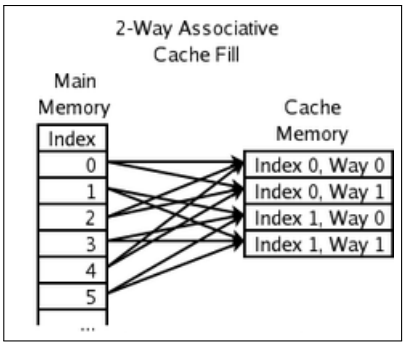

Fig. 3. 2-way associative cache

x subsets, each one having 2^n slots, with $n < N$ and $x = \frac{2^N}{2^n}$ Figure 3 depicts an example of a two-way associative cache.

Since it enables fast data access and better hit ratio than a usual cache [7], we choose this technique. By using this approach, the complexity of finding a block in $O(n)$ is reduced to $O(log(n))$ where n is the number of elements stored in memory. It is worth noting that, a standard *list structure* was a potential candidate technology too but it is not a feasible option because it results a systematic shift of all elements in the set ($O(n)$), which requires a lock during execution. The associative cache uses a technology called *Map* that is more efficient. It organizes the data table comprises two columns: *key* and *value*. This enables faster data access. The CedCom architecture promotes the operational complexity for finding or inserting an element: $O(log(n))$ and for deleting (without lock): $O(n)$.

II. Transit Area: The size of attraction memory is limited. Thus, it is possible that an excessive number of blocks with the same hash key stored in the same node will cause a buffer overflow. Since migrating data blocks of an attraction memory to another may generate high-traffic, we developed an area called transit area. When attraction memory of a node is full, CedCom moves its least recently used data blocks to transfer area and make enough space for new data to be loaded onto the memory.

The key purpose of implementing transit area is to increase the availability of attraction memory for the data blocks with high cache hit ratio.

We implemented transit area with a single map which contains the blocks and uses their identifiers as *keys*. This enables users to find, delete, and add a new block with a complexity of $O(log(n))$. A specific class: `secondary_memory.h` gives pre-built optimized functions to access, update, and delete different blocks that are available in the *Map*.

III. Secondary Storage: The CedCom architecture involves secondary memory for storing replicas. The key purpose is to make CedCom fault tolerant. The secondary devices of compute nodes store replicas of data blocks in *neighbor-safe* style. In this approach, a replica of a data block stored in the attraction memory of a node (say A) will be stored in the secondary memory of the neighbor node (say B). This will prevent the data loss if node A crashes.

B. Local Directory: The local directory has been introduced in CedCom architecture for indexing the locations and status of data blocks contained in compute nodes. The locations are: *Attraction Memory*, *Transit Area*, and *Secondary Storage*. The local directory reduces time to read, find, and, write data blocks. Without indexation, these operations are time consuming because the system must scan various locations sequentially. The directory provides exact locations of data blocks and hence, accelerate the speed to finding data.

C. Versioning System: A version management system has been developed to avoid data loss. The CedCom architecture enables a compute node to create its *snapshot* and storing them in secondary storage. The snapshots of compute nodes are called *versions*. Each snapshot corresponds to a specific restoration point. However, storing a snapshot can be expensive in terms of storage space. The storage can be overflowed. In order to avoid overflow of local storage, we limit the number versions of nodes to 3. In case the storage is full, the oldest version will be removed automatically. However, the users can migrate the old versions to another server.

D. Operating principle: The operating principle of a compute node is complex, because a number of parallel operations will be carried out by the node at runtime, especially maintaining the coherence of data is non-trivial. We implemented various *operational components* to parallelise the operations that a compute node performs at runtime. They are described as follows:

I. ComLoader: This is an important component of the CedCom architecture. It performs primary operations such as starting or stopping components of a compute node. It processes instructions received from the directory node. Also, it performs *save* operation to store the snapshots of recent state of compute nodes. For instance, if a compute node is *shutdown* normally, the *save* on *exit* option pops up and then the ComLoader saves the contents and node information as a file in the secondary storage and reloads it when the node is restarted. For each save operation, the ComLoader creates a new version of snapshot.

II. Block Manager: To keep data blocks contained in the compute nodes consistent, we implemented a *Block Manager*. The main tasks of this component are, moving and adding data blocks in memories. In CedCom, when the data blocks arrive, they are added in input queue which stores the blocks temporarily. The block manager calculates the hash keys of incoming blocks and stores them in the appropriate sets of attraction memories. If the sets are full, the block manager automatically moves the oldest blocks into *Transit Area*. This frees attraction memory to load new data blocks. Finally, the block manager updates the local directory to maintain a coherence between the block identifiers and their locations.

III. Transit Area Manager: The *Transit Area Manager* is implemented to manage the blocks stored in the *Transit Area*. The main purpose of developing this component is to manage transit areas of a cluster efficiently and to increase the availability of the attraction memories for data blocks with high cache hit ratio. This component manages transit storage and finds hosts to migrate the data blocks.

In order to migrate blocks, the transit area manager contacts the directory node of a cluster to find a compute node that can host data blocks located in a transit area. If the directory node finds host then it returns information to the requester node.

Then, the transit area manager transfers data blocks to the host node. The block stays in transit areas until a host node is found.

IV. Heartbeat Manager: This component is responsible for managing one-way messages from the compute node to directory node. It triggers messages at a regular interval and sends to directory node. This signal is an indication for directory node that compute nodes in a cluster is alive. The message body contains various information such as *availability of free spaces* in the nodes and *the memory usage ratio*. Note that, the manager will not initiate connection with the directory node on the Heartbeat port. This task is done at initialization phase.

V. Replication manager: Two main functionalities of the replication manager are: (i) it enables storing replicas of data blocks in secondary storage of a node and (ii) it enables transferring a replica of a data block to the attraction memory of a compute node. The manager enables to access data in replication area of secondary storage. No other data is allowed in this restricted area. A strict access control guarantees data consistency. In CedCom architecture, each node has a specific port to create an asynchronous server specifically for replication.

The replications are managed by directory node and therefore, this particular component is deployed on this node. We developed a replication directory in our architecture. This directory contains information such as data blocks that need to be replicated. The manager contacts the host compute node and requests to create replicas of data blocks. The compute node creates replicas and stores them in *replication area*. It is worth noting that only one replica of a data block is allowed in attraction memory.

VI. Backup Manager: A compute node may shut down. Since CedCom is an attraction memory based architecture, the information of the node (such as the information related to *Attraction Memory*, *Transit Area*, and *Local Directory*) and data blocks must be persisted. CedCom stores snapshot of compute nodes in their secondary storage devices. The node information and data blocks are stored at a regular interval to save the most recent changes in attraction memory. Also, the directory and the transit areas are stored.

Note that the information is saved only once. The Backup Manager updates the information periodically rather than writing the same information repeatedly on secondary storage. Since writing node information and data blocks are time consuming, the 'Block Manager' launch multiple threads to parallelize this operation.

VII. Block Registration Manager: The 'Block Registration Manager' is responsible for registering the inbound data blocks from clients. It contacts the directory node and assigns a unique identifier to new blocks. The CedCom architecture enables generating several threads to perform this task concurrently.

2) The Directory Node: The directory node is essentially a shared metadata server in our architecture. In CedCom, each compute node can act as a local metadata server because, each node has a local directory that supplies a limited amount of meta-information. However, the information is purely about the data blocks stored in the local node. Thus, a global metadata server has been developed to deal with some specific situations. For instance, the local metadata servers usually do not have sufficient information about data blocks stored in other nodes of a cluster. Conventional COMA does not have any directory node, rather, it relies on local metadata. In this case CedCom architecture adopts Hadoop's architectural principle *a global metadata server that receives requests coming from the compute nodes.*

Big Data applications can generate billions of blocks distributed to thousands of nodes. Thus, storing global metadata in compute nodes may not be a viable option. The reasons are two-fold: it will consume the attraction memories or transit areas of compute nodes and maintaining the updates of distributed metadata server can be a painstaking job and computationally expensive. The former is a well-understood problem whereas the latter is communication network related problem. For each write, all metadata servers have to be updated, which will promote an exhaustive number of messages exchanged between the nodes. This will cause high traffic in the network. Consequently, the job processing time will be increased. In some case, processing time can be increased dramatically.

Considering these issues, the CedCom architecture introduces a specific node to take the role of storing and managing a *Big Directory* which stores the location of all the blocks distributed across the compute nodes. We called this node *'Directory Node'*. This node is aware of both locations of different blocks and active blocks which is a similar approach to Hadoop [2] but simpler. Additionally, this node also manages the block replication system. Figure 4 shows the directory nodes.

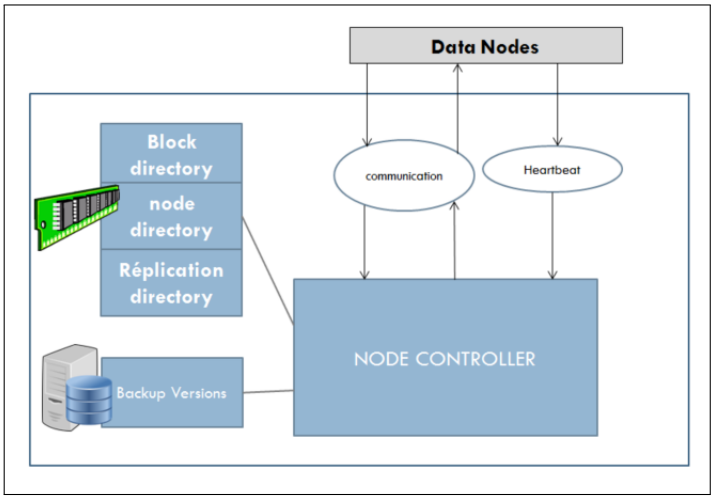

Fig. 4. Structure of the directory node

Below, we describe different aspects of the directory node.

A. Server Operations: In CedCom, the computes nodes are connected with the directory node using peer-to-peer technology. The connection between the directory node and the compute nodes are permanent. The connection has been classified into *Communication Connection* and *Heartbeat Connection*. Note that, the directory node communicate with compute nodes asynchronously. Therefore, the node is able to receive and

manage hundreds of requests from many nodes simultaneously, using multi-threading techniques.

I. Communication Connecter: The compute nodes contact directory node for various purposes such as, to know about the location of a data block, to request to find available space for data blocks *etc.*. In CedCom, a *Communication Connecter* has been implemented to establish connections between the compute and directory nodes. A connection is established by following a sequence of steps: first, the directory node checks the *IP address* of a compute node in its directory (note that the IP addresses are stored in the directory node through a registration process that is done when a compute node is connected with directory node for the first time). If an IP address is not found in the directory, the directory node starts registration process. It creates a new unique node identifier and assigns it to the corresponding compute node which opens the new connection. If an error occurs during this phase, the compute node is immediately turned off.

Once a connection is successfully established, the directory node performs the following tasks:

- turn off the node (with or without performing save operation)
- duplicate data blocks
- restore data blocks in the attraction memory or transit area
- create restoration point of compute nodes
- restore a compute node by reading the node information snapshot which was taken while switching off the compute node

On the other hand, the compute node uses the socket to carry out the following actions:

- request for information about a block location
- request for the IP address of a specific node
- register a new block, or a group of blocks
- disconnect the node from the directory node
- sending an update signal to notify the move of a block in the local attraction memory

We implemented a protocol to handle requests from compute nodes to the directory node. The request messages must comply the protocol. The requests messages are categorised into *command requests* and *data requests*. These are briefly explained in the following.

a. Command Requests: The commands requests are concerned with requesting information or operations. The size of the body of command request messages is small. It does not exceed fifty characters. The format used for this type of requests is as follows, *a four characters header* for writing operations and the remaining part of the header is used for instructions. *Regex* is used to identify and extract information from the request messages.

b. Data Requests: The *data requests* are concerned with sending data blocks from one node to another. Unfortunately, the *Regex* technology has been proved slow for data requests because, the size of message body is too large. Therefore, the analysis of messages consumes time. It is worth noting that upon receiving every request message is analysed. We cleverly avoid analysing the body of data request messages by separating it from the header. The CedCom architecture allows the analysis of meassge header only. Currently, we use $||-||$ separator.

c. Communication Protocol: The CedCom architecture has its own low-level communication protocol. So far, we have implemented the following protocols:

- The compute nodes open the connection with directory node by using the asynchronous connection provided by the 'boost.asio' API (Application Programming Interface).
- The directory node registers the connection by assigning a unique node identifier corresponding to the IP address of compute nodes
- The directory node sends the registration notification to the compute node:
  - If successful: the identifier of the node is created
    $$RSTR : N < (node\_identifier) >;$$
  - If fails: an error message pops up
    $$RSTR : ERR : (erro\_message);$$
- The compute node will use this connection to make request for information about the location of a block:
  - Request:
    $$BLCK : BLOCK < (block\_identifier) >;$$
  - Response:
    $$BLCK : BLOCK < (block\_identifier) >, NODE < (node\_identifier) >, ADDRESS < (ip\_address : port) >;$$
- The compute node will also use this connection, to obtain a group of unique identifiers for the new blocks received from an external client:
  - Request:
    $$BLRG : RANGE < (number) >;$$
    (Range is used to determinate the number of identifiers requested.)
  - Response :
    $$BLRG : MIN < (min_identifier) >, MAX < (max_identifier) >;$$
- The directory node can use this connection to turn off a compute node with or without previously saved contents:
  $$STOP : SAVE < true/false >;$$

II. Heartbeat Connection: In our architecture, a separate connection type is implemented to send heartbeats from the computing nodes to the directory node. The Heartbeats are essentially signals to the directory node, which indicates whether a compute node is *alive* or *dead*. As mentioned earlier, the compute nodes send *heartbeat* signal at a regular interval.

To register a heartbeat connection, the nodes must already be registered with the directory node. The registration will result in opening a socket between compute and directory nodes. By using this socket, the compute nodes send signals to the directory node. If registration of a compute node is unsuccessful, it will be turned off and an error message will pop up.

As mentioned in the beginning of this section, in addition to heartbeat signal, the compute nodes send other information to the directory node. The information are listed below:

- Free Space of the nodes

- Memory utilisation ratio of the nodes

The directory node generates and persists metadata and the time-stamp of compute nodes. For instance, it tracks the latest time-stamps of heartbeats and stores it. An arbitrary time is defined as *timeout* which allows the directory node to determine whether or not a compute node offline. This default timeout is currently set to *5 seconds*.

a. Heartbeat Protocol: We have developed a protocol for the communication between compute and directory nodes regarding heartbeats. This protocol uses the heartbeat ports. It is briefly explained below:

- The compute nodes open connection with the directory node by using asynchronous connection provided by the 'boost.asio' API.
- The directory node registers the *Heartbeat* by bundling the IP address of compute nodes with their unique identifier
- The compute nodes send an initialization message to the directory node to provide information about the port which the client should use to establish a connection:
$$INIT : C\_PORT < (Port\_number) >;$$
- The compute node sends heartbeat message at a regular interval along with the information of its memory utilisation
$$BEAT : FREE\_SPACE < (size\_in\_Mo) > , RATIO < (percentage\_number);$$
- The directory node receives heartbeats, stores information, and updates the latest heartbeat time-stamps
- The directory node triggers an error message upon occurrence on an error:
$$ERRO : (error\_message);$$
- If required, the directory node uses the *Heartbeat* connection to turn off a node (with or without saving its contents):
$$STOP : SAVE < true/false >;$$

III. Operating Principle: The operating components and principles of directory node are explained below.

- Like compute nodes, directory node has a component called *ComLoader* that starts and stops it other components. In addition, it saves the contents of the directory node on secondary storage if it is turned off.
- The directory node has a component for checking whether or not the computing nodes are alive. It checks the status by comparing the latest time-stamp with the predefined timeout. If a node is not alive, this component places it in the *restoration node queue*. Since the nodes should be restored efficiently, the CedCom architecture initiates a new thread to speed-up the restoration process.
- The CedCom architecture has a component for managing block reload. If a compute node stops responding, this component will restore all of its blocks in memory. The reloading process is composed of four steps. First, the component identifies the data blocks that were stored in the failed compute node. Then, it identifies the compute nodes that have the copies of the blocks. In the third step, it sends a request to these nodes to transfer the blocks to the failed compute node. Finally, it updates the directory.

C. Advanced Operations

This section presents a list of advanced operations which can be performed by the components that we have developed in this paper. The CedCom architecture enables performing these operations during transferring and creating data blocks. They are briefly explained in the subsections below.

1) Block Transfer: This section summarises the steps of transferring blocks from the source to target compute node.

The steps are listed below:

- A compute nodes uses communication connecter to establish connection with the directory node and request the locations of required blocks,
$$BLCK : BLOCK < (block\_identifier) >;$$
- The directory node uses the block directory to locate requested blocks, and then sends a response to the compute node
$$BLCK : BLOCK < (block\_identifier) >, NODE < (node\_identifier) >, ADDRESS < (ip\_address : port) >;$$
- The compute node opens a temporary connection with the compute node that is hosting required data blocks, and requests for the blocks by using the *block identifiers* as reference.
$$BLCK : BLOCK < (block\_identifier) >;$$
- Alternatively, a compute node can make a request for processing job to the compute node which have the copy of data block. However, in this case the host compute node must be available for performing the required operations. If yes, then the requester and host compute node process the transmission of *computations e.g., executable jar files* instead of transferring data packets.
$$BLCK : TRANS\_COMP\_REQ;$$
- If a block is transferred or just copied, the source node sends the block by using a standard data package format and specifies in the header of the package.
$$DATA : BLOCK < (block\_identifier) > , COPY < (true/false) > || - ||(data)$$
- Upon arrival, the requester compute node stores the data blocks in its attraction memory and then, sends a notification to the directory node to update the latest location of the blocks.
$$TRFR : BLOCK(block\_identifier);$$
- The directory node updates its block directory and sends an *invalidation* request to the compute nodes which have the old copies of the data blocks. Upon receiving the request, the nodes invalidate the copies. This avoid the data inconsistency and hence, the *dirty read*.
$$RMOV : BLOCK(block\_identifier);$$

2) Block Creation: This section summarises different steps which are carried out when a client sends data blocks to the compute nodes.

- The client establishes a connection with the directory node which sends information regarding a connection with the compute nodes
  $B\_SIZE < (size) >, [\{"address" : "(ip\_address)", "port" : "(port)", free\_space" : "(size)", "ratio" : "(percentage)"\}]$
- The client establishes connection with the compute nodes contained in the clusters. It partitions input files into different packages. Each package can contain maximum data size that is suggested by the directory node. The partition may result in a single package or many smaller packages, which depend on the size of input files and the predefined threshold of the package.
  - If data is sent as a single part:
    $$DATA : N < (filename) > || - ||(data)$$
  - If data is divided into many smaller packages:
    $$PDTA : N < (filename) >, P < (part\_number) > (LAST)|| - ||(data)$$
- To ensure the data integrity and flow, the compute nodes send an ACK signal to the client for each message is received by them.
  $$PDTA : ACK < (part\_number) >;$$
- The data blocks received by the compute nodes must be registered with the directory node. By using a system of range, a node can obtain successive identifiers. Additionally, the input filenames are transferred to associate the filenames with the data blocks in the internal directory of compute nodes. This essentially indicates that the node will always group them by filename during block transfer.
  $$BLRG : RANGE < (nb\_block) >, FILE < (filename) >;$$
- Upon receiving registration request from the compute nodes, the directory node provides block identifiers to the requesting compute nodes.
  $$BLRG : MIN < (identifier) >, MAX < (identifier) >;$$

V. Experiment

In this section, we present the results of our experiments which shows the robustness of CedCom in loading Big Linked Data [8]. Also, we demonstrate its performance in terms of *data loading time*, *data transfer rate*, and *data packet loss* during transmission.

A. Experiment Setting

We conducted our experiments on Grid5000 [9] which is a grid computing platform for high performance computing (HPC). The platform has different sites or resource centers. We performed experiments on two different sites *Lyon* and *Rennes*. [2,3] In Grid5000, upon receiving a job request, an automatic resource scheduler reserves resources based on availability. We conducted our experiments on three different clusters: *Taurus*, *Hercule*, and, *Parapide* that consist of seven, eleven, and and twelve nodes respectively. Note that the nodes are physical. The hardware specification of the nodes can be found in Lyon and Rennes resource sites.

Three datasets were used in our experiments: 19.775 GB, 200 GB, and 102 GB which contain approximately 153 Million, 1.3 Billion, and 900 million Notation 3 (N3) triples respectively. [4]

Experiment Steps

While running, CedCom must go through the following steps:

Step 1: The data distribution client opens a connection with directory node (DN). Once the connection is established, the following steps are performed.

1) The client contacts the directory node on a specified address and port. The client message contains the amount of data it wants to distribute to compute nodes.
2) The directory node receives the message, scan the metadata table to find available space, and then sends a pre-formatted string to the client containing the information as follows: maximum size of one data block and for each node, free space, IP address and port.

The IP and PORT are used to establish the data connection between the client and compute nodes. Client's distributor sends data to compute nodes according to free_space (in MB) and ratio (that are shown in the above).

Then, the client stops the connection with directory node

Step - 2: The client opens connection with compute nodes and distribute data blocks. The nodes perform the following steps to distribute data blocks.

1) The client opens a socket with each compute node in cluster using request header.
2) Then the client sends data to the compute nodes. A loop is defined to distribute data iteratively until all the blocks are sent to the compute nodes. Two important steps of data distribution are as follows:
   - The client partitions input files into many parts according to the maximum size suggested by the directory node. The client sends data packets begin with a header "DATA ;" (8 bytes).
   - The compute node receives data, stores it, and sends a message begins with "DATAACK;"(8 bytes) to the client.
3) Once the client closes socket connection, the compute nodes know immediately that the entire file has been transmitted.

B. Results and Discussion

Table I presents the results of experiments we performed with CedCom in this paper.

[2]Grid5000 Lyon: https://www.grid5000.fr/mediawiki/index.php/Lyon:Home,

[3]Grid5000 Rennes:https://www.grid5000.fr/mediawiki/index.php/Rennes:Home

[4]Notation 3: http://www.w3.org/TeamSubmission/n3/

TABLE I. RESULTS OF EXPERIMENTS WITH CEDCOM

Experiment No.	Size of the Dataset	Size of the cluster	Status	Total Packet Transfer	Transfer Rate (MB/sec)	Loading Time (hh:mm:ss)	Packet Loss	Packet Loss Rate
1	19.7	7 nodes	Successful	2706373	29.67	00:11:08	451	0.00
2	200	11 nodes	Failed	X	X	X	X	X
3	102	12 nodes	Successful	14344267	9.23	03:08:56	0	0.00

The experiments show that CedCom loaded datasets successfully on attraction memory. However, the second experiment was not successful. The Table I shows that CedCom performed better in the first experiment. The 19.2 GB dataset was loaded in 11 minutes 8 seconds. We found CedCom faster than HDFS loader which we experimented in [10]).

However, in the third experiment, the performance of CedCom in terms of data transfer rate was three times worst than the first one. Yet, the data packet loss was '0' which is relatively better than the first experiment. We performed the third experiment in *Grid5000 Rennes* site. Based on our observation, we identified two factors that might be responsible for the performance degradation in the last experiment. First, erratic bandwidth which we observed during experiment. Second, the nodes used for the final experiment have lesser processing power than the ones used for the first experiment.

The second experiment was failed due shortage of of space. To be more specific, the size of the dataset was greater than the sum of main memories of all nodes. In such cases CedCom stops data distribution.

It is worth noting that, CedCom retransmits the lost packets. For instance, in the first experiment 451 packets were lost during data transmission. However, they were retransmitted successfully. This implies that eventually, there was no packet loss.

VI. RELATED WORK

In this section we study the works related to system architecture and memory management of distributed, parallel, and high performance computing.

Shared memory architecture is a widely used one for handling high-performance applications. Until now, several approaches have been proposed. These are discussed in the following:

- *Virtual shared Memory (VSM)*: This term is commonly used to describe the systems which provide a shared address space by using hardware assistance [4]. It is implemented on top of this shared address space.
- *Shared Virtual Memory (SVM)*: Unlike VSM, SVM describes a system which provides a shared memory with a software implementation on top of the operating systems (OSs) [4]. This architecture employs a *MMU (Management Memory Unit)* to provide coherent shared address spaces. Unfortunately, this architecture is not OS transparent because it uses specific operating system functions to share memory with the other processors.
- *Distributed Shared Memory (DSM)*: It is another memory architecture that enables accessing shared data without replication. Replication is beyond the scope of this architecture, therefore, DSM does not maintain data coherence. In consequence, the memory address spaces in this architecture must be managed explicitly by the user, which is a *painstaking* task. This is one of the main reasons this architecture has not been adopted by many systems.

- *Cache Only Memory Architecture (COMA)*: In COMA, the memory organization is similar to *Non-Uniform Memory Access (NUMA)*. However, instead of storing data in a fixed location, COMA uses the storage spaces of different processors as a large cache [5] called *attraction memory*. The attraction memory enables accessing data blocks to processors faster than other memory architecture such as SVM. Additionally, compared to NUMA (which enables storing a block of data using a unique address and copying it in all processors' caches), COMA stores blocks only once and migrates them dynamically whenever a processor requires them. COMA reduces data copies to many processors. However, in COMA, a block can be duplicated in some specific cases for instance, data required by two or more nodes at the same time. It caches data in main memory, which can significantly promote the performance of data retrieval because the access time required to load data from secondary storage is eliminated in this architecture [11].

We found two main techniques for accessing memories: *Uniform Memory Access (UMA)* and *Non-Uniform Memory Access*. In UMA, accessing data depends on a single bus and all processors share the physical memory uniformly [5]. This essentially means that the data is stored in a location (often in a centralized server) that is accessible by all processors uniformly. This architecture is effective when many clients need to share data, yet it is insufficient for Big Data applications. The reasons are two-fold: the data size often exceeds the capacity of a single server and the nodes need a permanent access to data, which may cause network congestion.

In NUMA, the memory is divided between different processors, and each block has a fixed location. The processors will access their local memories, which is faster than remote access [5]. This architecture does not support data migration; more specifically, the data blocks cannot be migrated between nodes. The only way of making data available to the processors is to have a copy in the local cache of the processors. This architecture needs an efficient cache coherence protocol to guarantee the consistency of data blocks.

A distributed system comprises one or more clusters. Typically, each cluster consists of multiple physical/virtual nodes that can have several processors. While running, if a processor needs a block of data, it is copied to its cache. Consequently, a large number of copies of data blocks are diffused within and across the nodes of a cluster. These copies must be consistent to avoid any *dirty read* (incorrect data). If no *write* operation is performed, the blocks remain consistent without needing any management tasks. However, in case a data block should be changed, the copies of corresponding block needs to be updated to keep them consistent with the latest version of the data block.

Different methods to maintain data consistency in a distributed architecture already exist. The *Write-invalidate* protocol relies on *write-once read-many* principle. It allows only one *write* operation at a time. However, it allows multiple *read* operations. When a node updates a block, it sends an *invalidate signal* to all nodes that have the copy of this block and write new data in the block. The other nodes then flag their copies 'obsolete' and remove them. If the copy is needed for computation, the requesting nodes request a copy of the new block directly to the node that performed the update [4]. The main advantage of this protocol is the data is not broadcasted when an update occurs. Additionally, it allows recovering data progressively. In *Write-update* protocol, the node (which performs an update) sends the data to all nodes which have the copy of that block [4]. This guarantees the usability of data blocks because, all blocks residing in the nodes are essentially up-to-date. This protocol is hard to apply in large networks because it generates heavy traffic, which can quickly saturate the connection.

Of these two protocols, *Write-invalidate* is more suitable for a large architecture where data is cached by a wide number of processors simultaneously, whereas, the *Write-update* is preferable to those where data are not frequently cached, yet faster access is required. The systems which rely on central memories (as opposed to attraction memories) use one of these two protocols.

The *write-through* approach updates the central memory each time an update occurs. In *write through* systems, the main memory is always up-to-date [4]. The other approach is called *write-back* which does not synchronise main memory of a node *at once* [4]. Rather, the other processors contact the node(s) that have the latest copy of the block. Additionally, this protocol updates memory if the data block is removed from the cache. This protocol avoids unnecessary accesses to central memory, which is useful particularly for the architectures where data is often changed or modified.

There are some advanced protocols that optimize updates during the write operations. The *Write-once* protocol is one of them. It introduces several states (namely *'not modified*', *'possibly shared*', and *'reserved'*) of operations performed on memory pages. It reduces overall bus traffic by performing *write-update* operation during the first write and then it carries out *write-invalidate* operation [12] [4].

The protocols discussed in the above are not suitable for multi-bus architectures. Thus, a special type of cache coherence protocol called *directory based cache coherence* [13] was introduced. This protocol employs specific directories for maintaining coherence between caches. When an entry is changed, the directory either updates or invalidates the other caches with that entry.

In the *full map directory* architecture, the directories contain a list of processors. A single bit is used in the directories to know whether or not a processor contains different blocks [13]. In this architecture, when a *cache miss* occurs, the processor can contact any directory to locate the data block. Additionally, for each update, all directories must be updated. For this architecture, the size of the directories are very important because, they store the information of all processors. Another approach called *limited directory* is almost the same as the full-map directories, except for the directories in this architecture are not storing all system entries; instead, only a limited number of parts are stored. This solution reduces the storage space that is required by the directories, while it limits the number of blocks which can be cached simultaneously.

Finally, the *chained directory* distributes the directory between different caches. This approach addresses the size

problem of directories without restricting the number of shared block copies. Chained directories keep tracks of shared copies of a particular block by maintaining a chain of directory pointers.

The *Data diffusion machine* is a multiprocessor memory architecture which relies on COMA principle. The data is stored in different *attraction memories*; each of them is associated with a processor. When a block of data is needed by a processor, it is migrated from the source attraction memory location to the target location [4].

In DDM architecture, the nodes are not explicitly interconnected, however, they communicate with each other using a hierarchical system of directories. The directories store information of the data blocks that are stored in the leaf nodes [5]. The data is stored in set-associative memories. This approach enables the directories to locate data blocks efficiently. Various directories of this architecture contain input/output buffers, which enables storing intermediate results while transferring data from source to target locations. DDM provides a shared memory abstraction that encapsulates the underlying mechanisms and provides users a subset of total distributed data in a timely manner.

The main challenge of classical *hierarchical DDM* architecture is high-traffic, which may go beyond the control when too many data blocks need to be migrated at the same time. It can cause read and write buffer overflow, and can lead to a congestion and thus, it can slow the block transfer significantly. Several solutions have been proposed to resolve this problem. For instance, using routers for an optimized transfer of data block between different directories [14]. Another approach is called *Horn DDM* which essentially connects the physical nodes using a bus based point-to-point communication system [6]. This reduces congestion because the data can use different paths to be transferred between the nodes. However, this approach increases the number of intermediate nodes.

From the discussion it is clear that there are several efficient memory architecture such as NUMA and COMA for high-performance computing. We adopted COMA, however, we extended conventional COMA by proposing a new protocol and new functionalities such as replication function for storing copies of data blocks in secondary storage. In addition, we adopt some characteristics of hadoop distributed file system (HDFS), which are missing in COMA. These turn the CedCom architecture into functionally more efficient than existing COMAs.

VII. Conclusion and Future Work

In this paper, we presented the CedCom architecture. We provided a detail of how we developed the architecture. We discussed the implementation of its components such as directory node, compute node, *etc.*. Also, we described the implementation of attraction memory. We used an efficient technique called *n-way associative cache* which enables high-speed data access and increases the cache hit ratio significantly. Additionally, we implemented a data migrator that enables migrating data blocks automatically from the source to the target nodes in the clusters.

This architecture offers many functionalities yet, several works must be done. The works which we planned to carry out near future are as follows. The experiment that we presented in this paper is incomplete or in other words weak. Therefore, we plan to conduct a rigorous test to evaluate all the functionalities of the architecture. We plan to replace the set associative cache with the skewed associative cache which is a more efficient approach. Also, we plan to develop a component that can migrate data in an intelligent way such as by adapting the bandwidth. Finally, we will complete the development of the protocols of CedCom architecture.

Acknowledgment

This work was carried out as part of the CEDAR (http://cedar.liris.cnrs.fr/) Project (Constraint Event-Driven Automated Reasoning) under the Agence Nationale de la Recherche (ANR) Chair of Excellence grant Number ANR-12-CHEX-0003-01 at the Université Claude Bernard Lyon 1 (UCBL).

References

[1] P. Zikopoulos and C. Eaton, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*, 1st ed. McGraw-Hill Osborne Media, 2011.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450

[4] S. Raina, "Virtual shared memory: A survey of techniques and systems," Tech. Rep., 1992.

[5] E. Hagersten, A. Landin, and S. Haridi, "Ddm-a cache-only memory architecture," *Computer*, vol. 25, no. 9, pp. 44–54, 1992.

[6] H. L. Muller, P. W. Stallard, and D. H. Warren, "The data diffusion machine with a scalable point-to-point network," Tech. Rep., 1993.

[7] H. Muller, P. W. Stallard, and D. H. Warren, "The role of associative memory in vsm architectures: A price-performance comparison," in *In Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing*. IEEE, 1996, pp. 41–49.

[8] R. Haque and M.-S. Hacid, "Blinked data: Concept, characteristics, and challenges," in *In Proc. of Service Congress 2014*. IEEE, 2014.

[9] I. Sophia, "Grid'5000: Plate-forme de recherche exp ´erimentale en informatique," Tech. Rep., 2003, accessed: 2014-09-12. [Online]. Available: https://www-sop.inria.fr/aci/grid/public/Library/rapport-grid5000-V3.pdf

[10] H. Aït-Kaci, M.-S. Hacid, R. Haque, and D. Fourure, "Experiments with triplestores," Université Claude Bernard Lyon 1, Lyon, France, CEDAR Technical Report Number 5, October 2013, http://www.cedar-liris.fr/documents/ctr4.pdf.

[11] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum *et al.*, "The case for ramclouds: scalable high-performance storage entirely in dram," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.

[12] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *ACM SIGARCH Computer Architecture News*, vol. 11, no. 3. ACM, 1983, pp. 124–131.

[13] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, vol. 23, no. 6, pp. 49–58, 1990.

[14] H. L. Muller, P. W. Stallard, and D. H. Warren, "Implementing the data diffusion machine using crossbar routers," in *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*. IEEE, 1996, pp. 152–158.