The Data Management Entity: A Simple Abstraction to Facilitate Big Data Systems Interoperability

Damianos Chatziantoniou Florents Tselai
Department of Management Science & Technology
Athens University of Economics and Business (AUEB)
{damianos,tselai}@dmst.aueb.gr

ABSTRACT

Today's big data era is described by intense variety in data management systems, query languages and programming paradigms. Each system targets well a specific application area, reinforcing the belief that the era of one-size fits all has gone. Interoperability, systems' connectivity and high-level data models become once again the core of research initiatives. In this paper we present our vision for a layered architecture to support interoperability among different data management systems, generalized under the term *data management entities* (DMEs). DMEs range from JVMs running java programs to Hadoop systems employing complex MapReduce jobs to traditional RDBMS running SQL queries to stream engines and CEP scripts. The top layer consists of a universe of DMEs, communicating through a well defined http-like protocol: a DME *transparently* invokes another DME's data manipulation task, regardless task's nature. Communicating DMEs share/operate on a shared data object, a key-value set (KVS) - just a set of key-value pairs - which exists in the layer below and is referenced through a unique (internet-wide) address via a well-defined API. This layer serves as the transient common memory space for communicating DMEs and consists of *globally addressable* KVSs, organized in domains, sub-domains, etc. In a way, this approach constitutes a form of remote procedure call *by reference* (the KVS is the common reference). We argue that this architecture allows the construction of high level query languages and cost-based distributed query processing engines, involving completely heterogeneous data manipulation tasks. For example, we show that MapReduce evaluation algorithm and distributed relational query processing are just instances of the proposed architecture. We also claim that it can easily facilitate the end-to-end processing in big data applications, an established goal in the research agenda set by the Beckman report.

CCS Concepts

• **Information systems→Information integration**

1. INTRODUCTION

Currently, most big data deployments follow a highly ad hoc, non-disciplined approach, entailing a high degree of data replication and heterogeneity, both in terms of storing options and analysis tasks. The system administrator has to choose one (or more) data management systems from a plethora of alternatives and facilitate the enterprise’s reporting needs utilizing a wide range of query languages and analysis techniques. Data management systems involve traditional RDBMSs, cluster of Hadoops, NoSQL and others. Reporting and analysis tasks include plain SQL, spreadsheet scripts, MapReduce jobs, R/Java/Python programs, complex event processing queries, machine learning algorithms, and others. A not-so-new challenge resurfaces: interoperability. How can these systems interact? How can these systems interoperate? For example, how can an excel spreadsheet use the data produced by a MapReduce job in a *standardized* way, using a *well-defined protocol*?

This necessity has been identified by the current authors in [2] and more recently by the Beckman report [1] and Polystores [3]. Beckman report recognized the problems the "diversity in the data management landscape" creates and asserted "the need for coexistence of multiple Big Data systems and analysis platforms is certain" and that in order "to support Big Data queries that span systems, platforms will need to be integrated and federated." ODBC, a well-defined API between applications and RDBMs greatly contributed to the growth of relational systems. Prior to ODBC, applications had to rely on several, language- and DBMS-specific, APIs. Standardization of data connectivity greatly helped innovation and productivity, allowing developers to focus on the core of their ideas. What we need today is a similar breakthrough, with similar rewards.

We argue that a standardized and protocol-based approach can significantly facilitate the unified dissemination, federation and analysis of data. Towards this direction a new connectivity protocol between data consumers and data producers should observe the following generic properties:

Execution transparency: the data consumer should be completely unaware of the data producer's query language specifics. It should only know the signature of the data manipulation task at the data producer. In addition, the nature of the task should be completely transparent to the data consumer. This includes tasks involving stream data.
Schemaless and simple data representations: the structure and representation of the exchanged data objects should be as simple as possible. In addition, it should not encapsulate schema information in any way, either regarding the data consumer or the data producer.
Composability/Symmetricity: a data consumer should be able to participate as a data producer in a different connection instance and vice versa. I.e., whatever system manipulates/manages data should be able to play the role of the data consumer and data producer at the same time, interchangeably. This *composability* principle would allow the definition of *analysis workflows*, an essential property in facilitating the data analysis pipeline [1]. ODBC is asymmetric in this sense.

2. MOTIVATION

As a simple motivational example, consider two data management systems, a RDBMS (system A) used by some application that maintains data for a subset of facebook users and a remotely residing Hadoop (system B) that stores detailed data for all facebook users. Through a MapReduce (M-R) job M at system B, one can compute the average sentiment of the posts of each facebook user during the previous week. System A would like to use the output of M - only for those facebook users it manages - in its daily routines. With today's technology, a Hadoop job based on M has to be developed that reads in the user ids from system A and writes the results back to system A as a new column. For this to happen, custom connectors implementing hadoop's input/output specifications have to be developed, i.e. developers must implement hadoop's InputFormat and OutputFormat java interfaces for database access and pass the appropriate metadata (database url + port, table name, column name etc.) This approach has several disadvantages: a) for each data manipulation task aiming to use M of system B, custom-connectors and API implementations should be developed - an effort that involves significant human intervention, and b) M has to be aware of the schematic details of system A (in case of relational systems), thus limiting M's ability to act as an independent data producer. However, one can think of a different approach, easy to standardize, briefly depicted by Figure 1.

System A, using a predefined API, creates an empty set of key-value pairs, K (Step 1). This set can be referenced with an address (e.g. 1829), using this API. It then sends this address to B and declares its intention to execute M-R job M, possibly with some input parameters (Step 2). If system B agrees (as a reply) on the address suggested (because it may suggest a different address for the key-value set) and verifies the signature of M's invocation, system A populates K's keys with user ids and K's values with nulls (Step 3). Having done so, it requests system B to initiate job execution (Step 4). From this point on, Hadoop can access K (Step 5) to retrieve the specific keys and values for its computation. As long as the job executes, it writes results to the common key-value set, K (Step 6). When the job finishes, system B notifies system A about it, by issuing a specific request to it (Step 7). From now on system A can directly access specific key-value pairs from K (Step 8) and incorporate them to the native relational model. Note that system A could access K at any time after Step 4, without waiting for Step 7. This is particularly useful if system B represents a standing query, continuously updating this key-value set K.

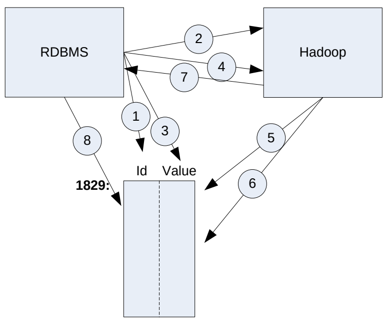

Figure 1. Communication workflow for a simple example

This call sequence resembles the interface between applications and RDBMSs via private buffers before the introduction of ODBC: to read an object, the application calls the database system, giving the object id and the address of a buffer in its address space. The system reads the object from the disk in its own buffer pool and then copies the object to the application's private buffer. A similar approach takes place for writing an object to the database. Similar sequences of this nature can also be found in cloud-base storage system such as Cloudy [5]. In the following Section, we provide the big picture of the proposed framework.

3. THE BIG PICTURE

We propose a *layer* of *addressable* **key-value sets** to be used as a *commonly-referenced* memory space for *communicating* **data management entities**.

A key-value set (KVS) is a set of key-value pairs (k_i, v_i), $i \in I$, such that $k_i \neq k_j \; \forall \; i, j \in I, i \neq j$.

KVSs are simple constructs, easily understood, with no schema information. Key-value stores can easily support the management of KVSs, however almost any modern data management system could do it. We explain below why a KVS is an excellent candidate to represent shared data among communicating DMEs.

We propose a layered architecture equipped with the appropriate protocols for intra- and inter-layer communication. The top layer (DME-layer) consists of DMEs, interacting with each other through a well-defined protocol (DME-to-DME protocol). Anything that manipulates data exists in this layer. The layer below (KVS-layer) consists of a collection of *addressable* key-value sets, shared among communicating DMEs. This layer can be thought of as the transient common memory space for interacting DMEs. Another protocol is required for KVS management by a DME (DME-to-KVS protocol). There are no specific operators within the KVS-layer. We foresee "domains" of KVSs, implementing their own algebras. Figure 2 presents a description of the proposed concepts.

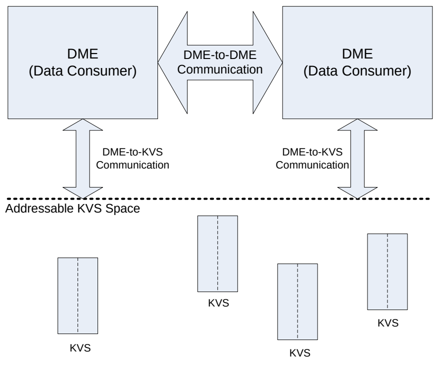

Figure 2. A layered architecture, DMEs and KVSs

The idea is simple: A DME (the data consumer, DC) wants to use a data manipulation task at another DME (the data producer, DP). This could be an SQL statement, a MapReduce job, a standing query, a python program, etc. For that purpose, the DC should be able to invoke a remote execution at the DP, in a similar fashion to http or other remote procedure call protocols. The two DMEs access a *shared* data structure to exchange data: the DC to provide input to the remote execution at the DP and the DP to provide the output to the DC. Both should be able to access this shared data structure at any time during the connection.

We foresee a specific pattern for such executions: the DC provides the DP with a set of keys and asks it to compute a value for each key. This pattern becomes commonplace in systems, queries and algorithms. Most modern applications simply produce a value that is associated to an ID (the key): a sentiment value of a document ID, the location of a facebook user, the average price of a 10-minute sliding window of a stock ID, the most recent reported value of energy consumption by a smart meter, etc. As a result, a key-value set seems quite appropriate for that shared data object.

Currently there are efforts pointing towards a separate addressable memory layer, such as RAMCloud [7] and Piccolo [8], which both share the notion of in-memory addressable “tables” supporting key-value operations. Clarifications, challenges and opportunities of the proposed architecture are presented below.

4. CHALLENGES & OPPORTUNITIES

As mentioned earlier, "we propose a *layer* of *addressable* key-value sets to be used as a *commonly-referenced memory space* for *communicating data management entities.*" There are several challenges and opportunities in this architecture. We start with the challenges and then discuss how this framework can be used to *generalize, encapsulate* - and *leverage* - distributed data management.

4.1 Challenges

Communicating Data Management Entities. This requires a well-defined protocol, consisting of a set of primitives to orchestrate the communication between the data consumer and the data producer. The DC can use this protocol to establish a connection with the DP, invoke an execution at the DP and terminate the connection. The DP can notify the DC for the completion of the execution and terminate the connection. A high level of the basic primitives follows. The first challenge involves a thorough design of this protocol, regarding primitives' signatures, replies, call sequence, security concerns, and other issues.

-*init()***:** this primitive is used to establish a connection between two DMEs and is issued by the DC. It should contain as parameters the address of the KVS to be shared and the invocation (name + arguments) of the program/query at the DP's side. We assume that some negotiation may take place during this phase. For example, the DP may propose a different address for the shared key-value set. This would be the case if DP's invocation is a standing query, already running, continuously updating a different KVS; or the DP has cached results for the specific execution in another KVS; or the DP already natively supports KVSs, and asks the DC to directly store data to one of its local KVS for performance reasons. Additional parameters for this primitive may include a key to encrypt/decrypt the contents of the KVS or other connection-related aspects.
-*execute()***:** this primitive is used to actually initiate the remote execution at the DP, *after* connection between DMEs has been established. It is issued by the DC. It is important to keep *init* and *execute* primitives separate, especially when there are time constraints on the remote execution (e.g. in case of stream data.)
- *notify_complete()***:** this primitive is issued by the DP and indicates the completion of the remote execution. This is required because access to the shared KVS is asynchronous by the DC and the DP, so the DC should be notified for the completion of the task.
-*terminate():* this primitive can be issued by either DME and terminate the connection.

Commonly-Referenced Memory Space. This implies the presence of a well-defined API so a DME can create and manage a KVS. While CRUDE-like operations such as *create, read, update* and *delete* methods are clearly understood, a discussion is required for the exact format and behavior of each. In particular, the read operation should allow some filtering of the KVS, either through a simple predicate over keys and values or by providing a set of keys to be selected. Our first implementation [2] allows filtering conditions over just the key, but in many industrial applications complex expressions involving values are not uncommon. Other issues that have to be addressed are: what if a DME does not delete a KVS that has created? Should the corresponding KVS management system implement garbage collection? Who is the "owner" of a KVS? What is the "lifetime" of a created KVS? Which DMEs are allowed to access this KVS and in what mode (read/write)? For example consider Webdis [9], an http interface for Redis that provides some insights on these issues. A similar kind of middleware between data producers and consumers in the form of publish-subscribe is suggested in [4]. A commonly-referenced memory layer is also proposed in Tachyon system [6], constrained however within a cluster.

A Layer of Addressable Key-Value Sets. This is a *conceptual layer*, consisting of systems that provide KVS management according to the proposed framework. To do so, it should (i) implement the DME-to-KVS API mentioned above, for a variety of DMEs, and (ii) allow access to a KVS through an address, internet-wide, following some standardized addressing scheme. There is no restriction on what such a system could be. It could store KVSs anywhere in the memory hierarchy: main-memory, distributed-cache, disk, etc. It could organize KVSs to domains, subdomains, etc. It could be one of the DMEs in a DME-to-DME connection, i.e. one of the DMEs could also play the role of the KVS manager. Such a system could guarantee (or not) fault-tolerance, availability, etc. In addition, it should provide answers on how it handles ownership, lifetime and access control of KVSs. Finally, a *standardized addressing scheme* should be designed in such a manner that captures location hierarchies (e.g. domains, sub-domains, etc.) and identifies the position in the memory hierarchy of a KVS. This information could be useful in a generalized query processor, discussed in the next Section.

Suitability for Stream Engines. A layered architecture like this, essentially introduces a referencing layer (i.e. indirection) between communicating DMEs. This is particularly appropriate for collaborating applications involving stream data: a stream management DME can continuously produce aggregated data (e.g. the average stock price over a sliding window of 10 minutes), consumed by a traditional DME, such as a RDBMS. The asynchronous access to the shared KVS, allows the data consumer to retrieve data whenever it deems appropriate (e.g. [7]). In addition, the shared KVS may be located near - or within - the stream processing DME (the data producer).

Transactionality Issues. A potentially challenging aspect in the proposed architecture is the issue of transactional consistency at the KVS layer; especially in the case of complex workflows where multiple DMEs constantly request execution from another. That is, in scenarios of multiple DMEs using the same KVS, one could argue that DMEs may have an inconsistent view (different versions) of the KVS layer. For example, consider two separate DME-to-DME connections with the same data consumer, sharing a common KVS - in other words both data producers populate the same KVS. As another example, if a DME is using a remote address to store a continuously running query that returns a huge set of key-value pairs, is that large result updated atomically or incrementally? If some data feeds are slow and some are fast, one might get an inconsistent (nonserializable) view of the KVS layer. While these problems already exist now, what (if anything) can the framework do to manage transactional requirements across systems? For instance, when a DME creates a KVS (and thus becomes "owner" of the KVS), it could also specify the required isolation level for that KVS.

4.2 Opportunities

The proposed architecture can be used to generalize various existing distributed data management frameworks, such as distributed relational query processors, MapReduce evaluation algorithm and column-oriented processing engines. However, given the diversity in DMEs, it opens up a wide range of interesting possibilities, both in terms of infrastructures and optimization opportunities.

For example, in the case of distributed relational query processing, each node would act as a DME. The node receiving user's query would act as the master DME (the data consumer), establishing connections with all the participating nodes (the data producers), sending a node-specific query (the remote execution) to each of them. For each connection, the data consumer would specify a KVS address for the data consumer to place the results, or let the data producers to specify a KVS address. The KVSs would keep the results of the node-specific query (e.g. row-id as the key and the record's contents as the value), which would be collected by the master DME, assembled, and provided to the user. In other words, with the use of the proposed architecture and accompanying protocols, one can build a distributed relational query processor - *possibly beyond cluster boundaries.* In addition, KVSs could reside somewhere "between" the data consumer and the data producer, not necessarily in disk.

The same holds for the traditional MapReduce evaluation algorithm: it could be represented as a collection of DMEs (Map/Reduce tasks), exchanging KVSs in a specific sequence, orchestrated by calls of the DME-to-DME protocol. One should also note that in the case of traditional Hadoop, HDFS nodes should be homogenous (i.e. running JVM instances). In our abstraction however, this is not the case; a node can be generally a DME (for example a JVM instance or a RDBMS) as long as it adheres to the protocol's primitives.

Going one step further, given: (a) the generality of DME's definition, (b) the flexibility of the KVS to handle both structured and unstructured data, (c) the ability to represent with the KVS the input, output or both of the collaborating process, and (d) the capability to position the KVS close to either the data consumer or data producer and anywhere in the memory hierarchy, one can think of truly *generalized*, *cost-based*, *distributed* query processors, workflow-like, involving heterogeneous systems, data formats, tasks and application profiles.

In another direction, one can think of scenarios where a KVSs' management system implements a set of operators to manipulate KVSs and provides some basic functionality to DMEs through these operators. Examples include filtering, outer-joins and intersection. A DME could map an operator in its native model to one of the KVS's management system's operators for performance How to specify this in the DME-to-KVS protocol is a challenge.

End-to-end processing and understanding of data is another crucial database research challenge [1]. The proposed architecture can significantly facilitate all data analysis tasks across the raw-data-to-knowledge data pipeline. Indeed, one can argue that each one of these steps is performed by a DME acting as a node of a unified data supply chain. Currently those nodes collaborate in an ad-hoc manner based on system or language-specific APIs and bindings. Once however communication between DMEs is standardized the whole process becomes truly seamless. Developers can focus on highly-specialized data tools implementing the appropriate API and need not to worry for specific bindings. Data scientists can experiment with novel combinations of those tools and systems leveraging the symmetry property of the DME-to-DME communication protocol. It is also important to note that the data pipeline is far from straightforward, it is rather an iterative process where an initial dataset can have multiple versions across the pipeline (for example different methods to handle missing data). Instead of re-computing every possible transformation on an initial dataset the data scientist can cache the results of each transformation at the KVS layer (different versions) and access different versions in subsequent steps of the pipeline. Furthermore, one can think of scenarios where multiple data supply chains share the common (global) KVS layer. This fact allows the definition of synergistic data flows and economies of scale in data management across systems and organizations. Of course such a “globalization” in the data economy poses new challenges in the area of privacy and data ownership at the KVS layer.

5. DISCUSSION AND CONCLUSIONS

In this paper we presented a layered architecture to data interoperability based on a ubiquitous universe of remotely accessibly KVSs. In essence, with the proposed architecture we completely decouple the computation and memory layer of any data management scenario. By doing so we are able to generalize, abstract and effectively encapsulate all the key components of distributed data computation, storage and management.

6. ACKNOWLEDGMENTS

The authors would like to thank Ted Johnson, Ken Ross and Diomidis Spinellis for their valuable comments during the preparation of this paper.

7. REFERENCES

[1] Abadi, D. et al. The Beckman Report on Database Research. In *ACM SIGMOD Record*, 43(3), 61-70, 2014

[2] Chatziantoniou, D., and Tselai, F. Introducing Data Connectivity in a Big Data Web. In *DanaC Workshop SIGMOD*, 2014

[3] Duggan, J. et al. The BigDAWG Polystore System. In *ACM Sigmod Record*, 44(3), 2015 (http://users.eecs.northwestern.edu/~jennie/research/bigdawg_record.pdf).

[4] Joshi, R. Data-Oriented Architecture: A Loosely-Coupled Real-Time SOA. *Real-Time Innovations, Inc*, (TR), 2007

[5] Kossmann, D., Kraska, T., Loesing, S., Merkli, S., Mittal, R., and Pfaffhauser, F. Cloudy: A modular cloud storage system. In PVLDB, 3(1-2), 2010

[6] Li, H., Ghodsi, A. , Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In SoCC, 2014, 6:1-6:15

[7] Ousterhout, J. et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. SIGOPS Op. Sys. Rev., 43:92–105, 2010

[8] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In OSDI, 2010

[9] Webdis http://webd.is