dataClay: a distributed data store for effective inter-player data sharing

Jonathan Martí, Anna Queralt, Daniel Gasull, Alex Barceló, Juan José Costa, Toni Cortes

PII: S0164-1212(17)30101-2
DOI: 10.1016/j.jss.2017.05.080
Reference: JSS 9967

To appear in: *The Journal of Systems & Software*

Received date: 2 May 2016
Revised date: 31 October 2016
Accepted date: 22 May 2017

Please cite this article as: Jonathan Martí, Anna Queralt, Daniel Gasull, Alex Barceló, Juan José Costa, Toni Cortes, dataClay: a distributed data store for effective inter-player data sharing, *The Journal of Systems & Software* (2017), doi: 10.1016/j.jss.2017.05.080

Highlights

- A distributed data store to share data in a secure and flexible way is proposed.
- The system is based on the concepts of identity and encapsulation.
- Parceled control breaks de facto assumption of centralized administration.
- Impedance mismatch issues are resolved based on an Object Oriented data model.
- A data access performance evaluation with popular benchmarks and applications.

dataClay: a distributed data store for effective inter-player data sharing

Jonathan Martí[a,*], Anna Queralt[a], Daniel Gasull[a], Alex Barceló[a], Juan José Costa[a], Toni Cortes[a,b]

[a] *Barcelona Supercomputing Center (BSC)*
[b] *Universitat Politècnica de Catalunya (UPC)*

Abstract

In the Big Data era, both the academic community and industry agree that a crucial point to obtain the maximum benefits from the explosive data growth is integrating information from different sources, and also combining methodologies to analyze and process it. For this reason, sharing data so that third parties can build new applications or services based on it is nowadays a trend. Although most data sharing initiatives are based on public data, the ability to reuse data generated by private companies is starting to gain importance as some of them (such as Google, Twitter, BBC or New York Times) are providing access to part of their data. However, current solutions for sharing data with third parties are not fully convenient to either or both data owners and data consumers. Therefore we present *dataClay*, a distributed data store designed to share data with external players in a secure and flexible way based on the concepts of identity and encapsulation. We also prove that *dataClay* is comparable in terms of performance with trendy NoSQL technologies while providing extra functionality, and resolves impedance mismatch issues based on the Object Oriented paradigm for data representation.

Keywords: data sharing, distributed databases, nosql, storage systems

*Corresponding author
Email address: jonathan.marti@bsc.es (Jonathan Martí)

1. Introduction

For many years, **databases have proven to be successful for sharing data** among different applications, **centralizing all the data of an organization**. Databases follow the ANSI/SPARC architecture [1], a layered structure that provides each user with access to a fragment of the database, and also abstracts how data is physically managed.

In this environment, where all the applications accessing the data live within the borders of the same organization, this notion of **centralized control is associated to the database administrator (DBA)**, an identifiable person who is responsible for the data. The DBA decides which data is stored in the system, and how and by whom it can be accessed. With this formula, DBA tailors the database to the needs of different applications, since their users are known and accessible.s

Based on this approach, **Data Services** [2] can be seen as its **successors in a globally connected world**, adapting the same ideas to the current scenario where data sharing is being extended from a single organization to an ecosystem of data owners and data consumers, what is usually referred as **large-scale data sharing** [3]. In particular, a data service is an additional layer on top of the database and/or other services that encapsulate access to data. This allows data owners to make data accessible to external players in a controlled way, basically by disallowing arbitrary queries on the underlying data. But in this case, the **data owner does not have access to all the possible users or applications, so he decides which data to store and to release according to his own criteria**.

However, in this context of large-scale data sharing, the fact that **all decisions (and their implementation) are exclusive to the data owner** does not seem to be the best solution. Data is one of the most valuable assets of a company and obviously its owner must control how it can be accessed, but **this should not prevent data consumers from adapting the accessible data** to their needs (one of the challenges of Big Data [4]). Specifically, it is vital that

consumers are allowed to **add additional views, new information that is not present, or additional functionality for data processing**; as long as the security and integrity of the existing data are not jeopardized, providers' do not lose control of their data, and existing users and applications are not affected by these additions (unless they want to).

Furthermore, offering this flexibility to consumers would help tackling another key aspect to take into account for effective data sharing: data integration. The lack of standards in conjunction with the vast amount of data formats and APIs compels developers to spend a significant amount of time on coding explicit middle-ware to adapt data for its correct processing or for interoperability between different services or applications. Therefore, enabling the consumer to adapt the accessible data from its source would ease the integration process.

In view of the above, and further detailed in the following sections, this paper expounds a novel solution to fill the gaps of current data services and database management systems in the area of data sharing. A novel distributed data store called *dataClay* is presented offering flexibility for the consumers to add new functionality without jeopardizing providers' requirements, and it is designed to exploit the best of near future storage technologies grounded in the concepts of identity and encapsulation.

dataClay relies on the structure and principles of the ANSI/SPARC architecture, which has proven successful in many aspects, but **breaks the de facto assumption that each DBMS has a central administrator** (or administrator team). In contrast with current database systems and data services, the main feature of *dataClay* is that **control is parceled, allowing different users to share and reuse data while maintaining full control on their assets, both regarding the schema and the data itself**. That is, any user can participate in the definition of the data schema by adding those missing concepts or functions he needs, but always based on the fragment that he is allowed to access as granted by the owner. These enhancements of the original schema are what we call enrichments, and can be shared with other users in the same way as the owner shares his original schema.

In this context, it is essential that data never moves outside the data store in order to guarantee full control on it (unless explicitly allowed by the owner). To this end, we propose an **identity-based data store**, in contrast to the usual value-based DBMSs. That is, **every single piece of data can be uniquely identified and individually manipulated** by means of the set of operations that encapsulate it, which are executed inside the data store. That is, data is manipulated in the form of objects, **exposing only the operations that can be executed on the data, instead of exposing the data itself**. **This encapsulation guarantees data independence** [1], and at the same time **avoids the impedance mismatch between the application and the specific data model of the data store**, which requires around 30% of the total of an application code to overcome [5]. Also,performance is increased since the amount and size of data transfers between the data store and the application is reduced.

The rest of the paper is organized as follows. Section 2 is a conceptual overview of the main pillars upon which *dataClay* is built. Section 3 describes the main design decisions and technical details to fulfill the requirements and implement the key *dataClay* features. Section 4 presents a performance evaluation comparing *dataClay* with trendy NoSQL databases. Finally, section 5 outlines the related work and the paper is concluded in Section 6.

2. Overview of the system

This section provides an overview of the main concepts on which *dataClay* relies. **For a better understanding, the explanation is based on the different levels of the well-known ANSI/SPARC** architecture [1] mapping them to the abstractions used in *dataClay* to structure and manage data. The main goal of the ANSI/SPARC architecture is to guarantee data independence, that is, the immunity of applications to changes in the way data is stored and accessed. To this end, the architecture proposes three levels of abstraction: the external level, the conceptual level, and the internal level. First, the conceptual

level of *dataClay* is presented, and then how it is exported to users (external level), and how it is mapped to the storage (internal level).

For the sake of clarity, we start exposing how *dataClay* maps with the three different levels of the ANSI/SPARC architecture as if there was a single administrator for the whole data store. In this way we explore how *dataClay* works analogously to other data stores in respect the essential characteristics of the ANSI/SPARC model that are still valid for us. However, we comment on the differences at each level and finally present a last subsection describing **how, in contrast to what is assumed in ANSI/SPARC, *dataClay* divides data management into different parcels that are separately controlled by independent users: *dataClay* parceled control**.

2.1. Conceptual level

The conceptual level represents all the information contained in the database, abstracting away physical storage details. This content is represented by means of the conceptual schema, which is defined using the conceptual data definition language (DDL).

The conceptual level in *dataClay* follows an object-oriented data model. The basic primitives are objects and literals or values. Each object has an **identity, represented by a unique object identifier (OID)**, and can be shared (referenced) by other objects. Every object is an instance of a type or class, and a class has a number of attributes or properties, which represent the internal state of the object, and a set of operations or methods that can be applied to objects of that type. A class can be a subclass of another class. Objects are encapsulated, that is, their internal structure is not visible outside the object and its users only know which methods can be executed on objects of that type. **This encapsulation provides data independence by basing the manipulation of data on its identity and the methods provided, rather than on its value**.

In contrast to objects, literals have no identifier and do not exist as first-class entities outside the object that contains them as values of properties. Examples

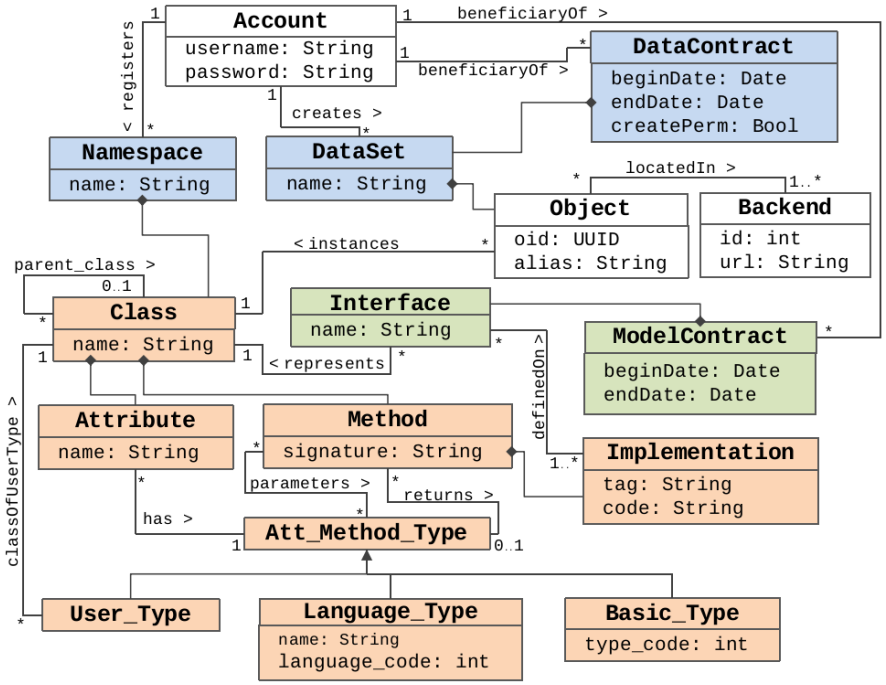

Figure 1: UML class diagram representing *dataClay*'s main entities. In orange, all classes related with the conceptual level. In green, classes related with external level. In blue, classes for parceled control. All relationships are navigational through methods (which have been omitted) or materialized attributes.

of literals are instances of primitive or built-in types, such as int, char or String.

Figure 1 depicts the information about classes and objects stored in *dataClay*, specified as a UML class diagram. The conceptual level corresponds to classes in orange color, and the rest of the diagram will be introduced in the following subsections.

Following the object-oriented model, *dataClay* is aligned with the idea that the ultimate objective of the conceptual schema is to describe not only the data itself, but also how it is used [1], and at the same time provides users with a transparent access to persistent data, thanks to the following principles [5]:

- **Persistence independence**: the persistence of an object is independent

of how a program manipulates that object, and vice-versa. That is, a piece of code should work with persistent and non-persistent objects indistinctly.

- **Persistence orthogonality**: all objects of all types should be allowed to be persistent and non-persistent.

The goal of *dataClay* is to fully support this object-oriented model in order to provide an identity-based data store. For this reason, the conceptual layer includes not only the structural part of classes, but also their methods. All this information is stored in *dataClay*, so that users can share and reuse not only data, but also the methods that allow applications to manipulate it. In addition, the fact that *dataClay* stores the methods associated to classes enables their execution within the platform when invoked on objects, which **guarantees that data is indeed encapsulated and protected by methods**, with no possibility of bypassing them. As a side-effect, **efficiency is improved due to the fact that data transfers outside the data store are avoided**.

The DDL that allows users to define the conceptual schema in *dataClay* can be any object-oriented language that permit to the users to describe classes and methods. The current implementation supports two well-known programming languages: Java and Python. Despite being procedural languages, they can be used to specify the structure of the information, as well as its associated behavior, in the form of classes. Data independence is then achieved by mapping these language constructs to an internal *dataClay* representation of classes, which is independent of the specific language used to define the class, as well as of the physical representation of the data. Thus, **each user may define his classes in the language that he prefers**.

Given that the schema definitions are translated into an intermediate generic representation, **the fact that the conceptual DDL is language-dependent is not a problem** but rather an advantage to the users, since they do not need to learn a new language to define the contents of the database. **Using directly the same languages used to write applications makes this task**

much easier for them. In addition, users can take advantage of the whole expressiveness of these languages in order to implement additional behavior in the conceptual schema itself, such as enforcing integrity constraints or doing security checks.

The fact that each user can access data only by means of a limited set of operations could represent a problem if some of the operations needed by the user are not available. However, and precisely because data is encapsulated, any user can implement a new operation using only the available ones, as well as additional data that this operation may need, to provide a new functionality that does not exist in the class, without any danger for the already existing data. In *dataClay* this is called ***enrichment***: **if existing classes and methods do not meet the requirements of user's application, he can expand these classes with new methods and/or new required properties**. These new properties and methods will be stored (and executed) in the data store as if they had been defined by the creator of the class. Thus, once they are part of the schema, they can be used and shared like any other method. In section 2.4, we describe how users can enrich existing classes, as well as adding new classes to the schema, while controlling how these new elements are shared.

Enrichments are also a way to compensate the flexibility lost by disallowing arbitrary queries in favor of methods: the enrichment mechanism can be used to structure objects according to the necessities of the application, by adding attributes or methods that provide access to objects that are already organized in the appropriate data structures. This provides a **more efficient way of accessing data, since queries can be avoided when they are known in advance**. Thanks to enrichments, we can also go one step further with regards to the object-oriented data model, and offer the possibility of having **several implementations for each method**. This novel feature enables users to enhance any existing method with their own algorithms, applying different constraints or considering specific environment variables that might be subject to particular conditions. For instance, a method can have an extra implementation optimized for specific hardware features, such as available memory or processor

type.

2.2. External level

According to the ANSI/SPARC architecture, the external level is concerned with the way data is seen by users. An external view is the content of the database as perceived by some particular user, and many external views may coexist on top of a single conceptual view of the database. Thus, each user perceives the same database in a different way.

Users interact with this level by means of a data sublanguage, which is embedded in a host language. This data sublanguage is a combination of an external DDL which, in the same way as the conceptual DDL, supports the definition of the database objects in the external schema, and a data manipulation language (DML), which allows processing or manipulating such objects.

In the case of *dataClay*, **the external level follows an object-oriented data model like the conceptual level, thus preventing unnecessary data transformations**. In order to define the external views, which are based on the data models of the conceptual level, providers populate their data models by means of *dataClay* interfaces and model contracts (as shown in figure 1 represented by green classes). An interface is a subset of the methods of a class that are made accessible to other users. Each class can have an unlimited number of associated interfaces. A model contract is a set of interfaces of different classes that are made accessible to a user for a certain period of time.

Therefore, a user's external view consists of the union of all his model contracts, which provides a subschema (a subset of the classes with a subset of the contents of each class) that the user is allowed to access at a given time. Thus, **several external views can exist at the same time for different users by means of distinct interfaces and model contracts**, and some users can share an external view by means of different model contracts including the same interface. In addition, **a user may define an external view based on another external view, by creating an interface of his own vision of the class and then including it in a model contract to other users**

(further detailed in subsection 2.4).

This mechanism also allows *dataClay* to guarantee data independence, since new properties, methods or classes can be created without affecting existing applications, which are based on the already existing external views.

Regarding the DML, in *dataClay* the same host language, either Java or Python as chosen by the user, is used as data sublanguage. Tight coupling is convenient for the user and, in fact, the data sublanguage should ideally be transparent [1], as is the case in *dataClay*.

To achieve this transparency, users retrieve the set of classes implementing their external view of the registered classes in *dataClay* according to their model contracts (more details in subsection 2.4). These classes only contain the methods visible to that user, and allow him to access shared objects as if they were his own in-memory objects. This is because these classes hide all the details regarding persistence and location of the objects. Thus, the DML in *dataClay* corresponds to the methods defined in the classes, and, the direct users of *dataClay* are application programmers.

2.3. Internal level

The internal level is a low level representation of the database close to physical storage, but still not in terms of any device-specific issues. The internal view is described by means of the internal schema.

dataClay relies on other existing data stores to implement this physical level, and thus there is a different internal schema for each data store. **For each different product, there is a *dataClay* component that maps the conceptual schema to the specific technology used to store the objects**. This mapping absorbs the changes in the internal schema, or even in the implementation of the internal database, and the conceptual schema remains invariant. This allows *dataClay* to take advantage of the advances in technology, for instance when new kinds of databases appear, or when current implementations of existing ones are improved, even if their interface changes. In the same way, **as soon as new storage technologies such as non-volatile memories (NVRAM)**

become a reality [6], *dataClay* will be able to exploit them without affecting already existing applications, and benefit from their impressive access times and fine access granularity just by implementing the appropriate mapping. Thus, any databases or storage technology can be interchanged at any point without affecting any other layer in *dataClay* or above, just by choosing the one desired in a configuration flag.

Currently we have already implemented mappings for the object-oriented database db4o [7], the key-value column oriented Cassandra [8], the Neo4j [9] graph database, and the relational database PostgreSQL [10]. But furthermore we have also implemented a mapping for the novel Seagate's Kinetic hard drives [11], which are disks with a key-value interface instead of a block interface, and we are also working in a mapping to NVRAM in a research collaboration with Intel. Both NVRAM and Kinetic mappings enable *dataClay* to directly access storage devices without the need to go through a 3rd party data store. **In the case of NVRAM mapping, considering *dataClay* design, there is a direct mapping between the conceptual and the internal level, thus mitigating the performance penalties due to unnecessary data transformations.**

Either way, and as will be seen in section 3.2, the fact that in *dataClay* objects are only accessed by their OID enables the simplification of the internal schema to its minimal expression. For instance, in the mapping to PostgreSQL the schema consists of a single table that contains all objects. This table has a column for the OID of the object, which identifies each row, and another column containing the object represented as a byte array, that codifies all the values for the properties of the object. There is a row for each identifiable object, and references to other objects are kept by means of the OID of the referenced object. In section 3 we explain how objects are distributed in several locations, and how their physical location is transparent to users.

2.4. Parceled control: decentralizing database administration

Traditionally, database administration is operated under the responsibility of a single person or a specific team: the database administrators (DBAs). The functions of DBAs traditionally include, among others, dealing with users to understand their requirements, defining the conceptual schema, defining security and integrity constraints, or defining the external schema.

However, a **DBA centralized approach becomes unfeasible in the context of data sharing within a completely open environment**, where users from different organizations may develop applications with requirements that are not always known by the data owner. In this scenario, the data owner cannot allow consumers to run arbitrary requests on the data, so a set of functions that limit how data is accessed must be implemented preventing data from moving outside the database without the appropriate control.

This scenario poses some difficulties both to data providers and to data consumers. On the one hand, data providers cannot foresee all the possible ways in which their data can be used, so they cannot implement all the functions that data consumers will need in order to build their applications. On the other hand, data consumers cannot depend on the knowledge or availability of the data owner to provide them with the functionalities they need. These functions may be very specific to the domain of the data consumer, or they may even be the base of his business model and, thus, nobody else can or should do this job.

Therefore, an alternative mechanism to maintain the security and integrity of the database is required. This alternative should provide flexibility to independent users to build applications based on shared data without requiring any kind of intervention of the data owner. This is what we call **parceled control: the same database stores objects from several owners, each of them controlling his part of schema and his objects, and possibly enriching and consuming objects from other providers**. Figure 1 shows in blue color the entities concerned in parceled control and explained hereafter.

In a first step to support parceled control in schema sharing, we defined a new abstraction for *dataClay*'s data model called ***namespace***. In par-

ticular, since *dataClay* is conceived for sharing data between independent users, and a class name is not a universal identifier, classes are grouped into namespaces preventing name clashing. Any user is allowed to create a namespace for the schema that defines his data, i.e. the schema on which his applications are based.

This schema can be composed of new classes created in the namespace by its creator, as well as already existing classes (enriched or not). That is, **classes from other namespaces can be imported in a specific namespace** if the owner of such a namespace has an active model contract including interfaces for them. From then on, they can be used in the context of the namespace according to the interface specification, and **the owner of the namespace can enrich them with new properties, methods, or implementations as he does with his own classes**. Furthermore, **these new elements enriching classes potentially from other namespaces, will be managed by the owner of the current namespace, who will be able to decide with whom they are shared and how long by means of new model contracts** in the same way as he does with the original classes of his namespace. This enables integrating data from different owners.

On the other hand, **an additional way of controlling access to data is by parceling not only the schema but also the data itself in an orthogonal way**. That is, in the same way that a user may create a namespace that contains a set of classes, a user can also create a **dataset**, which contains a set of objects from different classes. A dataset is simply a container of objects, there is no direct relationship between namespaces and datasets, which provides extra flexibility. For instance, a user can share the same interface for a given class with different users, but offer a different subset of the data to each of them. The dataset creator grants access to a dataset by means of a **data contract**, specifying the expiration date and whether he gives permission to create new objects on that dataset.

As a summary, **model contracts provide control on the schema, while data contracts provide control on the data**. In other words, data contracts

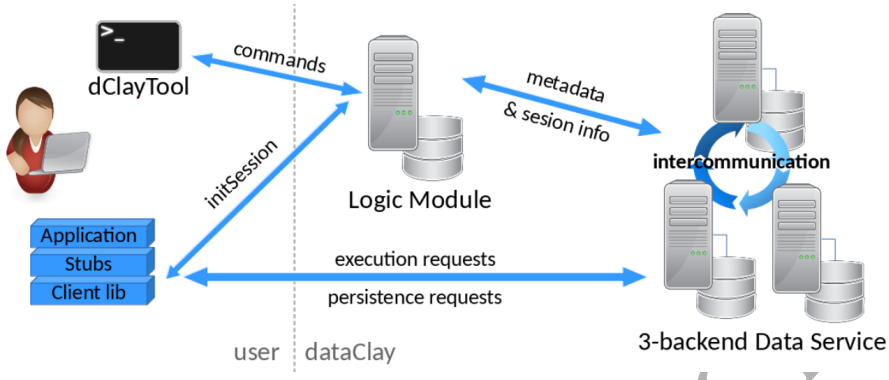

Figure 2: User-*dataClay* interactions with commands launched from *dClayTool* and session and execution requests launched from user-level applications. Logic Module and a 3-backend Data Service are deployed in different nodes with the corresponding data and metadata intercommunication.

allow providers to control which objects can be accessed by a user, while model contracts allow them to control how these objects can be manipulated. For instance, if no method to modify an attribute appears in the interface, the user will not be able to modify it. This kind of access control is very fine-grained because it can **grant write access to an attribute while only read access, or no access, to another attribute** in the same object.

3. Design and technical details

This section describes the main technical aspects for *dataClay* implementation and is divided in two parts: **Management Operations for Parceled Control** and **Application Execution**. For a better understanding, figure 2 illustrates all the components concerned and the interactions between them, and hereafter is shortly summarized before going into detail.

dClayTool is a user-level application that facilitates Management Operations related with the **data access control and the definition, evolution and sharing of the conceptual and external schemas**. Users are also provided with client libraries so their applications become enabled to open sessions and submit requests to *dataClay*.

The resulting requests from *dClayTool* are sent to **Logic Module (LM)**, a service that after authenticating the user,processes them and stores the derived information in its own database. On the other hand, application requests are sent to the **Data Service (DS)**, a distributed service deployed in several nodes (DS backends) that handles all object operations including **persistence requests**: store, load, update and delete objects in the underlying database (i.e. CRUD on objects); and **execution requests**: executing arbitrary code from methods of user classes on target objects. Persistence and execution requests can be also produced from nested methods within a single DS backend or backend-to-backend (intercommunication).

LM also handles a central repository of **object metadata**, thus DS backends send LM the object metadata of new persistent objects, or retrieve missing object metadata from LM.

3.1. Management operations for Parceled Control

Management operations are **those related with parceled control: schema registration, enrichments, and data access granting**. Schema registration includes creating **namespaces** and registering **classes**, definition of **interfaces**, and **model contracts** signing. **Enrichments** include the registration of **new attributes and methods** to existing classes, or new implementations for existing methods. Data access granting refers to **dataset** registration and **data contracts** signing.

As stated before, *dClayTool* provides the user with a command-line application to handle Management Operations. In particular, the most relevant commands are shown in table 1. Throughout this section it is assumed that *dClayTool* sends user's credentials to LM along the requests, so that LM can authenticate the user. Finally, subsection 3.1.1 describes how users retrieve the classes corresponding to their available data models, which is a key point to understand the details of next section 3.2 about Application Execution.

	Schema Sharing
1	dClayTool -newNamespace <name>
2	dClayTool -newClass <namespace> <classname> <directorypath1, 2, ...N>
3	dClayTool -newInterface <interface-filepath> <interfacename>
4	dClayTool -newContract <begindate> <enddate> <beneficiary> <interface1, 2, ...N>
5	dClayTool -importContract <contract> <namespace>
6	dClayTool -importClass <contract> <class> <namespace>
7	dClayTool -enrichClass <namespace> <enrichmentclass-filepath> <classtoenrich>
8	dClayTool -getstubs <ddl> <contract1, 2, ...N>
	Dataset Access Control
9	dClayTool -newDataset <datasetname>
10	dClayTool -grantAccess <begindate> <enddate> <dataset> <beneficiary> <permissions>

Table 1: Syntax for most relevant *dClayTool* commands related to schema sharing and dataset access control.

3.1.1. Schema registration

Given a user wanting to register a schema defined with the classes shown in figure 3 (developed in Java or Python), he uses *dClayTool* to firstly create a **namespace** that contains them naming it with a string (command 1). Logic Module verifies that there is no other namespace with the same name and registers it in its database.

Now the user proceeds to register his **classes** in the created namespace specifying the directory paths where they are stored (command 2).

At this point, ***dClayTool* analyzes the code** (source code in Python or byte-code in Java) to verify that all class dependencies are found in the given paths, and automatically registers all the required classes transparently for the user. In particular, *dClayTool* performs a dependency analysis that generates a dependency tree by looking up references to other classes, which in the implementation of the class will appear as: **types of attributes, parameters or returning values of methods, types used in local variables, and superclasses of the current classes to be registered**.

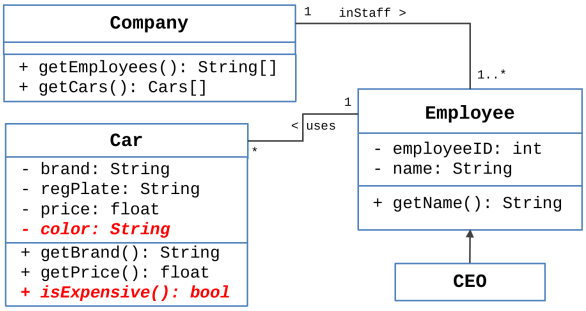

Figure 3: Schema example with different kinds of associations and enrichments highlighted. *Company* class is registered associated to a set of employees with at least one CEO. Employees may use some cars, but it is assumed that a car cannot be used by different employees.

The leaf nodes of this dependency tree are either classes already registered or language special classes (e.g. Object.class in Java). Already registered classes are not registered again, and in the case of language classes, it is assumed that they are always accessible from the programming language itself, so they are not registered either. On the contrary, non-leaf nodes correspond to user classes that are transparently registered.

For instance, assuming that all the classes are in the directory /home/user/-classes, the user executes:

```
dClayTool -newClass myNameSpace Company /home/user/classes
```

The dependency analysis detects *Employee* as a dependency, and recursively also finds *Car* and *CEO* classes, and all of them are registered automatically in *myNameSpace*. In order to resolve cycles in class dependencies, classes are marked as *in-progress* before registering their dependencies. Therefore, assuming that the implementation of class *Employee* has an attribute referring to his company and class *Company* has an attribute referring to its employees, *Company* class is marked as *in-progress* until *Employee* is registered. In this way, *Company* can be omitted when the dependency analysis finds that *Employee* depends on it, preventing *Company* from being revisited.

Eventually, all class information is sent to Logic Module that checks that

there is no class clash in the namespace and deploys the classes to Data Service (further detailed in section 3.2.3).

At this point, the user might **share his classes by registering model contracts. To this end, he creates one interface per class** (command 3), writing one XML file per interface defining the **visible methods** (with operation **signatures**) and **implementations** (with implementation **identifiers**) for each corresponding class. Notice that applications access *dataClay* objects only through methods, consequently interface definition does not include class attributes. Given that multiple interfaces can be defined for a single class, the user assigns an **interface name** to easily identify them and, in this way, also facilitates the **model contract** registration, providing the names of the interfaces for the contract, **begin and expiration dates**, and the **beneficiary** (command 4).

3.1.2. Enrichments

In line with the previous example, now the user wants to enrich class *Car* to include a new attribute for the color and a new method *isExpensive()* that returns a *boolean*, *true* if the price is higher than 20K, *false* otherwise. For the new method, the user knows that the interface of class *Car* present in his model contract includes the method *getPrice()* that retrieves the price of the car.

It might be the case that class *Car* was already registered in another namespace and the user got access to it from one of his model contracts. In this context, he would previously **import the class *Car* into his namespace**. To this end, he might either import a **whole model contract** containing an interface for class *Car* (command 5), **or only the class *Car* from a specific model contract** (command 6). As part of this process, Logic Module validates that model contracts have not expired yet and that the same class names are not already present in the target namespace of the user.

With class *Car* already present in the user namespace, either because he is its owner or he imported it, he is now enabled to enrich it. The user codes the enrichments within a regular **class *E* extending from *Car*** (or, more precisely,

the view of class *Car* according to his contracts). In particular, class *E* would define the *color* attribute and the *isExpensive()* method. In this way, this class E benefits from being a subclass of class *Car* enabling access to *getPrice()* method needed for the implementation of *isExpensive()*, and if method *isExpensive()* was already present in class *Car*, the user could use overriding techniques to redefine it with a new implementation.

Finally, the user **registers the enrichments contained in class *E*** with command 7 (class *E* is not registered, it is intended for enrichment coding purposes), and they **become part of the vision of class *Car* in his namespace**. The same **dependency analysis** used for class registration is applied, since any new enrichment may require some classes for attribute types, method parameters or return values, or local variables in new implementations of existing methods. If some class needs to be registered it is performed automatically.

Analogously to class registration, enrichment information is also sent to Logic Module for the corresponding checks **and the deployment of the enrichments** to DS backends (further detailed in section 3.2.3).

3.1.3. Dataset Access Control

As explained in previous sections, every object registered in *dataClay* belongs to a single **dataset**, so that **data contracts** granting access to a dataset are used to define common permissions for all the objects belonging to it. Therefore, management operations also include dataset registration with a unique name (command 9), and definition of data contracts (command 10) indicating the dataset for which the contract grants access, the *begin* and *end* dates of the validity of the contract, the beneficiary and permissions. Current version of *dataClay* assumes that a data contract grants the beneficiary to execute class methods on all the objects registered within the offered dataset. However, create/delete permission (*createPerm* in figure 1) is configurable to further control whether the beneficiary of the data contract may create/delete objects in the dataset or not.

3.1.4. Stub classes

Stub classes (or stubs) implement the external schema representing the particular vision of a user for the classes registered in *dataClay*, having one stub per accessible class and containing only those methods for which the user has been granted access via model contracts.

Following the example described in previous subsections, let us assume that a user is beneficiary of several model contracts including different interfaces for the same class *Car*. Now, the user is developing an application that requires accessing objects of class *Car*, so by means of command 8 he is able to retrieve a stub of class *Car* for any of the supported languages (interoperability details in section 3.2.5). This stub is generated and returned by the Logic Module as the union of the methods visible from all interfaces of *Car* included among his model contracts.

At this point, the application can be compiled and executed using the stub of class *Car* either to **instantiate and persist new car objects, or to instance references to existing car objects. In both cases, the application is then enabled to access persistent car objects through the visible methods available in the stub**.

Regarding the management of object persistence, stub classes extend from a global *dataClay* class called **DataClayObject** supplied in a **client application library**. Currently, this client library has the form of jar file *dataclay.jar* for Java and a *dataClay* package for Python. *DataClayObject* offers a set of methods that can be called from any stub instance, the most important are:

- **makePersistent (ds_backend, [alias])**: requests that the current stub instance (object) is made persistent (becoming a persistent object) in the specified DS backend. Optionally, a string alias can be provided as a user-friendly way to identify it.
- **deletePersistent**: removes the object referenced by current stub instance from *dataClay*.

In the case of *makePersistent* method (further detailed in section 3.2.1), both

parameters are optional, if the *ds_backend* is not provided *dataClay* chooses a random backend, and if no alias is given the object is only retrievable by following a reference from any other object.

If the alias is supplied, the object can be retrieved afterwards using a **static class method** called ***getByAlias*** present in all stub classes by default. This method starts by submitting the corresponding request to the Logic Module, which returns the object metadata; and then **builds a new instance** of the stub class with the returned **OID** and **locations** (which are **attributes inherited from DataClayObject** as well).

Regarding the execution of methods not related with persistence management, stub methods have a different behavior depending on whether the current stub instance refers to a persistent object or not. **If the object is persistent**, the stub method generates an execution request for Data Service containing any possible parameters for the execution of the method. But on the contrary, **if the object is not yet persistent**, the stub method must be executed locally (that is, in the context of the application). To this end, the stub methods are provided with one of the method implementations available according to the model contracts of the user. Since there may be several implementations available from different contracts, the LM resolves the collision **when generates the stub class by assuming that model contracts are sorted by priority order** in the *getStubs* command, with the first model contract having the highest priority. Therefore, the user has the possibility to affect the choice of the implementation to be eventually executed among his accessible ones.

In order to make stub methods aware of whether the target object is persistent or not, *DataClayObject* exposes a boolean attribute called *isPersistent*, which is set to true either after the execution of *makePersistent* request, or from a return value of a stub method containing references to existing persistent objects, or when using the *getByAlias* method.

Last but not least, stub classes also contain the implementation of two private methods (not visible for the application) inherited from *DataClayObject*:

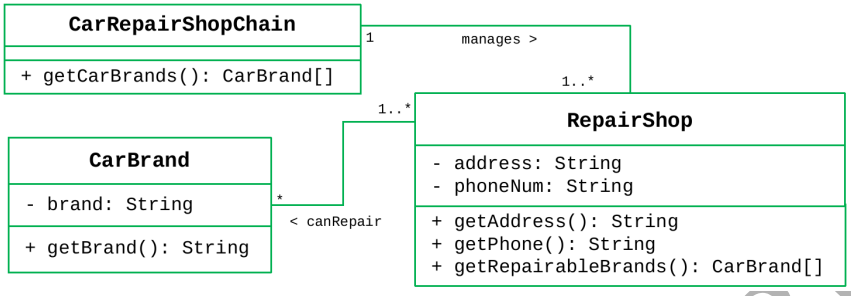

Figure 4: Data model of the chain of car repair shops.

serialize and deserialize. The serialize method prepares **a byte-array for the binary representation of the attribute values of the stub instance**, so it can be transferred within an execution request as the parameter of any stub method or for a *makePersistent* call. The deserialize method is used to build a local object from its binary representation (serialized) coming **from the return value** of a stub method.

3.1.5. Data sharing and integration examples

Following the company data model presented in section 3.1.1, let us suppose that there is an important chain of car repair shops that signs an agreement with the previous company, so that all cars of the company will be repaired by the car repair chain. The data model of the car repair chain is illustrated in figure 4. *RepairShop* class represents the information related to car repair shops (address and phone number), and it is related to *CarBrand* class which represents the information of all the covered car brands, i.e. cars that can be repaired in the available repair shops. Due to the agreement between the two companies, both models should be integrated to provide a unified view of their data. With *dataClay*, this integration can be done at different levels, ranging from a loose integration in which only the company uses *dataClay*, to a tight integration in which all the data is stored in the same datastore.

In the first scenario, databases of the chain and the company are managed in different infrastructures, and the chain does not use *dataClay* but offers a

Data Service to consult the repair shops given a car brand. In this context, a developer of the company codes a method for *Car* class connecting to this Data Service to get available car repair shops using the car brand information from the class. This method is used every time a car needs to be repaired to get the address and phone number of an available repair shop for the broken car, thus the integration is on the basis of the interoperability between the company and the existing chain services. Although data migrations are avoided, the drawback is that this approach requires data transformations to perform Data Service dependent requests and responses.

A tighter level of integration can be achieved if the chain also uses *dataClay* to store its data. The chain starts creating a model contract to grant the company access to the method *getRepairShops(String carBrand)* thus providing a functionality analogous to the previous Data Service. The model contract also includes an interface for *RepairShop* class with access to getter methods for the phone and number attributes. Finally the chain creates a data contract to grant the company access to the object of *RepairShop* class. In this context, the developer of the *Company* gets the stubs corresponding to the model contract and codes the method to get available repair shops using them. In this occasion the external communication with the infrastructure of the chain is done transparently through *dataClay* stubs, preventing user-level transformations.

Finally, full integration can be achieved if both the company and the chain store their data in the same data store, for instance externalizing it to a *dataClay* service in the cloud. They use different namespaces for their data models and different datasets for their objects to ensure that their data is isolated unless they want to share it, but still want to integrate their data without losing control (as shown in figure 5).

In this context, the company creates a model contract granting the chain access to *Car* brand name attribute through its getter, plus the *getCars* method of *Company* class; as well as a data contract to offer its dataset. With these contracts, the chain **enriches the *Car* class with a new relationship attribute *availableRepairShops*** and codes an application that matches com-

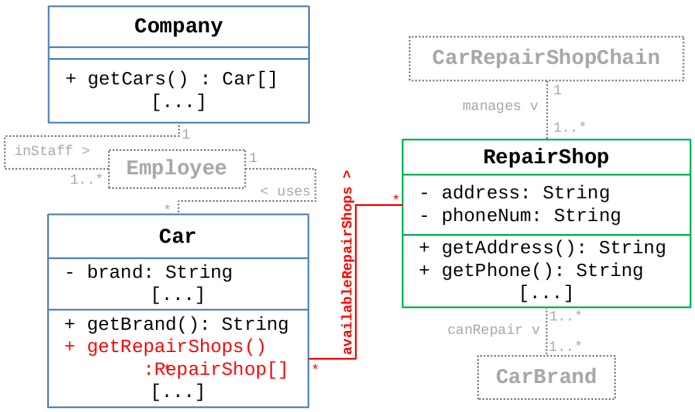

Figure 5: Schema of the data integration between the company and the chain of car repair shops. Classes and associations in dashes means that are hidden for the non-proprietary of these entities. In red, the enrichments added to the company data model to get repair shops when needed.

pany cars with chain *CarBrand* information, filling the *availableRepairShops* with references to *RepairShop* objects. Now the chain creates a model contract analogous to previous scenario but includes the getter method of *availableRepairShops*, thus the company application uses it to navigate from car objects to their corresponding repair shops.

This approach enables the company to access up-to-date information from the chain (as in the first scenario), but also prevents user-level data conversions (as in the second scenario). With encapsulation and *dataClay* parceled control, both the company and the chain are enabled to isolate the parts of the schema and data that do not have to be visible from the non-proprietary party. Enrichments facilitate the integration between both data models, letting the chain to adapt *Car* class to its needs and adding new functionality accessible for the company.

3.2. Application Execution and Object handling

This section details how the user-level applications actually interact with *dataClay* to manage **object persistence and stub method execution** on

existing objects. It is assumed that the schema presented in previous section has been successfully registered and the owner of the namespace has defined a model contract to **share his classes with** a certain user. The user beneficiary of such a model contract has downloaded the corresponding stub classes and develops two applications linked with them:

- *AppInsertCompanyData* (figure 6): creates and stores the data of the employees in a company.
- *AppGetCompanyEmployees* (figure 7): retrieves the names of all the employees of a company.

Analogously to the user authentication for Management Operations, applications wanting to interact with *dataClay* also need to perform an authentication process that, in this case, is performed through a **session-based mechanism detailed in section** 3.2.4. Throughout this section it is assumed that the application is already authenticated and has a session identifier for its execution requests.

3.2.1. Data generation

When executing *AppInsertCompanyData*, it starts creating local instances for the stub classes of *Employee* and *Car* (with their corresponding associations). In order to illustrate different ways to persist objects, the application calls *makePersistent* for each employee which transparently makes the associated car to be persistent too, i.e. serializing both car and employee data to be sent within the request. In contrast, *makePersistent* for the company object omits serializing employees data since at this point they are already references to persistent objects, i.e. OIDs. Thus, only these OIDs are serialized along with the company name (which is also used as the **alias** for *myCompany* object) as the state of *myCompany* object. In all cases, no particular DS backend is provided for *makePersistent* requests, so they are submitted to a random one based on a hash function. The request includes the **session ID for the proper checks and to infer the dataset** where objects are registered (more details about

```
public class AppInsertCompanyData {
  public static void main (String[] args) {
    String user = args[0]; String pass = args[1];
    String datasets[] = { "myCompanyDataset" };
    String datasetStore = datasets[0];
    dataClay.initSession(user, pass, datasets, datasetStore);
    List<Employee> employees = new LinkedList<Employee>();
    int employeeID = 0;
    for (int i = 2; i < args.length; i = i + 2) {
      String employeeName = args[i];
      Float carPrice = new Float(args[i+1]);
      Car c = new Car(carPrice);
      Employee emp = new Employee(employeeName, employeeID, c);
      emp.makePersistent(employeeName); // remote request
      employees.add(emp);
      employeeID++;
    }
    Company myCompany = new Company("myCompany", employees);
    myCompany.makePersistent("myCompany"); // remote request
  }
}
```

Figure 6: AppInsertCompanyData Java code that initializes a session and generates persistent objects corresponding to the employees of the Company and their cars.

session management in section 3.2.4). As objects are stored in the assigned DS backends (further detailed in section 3.2.3), they send the corresponding object **metadata to Logic Module (LM)**.

LM keeps an up-to-date metadata repository, equivalent to the database catalog, with the following information per *dataClay* object: **OID**, a Universally Unique Identifier (UUID) generated when the stub class is instantiated; **dataset**, for permission checks; ***ds_backends***, to know the locations of the object; and finally **aliases**, in order to resolve *getByAlias* requests.

```
public class AppGetCompanyEmployees {
  public static void main (String[] args) {
    String user = args[0]; String pass = args[1];
    String datasets[] = { "myCompanyDataset" };
    String datasetStore = datasets[0];
    dataClay.initSession(user, pass, datasets, datasetStore);

    // Initialize stub instance comp using the company alias
    Company myCompany = (Company) dataClay.getByAlias("myCompany");

    // remote execution request
    String[] employeesNames = myCompany.getEmployees();
    for (String employeeName : employeesNames) {
      System.out.println(employeeName);
    }
  }
}
```

Figure 7: AppGetCompanyEmployees Java code that initializes a session with the dataset where Company objects have been previously registered, and after retrieving the reference to myCompany object obtains its employees' names through remote execution request.

3.2.2. Remote execution

With the objects already stored, the user executes *AppGetCompanyEmployees*, which starts retrieving the Company object from its alias *myCompany* by means of the static method *getByAlias* (introduced in section 3.1.4) accessible from Company stub class.

From then on, stub methods of Company behave as Remote Procedure Calls (RPCs) for the *myCompany* object, and the resulting **execution requests** are submitted to one of the DS backends where the object is located, including the **parameters for the method and the session information, which are serialized in TCP packets in binary format** for the underlying binary communication protocol used in *dataClay* (e.g. in Java, client library uses Netty [12] framework). Types of parameters might be **literals, language classes,**

or user registered classes. Both literals and language classes are serialized to *dataClay* compliant types in order to translate their contents to a common binary representation, whereas parameters of user registered classes use the serialize method from the corresponding stub classes. In case of persistent objects, only the OID is serialized. However, in the example no parameter is needed and the application directly calls *myCompany.getEmployees()*.

At this point, DS backend DS1 receiving the resulting execution request **validates the session** and, considering permissions of the corresponding data contracts, checks that *myCompany* object is actually accessible. To this end, DS backends are allowed to access to **session information and object metadata from Logic Module**, that are requested on-demand (**when they are missing or out-of-date**) and saved in internal **LRU caches** in order to improve performance avoiding subsequent requests, and to prevent Logic Module from becoming a bottleneck.

With session validated, DS1 loads *myCompany* object in its memory context and executes *getEmployees()* method. To this end, DS1 has so-called **execution classes analogous to stub classes** used to load objects from database to memory or to update objects from memory to database. In both cases, analogous serialization and deserialization mechanisms as for the communication protocol are used (further detailed in section 3.2.3). Therefore, *getEmployees()* method iterates through the accessible Employees from *myCompany* object and for each employee object *emp* executes its method *getName()* to eventually return all employees' names.

It might happen that some of the objects are not present in DS1, so instances of *Employee* execution class are actually referring to objects stored in other backends. In this case, DS1 generates requests for *getName()* to other DS backends in the same way as an application does. That is, execution classes use the same serialization and communication protocols as stub classes and the **session identifier is propagated** so the target backend is also enabled to **check the corresponding dataset permissions** for the required employee objects.

On the other hand, employee objects already present in DS1 are **loaded in the same memory space** as *myCompany* object (which is also present in DS1) **and the references from *myCompany* to employee objects will be materialized into native language references**. In **Java**, objects will meet in the heap of the Java Virtual Machine (JVM), and in **Python**, objects will be mapped and navigable within the memory space of the interpreter. In this way, execution workflow is analogous to a user-level application with a single memory space for the objects present in the same DS backend.

Consequently, the JVM heap memory acts as an object cache and objects are present there until the **Garbage Collector** "decides" to remove them. In this case the ***finalize()*** method, (present in all Java objects from Object.class) is called, and in *dataClay* it is overridden through *DataClayObject* class in order to propagate any missing updates to the database before the object is actually removed. Analogously, in Python the method ***__del__()*** (present in all Python objects) is executed when the **reference count** reaches zero, meaning that the object is inaccessible and can be deleted from the main memory, thus this method is overridden to update the values in the database before the object is actually deallocated.

Finally, all employees' **names are serialized and returned** from DS1 to the application *AppGetCompanyEmployees*. In this case, the returning value is an array of literals (strings), so the *getEmployees* method of the execution class serializes all the strings in binary format and produces a return message containing them along with the size of the array, so the stub method *getEmployees* executed from *AppGetCompanyEmployees* deserializes the array and the execution is resumed.

3.2.3. Execution classes - object handling

Execution classes are used in DS backends analogously **to user stub classes** at application level. That is, an execution class is instantiated in order to load objects in memory and has the serialization and deserialization functionalities to update or read objects from the underlying storage or to transfer them

either to client applications or other backends (as a return value or parameters). Moreover, execution classes as well as user stubs, contain the proper code to execute RPCs on the methods that involve remote backends (when accessing objects not stored in the current backend).

However, there is a major difference between an execution class and a user stub, which is that the former contains the **whole set of attributes and methods** of the corresponding registered class (including all its enrichments) while the latter is only based on certain user's model contracts. This is a requirement, since DS backends must be able to execute any possible method of any class, and are not specialized to store objects instantiated from any particular stub class. To this end, whenever a class is registered or enriched, **Logic Module deploys the corresponding execution class to all DS backends** which store it as a regular class file in their local **filesystems** separated by namespaces, i.e.one directory per namespace.

In the case of enrichments, the deployment process updates the corresponding execution class stored in DS backends. Objects already stored with previous versions of the class are loaded considering that any possible new attribute is initialized to default value of its type. When objects are updated in the underlying database, values of the new attributes are also stored (if any). Consequently, **objects are eventually in the form of the newest version of the class** without breaking any constraint while they are still in the old form.

In Java, classes are managed via ***ClassLoaders***, which are part of the Java Runtime Environment and dynamically load Java classes into the Java Virtual Machine. In *dataClay*, DS backends handle one *ClassLoader* per namespace and load the corresponding execution classes from the underlying filesystem on demand. Given that *ClassLoaders* cannot reload a class dynamically, there might be objects in memory being instances of early loaded classes prior to enrichments. In order to overcome this issue, *dataClay* reacts to enrichments by pausing upcoming execution requests that depend on newer versions of loaded class, waits to in-progress execution requests to finish (that were using previous versions of the classes), and when they finish creates a new *ClassLoader* instance

to load newest classes and resume the previously paused execution requests. Notice that this will only affect the objects instancing classes of the namespace of the enrichments, and all other objects will see no delays in their execution . However, in order to avoid pausing execution requests in highly utilized classes, *dataClay* can be configured to apply enrichments once a day (at a particular hour).

On the other hand, **in Python**, there is no exact counterpart for the concept of *ClassLoader* and two objects might be loaded in memory as instances of different versions of a class while still being considered of the same class. The only problem is the built-in *isInstance* function, which is expected to return the same result regardless the version of the class, but stub classes override this method to make it behave as expected.

Regarding the mapping of objects to the underlying storage, DS backends store the serialized objects (with **serialize** method) coming from *makePersistent* requests (from a stub or an execution class) directly to the database. That is, objects are stored in the database without having to deserialize them in memory, in a table with two columns using the OID as the primary key and a byte-array for the object data (all other attribute values or references). Analogously, when an object is read from the database, the **deserialization** method in the execution class is used to load it in the **backend execution memory context**: heap of the Java Virtual Machine in Java, or the memory space of the Python interpreter in Python.

Similarly if a **method execution** (different from *makePersistent*) requires sending an object as a return value or parameter **from one backend to another** (return value or parameter of an execution request) the source backend uses the **serialization method** and transfers the serialized object to the target backend. Then, the target backend deserializes the object with the **deserialization method** and loads it in the corresponding execution memory context. **If it is a persistent object only the OID is serialized/deserialized (as a reference)** thus reducing data transfers significantly.

3.2.4. Session Management

In the context of the *dClayTool*, the credentials are always included within the requests so that Logic Module authenticates the user for every management operation. On the contrary, in the context of a user-level application where the execution time is critical, we have implemented a **session-based authentication mechanism** in order to authenticate the user only once for the lifetime of the application. In particular, the application uses a client library provided with *dataClay* that offers the following function:

```
dataClay.initSession(user, password, datasets, dataset_for_store)
```

This method submits a request to Logic Module, which validates the user and checks that he can access the provided **datasets** from any of his **current data contracts** (not expired ones). The user also defines one of these datasets as the dataset used by default in the session, meaning that all the objects made persistent in the context of this session will be registered in the indicated dataset (***dataset_for_store***). Finally, LM generates a **session identifier** (***sessionID***) which is returned and saved in the client library and, from then on, it is sent serialized within the data of any upcoming client requests (e.g. along the parameters of a stub method execution). LM infers the **session expiration date** by taking the most restrictive expiration date among the data contracts corresponding to the datasets specified for the session.

3.2.5. Interoperability

With the abstraction of the conceptual schema, *dataClay* allows to retrieve stub classes in any of the supported languages regardless the language used to code their corresponding registered classes. This means that, for instance, a class implemented in Python, as well as its existing objects, can be used from a Java application, and vice versa in transparent way to the programmer.

In previous example, let us assume that the user who registered the conceptual schema coded the classes in Java, registered them, and created an interface of Company class to offer the method *getEmployees()* through a model contract.

Its beneficiary codes *AppGetCompanyEmployees* either in Java or in Python by previously downloading the Company stub class in the required language. The stub instance *myCompany* of Company class is accessed via *getByAlias* method, and the application will access it as a Java object or Python object, so that *getEmployees()* is accessible in the same way because internally the string array of Employees names will be deserialized taking the language into account. This is plausible because string arrays are one of the types supported to be serialized and sent from the Java execution class of the DS backend (where *myCompany* object is actually located) to be deserialized in the Python stub class of Company for the application.

Beyond string arrays stated in the previous example, *dataClay* currently **supports the translation between any basic type** (integer, boolean, float, etc.), **arrays of basic types**, and equivalent **built-in types in both languages** (e.g. *LinkedList*), **and user-defined classes**. However, other complex types with a non-straightforward equivalence (e.g. *ConcurrentLinkedQueue*) cannot be converted from one language to the other in our current implementation. Consequently, the retrievable stub methods for a language different than that of the corresponding class, are those having parameters and return values compatible between both languages and supported by *dataClay*.

As described in section 3.1.4, stub methods are provided with one of the available implementations for local execution (i.e. for method calls made while the object is not persistent). This means that, in addition to serialization of parameters and return values, another issue to be faced is code translation. Today, this is a work-in-progress feature, so stub classes generated in a language different than the original class do not contain implementations for their methods. Consequently, these stub classes cannot be used to create local instances but to refer to already persistent objects. However, *dataClay* provides an intuitive way to persist new objects from these stub classes. In particular, these **stub classes are supplied with a customized implementation of the constructors that produce a remote execute request for the original constructor and immediately after a *makePersistent()*** call is submitted to persist the

object. From this moment, the object can be accessed as any other persistent object with the only limitations about serialization compatibility commented before.

3.2.6. Concurrent execution

It might be the case that simultaneous execution requests are targeting the same object (e.g. from different user level applications). In this context, *dataClay* allows concurrent object access as in a regular multi-threaded application. That is, *dataClay* does not force any locking or transactional mechanism, but the schema developers are allowed to use any built-in mechanisms of the programming language to fulfill their concurrency requirements. For instance, in Java, a user might register a class using the *ReentrantLock* built-in type to force a reentrant mutual exclusion behavior on a particular piece of code within a method; and in Python, the *threading* module of the standard library offers analogous mechanisms to control concurrent execution. This prevents users from having to pay the penalty of concurrency control in those cases where it is not required by their applications.

4. Evaluation

This section presents two evaluation studies. The first study is conducted with Yahoo Cloud Serving Benchmark (YCSB [13]), a well-known benchmark in recent literature [14] that allows us to compare *dataClay* with popular NoSQL solutions in terms of latencies and throughput. The second study shows how *dataClay* improves the performance in the execution of data analytics applications implemented as *Map-Reduce* workflows.

4.1. CRUD operations - Latencies and Throughput

First of all, we want to prove that besides novel features presented along the paper, *dataClay* can be compared in terms of I/O performance with other trendy databases. In particular, we show a comparison between *dataClay*, Cassandra

(version 2) [8] and MongoDB (version 3) [15]; both in terms of throughput (operations per second) and latencies when performing read and update operations. We chose Cassandra and MongoDB because they are two of the topmost popular NoSQL databases [16] and have become key technologies to resolve common storage issues in Big Data scenarios [17] [18]. Finally, we also present an analysis on the potential overhead produced by enrichments, a key *dataClay* feature for effective data sharing.

4.1.1. Methodology - YCSB

The kernel of YCSB has a framework coded in Java with a workload generator that creates the test workload and a set of workload scenarios. YCSB assumes that every DBMS to be tested exposes a table where each record is represented by one string field as the primary key and a set of byte-array fields (values). The workloads of YCSB are characterized by the following main options: number of records (for the table), number of operations (to be executed on records),percentage of operations of each type and access distribution (zipfian, uniform, etc.). A *Zipfian* distribution is the most commonly used [19] since it represents a realistic behavior where some records are more popular than others, therefore is the default distribution and the one we used in our tests.

Once a workload is defined, YCSB launches a multi-threaded workload executor that calls the operations to the database following the workload specification with equal load-balancing between the threads. To this end, the workload executor relies on an abstract class called DB that defines a set of methods to be implemented by specific driver classes, having one driver class per database, i.e. each driver overrides DB methods with its database-dependent implementation.

In the case of Cassandra and MongoDB, YCSB already provides the corresponding drivers to execute the benchmark, and for *dataClay* **we implemented our own driver analogous to the other two**. Every 10 seconds, YCSB outputs stats on global throughput in operations/second (aggregating throughputs from all threads) and the latency per operation in microseconds.

In our tests, we used a cluster of 4 nodes interconnected with a 10-Gbit

Ethernet switch (0.3 milliseconds RTT) and each node equipped with 16 Intel Xeon processors with 24GB of DDR3 RAM. We dedicated **1 node for the client** running the YCSB workload executor, and **3 nodes for the database services** (i.e. 3 nodes for distributing the records of the table) to deploy a distributed environment with all the different actors for each DBMSs, in *dataClay*: LM and 3 DS backends (LM sharing a node with one DS backend); in MongoDB: 1 *config* server, and 3 *router+shard* servers (*config* server is in a shared node); and in Cassandra: 1 seed and 2 non-seed nodes. In the case of *dataClay* we used PostgreSQL in the DS backends for the internal level representation, i.e. where objects are eventually stored. Given that YCSB uses only one node for the client-side, spreading data in more than 3 nodes did not produce any significant effect in the performance since the amount of concurrent requests is limited.

We specified two main workloads, workload *WR* and workload *WU*, to analyze the performance outcome in basic data I/O operations. *WR* is based on *workloadC* of the benchmark, focused on **read operations**. In contrast, *WU* only performs **update operations**. Both workloads are configured to execute 1,000,000 operations on a distributed table of **1,000,000 records**, and each record with **1,000 bytes** in size distributed in 10 fields. The values in each field are random strings of ASCII characters, 100 bytes each. The workload executor uses up to **16 threads** (number of CPUs) and all threads execute the same amount of operations, thus we have: 1 single thread executing 1M operations (**1*1M**), 2 threads running 500K operations (**2*500K**), 4 threads with 250K operations each (**4*250K**), 8 threads doing 125K (**8*125K**), and 16 threads performing 62,500 each (**16*62,5K**).

4.1.2. Comparison with NoSQL databases

YCSB binding for Cassandra assumes that there already exists a table (which is created using CQL) on a specific *keyspace* containing 11 *varchar* fields, one for the primary key and the other for the values. In the case of MongoDB, records are embodied in BSON documents grouped in a collection that represents the

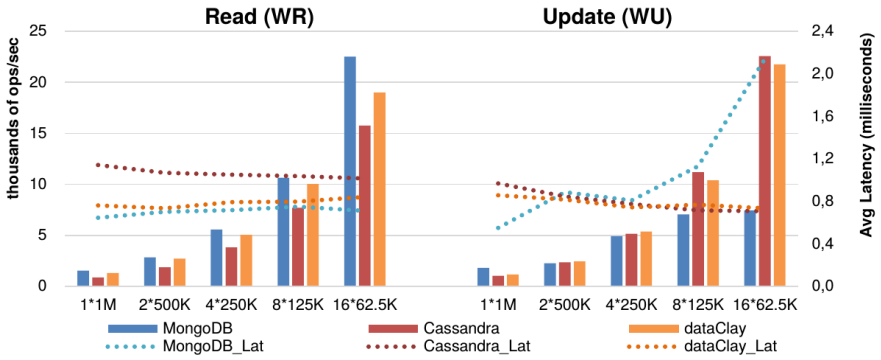

Figure 8: Results of WR and WU workloads. In bars, throughput in thousands of ops/sec (left axis); in pointed lines, latencies in milliseconds (right axis). X-axis shows #threads * #ops per thread of the evaluated subcases.

table; each document is identified by the key field and is filled with 10 byte-array entries for the values. In *dataClay*, every record is an object instancing a user-defined class with 10 byte[] attributes for the values, and the key field is stored as the object alias.

The results obtained for workloads *WR* and *WU* are presented in figure 8 showing throughputs and latencies for all the cases stated in previous subsection.

In the case of read-only workload WR, **all DBMSs present a linear scalability and *dataClay* achieves between 33% and 17% better throughput than Cassandra, but performs between 10% worse than MongoDB**. This is also reflected in latencies, where MongoDB keeps values between 600 and 800 microseconds, *dataClay* between 700 and 900, while Cassandra achieves latencies around 1 millisecond. It is worth mentioning that early results on Cassandra were similar to *dataClay* and MongoDB when using *cassandra-10* client binding of YCSB, but it turned out that Thrift API used in this binding was deprecated and *cassandra2-cql* must be used instead (CQL API), which obtains the presented results. Anyhow, the maximum performance is obtained in *dataClay* and MongoDB with variations produced due to differences in serialization (different ways to represent data) and communication protocols.

On the other hand, in update-only workload *WU*, MongoDB achieves best results in the single thread execution but tends to increase its latencies and gets stalled when the amount of threads is greater than 4. This is due to reader-writer locks used in MongoDB, which allow concurrent readers accessing to a collection of documents, but force exclusive access in write operations. Consequently, *zipfian* distribution, which makes some objects or documents particularly popular, tends to penalize MongoDB. On the contrary, **Cassandra and *dataClay* achieve similar results outperforming MongoDB when using 4 to 16 threads**, and present again an almost linear scalability. In the case of *dataClay*, this is a consequence of having no implicit control or locking for concurrent accesses to objects (as explained in section 3.2.6). In Cassandra, this is due to its eventual consistency in transactions and concurrency granularity at row level, and also because Cassandra *upserts* (operation for insert or update) do not need to read rows before updating them. Therefore, Cassandra ends up obtaining results for WU 25% better than *WR*.

4.1.3. Enrichment overhead

One of the key features of *dataClay* is the possibility to enrich data models, and thus it is important to evaluate the overhead of using such mechanisms. Especially because many enrichments may be performed without the data user being aware of them.

Figure 9 compares the original *dataClay* scenario presented in previous results with other scenarios based on enrichments. In particular, the class for record objects is now registered empty and enriched afterwards with the ten byte-array attributes. Enrichments can be applied in several enrichment steps until the final class represents the record schema. In particular, figure shows the cases for one (*Enriched1*), two (*Enriched2*) and five (*Enriched5*) enrichment steps. Thus, *Enriched1* means that a single class extending from the record class has been used to specify all missing 10 byte-array attributes. *Enriched2* means that the record class has been enriched first with 5 byte-array attributes, and afterwards with the other 5. Thus *Enriched5* means that the record class has

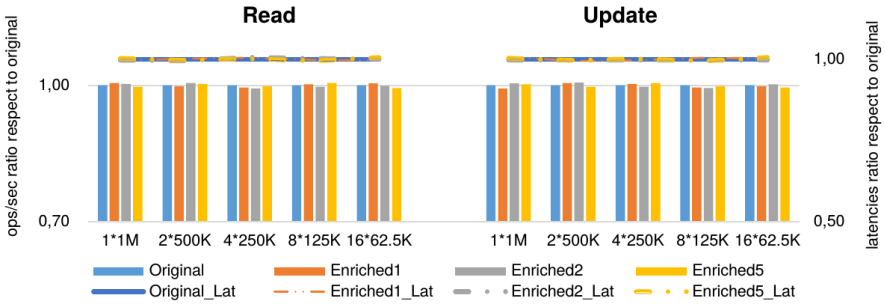

Figure 9: *dataClay* with and without enrichments for WR (left) and WU (right) workloads. Bars represent the ratio of the throughput respect to the original *dataClay* execution (left axis), lines represent the ratio of the latencies (right axis). X-axis shows #threads * #ops per thread of the evaluated subcases.

been enriched 5 times with 2 attributes in each case.

Results show that both the **throughput and latencies are almost the same** in all the evaluated cases, which follows that using enrichments incurs no extra penalty and no matters how many enrichment steps are used to enrich a class. This was the expected behavior since the execution class resulting after deploying all the enrichments has exactly the same code as the execution class deployed from the record class registered with all the attributes from the very beginning.

The only penalty to be considered regarding enrichments is produced when an object created with a previous version of the class is loaded for the first time with a newer version of the class containing new attributes. These new attributes must be initialized to fulfill the new schema, basic type attributes to the corresponding default values and non-basic types (references) to null. However, the elapsed time to initialize an attribute (in 1K executions on one of the cluster nodes) is about 2 microseconds in Java and 1 microsecond in Python, which are negligible times unless there are hundreds of enrichments being applied simultaneously.

4.2. Map-Reduce applications

In this section we present a weak scaling study on the execution of two well-known data analytics algorithms implemented in Java as *Map-Reduce* workflows: Wordcount and K-means. We coded both of them following a parallel programming model to separate *Map* methods from *Reduce* methods. In this way, we measured the elapsed times required to compute the applications in different scenarios, and compared them with the results obtained when using *dataClay* as the execution environment and underlying data store (i.e. following the processes described in 3.2).

The experiments were conducted in the *Mare Nostrum III* supercomputer [20] using 16-CPU nodes with 32GB of DRAM each (2GB DRAM per CPU) interconnected via an FDR10 Infiniband network. The applications are executed using COMP Superscalar (COMPSs [21]), a framework developed in the Barcelona Supercomputing Center that provides a master-worker runtime to orchestrate parallel workflows considering task/method dependencies and available computation resources. In this way, applications are managed from a master thread (in a dedicated node) which submits their parallel tasks to worker threads, mapping one worker thread per CPU.

4.2.1. Wordcount

Wordcount is a classical algorithm to count the appearances of all the different words in a file or a set of files. The application parses the input files splitting their text lines into words and maintaining a data structure to keep a counter per word. It finally produces an output presenting the final counters for each unique word.

The application is implemented with a *Map* stage comprising the parallelization of word counting in a per file basis (one task per file). In the *Reduce* stage, all the partial results obtained from the *Map* tasks are put together following a binary tree strategy to combine all partial results. That is, results are combined in pairs so that all files processed by a worker are locally merged, then workers work in pairs to combine their results, and finally the last result (the

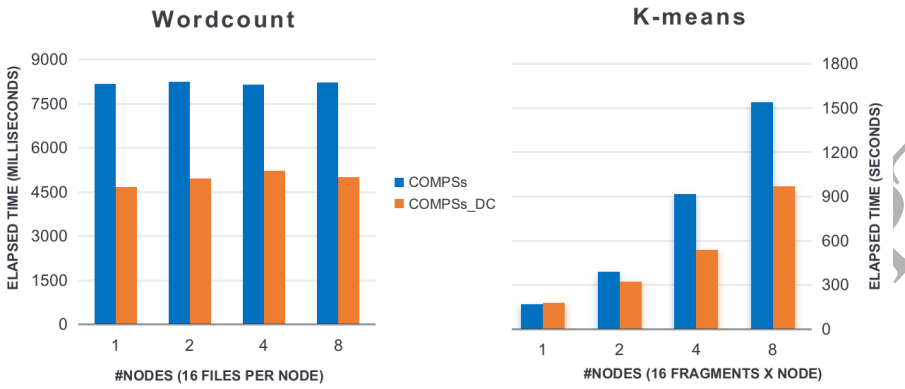

Figure 10: Weak scaling study of the integration of COMPSs with *dataClay*. Y-axis represents the elapsed times, in milliseconds for Wordcount, in seconds for K-means. X-axis shows the number of nodes. In Wordcount, 16 files of 256MB are processed per node. In K-means, 16 fragments of 12800 100-dimensional vectors are processed per node.

tree "root") is sent to the master (application executor).

When using *dataClay* files are no longer necessary. Instead, their contents are mapped to persistent objects instancing a *TextFile* class, which represents the text of a file as a list of strings. Given that texts are represented as objects of a user class, *dataClay* allows to define methods for them. Therefore the method *Map<String, Integer> wordCount()* is provided from the *TextFile* class, which is called from COMPSs workers to compute the result of each text object within the *dataClay* execution environment.

Left side of figure 10 illustrates the elapsed times (in milliseconds) obtained in Wordcount evaluation with 1, 2, 4 and 8 nodes. Every node computes 16 files of 256MB each (4GB per node), achieving a parallelization of 1 file per computing unit, i.e. one *Map* task per CPU. The *Reduce* stage is negligible in comparison to the *Map* stage since texts have been created with a Lorem Ipsum word generator of 400 words, thus every partial result (a file wordcount) produces a short map of 400 counters. This is intended to focus the problem on the impedance mismatch issues, whereas next section covers the effects of data serialization and inter-node communication (with K-means evaluation).

Results show that using *dataClay* instead of files, the execution is boosted reducing the elapsed time up to a 43%. This is due to the lack of data transformations since *dataClay* works directly with the text stored within a *TextFile* object. Without *dataClay*, although files are cached and accessed in parallel (Mare Nostrum III uses GPFS [22]) every read operation incurs one I/O system call producing the observed impact in the execution times.

4.2.2. K-means

K-means clustering is a method of vector quantization that is popular for cluster analysis in data mining. The algorithm partitions N observations represented as multi-dimensional vectors into K clusters, in which each observation belongs to the cluster with the nearest mean. These cluster means are also represented as a multi-dimensional vector that serves as a prototype of the cluster.

A typical implementation of the K-means algorithm [23] uses an iterative approach that starts defining K random means in the range of the observation values, and recomputes them in every iteration until it converges up to a certain epsilon value (sum of variances) which represents the maximum tolerated distances between observations and the final means. The problem is NP-hard so the number of iterations is limited to a maximum amount regardless of the epsilon value. In this context, the *Map* stage of each iteration performs the computation of all the distances between the N observations (grouped in fragments to enhance parallelism) and the current selected K means, while *Reduce* stage gathers the resulting values and adjusts the K means for the next iteration. At the end of each iteration the algorithm checks whether the result has already converged or the maximum number of iterations has been reached. In either case it finishes and returns the final K means.

Unlike in the Wordcount study, we now focus on the problems derived from the communications produced in the *Reduce* stage when gathering the partial results obtained in the *Map* stage. To this end, instead of parsing files, our K-means tests (with and without *dataClay*) start creating a random set of values on behalf of the N observations to be clustered. These N observations are

fragmented into objects of the same size, forcing the *Map* stage to comprise one fragment per core (CPU) to best exploit the parallelism. Fragments are represented as Java objects instancing a *Fragment* class that contains, for each observation in the fragment, one double-typed array storing the values of each dimension.

Whent no using *dataClay*, fragments are created in COMPSs workers memory heap so that *Map* stage is fully parallelized having one computation task per fragment (i.e. one task per CPU). The *Reduce* stage is executed combining all partial results in pairs as a binary tree that starts from its leaves up to the root. The "root" or final result is then gathered by the master node to normalize it and start the next iteration.

When using *dataClay*, fragments are represented as persistent objects of the *Fragment* stub class and are distributed analogously along the Data Service backends (one DS backend per node). Partial results are also persistent objects to reproduce the same behavior in the *Reduce* stage.

Right side of figure 10 illustrates the elapsed times (in seconds) of the K-means computation with 1, 2, 4 and 8 nodes. Every node computes 16 fragments, thus having 16, 32, 64 and 128 fragments respectively. Each fragment represents 12800 different points (observations), and each point is a 100-dimensional double-typed array, resulting in 10MB per fragment. K is set to 1000 clusters and the maximum amount of iterations is set to 10. All 10 iterations are computed since we configured the epsilon value to be extremely low (0.0001), so we can focus on the serialization problem of the *Reduce* tasks of every iteration.

The impact of inter-node communications is revealed considering the bad scalability shown in the results. Unlike Wordcount application, K-means requires transferring a larger amount of data between nodes for the *Reduce* stage (which in addition is repeated 10 times, once per iteration). However, results show that thanks to serialization mechanisms provided by *dataClay* through stub/execution classes we can outperform Java serialization mechanisms used to transfer the partial results. The stub classes resulting from registered classes an-

alyze class attributes to create a customized serialization of the objects, whereas Java serialization uses reflection techniques that penalizes the *Reduce* stage resolution. For this reason, when only one node is used the difference is almost irrelevant since no communications (and thus, no serialization) are actually performed, but on the contrary, when 2, 4 or 8 nodes are involved in the computation *dataClay* boosts the execution times achieving an improvement of up to a 39%.

5. Related Work

Current solutions for **efficiently** sharing data in a **flexible** way, **without jeopardizing** data providers' constraints, are limited for the following reasons:

- Providers have no easy way to restrict how the consumers **access data** (which specific methods or implementations are accessible and for whom).
- Providers cannot enable the consumers to **enrich existing schema** (with new attributes, methods or implementations) while keeping full control and different rights per consumer.
- Developers have to understand and deal with **different data representations**, as well as the mappings between the persistent and the non-persistent environments.
- Some storage systems or DBMSs cannot execute stored procedures or processing methods to exploit **data locality** and prevent data from leaving the storage infrastructure.

5.1. Security models

Current Database Management Systems (DBMSs) have a centralized security management system based on the Database Administrator (DBA) who is in charge of granting certain rights to the users registered in the DBMS. To this end, Role-based access control (RBAC) is the mainstream security model in both SQL databases like Oracle [24] or Microsoft SQL Server [25] and NoSQL

databases like Cassandra [8] or MongoDB [15]. RBAC is normally based on a set of rights profiles offered by the DBMS (in some cases can be customized by the DBA) that define special permissions. These profiles can be grouped as DBA deems suitable by defining *roles*, so when users need some of the capabilities to perform a particular operation requiring special privileges, DBA assigns them the corresponding roles.

The drawbacks of this approach are the **centralization of the system** and the **importance of the DBA**. The security management is centralized because all the constraints rules or permissions are checked and validated in the core of the system. Of course this can be an advantage for a Data Service or DBMS that acts as the unique data provider, but when dealing with a system where every user is a potential model and/or data provider, the centralization would incur an ineluctable bottleneck effect.

The **parceled control in *dataClay*** does not assume that there is a single DBA acting as the "root" of the DBMS, but on the contrary data administration is actually **managed by the providers themselves**. Both the schema developers and the data owners, by defining model contracts and data contracts (respectively), are acting as the **"DBAs" of what they are sharing**.

In addition, the security management is decentralized thanks to the **per-method validation** (methods being the only way to access data) and with the support of the **session-based authentication mechanism**.

5.2. Data locality

Computing close to the data has become a must in the last decade, not only because of the vast amount of data generated, but also because it is usually produced so fast that moving it to an external infrastructure becomes unfeasible. In this sense, traditional RDBMs, like PostgreSQL [10], already support stored procedures that are actually computed inside the database engine, or NoSQL databases like Neo4j [9] which enable the administrator to install plugins or coprocessors; also solutions like Hadoop [26], that is designed to enable the MapReduce [27] workflow abstraction to allow processing on immobile datasets.

On the other hand, Active Storage [28] is becoming a ground-breaking technology that aims to move the computation to the storage devices, thus freeing up the CPU for other tasks, eliminating the need to move data into main memory and removing interferences on the different links and caches.

***dataClay* offers an easy and effective way to execute code in the same infrastructure where the data is, but also in a transparent way to the programmer. The application code does not need to be adapted to the underlying data engine since the execution is performed through stub classes which are like regular classes for the application.**

5.3. Sharing while enriching

Data enrichment is a general term that refers to processes used to enhance, refine or improve existing data. In many cases, the term refers to annotating data with additional semantic information, either manually or automatically, to improve search and retrieval of different kinds of data [29] [30] [31]. In dataClay, we use the term enrichment in a slightly different sense, understanding the additional semantic information as part of the data itself, not as metadata as is usually done, which requires the ability to extend the schema.

In this sense, current schema-less DBMSs like MongoDB [15] and Neo4j [9] support the free addition of new attributes to documents or graph nodes respectively, and stored **javascript functions or plugins** (i.e. stored procedures being called via REST API or RPCs), but only permit the system administrator to install and share them with others.

In the same way as RDBMSs, schema-based NoSQL solutions like Intersystems Caché [32], db4o [7], HBase [33] and Cassandra [8] support schema evolution by offering the possibility to add or delete attributes from existing schemas dynamically (i.e. alter table or view). However, none of them supports sharing the schema in a way that it can be enriched with the enrichments being controlled by their creator.

On the contrary, *dataClay* provides the flexibility to deal with and publish enrichments as any of the original attributes or methods

based on the contract-based model sharing.

On the other hand, there are Dynamic Software Updating (DSU) solutions like JavAdaptor [34] or Rubah [35] that enable the developers to modify the schemas of their applications in real time directly affecting any execution in progress for instance to apply security fixes on a live vulnerable system. The drawback is that these modifications affect to all running applications, so that there is no way to keep different versions of the same data in the case that users are concurrently accessing it.

To tackle this problem, *dataClay* keeps objects in memory while the associated execution requests are running, and realizes the changes derived from the enrichments in background (enrichment deployment to execution classes) so that they are eventually available to upcoming execution requests.

5.4. Persistent vs. volatile data

Traditionally, data has been represented and designed in a different way depending on whether it is treated within a **persistent** (non-volatile) environment or a non-persistent (volatile) one. In a non-volatile environment, the application has to deal with a persistent storage such as a file system or a database. In the first case, the data is contained in files with different formats (or not formatted at all) and the programmer handles them by performing specific direct I/O operations or, in the best case, using existing parsers or serialization mechanisms. In the second case, data is handled within a database with specific structures and relationships like in relational databases and it can be accessed with specific query languages; or it can be stored and accessed in newer NoSQL databases for instance via a REST API.

On the other hand, in a volatile environment the applications allocate free memory to load the data in, for example creating a set of objects in an Object Oriented programming language where the data model is designed to navigate through object references, iterators, etc.; thus processing and analyzing data efficiently.

This situation compels the programmers to face the problem of dealing with two different data models and the mappings between them. For this reason, in the 80s emerged **Object Relational Mapping (ORM)** frameworks emerged to provide the necessary mechanisms to automatically convert data between different type systems. However, current approaches like Hibernate [36] or DataNucleus [37] are not enough to fulfill our goals. For instance, Hibernate addresses object-to-database **impedance mismatch** problems by replacing direct persistence-related database accesses with high-level object handling functions. However, the user must explicitly specify how the objects map to the database, thus not preventing the programmer from handling two different data models. In addition, persistence-related issues such as indexes, primary keys, or foreign keys must be explicitly handled by the programmer by means of annotations in the classes.

In this case, OODBMSs such as db4o [7] or Intersystems Caché [32], make the persistence of objects transparent to the programmer. Furthermore, Intersystems Cach also offers a good level of interoperability between different languages such as C++ and Java, thus having the capability to store and access objects from applications coded in any of the supported languages, as *dataClay* does. However, both db4o and Intersystems Cach present a lack of flexibility in model sharing, as stated in the previous section, so makes them not valid to fulfill all our requirements.

Last but not least, it is worthwhile to mention emerging hardware solutions like new storage devices such as **NVRAM** or Storage Class Memories (SCM) [38] that are included into the storage/memory hierarchy. Given that the nature of these new devices will be closer to memory than to storage (low latencies, high bandwidth, and byte-addressable interface) using a data model close to applications, like OO, might exploit better its benefits (instead of using them as block devices for a file system). *dataClay* **is perfectly in tune with these new upcoming memory technologies, since it is intended for in-memory working by design.**

6. Conclusions

In this paper we present *dataClay*, a distributed data store based on the concepts of identity and encapsulation, designed to share data in a secure and flexible way thanks to parceled control.

Controlled access to data is ensured by making data accessible only and through methods with explicit granted access. To this end, *dataClay* uses stub classes to connect user-level applications with the stored objects in the backends. On the other hand, flexibility is effectively supported by allowing schema evolution through enrichments. Enrichments allow to expand classes with new attributes, new methods, or even new implementations of existing methods; and can be applied by any user that is granted access to the corresponding classes through model contracts.

To be competitive in terms of data access performance, *dataClay* exploits data locality and overcomes impedance mismatch issues with an Object Oriented data model for data representation supporting interoperability between different languages (currently Java and Python). To analyze data access performance, we executed YCSB in different scenarios comparing *dataClay* with trendy NoSQL databases: MongoDB and Cassandra. Results show that in a read-only workload *dataClay* achieves similar throughputs and latencies to those obtained by MongoDB, which was the best in this case. On the other hand, in update-only workload *dataClay* obtains similar performance to Cassandra's, which was the best in this case. In all cases, *dataClay* shows an almost linear scalability regarding the number of threads, whereas MongoDB gets stalled in update-only workload.

With YCSB, we have also shown that enrichments, one of the key features in *dataClay*, have a negligible effect on the performance results. This was the expected behavior since they are actually deployed as part of the code of any registered class in the system with no special treatment.

Finally, we have presented a performance analysis on the basis of two well-known data analytics applications: Wordcount and K-means. Both applications

are implemented as Map-Reduce workflows to evaluate their execution times on an HPC environment and throughout different workloads. This study shows how *dataClay* overcomes impedance mismatch difficulties and improves data serialization for inter-node communications. The former is proven by using *dataClay* instead of files in the computation of Wordcount, which reduced the elapsed times in up to a 43%. The latter is revealed by using *dataClay* instead of default language serialization techniques, which boosted up to a 39% the K-means execution.

Acknowledgements

This work has been supported by the Spanish Government (grant SEV2015-0493 of the Severo Ochoa Program), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316) and by Generalitat de Catalunya (contract 2014-SGR-1051).

Special thanks go to Dr. Oscar Romero (Universitat Politècnica de Catalunya) for providing helpful feedback on the paper.

References

[1] C. J. Date, An introduction to database systems, 3rd Edition, Addison-Wesley Pub. Co, 1981.

[2] M. J. Carey, N. Onose, M. Petropoulos, Data Services, Commun. ACM 55 (6) (2012) 86–97. doi:10.1145/2184319.2184340.

[3] I. M. Faniel, A. Zimmerman, Beyond the data deluge: A research agenda for large-scale data sharing and reuse, International Journal of Digital Curation 6 (1) (2011) 58–69. doi:10.2218/ijdc.v6i1.172.

[4] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, C. Shahabi, Big data and its technical challenges, Communications of the ACM 57 (7) (2014) 86–94. doi:10.1145/2611567.

[5] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, R. Morrison, An Approach to Persistent Programming, The Computer Journal 26 (1983) 360–365. doi:10.1093/comjnl/26.4.360.

[6] S. Pelley, T. T. F. Wenisch, B. T. B. Gold, B. Bridge, Storage management in the NVRAM era, Proceedings of the VLDB Endowment 7 (2) (2013) 121–132. doi:10.14778/2732228.2732231.

[7] Versant Corporation, db4o (database for objects), db4o.com, (accessed 2015-09-30).

[8] Apache Software Foundation, Apache cassandra project, cassandra.apache.org, (accessed: 2015-10-2).

[9] Neo Technology, Neo4j, neo4j.com, (accessed: 2016-1-14).

[10] The PostgreSQL Global Development Group, Postgresql, postgresql.org, (accessed: 2016-3-8).

[11] Seagate, Kinetic hard drive for scale-out object storage, seagate.com/products/enterprise-servers-storage/nearline-storage/kinetic-hdd, (accessed: 2016-2-1).

[12] The Netty Project, Netty, netty.io, (accessed: 2015-12-2).

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10 (2010) 143–154doi:10.1145/1807128.1807152.

[14] C. Bazar, C. S. Iosif, The Transition from RDBMS to NoSQL. A Comparative Analysis of Three Popular Non-Relational Solutions: Cassandra, MongoDB and Couchbase, Database Systems Journal V (2) (2014) 49–59.

[15] MongoDB Inc., Mongodb, mongodb.org, (accessed: 2015-11-7).

[16] DB-engines, Knowledge base of relational and nosql database management systems, db-engines.com/en/ranking, (accessed: 2016-3-18).

[17] T. Rabl, S. Gómez-Villamor, Solving big data challenges for enterprise application performance management, Proceedings of the VLDB Endowment 5 (12) (2012) 1724–1735. arXiv:1208.4167, doi:10.14778/2367502.2367512.

[18] Y.-S. Kang, I.-H. Park, J. Rhee, Y.-H. Lee, MongoDB-Based Repository Design for IoT-Generated RFID/Sensor Big Data, IEEE Sensors Journal 16 (2) (2016) 485–497. doi:10.1109/JSEN.2015.2483499.

[19] L. a. Adamic, B. a. Huberman, Zipf's Law and the Internet, Glottometrics 3 (2002) 143–150.

[20] B. S. Center, Mare nostrum iii, https://es.wikipedia.org/wiki/MareNostrum.

[21] E. Tejedor, R. M. Badia, COMP superscalar: Bringing GRID superscalar and GCM together, in: Proceedings CCGRID 2008 - 8th IEEE International Symposium on Cluster Computing and the Grid, IEEE, 2008, pp. 185–193. doi:10.1109/CCGRID.2008.104.

[22] F. Schmuck, R. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, Proceedings of the First USENIX Conference on File and Storage Technologies (January) (2002) 231–244.

[23] W. Zhao, H. Ma, Q. He, Parallel K-means clustering based on MapReduce, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 5931 LNCS, Springer Berlin Heidelberg, 2009, pp. 674–679. doi:10.1007/978-3-642-10665-1_71.

[24] Oracle, Oracle database, oracle.com/database, (accessed: 2015-12-3).

[25] Microsoft, Sql server, microsoft.com/es-es/server-cloud/products/sql-server, (accessed: 2015-12-4).

[26] Apache Software Foundation, Apache hadoop, hadoop.apache.org, (accessed: 2016-1-11).

[27] J. Dean, S. Ghemawat, MapReduce : Simplified Data Processing on Large Clusters, Communications of the ACM 51 (1) (2008) 1–13. arXiv:10.1.1.163.5292, doi:10.1145/1327452.1327492.

[28] J. Piernas, J. Nieplocha, E. J. Felix, Evaluation of active storage strategies for the lustre parallel file system, Proceedings of the 2007 ACMIEEE Conference on Supercomputing SC 07 (1) (2007) 1. doi:10.1145/1362622.1362660.

[29] K. Guo, W. Pan, M. Lu, X. Zhou, J. Ma, An effective and economical architecture for semantic-based heterogeneous multimedia big data retrieval, Journal of Systems and Software 102 (2015) 207–216. doi:10.1016/j.jss.2014.09.016.

[30] G. Touya, A Road Network Selection Process Based on Data Enrichment and Structure Detection, Transactions in GIS 14 (5) (2010) 595–614. doi:10.1111/j.1467-9671.2010.01215.x.

[31] R. C. F. Wong, C. H. C. Leung, Automatic semantic annotation of real-world web images, IEEE Transactions on Pattern Analysis and Machine Intelligence 30 (11) (2008) 1933–1944. doi:10.1109/TPAMI.2008.125.

[32] Intersystems Corporation, intersystems.com/our-products/cache, (accessed: 2016-9-8).

[33] Apache Software Foundation, Hbase, hbase.apache.org, (accessed: 2016-1-5).

[34] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schröter, G. Saake, JavAdaptor - Flexible runtime updates of Java applications,

Software - Practice and Experience 43 (2) (2013) 153–185. doi:10.1002/spe.2107.

[35] L. Pina, L. Veiga, M. Hicks, Rubah: DSU for Java on a stock JVM, Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14 (2014) 103–119doi:10.1145/2660193.2660220.

[36] Red Hat, Hibernate, hibernate.org, (accessed: 2016-1-14).

[37] DataNucleus, Datanucleus, datanucleus.org, (accessed: 2016-2-5).

[38] R. F. Freitas, W. W. Wilcke, Storage-class memory: The next storage system technology, IBM Journal of Research and Development 52 (4.5) (2008) 439–447. doi:10.1147/rd.524.0439.

Vitae

Jonathan Martí

Jonathan Martí is a researcher and PhD student in the Storage-System Group at the Barcelona Supercomputing Center (BSC) since 2006. He holds a BSc in Computer Science and a MSc in Computer Architecture Networks and Systems, both received from Universitat Politècnica de Catalunya (UPC). His research interests are programming models and distributed and parallel data stores for data-sharing and data integration.

Anna Queralt

Anna Queralt is a senior researcher in the Storage-System Group at BSC since 2012. She received her PhD in Computer Science from UPC in 2009, where she was a faculty member. She also was a part-time lecturer at the Open University of Catalonia, and nowadays she lectures in Big Data courses at the UPC School of Professional and Executive Development, and at the ESADE Business School. She is also a member of the Steering Committee of the Standard Performance Evaluation Corporation (SPEC) Research Group. Her research interests are data-sharing and integration of persistent data in the programming model.

Daniel Gasull

Daniel Gasull is a developer in the Storage-System Group at the BSC since 2012. He holds a BSc in Computer Science received from UPC in 2012. His topics of interest are design, specification, leadership and development of research projects, applications and simulations.

Alex Barceló

Alex Barceló is a Junior Developer and PhD Student in the Storage-System Group at BSC. He has studied in the UPC and currently holds a Mathematical degree and two Engineering degrees in Computer Science and Telecommunications. His research interests are computer architecture and programming models.

Juan José Costa

Juan José Costa is a Collaborating Lecturer at UPC since 2010. He received his PhD in Computer Science from the same university in 2011, where he has also been teaching operating systems courses since 2002. He also collaborated with the Storage-System group at the BSC since 2014 to 2016. His main research interests are distributed shared memory, cluster computing and operating systems.

Toni Cortes

Toni Cortes is the manager of the Storage-System Group at the BSC and associate professor at UPC. He received his PhD in computer science in 1997 from UPC. Since 1992, Toni has been teaching operating system and computer architecture courses at UPC where he also served as vice-dean for international affairs. His research concentrates in storage systems, programming models and operating systems. He has published more than 125 technical papers. In addition, he has also advised 10 PhD thesis and has been involved in several EU and industry projects.