dataClay: next generation object storage

Jonathan Martí

Advisors:
Dr. Toni Cortes Dr. Anna Queralt

A dissertation submitted in partial fulfillment of
the requirements for the degree of
Doctor from the Universitat Politècnica de Catalunya

Doctoral program of the Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Barcelona, January 9, 2017

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Abstract

Existing solutions for data sharing are not fully compatible with multi-provider contexts. Traditionally, providers offer their datasets through hermetic Data Services with restricted APIs. Therefore, consumers are compelled to adapt their applications to current functionality, and their chances of contributing with their own know-how are very limited.

With regard to data management, current database management systems (DBMSs) that sustain these Data Services are designed for a single-provider scenario, forcing a centralized administration conducted by the database administrator (DBA). This DBA defines the conceptual schema and the corresponding integrity constraints, and determines the external schema to be offered to the end users. The problem is that a multi-provider environment cannot assume the existence of a central role for the administration of all the datasets.

In terms of data processing, the different representations of the data model at different tiers, from the application level, to the Data Service or DBMS layers; causes the applications to dedicate between 20% and 50% of the code to perform the proper transformations. This causes a negative impact both on developers' productivity and on the global performance of data-intensive workflows.

In light of the foregoing, this thesis proposes three novel techniques that enable a data store to support a multi-provider ecosystem, facilitating the collaboration within all the players, and the development of efficient data-intensive applications. In particular, and after the convenient decentralization of the database administration, this thesis contributes to the community with: a) the proper mechanisms to enable consumers to extend current schema and functionality without compromising providers constraints; b) the proper mechanisms to enable any provider to define his own policies and integrity constraints in a way that never will be jeopardized; c) the integration of a parallel programming model with the data model to drastically reduce data transformations and being designed to be compliant with near future storage devices.

Keywords: data sharing, data integration, database management systems, object oriented persistence, multi-provider storage, data-intensive workflows

Acknowledgements

This thesis has been possible thanks to the guidance and experience of its advisors: Dr. Toni Cortes and Dr. Anna Queralt. After a 10-year professional relationship with Dr. Cortes, we transcended the limits of the day-to-day work and he is now like a brother for me. On the other hand, my relationship with Dr. Queralt is more recent but still intensive, her strong dedication and her multifaceted profile are admirable. For these reasons, it is an honor for me to work with both of them.

In addition to the advisors, there are other people who deserve my sincere gratitude. From my professional life in the Barcelona Supercomputing Center, I thank all the members of the Storage Systems Research Group for the excellent working atmosphere they built, from early colleagues: Jesús Malo, Ernest Artiaga, Alberto Miranda and Ramón Nou; to my *dataClay* teammates: Alex Barceló, Juan José Costa, Rizkallah Touma, Iván López and Paola Garfias; and with a special mention for Daniel Gasull, whose invaluable commitment to the development of *dataClay* technology has been crucial for the realization of this thesis.

I also want to express my gratitude for the feedback received from people who reviewed this thesis at its different stages: Dr. David Carrera, Dr. Leandro Navarro, Dr. Rosa María Badía, Dr. María S. Pérez Hernández and Dr. Gabriel Antoniu.

Moving out from my professional life, and as could not be otherwise, this thesis has also nourished from a good balance between work and leisure time. In this regard, I am profoundly grateful for all the experiences and adventures lived with some of my friends: Paco, Mari Carmen, Martí, Juan, Augusto, Mario, Juan Luis, Marta, Judit, Enric, Cristina, etc. With special thanks to my second family of the Sala d'Armes de Montjuic for teaching me the noble art of fencing while meeting lovely people of all ages, interests and concerns.

Undoubtedly, this thesis would have never been possible without the education, values and wholehearted support I received from my family, especially from Esther, Ramón and Silas. I will always be indebted to you.

And last but not least, I want to especially thank Sílvia Bonàs for her endless energy and her daily support; which have been of vital importance in moments of discouragement inherent in doctoral studies. Thank you very much for giving me the chance to live passionately in a chaotic but exciting universe.

Contents

List of Figures

Chapter 1

Introduction

Sharing data so that third parties can build new applications and services on top of it is nowadays a cornerstone in both academic and business worlds to obtain the maximum benefits of the produced information. Although most data sharing initiatives are sustained by public data, the ability to reuse data generated by private companies is becoming increasingly important as organizations like Google, Twitter, the BBC or the New York Times, are providing access to part of their data and making a profit from it. Indeed, all kind of organizations and institutions collaborate by sharing their data either by direct file transferring or more commonly through Data Services, which were born from Web 2.0/3.0 services [1] and are feeding todays Big Data ecosystem [2].

Data Services publish suitable APIs to enable the consumer to access data remotely (e.g. REST APIs [3]) and are basically offered in two flavors: over the Internet with key vendor storage and functionality tools located in the cloud, or directly from providers' infrastructures. Setups in the former case help to provide agile services that can perform well while providers have to trust on the spaces over which their data traverses (which are arbitrary). In the latter case, providers keep full control on their data since it never leaves the infrastructure. In both cases, however, consumers are still restricted to the set of provided functions which might meet their current needs but hinder forthcoming collaborations requiring new functionality or certain adaptations to existing data.

Another key aspect to take into account for effective data sharing is related to data integration. The lack of standards, in conjunction with the vast amount of data formats and APIs, compels developers to make huge efforts to code explicit middle-ware or pre/post-processing applications to adapt data for its correct processing

and for interoperability. Moreover, this leads to an ineluctable performance decrease due to systematic data transformations involving massive data transfers through the communication layers among Data Services or between a Data Service and its end-user applications.

In addition to inter-service issues, performance penalties also arise within services themselves due to well-known impedance mismatch difficulties [4] present through different layers of the I/O stack. From the application or Data Service data structures resident in volatile memory to the lowest layer of the underlying persistent storage, data is continuously transformed to meet different representations, data models or schemas.

Furthermore, the increase in computing power has far outpaced the increase in the speed of storage systems so that data is usually produced faster than could be stored or retrieved from disks, which also pushes the integration problem to its limits and encourages new solutions to overcome the situation. In this regard, new research strategies can be decisive to resolve these issues, both in storage systems or hardware and in workflow management or software.

From the perspective of the persistence layer, the expected proliferation of NVRAM technology (non-volatile random access memory) will boost the I/O stack [5] achieving performance results similar to volatile memories and improve energy efficiency [6]. Therefore, cumbersome data transformations could be avoided by letting the applications work with the storage system in the same way as they do in memory, so that developers should only concern about designing a data model consistent with the requirements of such applications. Consequently, this is a good time for in-memory compliant databases to outperform their competitors.

On the other hand, in-situ processing strategies [7] [8] are gaining momentum in High Performance Computing (HPC) environments, where developers are forced to redesign their pipelines to perform post-processing tasks (post-analysis, visualization, etc.) on data produced in prior stages and while it is present in memory (or being generated). Otherwise these post-processing tasks are or were usually performed in external clusters and/or institutions after dumping all outputs in disks or through long-winded data transfers to retrieve the required data, which by the way, closes the circle of data sharing difficulties since this data externalization is sometimes problematic due to privacy policies.

In view of the above this thesis presents *dataClay*, an object-oriented (OO) data store that fills the gaps of current Data Services and DBMSs with novel techniques for sharing

structured data in a multi-provider ecosystem. By means of the implementation and evaluation of *dataClay*, this thesis validates three novel contributions.

The first contribution responds to the need of flexibility to adapt or integrate data models from different sources in a single multi-provider data store. Traditionally providers define how data is and how can be processed, but to foster data sharing in a multi-provider context it is important that the different players are enabled to extend the different data models to meet their requirements. To this end, a novel solution is proposed on the basis of a particular decentralization of the data model administration.

The second contribution arises from the basis that, in such a multi-provider data store where data is potentially interrelated across different datasets, enabling every provider to define his particular access policies on his dataset fosters data sharing but might jeopardize data integrity constraints of the data model. In particular, constraints depending on references and accessibility between data records belonging to different datasets, might be compromised due to potential differences in access permissions. In this context, this thesis proposes a novel mechanism to decentralize access control while enabling the providers to define extra functionality to ensure data integrity.

Last but not least, the third contribution responds to current difficulties of data-intensive applications. These applications not only suffer from the unstoppable growth of data volumes, requiring parallel computing models and/or exploiting data locality (in-situ or near-data processing), but also from the differences between the application and the persistent data models, forcing developers to code explicit data transformations. To tackle this problem, this thesis proposes a novel integration of a parallel programming model with a persistence OO data model. This integration is proven successful to improve the performance of data-intensive workflows.

1.1 Background

The scenario in which this thesis falls within is based on the three pillars of todays mechanisms for data sharing: Data Services, Database Management Systems, and storage technologies (figure 1.1). In this regard, this section explores their main characteristics to highlight the concepts related to the thesis contributions, and chapter 2 further exposes them through the related work in the state of the art.

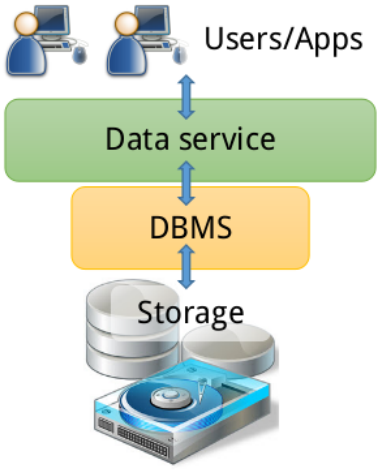

Figure 1.1: *Application to storage data layers*

1.1.1 Data services for large-scale data sharing

Data Services [9] appeared as successors to Database Management Systems (DBMSs) in todays globally connected world, adapting database ideas to the current scenario where data sharing is being extended from a single organization to an ecosystem of providers and consumers. In particular, a Data Service offers an interoperable service-oriented platform for data exchanging that enables the data provider employing it to effectively offer its datasets to consumers or 3rd parties through a specific API.

To this end, Data Services are deployed as an additional layer of a DBMS to parameterize and control access to data, and making it available for external applications, which do not require to know how data is structured in the underlying data store. Consequently, Data Services had become the default solution for the specialization of web services deployed on top of data stores, other services, and/or applications to encapsulate data-centric operations.

Data services are mainly grounded in the concepts of: data dissemination, data integration and data virtualization.

Data dissemination refers to the distribution or transmitting of data to end users, either from open/public data usually based on open formats, or from proprietary data that is released under specific constraints or using proprietary formats that can only be processed with specific software provided by the organizations to users.

Data integration involves combining data residing in different sources and providing users with a unified view of these data. This process becomes significant in a variety of

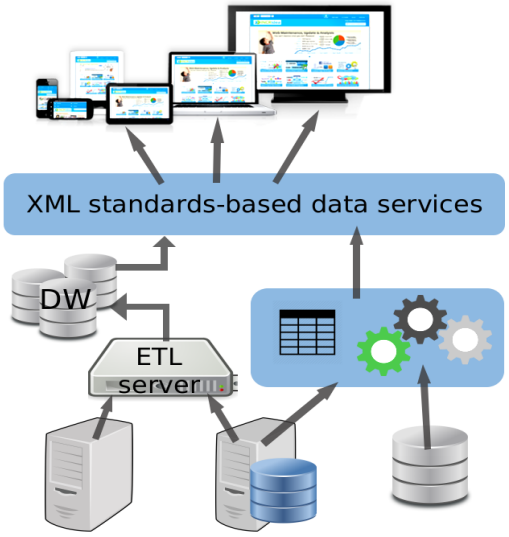

Figure 1.2: ETL Data Service

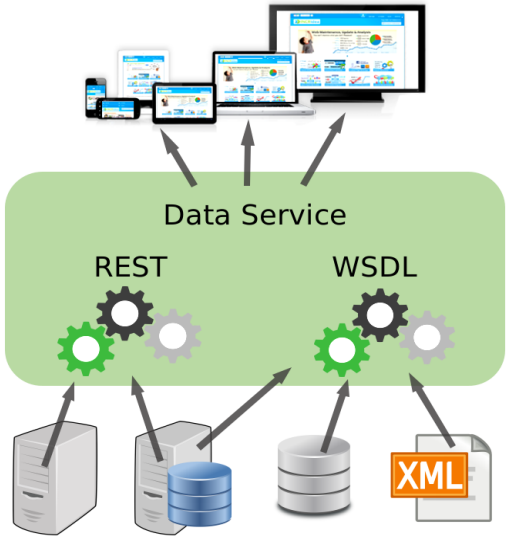

Figure 1.3: REST Data Service

situations, which include both commercial (two companies that merge their databases) and scientific domains (combining research results from different repositories). Data integration appears with increasing frequency as the volume and the need to share existing data explodes.

Data virtualization allows an application to retrieve and manipulate data without requiring technical details about the data, such as how it is formatted at source, or where it is physically located. Unlike the traditional extract, transform, load (ETL) process, the data remains in place, and real-time access is given to the source system for the data, thus reducing the risk of data errors and reducing the workload of moving data around that may never be used. Unlike Data Federation it does not attempt to impose a single data model on the data (heterogeneous data). To resolve differences in source and consumer formats and semantics, various abstraction and transformation techniques are used.

Practical examples of Data Services are shown in figures 1.2 and 1.3. The former figure shows an ETL Data Service where data is extracted from homogeneous or heterogeneous data sources, transformed for storing it in the proper format or structure for the purposes of querying and analysis, and loaded into the final target (database or another data service). The latter figure shows a REST Data Service where, unlike the ETL Data Service, data does not leave the infrastructure and the API provided offers the whole functionality available to process it.

In both cases, data providers are the central actors that decide how data is and how it is published to consumers, whereas **consumers are only allowed to retrieve or offer**

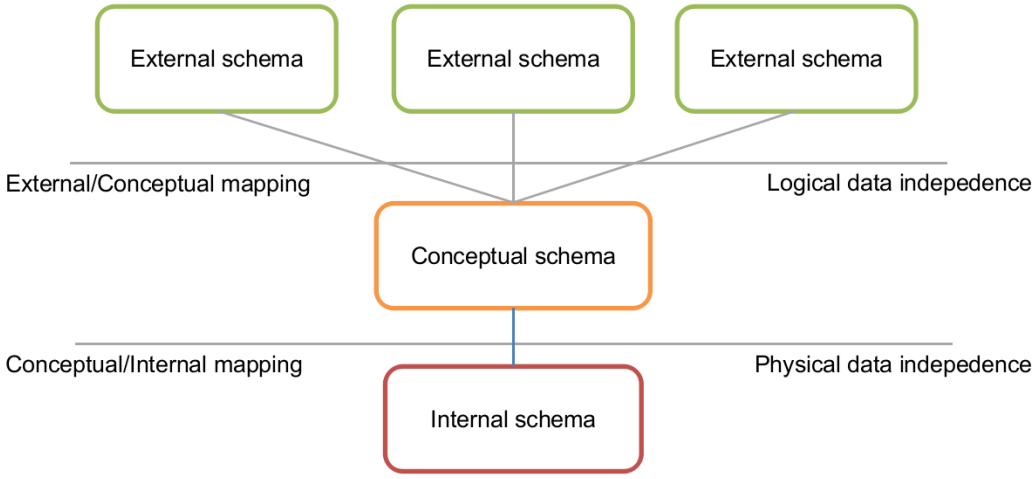

Figure 1.4: *ANSI/SPARC architecture*

new data remaining offside to contribute with new functionality based on their own intellectual property. Consequently, 3rd-party collaboration is harmed with tedious processes of data integration and external adaptations which sometimes makes it totally unfeasible.

1.1.2 ANSI/SPARC architecture for DBMSs

Current DBMSs are based on the ANSI/SPARC [10] model, an abstract design first proposed in 1975, that, although never became a formal standard, its idea of logical data independence was widely adopted. 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, providing each user with access to a fragment of the database and abstracting how data is physically managed. To this end, and illustrated in figure 1.4, the architecture proposes three levels of abstraction: the external level, the conceptual level, and the internal level.

The external level provides user's view of the database describing a part of the database that is relevant to a particular user. It excludes irrelevant data as well as data which the user is not authorized to access. This level is specified by the database administrator (DBA) that defines how data is exposed and which users can access it.

The internal level, in the opposite side, involves how the database is physically represented on the computer system. It describes how the data is actually stored in the database and on the computer hardware. This level is then enforced by database vendors that implement the database internals and usually provide the proper mechanisms or tools

that enable the DBA to tune the corresponding features considering the underlying storage system.

The conceptual level, between the external level and the internal level, describes what data is or can be stored within the whole database and how the data is inter-related. To this end, the DBA is in charge of defining the schema or data model represented in the database.

In view of the above, the DBA plays a central role in the administration of the system being the responsible for defining the external and the conceptual levels, and for tuning the internal level to meet the provider's requirements. Thus in these circumstances, **users are compelled to adapt their applications to the external view of the database which preserves the data safe in exchange for losing flexibility.**

Furthermore, from time to time the DBA has to apply certain changes to the data model or schema of the conceptual level, so-called **schema evolution**. In the case of SQL, some operations are provided that enable the DBA to "alter" tables in order to add, drop or modify (change type) columns. This feature is sometimes enough in order to deal with basic adaptations, but the **data model behind the scenes is still restrictive in the sense that all rows within the table must be compliant with current column design.** On the contrary, most NoSQL data stores are characterized by not having a fixed schema. Each "row" (object, node, document, etc.) might have a different number and even different type of "columns" (or attributes) so that it is said that data modeling in some NoSQL databases is fully elastic. The drawback is that **the programmer has to take into account these potential differences between the elements of the database**.

1.1.3 Data integrity in DBMSs

Data integrity refers to a condition of the persistent data in which all its elements are accurate and consistent in respect to its target state of the real world. In this way, data with "integrity" is said to have a complete or whole structure, that is, data values are standardized according to a data model and/or data types and all characteristics of the data must be correct. In this sense, mechanisms to enforce data integrity include referential integrity defining the relations a datum can have with other data (e.g. Customer related to purchased Products), type checks based on a fixed schema (e.g. numeric property cannot accept alphabetic data), automatic correction checks based on triggers or callbacks, or

derivation rules (e.g. materialized views).

Data integrity is not widely adopted by NoSQL solutions which normally delegate this responsibility to applications, especially those offering a schema-less approach thus not forcing a fixed schema. On the contrary, schema-based DBMSs such as object oriented DBMSs (OODBMSs) and relational DBMSs (RDBMSs), traditionally support the following types of integrity constraints.

Domain constraints specify that attributes or columns must be declared upon a defined domain or type (e.g. a boolean typed column cannot contain a string).

Entity constraints concern the concept of a primary key, a rule which states that every record must be uniquely identified. In the relational model, every table must define a column or a set of columns as the primary key, forcing every row to be identifiable by its content. In the OO model there is no such a concept of primary key, since every object is identified by its unique OID (object identifier) in a transparent way for the user.

Referential constraints state that an attribute-column/s represents a reference to an object/row of another class/table. In RDBMSs referential integrity is ensured based on foreign keys which define a column or set of columns that refer to another table. In this way, the system prevents rows to be removed while other rows are referring to them, unless triggers are used to define so-called *cascade* behaviors to propagate deletions. In OODBMSs, there is no concept of foreign keys, relationships are built through OID references enabling objects to directly refer other objects through their identity. The problem comes when removing an object. Although early OODBMSs [11] did not support explicit object removal, more recent systems ensured referential integrity on the basis of reference counters, *inverse members* [12], etc.

Check constraints enable the users to define extra domain rules such as accepted value ranges (e.g. age attribute defined for integer values between 0 and 120)

User-defined constraints on the basis of **procedural rules** are used to define extra checks or behaviors based on user-defined algorithms that are resolved in a non-interactive way under certain circumstances. Triggers and materialized views are well-known features in this regard.

In light of the foregoing, a schema-based data store should support all these integrity constraints, but following sections reveal the extra complexity of ensuring data integrity in a multi-provider context.

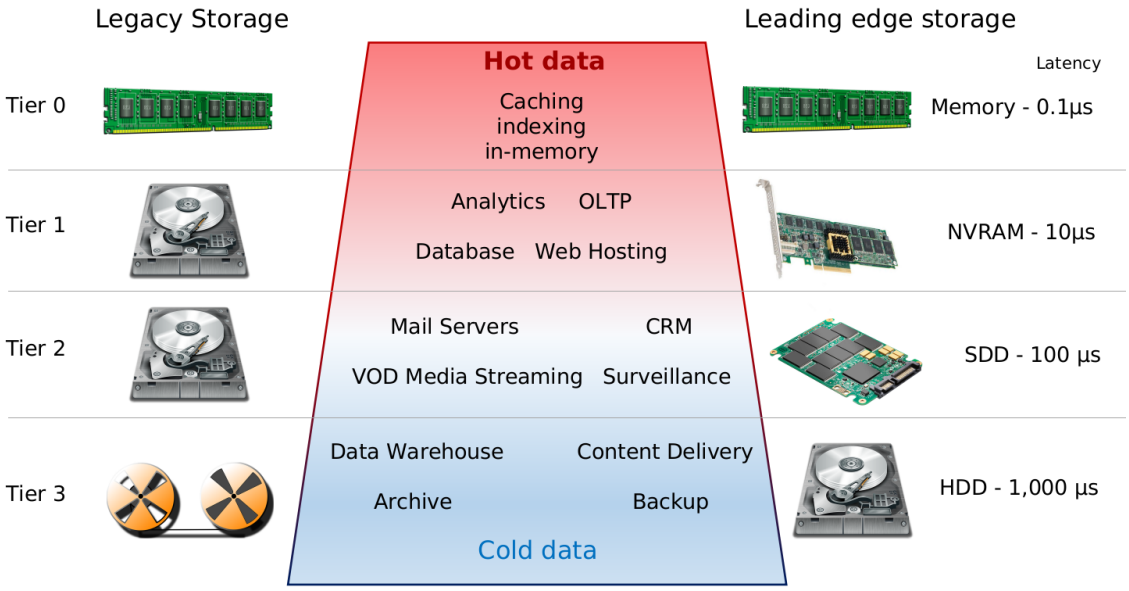

Figure 1.5: *Redefinition of persistence layers*

1.1.4 New storage technologies

Newer storage devices with no mechanical parts are unveiled today to break from previous storage technologies and to redefine the I/O stack (as shown in figure 1.5).

In the first place, the proliferation of Solid State Disks (SSDs) displaced Hard Disk Drives (HDDs) from personal computers, laptops and smart* gadgets due to its performance and lower battery consumption [13], but also gained importance in HPC environments with hybrid SSD-HDD technologies [14].

But nowadays it is expected that the inclusion of non-volatile memories (like NVRAM [15]) into the storage/memory hierarchy will boost the I/O stack to definitely obtain similar results as traditional memories used for the application layer. NVRAM is random-access memory that retains its information when power is turned off (non-volatile), in contrast to dynamic random-access memory (DRAM) and static random-access memory (SRAM) which both maintain data only for as long as power is supplied. Therefore, NVRAM should offer the durability of disk with the high performance and byte addressability of volatile RAM.

Indeed, as cost and yield improve, NVRAM may become ubiquitous across the computing spectrum: they could become the preferred storage for small devices in the Internet of Things (IoT), and similarly could become critical to performance and recoverability in cloud systems and HPC environments. Byte-addressable NVRAMs enable

the construction of high-performance, in-memory, recoverable data structures (RDS). Conventionally, RDSs have been placed either on disk or Flash and accessed through a slow, serial, block-based interface. With NVRAM, these data structures could be accessed at word granularity via loads and stores orders of magnitude faster than accesses to HDDs or SDDs.

This paradigm shift in access latency and interfaces encourages the redesigning of RDSs. For example, recent research has looked at ways to optimize file systems for NVRAMs [16] [5]. Programs written for such systems will have to manage the transfer of data between volatile and nonvolatile memories. Whereas programmers are familiar with designing recoverable systems using block-oriented file-system interfaces, NVRAM's byte addressability could create the potential for new and higher-performing RDSs and calls for new programming interfaces to succinctly express ordering requirements. Future systems may incorporate two explicit memory spaces for volatile and nonvolatile data, or further better they may use a unified address space and precede nonvolatile memory spaces with volatile caches to enable write coalescing.

Either way, NVRAM technologies **will enable in-memory databases to keep running at almost full speed while maintaining data in the event of power failure**. This uprising of in-memory processing should encourage the exploitation of **data models closer to the application-level, thus tackling the impedance mismatch handicap resulting from the differences between data representation at the application layer and its actual representation in the persistence layer**.

1.1.5 In-situ processing

The growing power of parallel supercomputers gives scientists the ability to handle more complex applications at higher accuracy. To maximize the utilization of the vast amount of data generated by these applications, scientists also need scalable solutions for studying their data to different extents and at different abstraction levels. As we move into *exascale* computing, simply dumping as much raw simulation data as the storage capacity allows for post-processing analysis or visualization is no longer a viable approach. A common practice is to use a separate or external parallel computer to prepare data for subsequent processing, but this strategy not only limits the amount of data that can be saved, but also turns I/O into a performance bottleneck when using a large parallel system. The most plausible solution for the *exascale* data problem is to reduce or transform the data

in-situ [17] to perform subsequent processing locally or even while it is being generated. Thus the amount of data that must be transferred over the network is kept to minimum.

For instance, for large-scale simulations [18], in-situ processing is the most effective way to reduce the data output since the reduction can benefit all the subsequent data movement and processing. In-situ processing may also be used to extract some specific information of interest, which is generally much small than the original data and can be used for offline examination and assessment. Compared to the conventional post-processing approach (in an external cluster), performing data reduction and information extraction in-situ is a more appealing approach, provided that the post-processing calculations take a small portion of the supercomputer time. Furthermore, with in-situ processing there is no need to keep the full raw data as it can be consumed as long as it is post-processed, thus saving a considerable amount of disk space for data storage and network bandwidth for data transmission.

Therefore, it is important that new data stores are capable of managing in-situ processing by offering the proper execution environment in conjunction with data persistence.

1.2 Problem statement

In the context of data sharing for multi-provider ecosystems, the effective collaboration between data providers and consumers will be achieved only if the interests of all players are accomplished. This means that on the one hand, data providers must be able to keep full control on their datasets in order to trust that they can decide how to share them and with whom at any moment. And on the other hand, and without breaking providers' policies, consumers should be allowed to adapt data to their needs to the point of being capable of creating new services on top of it, or becoming data providers for 3rd parties. That is, by means of guaranteeing control over the data while maximizing flexibility for its adaption, the key requirements of both providers and consumers are fulfilled thus nourishing the spiral of collaboration.

In addition, large-scale data sharing involves an unstoppable growth of data volumes requiring parallel computing models and/or exploiting data locality (in-situ or near-data processing) in order to reduce data transfers. In this context, applications also suffer from the differences between high-level and persistent data models, requiring an excessive amount of data transformations.

Hereafter this section digs deeper into these identified drawbacks in the scope of this thesis.

1.2.1 Centralized administration

To begin with, **the role of a centralized DBA used in current DBMSs is no longer the best solution**. In a context with several data providers they, as the data owners, should be the ones responsible for the control on the data to be shared and how can be accessed or processed. Therefore, a data store for **a multi-provider context should delegate these rights, traditionally assigned to the central DBA, to the actual data owners**.

1.2.2 Schema evolution in a multi-provider context

Current Data Services **restrict consumers by the set of APIs defined by the provider**. That is, consumers are limited to how providers decide to offer their data both regarding structure and functionality, which consequently narrows the potential of consumer applications and blocks further collaborations with 3rd parties since consumers are not allowed to share their know-how.

On the other hand, at DBMS level, schema evolution is widely supported enabling the administrator to extend existing data with additional views, new information that is not present, or additional functionalities for data processing (e.g. an *alter table* in RDBMSs).

In view of the foregoing, the challenge is to export the concepts of schema evolution into a multi-provider context, thus offering a solution valid for Data Services. In particular, consumers should be able to extend current schema without jeopardizing existing providers' constraints and at the same time becoming providers of their intellectual property.

1.2.3 Data integrity with decentralized access control

The decentralization of access control enables the providers to define their particular access permissions on their own datasets. In this context, ensuring data integrity requires special attention, since data might be interrelated across different datasets having potentially different accessibility policies. In particular, **the integrity of the accessible data by a consumer can be jeopardized in regard to referential constraints and procedural rules if the interrelated/involved data come from datasets with**

different permissions.

Regarding **referential constraints**, a reference between a datum from a dataset pointing to a datum of another dataset, might be compromised for users that have no access to the second dataset. Therefore, providers' administration based on a naive approach using traditional mechanisms for access permissions (such as ACLs) is not enough. For example, certain user Alice codes an application to access a datum from dataset D1 that points to datum from dataset D2. If Alice has the proper permissions to access both D1 and D2, she can develop an application that navigates through a reference from D1 to D2. On the contrary, **if Alice has no access to D2 the reference is broken** and the application might produce unexpected errors.

On the other hand, **procedural rules** for user-defined triggers and derived data potentially entails accessing d from different datasets to compute the corresponding algorithm. For instance, a data model could implement the observer pattern for some datum to be automatically updated under certain circumstances (e.g. a trigger updating a counter). If the algorithm requires data from different datasets for the calculation of the derived datum, the derived datum might be compromised if the accessibility permissions are not the same in all the datasets involved. For example, user Alice now codes an application that accesses some derived datum of dataset D1 representing a counter of occurrences of certain elements in dataset D2. If Alice has the proper permissions to access both datasets, the application retrieves a consistent value for the datum counter, but otherwise the **application might see a value greater than 0 although Alice has no access to D2** which could incur an unexpected behavior.

1.2.4 Data model of data-intensive applications

Data Services usually move the computation of the offered functionality to HPC environments in order to tackle the difficulties inherent from Big Data challenges. In particular, data-intensive applications/functionalities suffer from the fast and continuous growth of data volumes, which forces developers to code them on the basis of parallel programming models to take full advantage of distributed environments such as HPC clusters.

In-situ processing techniques, in-memory data management, parallel computing or the recent changes in the I/O stack; already entail significant improvements in performance, preventing expensive data transfers and exploiting data locality. However, parallel

computing is still based on file transferring or I/O APIs that require a significant waste of time for developers to write the proper code that translates the application data model into persistent or communication data model.

Indeed, even the simplest user-level applications communicating with Data Services through REST APIs or with DBMSs through drivers require a waste of time for programmers to build the corresponding system-dependent requests and to parse the subsequent responses. These conversions have proven to require between 25% and 50% [19] [20] of the total of the code to perform the corresponding mappings between the application and the persistent data models.

1.3 Requirements

Given the difficulties and opportunities highlighted in section 1.1 and the major problems to be resolved exposed in section 1.2, this section highlights the set of requirements that a new multi-provider data store should fulfill.

R1 Decentralized database administration. DBA role is no longer valid in current scenario, with different data providers and consumers working together possibly having different requirements. Requirement defined from problem described in section 1.2.1.

R2 Facilitate data sharing. The system must provide the proper mechanisms to facilitate data sharing in the multi-provider environment, concerning data access and functionality.

R3 Maximize flexibility of data modeling. The system must provide the proper mechanisms to help users to extend data structures and functionality. This must be done without jeopardizing providers' constraints and in a way that can be shared with other users. Requirement defined from problem described in section 1.2.2.

R4 Ensure data integrity. Considering the decentralization of access control, the system must support the proper mechanisms that enable providers to define any required functionality to ensure data integrity of their data models. Requirement defined from problem described in section 1.2.3.

R5 Prevent data transformations. The system must implement a data model compatible with parallel data-intensive applications that reduces the code dedicated to data

transformations (in response to problem described in section 1.2.4). Therefore this requirement must consider:

R5.1 Resolve impedance mismatch issues. The system must resolve the impedance mismatch difficulties between the application and the persistent data models.

R5.2 Compatible with programming models. The data model must be compatible with top-most used programming languages and with parallel programming models.

R5.3 In-situ processing techniques. The system must offer an execution environment enabled to exploit data locality.

R5.4 Data model of new generation storage devices. The data model should be enabled to take full advantage of new technologies.

1.4 Solution approach

This thesis presents *dataClay*, a distributed Object-Oriented data store that fulfills the requirements stated in section 1.3 and thus serves to validate the three stated contributions at the beginning of this chapter and that are further described in section 1.7.

dataClay relies on the structure and principles of the ANSI/SPARC architecture (exposed in section 1.1.2), which has proven successful in many aspects, but breaks the assumption that each DBMS has a central administrator (or administrator team). In contrast with current DBMSs and data services, *dataClay* decentralizes the administration (requirement **R1**) thus enabling different users to share and reuse data while maintaining full control on their assets, both regarding the schema and the data itself. In this way, any user can participate in the definition of the schema (requirement **R3**) adding those missing concepts or functions he needs but always based on the fragment that he is allowed to access as granted by the provider. These enhancements, also called schema *enrichments*, can be also shared with other users in the same way as the provider 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 provider). To this end, an identity-based data model is proposed making every single piece of data to be uniquely identified and individually manipulated. In particular, *dataClay* is based on an Object Oriented (OO) data model, which not only defines a unique identifier per object (OID),

but also enables the encapsulation of the object state (the actual data) by exposing only the methods that can be executed on such an object (i.e. class methods targeting object instances).

This encapsulation enables providers to ensure data integrity constraints from within the methods and by considering the different consumer views depending on access permissions (requirement **R4**). For instance, in the case of referential integrity with the example of section 1.2, Alice has access to a datum d1 in dataset D1 pointing to a datum d2 in dataset D2, but if she has no access permissions to D2 the reference from d1 to d2 is broken. In *dataClay*, data is represented with objects and a reference from d1 to d2 would involve an attribute of d1 referring to the OID of d2. Given that the only way to access objects is through methods, this attribute will never be exposed but only its getter method. Therefore, *dataClay* enables the provider to implement such a getter method to check Alice permissions and react accordingly, e.g. returning a null reference if she has no access or a reference to d2 otherwise.

Analogously, this solution is then also valid for user-defined constraints on the basis of *procedural rules*, since these procedures are actually methods that might be implemented on the same basis. For instance, *dataClay* supports triggers on the basis of Event-Condition-Action rules where condition checkers and actions to be taken are regular methods defined by providers.

Finally, requirements related to preventing data transformations (**R5**), are resolved with a novel integration of a parallel programming model with the proposed OO data model for persistent data (requirement **R5.2**). In the first place, having an OO data model enables the full integration of the external schema (of OO applications) with the conceptual schema (within the system), which minimizes explicit data transformations (in connection with requirement **R5.1**). Secondly, an OO data model is compatible with upcoming byte-addressable NVRAM technologies (requirement **R5.4**), making possible the integration of conceptual and internal levels that would prevent data transformations between the system and the underlying data storage. Thirdly, the integration of the OO data model with a parallel programming model is grounded in the concepts of in-situ processing in order to exploit data locality (requirement **R5.3**).

1.5 Research questions and hypotheses

During the inception of each research topic, the following research questions have been raised:

Q1 Can consumers extend existing schema without jeopardizing providers' constraints?

Q2 Can consumers extend functionality for data processing without jeopardizing providers' constraints?

Q3 Can consumers provide/share their extensions without jeopardizing prior providers' constraints?

Q4 Can providers define integrity constraints across multiple datasets with a decentralized access control?

Q5 Can the data model be integrated with a parallel programming model?

Based on the solution approach described in section 1.4, this thesis responds these questions by formulating the following hypotheses:

H1 In a multi-provider data store with decentralized administration, consumers can be enabled to extend accessible schemas and share these extensions without jeopardizing providers' constraints both by:

- **H1.1** Extending the provided data structures without jeopardizing providers' constraints.
- **H1.2** Extending the provided data functionality without jeopardizing providers' constraints.

H2 In a multi-provider data store with decentralized administration, encapsulating data through methods, in conjunction with the proper mechanisms to check access permissions, will enable providers to ensure data integrity constraints across multiple datasets.

H3 An object-oriented data model can be integrated with a parallel programming model and making the data store capable of:

- **H3.1** exploiting data locality for OO applications.
- **H3.2** avoiding data transformations between external schema and conceptual schema.

H3.3 eventually avoiding transformations between conceptual schema and internal schema considering upcoming storage devices.

1.6 Methodology

This thesis follows an iterative methodology. Considering the scenario described in section 1.1 and the related work exposed in chapter 2, the following iterative process is engaged for each of the research hypotheses:

1. Analysis of the requirements that aim to verify the hypotheses.
2. Design and implementation of the system modules required to fulfill the requirements and deal with the research objectives.
3. Design and implementation of the proper tests to verify the robustness of the system.
4. Experimental evaluation with standard benchmarks and common applications to validate the performance of the system.
5. Are the results satisfactory? If not, use the feedback from the experiments to assess and improve steps 1 and 2, and repeat step 3 and 4.
6. State the conclusions, and use them as input for further research goals.

1.7 Contributions

This thesis proposes three novel contributions for the scientific community that aim to prove the hypotheses formulated in section 1.5. For their validation, this thesis exposes a novel distributed OO store called *dataClay* which is implemented following the methodology of section 1.6 and fulfills all the requirements stated in section 1.3.

C1 - Flexibility to extend schema

The first contribution responds to the need of flexibility to adapt or integrate data models from different sources in a single multi-provider data store, where the data administration is assumed to be decentralized and the underlying data model is based on the OO encapsulation through methods.

In this context, providers decide what they share with their consumers by defining the external schema comprising only the relationship between classes and the methods that can be executed. Consumers are not enabled to access attributes directly,

thus they cannot jeopardize providers' data and are enabled to add new attributes and new methods to accessible classes. Actually what providers grant are access permissions to specific implementations of their methods, and consumers are enabled to extend functionality even with new implementations of their accessible methods. In this way, flexibility is pushed to the limit while data cannot be compromised and consumers are then enabled to share these extensions becoming new providers of the data store.

This contribution aims to prove the hypotheses **H1**, **H1.1** and **H1.2**.

C2 - Data integrity in a multi-provider data store

The second contribution responds to the problems of maintaining data integrity in a multi-provider data store, where data is potentially interrelated across different datasets while administration of access permissions is decentralized.

In this context, the encapsulation through methods of the OO data model enables providers to ensure data integrity with the sole condition that the data store provides a mechanism to check access permissions from within the method code. In this way, this solution is suitable for both referential integrity constraints with relationships across multiple datasets; and for derivation rules involving data elements from different datasets.

This contribution aims to prove the hypothesis **H2**.

C3 - Integration of the parallel programming and data models

The third contribution responds to current difficulties of data-intensive applications, requiring the exploitation of data locality with parallel programming models and suffering from excessive data transformations which compel data programmers to waste significant amounts of time coding these conversions.

Considering that top-most popular languages for programming user-level applications are OO languages (Java and Python according to PYPL index [21]), the integration of a parallel programming model with an OO data model for persistence facilitates the exploitation of data locality in distributed environments through parallelization, while reduces the data conversions required to access stored data.

Furthermore, a persistence data model close to the application data model is of interest in order to take full advantage of new generation storage devices such as

NVRAMs.

This contribution aims to prove the hypotheses **H3**, **H3.1**, **H3.2** and **H3.3**.

1.8 Organization

This thesis is organized as follows. Chapter 2 presents current approaches related to each research topic in the scope of this thesis, while emphasizing their disadvantages and/or missing features to underline the problems introduced in section 1.2. Then chapter 3, exposes a first overview of *dataClay*, a distributed OO data store that fulfills the requirements defined in section 1.3 and that serves to validate the contributions of this thesis stated in section 1.7. This overview is further expanded afterwards in chapter 4 with design decisions and implementation details.

Thereupon, chapter 5 presents several performance studies conducted with *dataClay* in order to validate the contributions of this thesis. *dataClay* is firstly validated using the Yahoo Cloud Storage Benchmark (YCSB [22]) to compare it in terms of latency and throughput with popular NoSQL solutions, and to prove that extensions of the schema can be offered without penalizing the performance. Once *dataClay* is proven valid, two popular applications are deeply analyzed comparing their execution behaviors between their common implementation based on existing parallelization techniques and a further improved implementation taking advantage of the integrated data model proposed in the third contribution.

The last block begins with chapter 6, which outlines the conclusions achieved with the realization of this thesis by pointing to the defined requirements and expected hypotheses to be responded. Thereafter chapter 7 presents some work-in-progress and future work that is out of the scope of this thesis but expected to be done proximately. Finally, chapter 8 highlights the publications related to the thesis and its expected impact in the near future both in academic and business worlds.

Chapter 2

Related work

This chapter briefly describes previous work in database management systems and afterwards exposes the state of the art of current solutions in relation to the main topics of this thesis. In particular, and following a similar order as in section 1.2 of identified problems, sections below explore related work on:

- The ANSI/SPARC architecture, data representation and administration.
- Schema evolution for data structures and functionality.
- Data integrity: declarative constraints, triggers and derivation rules.
- Data processing: data locality, data transformations, and parallel programming models and runtimes.

2.1 Database Management Systems

Database management systems (DBMSs) have experienced a fast evolution (figure 2.1) from its first commercial approaches in the 60s. In 1964, Integrated Data Store (IDS), developed at General Electric, based upon an early network data model developed by C.W.Bachman ([23]). In the late 1960s, IBM and North American Aviation (later Rockwell International) developed the first commercial hierarchical DBMS: Information Management System (IMS). Both kinds of DBMSs (hierarchical and network) were accessible from the programming language (usually Cobol) using a low-level interface. This made the task of creating an application, maintaining the database as well as **tuning and development controllable, but still complex and time-consuming**.

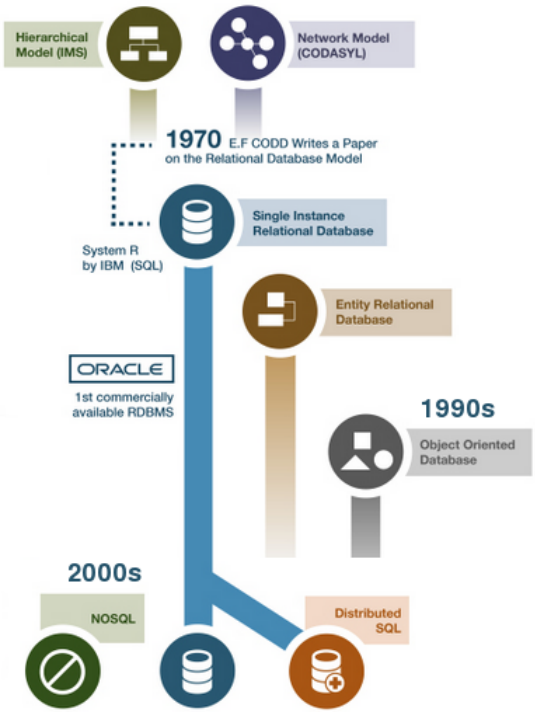

Figure 2.1: *Evolution of DBMSs*

In 1970 Edgar F.Codd published a fundamentally different approach [24], suggesting that all data in a database was represented as a tabular structure on the basis of tables with columns and rows, which he called relations, and that these relations could be accessed using a high-level non-procedural (or declarative) language: SQL (structured query language). This was the origin of the relational model which enabled to access data through predicates that identified the desired records or combination of records. In the 80s, the relational model led to the emergence of several commercial Relational DBMS (RDBMS), also referred to as SQL databases (e.g. Oracle [25], Informix [26], Ingres [27] or DB2 [28]).

In the 90s, relational databases adopted the Object Oriented paradigm towards Object-Relational DBMSs (ORDBMSs) [29]. That is, an ORDBMS is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, just as with pure relational systems, it supports extension of the data model with custom data-types and methods. An object-relational database can be said to provide a middle ground between relational databases and object-oriented databases (object database). In ORDBMSs, the approach is essentially that of relational databases: the data resides in the database and is manipulated

collectively with queries in a query language.

In contrast, at the other extreme are Object-Oriented DBMSs (OODBMSs) [11]. In this case, the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying. For the first time, databases were concerned about **overcoming impedance mismatch issues within the application layer and the actual data representation in the persistent layer**. OODBMSs provide more flexibility in data modeling and are designed specifically for working with object-oriented values, thus eliminating the need for converting data to and from its SQL form, as the data is stored in its original object representation and relationships are directly represented, rather than requiring join tables/operations.

Also in the 90s, appeared the first in-memory databases (IMDBs) also known as main memory database systems (or MMDBs or memory resident database) [30], which primarily rely on main memory for computer data storage in contrast to other DBMSs that employ a disk storage mechanism. Main memory databases are faster than disk-optimized databases because the disk access is slower than memory access, the internal optimization algorithms are simpler and execute fewer CPU instructions. Accessing data in memory eliminates seek time when querying the data, which provides faster and more predictable performance than disk.

Finally, in 2009 was introduced NoSQL [31] as we know it today, the name attempted to label the emergence of an increasing number of non-relational distributed data stores (as previous OODBMSs). A NoSQL database (referring to “non SQL”, “not only SQL” or “non relational”) provides a mechanism for storage and retrieval of data which is modeled in means other than the tabular relations used in relational databases, such as key-value, wide column, graph or document; making some operations faster in NoSQL by better suiting to the the problem it must solve. One of the motivations of this approach is presenting a simpler “horizontal” scaling to clusters of machines, which is a well-known problem for relational databases [32].

2.1.1 ANSI/SPARC architecture

The ANSI/SPARC architecture (introduced in section 1.1.2) defines that DBMSs require a central role for database administration: the Database Administrator (DBA). The DBA is in charge of the definition of security policies (i.e. including the granting of certain

rights to users) and the design and implementation of the database conceptual schema.

In terms of security, role-based access control (RBAC) is the mainstream model in both SQL databases, like Oracle or Microsoft SQL Server [33], and NoSQL databases like Cassandra [34] or MongoDB [35]. RBAC is normally based on a set of right 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. When users need some of the capabilities to perform a particular operation requiring special privileges, DBA assigns them the corresponding roles.

This centralized administration might be appropriate for scenarios where a Data Service (or DBMS) is acting as the unique data provider, but **this centralization is unacceptable when dealing with a multi-provider system where all users are potential model and/or data providers that want to keep control on their assets.**

With regard to the design of the conceptual data model, in some NoSQL DBMSs offering a "schema-free" approach, like MongoDB or Couchbase [36], certainly the DBA has no major bearing. However, the schema must be handled from the end-user applications which eventually need to know how data is structured (in order to update contents, to create indexes, etc.). Therefore, conceptual model is not actually managed from the DBMS but at user level, thus from the ANSI/SPARC perspective conceptual level and external level are mixed together.

This still has the advantage of decentralizing the conceptual data model so that provider keeps full control on it with no DBA central role. However, **the description of the external model is lost and maintaining consumer applications up-to-date can be tedious specially if the conceptual schema is altered regularly**.

2.2 Flexibility to add new functionality

As exposed in section 1.2.2, one of the most important aspects to be exploited for an effective data sharing in a multi-provider context is the flexibility to adapt current data models and functionalities to consumers' needs.

In this regard, 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 [37] [38] [39].

In current *schema-free* DBMSs, like MongoDB and Neo4j, enrichment is supported through the free addition of new attributes to documents or graph nodes respectively; and stored javascript functions or plugins as stored procedures that can be called via REST API or RPCs. However, **they only permit the system administrator to install and share them with others, which is an inconvenient as exposed in section 2.1.1.**

On the other hand, and in the same way as RDBMSs, *schema-based* NoSQL solutions like Intersystems Caché [40], db4o [41], HBase [42] and Cassandra support schema evolution by offering the possibility to dynamically add or delete attributes from the existing data model (i.e. alter table or view). However, **none of them supports sharing the schema in a way that it can be enhanced with new functionality that remains controlled by its creator.**

On the other hand, there are Dynamic Software Updating (DSU) solutions, like JavAdaptor [43] or Rubah [44], that enable the developers to modify the schema of their applications in real time by 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.

2.3 Data integrity constraints

Integrity constraints are used to ensure that all data in a database complies with rules that have been set to establish accurate and acceptable information. In this way, since the SQL standard definition [45] until now, DBMSs offer specific features to implement data integrity constraints. In particular, this section explores the two common ways to specify data integrity in current data stores: declarative constraints and active behaviors.

Declarative constraints are related to those originally defined by the SQL standard. As exposed in section 1.1.3, examples of declarative constraints are: domain constraints, entity constraints, referential constraints and check constraints.

On the other hand, **constraints defined with procedural rules** are commonly based on user-defined algorithms that are executed in non-interactive way, i.e. by means of so-called *active behaviors*. From the Event-Condition-Action (ECA) rule engines introduced in the 80s [46], to modern event-driven architectures (EDA) [47] fulfilling the Internet of Things (IoT) [48]; the support to active behaviors enable the users to program integrity constraints that cannot be resolved from the mere definition of the data structure.

Traditionally active behaviors are related to two well-known features: **triggers** fired under certain circumstances, and **derivation rules** to define derived attributes and materialized views.

2.3.1 Declarative constraints

Mostly all RDBMSs (SQL-based) support data integrity constraints since they are part of the SQL standard, but on the contrary, in NoSQL databases integrity constraints are not widely supported since integrity checks are expensive, especially in highly distributed systems. Neo4j allows to specify a *existence property* constraint to ensure that a property exists for all nodes of the graph with a specific label or for all the edges with a specific type; and *unique property* constraints to ensure that property values are unique for all nodes with a specific label. Cassandra allows to specify one primary key per table. On the contrary, MongoDB or Couchbase do not provide any kind of built-in constraints, forcing the applications to ensure uniqueness and correctness.

In the same way as schema-less NoSQL databases, modern OODBMSs such as Versant (or db4o) also delegate most of data integrity checks to user-level applications. However, early OODBMSs like GemStone [11] or ObjectStore [12] and more recent OODBMSs like EyeDB [49] offered constraint support trying to mimic RDBMSs. For instance, EyeDB supports *not null* or *unique* constraints.

Regarding referential constraints, in OODBMSs there is no concept of foreign keys as in RDBMSs, relationships are built through OID references enabling objects to directly refer other objects through their identity. Therefore, early OODBMSs did not support explicit object removal, but more recent systems ensured referential integrity on the basis of *reference counters* or *inverse members*. Reference counters might be used to know the number of references pointing to an object, thus preventing that object to be removed if some other objects are referencing it. Inverse members establish a hidden reference between the pointed object and the object pointing it, thus having a double reference per actual reference. If an object is about to be removed the system can check the existence of inverse references (as reference counters do) to prevent the removal, if no inverse member is found then navigates through its actual references to remove others' inverse references pointing to it.

As exposed in 1.2.3, a multi-provider data store requires special attention on referential constraints defined on data that can be potentially interrelated across different datasets.

Current state of the art does not offer a valid approach for this case.

2.3.2 Triggers

With regards to triggers, Oracle pushed this idea to the limit since not only permits to react to data changes but also allows triggers that fire when schema level objects (i.e. tables) are modified and when user *logon* or *logoff* events occur. Microsoft SQL Server also supports Data Definition Language (DDL) Triggers [50] which can be fired in reaction to a very wide range of events including those supported in Oracle. PostgeSQL and MySQL only support data triggers, although PostgreSQL offers a workaround to schema triggers by letting the table operations to be overridden by customized operations.

On the other hand, NoSQL databases are still a bit immature in this aspect. For instance, MongoDB has no built-in implementation of triggers although there are external solutions like *mongo-triggers* [51] which offers a workaround based on a middleware intercepting the requests to database (allowing to perform any necessary checks or changes to the whole query) and a callback-based mechanism or listeners waiting for the eventual responses (allowing to execute any further requests). In contrast, in 2015 Amazon DynamoDB [52] launched its *streams* feature that implement *table update notifications*, which in conjunction with AWS Lambda functions that can be linked to these notifications, provides the database with triggers for data updates. This is similarly implemented in Neo4j with the *TransactionEventHandler* interface that supports reacting to update events that are to be committed in the database through a transaction.

As exposed in 1.2.3, a multi-provider data store requires special attention on procedural constraints involving objects from different datasets. In case of supporting a feature like *triggers*, solutions in the state of the art cannot be directly mapped to a multi-provider context.

2.3.3 Derivation rules

Section 1.2.3 exposes why a multi-provider data store requires special attention on derivation rules when involving objects from different datasets. Current state of the art can be used to understand the expected features to be covered, although all of them must be revised for a multi-provider context.

Derived attributes

Derivation rules enable the users to define data attributes/columns to be calculated with an algorithm involving other attributes/columns.

In RDBMSs, such as as Oracle or DB2, derived columns are supported by means of *as* or *generated as* clauses used in the definition of a table attribute. In MySQL, this kind of derived columns are supported from views.

Regarding OODBMs, derived attributes can be offered from the data model since OO programming defines the encapsulation of data through methods which means that all object attributes are (or should be) accessed through methods. Therefore, derived attributes can be supported by exposing only the corresponding getter methods from which the actual value can be calculated every time the method is requested.

In NoSQL databases, derived attributes are not widely supported. However, the following section 2.3.3 exposes some examples on how to obtain derived information in such systems.

Materialized views

Materialized views are an important feature for SQL databases [53] where popular joins might become especially expensive. In this context, materialized views are user views defined on a query that are automatically updated considering the base tables of the query. For instance, a materialized view can be used for maintaining up-to-date stats, logging, or in general dynamic information.

Oracle supports materialized views and Microsoft SQL Server offers the same functionality with *indexed views* as well as IBM DB2 provides *materialized query tables* [54]. In the case of MySQL and PostgreSQL, materialized views are also supported but they must be refreshed manually (or through triggers defined on the base tables).

In respect of NoSQL databases, MongoDB for instance offers so-called *Aggregations* which are operations (e.g. a map-reduce function) that can group values from multiple documents together and perform a variety of operations on the grouped data to return a single result. Other NoSQL databases like recent version 3.0 of Cassandra supports immediately updated materialized views [55], or CouchDB which offers *MapReduce views* [56] based on incremental map-reduce operations which eventually compose the contents of the view.

2.4 Data processing

This thesis not only focuses on data management for effective data sharing in a multi-provider context, but also concerns about data processing solutions for data-intensive workflows. This includes the exploitation of data locality with user-defined functionality, identifying and overcoming current difficulties related to data transformations, and analyzing current approaches for the parallelization of data-intensive applications.

2.4.1 User-defined methods

Computing close to the data has become a must in the last decade [57] [58], 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. To this end, DBMSs traditionally offer some kind of implementation of so-called *stored procedures* that allow the execution of arbitrary code within the database.

In relational DBMSs, stored procedures are usually offered through procedural languages that extend SQL statements with elements such as conditions and loops, declaration of constants and variables, etc. For instance, Oracle offers its PL/SQL [59], Sybase ASE and Microsoft SQL Server have Transact-SQL [60], PostgreSQL has PL/pgSQL [61] (which emulates PL/SQL to an extent), and IBM DB2 includes SQL Procedural Language which conforms to the ISO SQL's SQL/PSM standard [62].

The drawback of these solutions is the lack of interoperability, since applications need to create DBMS-dependent request statements to call procedure methods. Furthermore, any obtained result needs to be parsed or iterated following the rules of DBMS drivers (e.g. a *resultset* representing a set of rows matching a query).

Stored procedures are also supported in some NoSQL databases. For instance, Neo4j [63] manages stored procedures written in Java from version 3.0, and MongoDB allows storing JavaScript functions on the server side (although they do not recommend the overuse of this feature due to "performance limitations").

Nevertheless, there is still a drawback related to previous section: DBA centralization. **It is assumed that using arbitrary code programmed in common programming languages might jeopardize the database, therefore only its DBA or users with permissions explicitly granted by the DBA, are enabled to store procedures in the system.**

2.4.2 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, applications have to deal with 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.

Therefore, programmers are compelled to face the problem of handling two different data models and the mappings between them. For this reason, in the 80s, Object Relational Mapping (ORM) frameworks emerged to provide necessary mechanisms to automatically convert data between different type systems. However, current approaches like Hibernate [64] or DataNucleus [65] present some important drawbacks. 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 [41] or Intersystems Caché [40], make the persistence of objects transparent to the programmer. Furthermore, Intersystems Caché also offers a good level of interoperability between different languages such as C++ and Java, thus having the capability of storing and access objects from applications coded in any of the supported languages. However, **both db4o and Intersystems Caché present a lack of flexibility in model sharing** as exposed in section 2.2.

Some DBMSs such as Oracle, Microsoft SQL Server , or Redis [66]; already offer the possibility to be configured as in-memory databases. With the inclusion of NVRAM technology in the I/O stack, **in-memory databases should be able to run at full speed and maintain data in the event of power failure**.

In that connection, the gains of NVRAMs are already proven successful in filesystems, which traditionally assumed the presence of block-devices as the underlying storage hardware accessing them through block I/O operations leading to large amount of data migration (depending on the block-size). However, newer prototypes such as SCMFS (Storage Class Memory filesystem [67]) or RAFS (random access filesystem for NVRAM [16]) are directly built on top of the memory access interface thus achieving better performance results.

In view of the above, **any future data store should be designed with a data model capable of taking full advantage of newer storage devices**.

2.4.3 Parallel programming and runtimes

The scope of this thesis with regard to parallel programming is focused on the different frameworks or tools that offer programming models for the parallelization of applications, as well as the runtimes required to orchestrate the resulting workflows.

For instance, Ibis [68] offers a programming model for compute-intensive workloads and includes a framework for programming and executing high-throughput applications. The drawback is that porting an application to Ibis requires the user to **explicitly implement an API corresponding to its specific pattern** and then compile the code using specific scripts. Another example is Swift [69], a scripting language oriented to scientific computing, which can automatically parallelize the execution of scripts and distribute tasks to various resources, exploiting implicit parallelism. However, Swift also entails **porting the applications to its scripting language**.

On the other hand, programming models for big data include models and frameworks related to the processing and generation of large data sets. In this group, the MapReduce programming model together with frameworks based on Hadoop [70] or Spark [71], are widely used and implemented. These frameworks provide good performance on cloud architectures, above all for data analytics applications on large data collections. The drawback of these popular solutions is precisely that they are **limited to MapReduce applications**.

Regarding workflow managers, Taverna [72] is a workflow language and computational model designed to support the automation of complex, service-based and data-intensive processes. It automatically detects tasks that are free of data-dependences, and it executes them in parallel. Pegasus [73] is another workflow manager that automatically maps high-level workflow descriptions, provided as XML files, onto distributed resources. The problem of these approaches is that the **workflow of the application must be statically predefined** in a way that the programmer requires specific knowledge of multi-threading, parallel and distributed programming or service invocation.

In light of the foregoing, it is worth introducing COMP Superscalar [74] (COMPSs). COMPSs is a framework that provides an easy-to-use programming model for task-based workflows as well as a suitable runtime to orchestrate their execution. In contrast to Ibis or Swift, its users just specify the tasks of their sequential applications that can be executed in parallel, and they are not limited to *MapReduce* or any specific programming model. Furthermore, in contrast to Taverna or Pegasus, COMPSs detects task dependencies at execution time. In this way, it exploits the parallelism at task level and distribute the corresponding tasks along the available nodes of the execution environment.

Nevertheless, in spite of all these advantages and before the realization of this thesis, COMPSs still forced the applications to deal with persistent data in the form of files, and thus it became a good candidate to design, implement and evaluate the contribution **C3** (detailed in chapter 5).

Chapter 3

dataClay: overview

This chapter exposes an initial overview of *dataClay* before going into detail about its design and technical aspects, which are extensively described in chapter 4. In the first place, concepts on which *dataClay* relies are exposed through the different levels of the ANSI/SPARC architecture (introduced in section 1.1.2) and, subsequently, *parceled control* is presented as the mechanism to decentralize the schema administration in the data store.

Thereafter, and on the basis of *parceled control*, the main pillars of the contributions of this thesis are established through next sections by addressing these topics:

- Schema sharing and flexibility to extend accessible ones without compromising existing data and prior functionality (regarding contribution **C1**).
- Management of data integrity constraints considering the decentralization of the administration (with regard to contribution **C2**).
- Execution environment of *dataClay* specifically designed for an object-oriented data model (in connection with contribution **C3**).

3.1 ANSI/SPARC architecture in dataClay

This section describes how the three layers (external, conceptual, internal) of the ANSI/SPARC architecture are mapped to the *dataClay* abstractions used to structure and manage data. To begin with, the conceptual level of *dataClay* is introduced given its direct connection with the main contributions of the thesis. Secondly, it is presented how the conceptual model is exported to users (external level) and, finally, how it is mapped to the underlying storage (internal level).

For the sake of clarity, the mappings with each of the ANSI/SPARC levels are described as if there was a single administrator for the whole data store. This provides a clear picture of how *dataClay* works analogously to other data stores with respect to the essential ANSI/SPARC characteristics that are still valid for its design. However, some differences are clarified at each level to finalize the section exposing how, in contrast to what is assumed in ANSI/SPARC, *dataClay* divides data management into different parcels that are separately controlled: ***dataClay* parceled control**.

3.1.1 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 to represent the internal state of the object, and a set of operations or methods that can be applied to objects of that type. Multi-level inheritance is also supported allowing any class to be a subclass of another class or subclass.

The internal state of the object (i.e. attribute values or object's data) is encapsulated through its corresponding class methods. In this way, neither the class attributes nor their values are directly visible outside the object, on the contrary, users are only allowed to execute its class methods. **This encapsulation ensures data independence by basing the manipulation of data on its identity and functionality 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 of literals are instances of primitive or built-in types, such as int, char or String.

Figure 3.1 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.

On the basis of 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

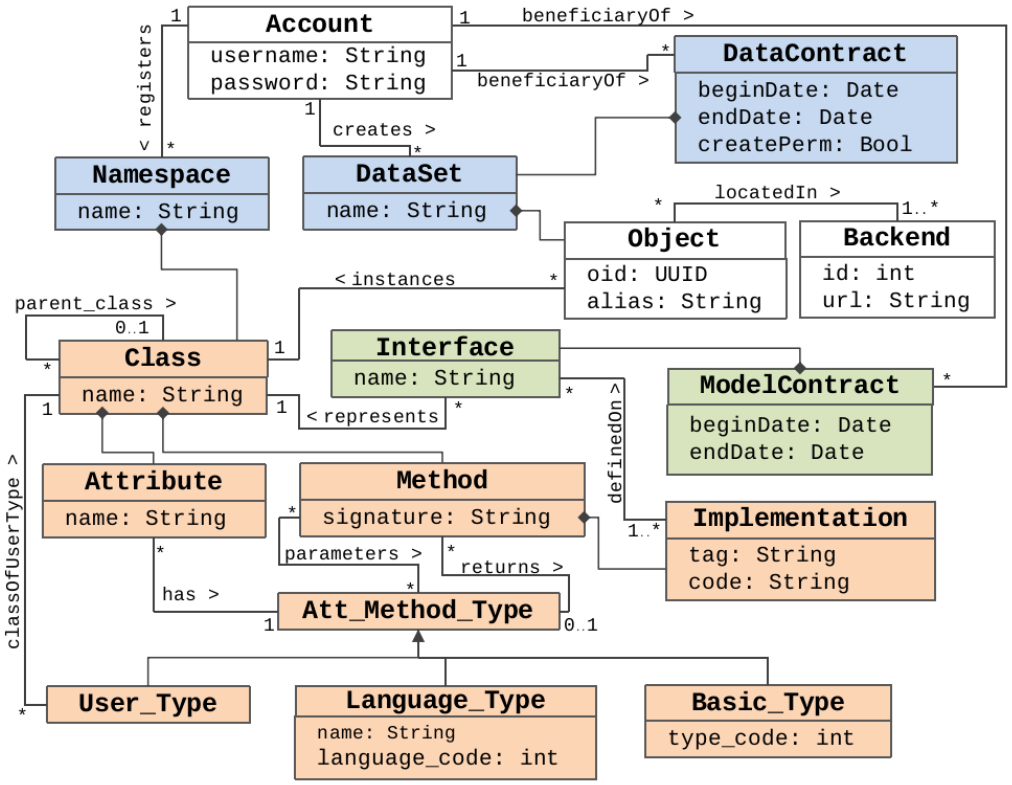

Figure 3.1: *UML class diagram representing dataClay's main entities. In orange, all classes related with the conceptual level. In green, classes related with external level. In blue, classes for parceled control. All relationships are navigational through methods (which have been omitted) or materialized attributes.*

also how it is used [10], and at the same time provides users with a transparent access to persistent data, thanks to the following principles:

- **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 facilitates 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 minimized.

The DDL that allows users to define the conceptual schema in *dataClay* can be any object-oriented language that enables them to describe their classes and methods. The current implementation supports top-most popular 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 security checks.

The fact that each user can access data only by means of a limited set of methods is a potential problem if the user requires a specific method that is not available. However, and precisely because data is encapsulated, any user is allowed to implement a new method using only his available ones, as well as additional data that this method may need, to provide new functionality without any danger for the already existing data. This is so-called ***enrichment*: if existing classes and methods do not correspond to what the user needs for his applications, he can expand a class with new methods and the necessary additional 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. Section 3.2 describes 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.

Going one step further with regards to the object-oriented data model, enrichments provide the possibility of having several implementations for each method. This 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.

3.1.2 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 model of the conceptual level, providers populate their data models by means of *dataClay* interfaces and model contracts (as shown in figure 3.1 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, and a subset of the contents of each class) that the user can 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 section 3.2).

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 [10], 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 section 3.2). 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.

3.1.3 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, *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.

In fact, some mappings are already implemented for the object-oriented database db4o [41], the key-value column oriented Cassandra [34], the Neo4j [63] graph database, and the relational database PostgreSQL. An additional mapping has been also implemented based on the novel Seagate's Kinetic hard drives [75], which are disks with a key-value interface instead of a block interface, and are also working in a mapping to NVRAM in a research collaboration with Intel. This shows that *dataClay* is able to directly access storage devices without the need to go through a 3rd party data store, and benefit from advances in storage technology. However, these novel storage technologies are not yet easily accessible, so it is necessary to rely on an off-the-shelf database to implement the internal level.

Either way, and as will be seen in section 4.2, the fact that in *dataClay* objects are only accessed by their OID facilitates the simplification of the internal schema to its minimal expression since most of the work is done in memory and with cached objects that are already instantiated. For instance, in the mapping to PostgreSQL the schema consists of a single table that contains all the objects. This table is defined with a primary key column on behalf of the OID, and a second column for the object data represented as a codified byte array containing all the values of its properties. References to other objects, such as a property which value points to another object, are analogously stored within the byte array by only saving their OIDs, since an OID is the minimum information required to look for any *dataClay* object. Chapter 4 describes how objects are distributed in several locations, and how their physical location is transparent to users.

3.2 Schema sharing and evolution via parceled control

As introduced in section 1.1.2, database administration is traditionally centralized under the responsibility of the database administrator (DBA), which is the role in charge of dealing with users to understand their requirements and define the conceptual schema and external view, as well as the security and integrity constraints.

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 provider. In this scenario, the data provider 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 provider to offer them the required functionality, which might be very specific to the domain of the data consumer, or the base of his business model hence nobody else can or should do this job.

In this context, an alternative mechanism to maintain the security and integrity of the database is needed so the requirement **R1** (section 1.3) is fulfilled. 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 (requirement **R2**), and at the same time it must be considered to offer the proper mechanisms for data integrity across providers' datasets.

In *dataClay* this mechanism is referred as **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 3.1 shows in blue color the entities concerned in parceled control and explained hereafter.

In a first step to support *parceled control* in schema sharing, it was defined a new abstraction for *dataClay*'s data model called **namespace**. In particular, 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 (requirement **R3**, contribution **C1**). 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 (requirement **R2**). 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 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.

3.3 Data integrity with decentralized administration

As introduced in section 1.1.3, and further extended with related work in section 2.3, ensuring data integrity is a widely supported feature of current DBMSs, especially those that are schema-based. In *dataClay*, integrity constraints are supported either from the implicit characteristics of the OO data model, or by providing schema developers with the proper mechanisms to ensure them.

3.3.1 Schema constraints

Schema constraints are those related to originally defined as declarative constraints for RDBMSs, comprising: entity constraints, domain constraints, check constraints and referential constraints.

Entity constraints were introduced in RDBMSs to ensure that there are no duplicate rows in a table. However, OODBMSs (and modern NoSQL databases) do not support them since this concept of uniqueness is only required globally to identify object references, which are based on their OIDs. Analogously, since *dataClay* is based on an OO data model, global unique identifiers are used to specify objects' OIDs. However, *dataClay* also enables the users to define optional alias names for their objects in order to distinguish them in a per-class fashion. If the schema developer still needs to define an attribute or set of attributes to identify an object from other objects of the same class, he can implement secondary objects in the form of indexes or dictionaries and check uniqueness from class constructors. For instance, a developer registering a *Book* class defines the *isbn* attribute as identifier through, and every time a new *Book* object is created the constructor checks an external *BookIndex* object containing the set of existing *isbns*.

Domain constraints are implicitly resolved using an OO data model by means of defining class attributes with a specific type, which in *dataClay* can be either language types provided by the programming language (basic types or classes) or user-defined types from the registered classes of providers' schemas.

Check constraints can be enforced from *setter* methods, since *dataClay* enforces the encapsulation of object's state through its corresponding class methods. Defining a *setter* method per attribute, developers can register the proper algorithm that serves to check the correctness of their assigned values.

Finally, and as introduced in 1.2.3, the only problematic declarative constraints are the referential ones. In a multi-provider context with a parceled control as described in 3.2, objects can be stored in multiple datasets with potentially different permissions (different data contracts). Therefore, not only *removes* can be problematic as in prior state of the art, but further complex is the fact that every consumer has a potentially different vision of the stored objects depending on his particular access permissions. In this context, referential constraints can be compromised if, for instance, two interrelated objects are stored in different datasets and the user trying to navigate from one object to the other only has permissions for the first one. This means that referential constraints must be resolved in

a per-consumer basis, since another user might have permissions to access both datasets thus presenting no difficulties. To this end, and related to contribution **C2**, *dataClay* offers the mechanisms to enable schema developers to check permissions from their methods, which in conjunction with data encapsulation through methods, allows programming the corresponding *getter* methods (e.g. "obj.getAttributeX()" where attribute X refers to another object) by retrieving consumer permissions to determine what to do. For instance, if the user has no access to referred object, the schema developer could decide returning a null value, or an empty object on behalf of the real object, throwing an exception, etc.

3.3.2 Derived attributes

The solution proposed for referential constraints is also valid for derived attributes since, as exposed in 2.3.3, in an OO data model enforcing encapsulation through methods, they can be offered through *getter* methods. Therefore, a derived attribute calculated from other objects data, should check access permissions of any involved object from the corresponding algorithm implemented within the *getter* method. In this way, the developer might decide that final value is composed only by the accessible ones, or an exception must be thrown if some involved object is not accessible, etc.

3.3.3 Active behaviors: triggers and materialized views

In *dataClay*, triggers and materialized views are supported on the basis of an event-driven subsystem (detailed in chapter 4.3) through Event-Condition-Action (ECA) rules defined by the users. Specifically, developers might subscribe their classes to particular events offered by the system (further detailed in chapter 4). In these classes, developers define the *condition* to be checked on the object related to the produced event (if any), and the *actions* to be performed if the condition is met. Both *condition* checkers and *actions* are also methods of the same class.

The following example, illustrated in figure 3.2, serves to expose the integrity problems described in section 1.2.3 with regard to accessibility permissions in user-defined active behaviors. It actually shows the particular implementation of materialized views in *dataClay*, but this includes the logics involved in triggers.

To begin with, a developer codes a *RedCars* class representing a collection of red cars and wants to keep its objects' instances up-to-date any time an accessible red car is stored in or removed from the system. The developer codes the class to be registered to the

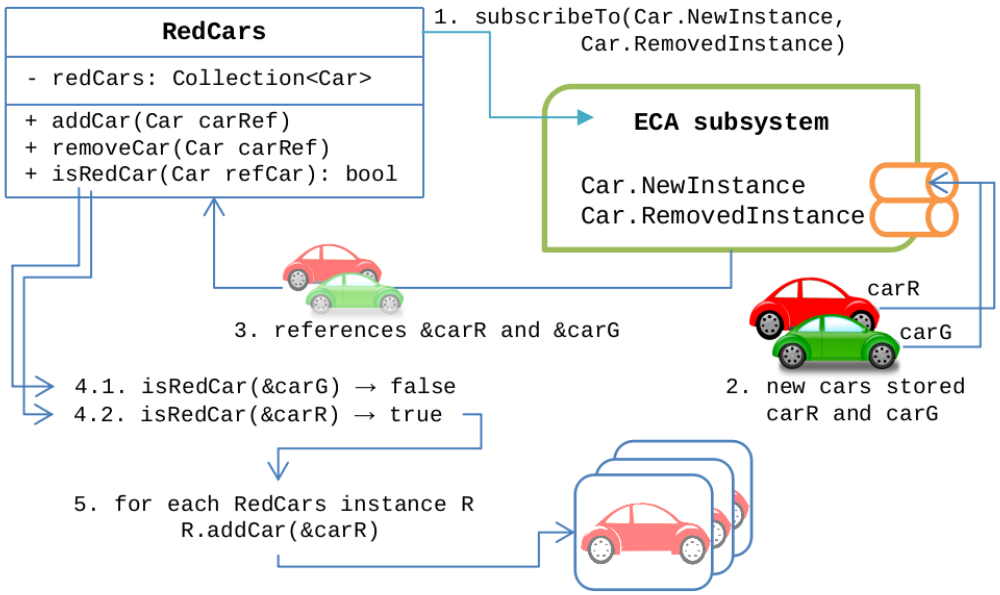

Figure 3.2: *Up-to-date collection of red cars with ECA subsystem.*

events *NewInstance* and *RemovedInstance* of class Car, which are events supported by the event-driven subsystem for any registered class. The developer codes a method in *RedCars* class that acts as the *condition checker*, and names it *isRedCar*. This method receives the reference of the new or removed instance of car whenever the notifications of the subscribed events are produced, and checks if the car is red in order to determine whether it must be added to or deleted from the collection. In the former case, if *condition checker* resolves to true the *action* to be taken corresponds to the execution of the method *addCar* present in the *RedCars* class and targeted to objects instancing it (which must be kept up-to-date). Analogously, if a red car is removed the *action* would be executing *removeCar* method.

In this context, **active behaviors involve the same difficulty as referential constraints**. Car object references might have different access permissions for different users and thus these triggers could compromise data integrity. However, analogously to the solution proposed for referential constraints, and in regard with the contribution **C2**, developers are enabled to check access permissions from the action methods. That is, *addCar* or *removeCar* might check the accessibility to car object (i.e. its dataset) before performing the corresponding action. This entails some particular issues that are further detailed in chapter 4.

3.4 Execution environment

On the basis of the *parceled control* presented in section 3.2 and the object encapsulation of the conceptual level introduced in section 3.1.1, *dataClay* offers an execution environment to carry out all the computation related to stored objects. This execution environment can be distributed along several nodes which are enabled to both storing objects and invoking their methods.

In this context, a user authorized to access the objects within a dataset (considering current data contracts) can code an application that executes their corresponding class methods (considering current model contracts) encapsulating them. These execution requests are submitted through provided client libraries to be eventually handled by *dataClay*.

To this end, and in order to make the applications to be easily adapted to *dataClay*, **model contracts are translated to actual classes that can be used from within the applications as any other regular class**. These contract classes, so-called **stub classes, are available for the user as long as the corresponding model contract is active** (i.e. has not expired). In addition, these stub classes are provided with all the extra methods coupled to the *dataClay* I/O interface, thus enabling the applications to generate new objects or delete accessible ones (according to current data contracts).

Through the integration of application and data models, *dataClay* aims to tackle the **difficulties of impedance mismatch present at multiple levels from the application and up to physical data representation**. On the one hand, avoiding common explicit transformations required to connect applications with current Data Services, which traditionally need to serialize and transform data with specific formats, such as JSON or XML in REST APIs. On the other hand, the OO data model is also chosen to exploit the characteristics of upcoming NVRAM technologies, which will help mitigating the data conversions between the conceptual level and the persistent storage.

In view of the above, the following chapter 4 describes the main design and technical details about the execution environment mechanisms to fulfill all these application requirements, on the basis of a bidirectional interaction between applications and *dataClay* through stub classes and/or *dataClay* libraries, and the execution request handling that is performed within the distributed execution environment.

Afterwards, chapter 5 evaluates it through different performance studies, and analyzes

different applications to validate the integration of OO data model with a parallel programming model according to the contribution **C3**.

3.5 Summary

This chapter introduces *dataClay*, a novel object-oriented data store that serves to validate the proposed contributions of this thesis. In the first place, it exposes the mappings between the three-layered data modeling proposed in ANSI/SPARC architecture and the abstractions defined within *dataClay*. Considering that the majority of applications are coded with object-oriented programming languages, an object-oriented data model resolves the common impedance mismatch difficulties by coupling the external and conceptual layers with the same data representation. Moreover, near future storage technologies, such as NVRAMs, will presumably facilitate the persistent data to be stored in an object-oriented fashion thanks to its byte-addressability, and therefore all data abstractions could be presented with a single unified data model.

Thereafter, it is introduced a first overview of the so-called *parceled control*, which provides specific mechanisms to decentralize the schema administration as opposed to having a central administrator as defined by the ANSI/SPARC architecture (i.e. the DBA). In particular, *parceled control* enables any user of *dataClay* to manage his own persistent data and schema, as well as the permissions to share both with other users. In order to facilitate the management of persistent data, *dataClay* offers the abstraction called *dataset* to define common permissions for a set of objects (associated to it) and *data contracts* to share them. On the other hand, the schema administration entails the management of user-defined classes with so-called *namespaces*, which can be shared through *model contracts*.

Thanks to data encapsulation through class methods, flexibility (contribution **C1**) and integrity (contribution **C2**) are supported without compromising neither existing persistent data nor current schemas. In respect of flexibility, the data model in conjunction with the *parceled control* enable authorized users to extend schema by means of new structures and functionality without jeopardizing current ones. These extensions are called *enrichments* and can be also shared as any other part of the schema. On the other hand, the required mechanisms to support user-defined data integrity constraints are also presented, with special attention to referential constraints that serve as the basis to fully support derived attributes and active behaviors in the context of multi-user scenarios with

potentially different permissions.

Finally, this chapter also introduces the main ideas behind the *dataClay* execution environment. From the application perspective, stub classes are used for the external schema built from the *model contracts* available for each user, and acting as the main entry points to the system by producing execution requests to registered class methods. Internally, this execution environment is eventually in charge to conduct the actions programmed within these methods and user-defined active behaviors, enabling near-data processing and thus facilitating contribution **C3** as it is described in the next chapters.

Chapter 4

dataClay: design and technical details

This chapter describes the key technical aspects of *dataClay* for the implementation of the concepts exposed in previous chapter 3. Along the chapter, references to the contributions are explicitly established in order to easily identify which are involved for each concept.

For a better understanding, figure 4.1 illustrates the main components concerned and the interactions between them.

To begin with, *dClayTool* is a user-level application that facilitates Management Operations related with data access control and the definition, evolution and sharing of the conceptual and external schemas. 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 local database.

On the other hand, users are also provided with a client library that enables the development of their applications, using an API to communicate with *dataClay* to initiate a session and submit any of the supported requests. Specifically, 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 (i.e. backends are intercommunicated).

LM also handles a central repository of object metadata and the ECA subsystem for active behaviors. The metadata repository gathers information about persistent objects (e.g. location), and serves DS backends requiring any metadata information (further

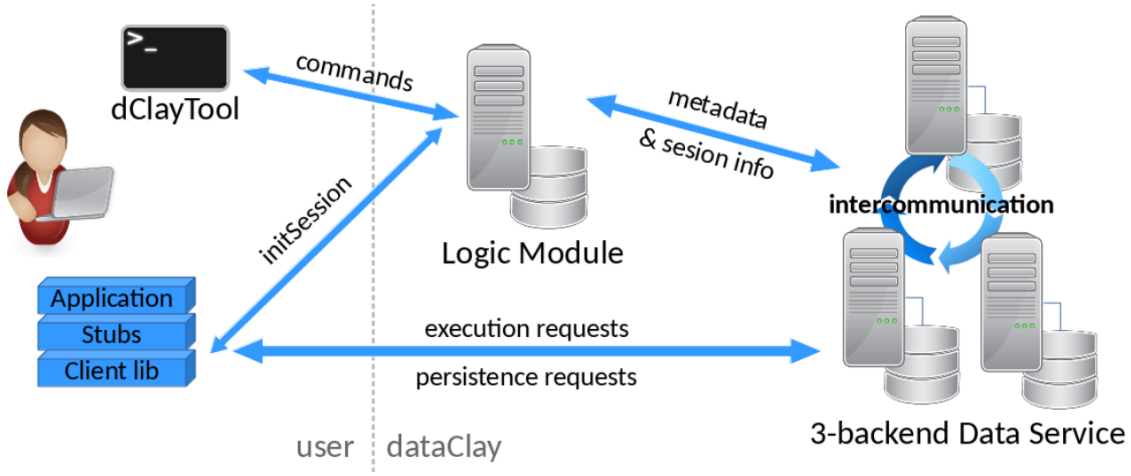

Figure 4.1: *User-dataClay interactions with commands launched from dClayTool and session and execution requests launched from user-level applications. Logic Module and a 3-backend Data Service are deployed in different nodes with the corresponding data and metadata intercommunication.*

discussed in section 4.2.2). Regarding the ECA subsystem, LM collects the available events and subscriptions to them defined from classes requiring certain active behaviors (further detailed in section 4.3).

Having *dataClay* components already introduced, the main design and implementation details behind the proposed contributions of this thesis are explored along two major sections:

- Parceled control and data integration, describing the mechanisms to effectively decentralize the schema administration for a multi-provider scenario, with particular attention to the flexibility to enrich existing schemas.
- Execution environment for data processing, with a distributed architecture for parallel computation and object handling, as well as the capability to effectively manage user-defined data integrity constraints and language interoperability.

Finally, and for the sake of clarity, the ECA subsystem, and its particularities related to the contributions, is completely exposed in a separated section 4.3 as an extension of previous ones.

4.1 Parceled Control and data integration

Management operations are those related with **parceled control**: schema registration, enrichments, and data access granting. Schema registration includes creating namespaces, registering classes and defining interfaces and model contracts for **schema sharing**.

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

Figure 4.2: *Syntax of dClayTool commands related to schema sharing and dataset access control.*

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 for data sharing.

Schema sharing through model contracts and the flexibility to enrich accessible schemas materialize the contribution **C1**, whereas data access operations will be considered along the next section 4.2 and thus fulfilling contribution **C2**.

For a better understanding, this section exposes management operations through the *dClayTool*, a user-friendly command-line tool that serves providers to handle them. In particular, the most relevant commands are shown in table 4.2. Throughout this section it is assumed that *dClayTool* sends user's credentials to LM along the requests, thus LM authenticates the user and performs any necessary checks. Finally, subsection 4.1.4 describes how users retrieve those classes (i.e. stub classes) according to their accessible external schema, which is a key point to understand the details of next section 4.2 about the execution runtime for data handling and processing.

4.1.1 Schema Registration

Given a user wanting to register a schema defined with classes shown in figure 4.3 (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 local 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

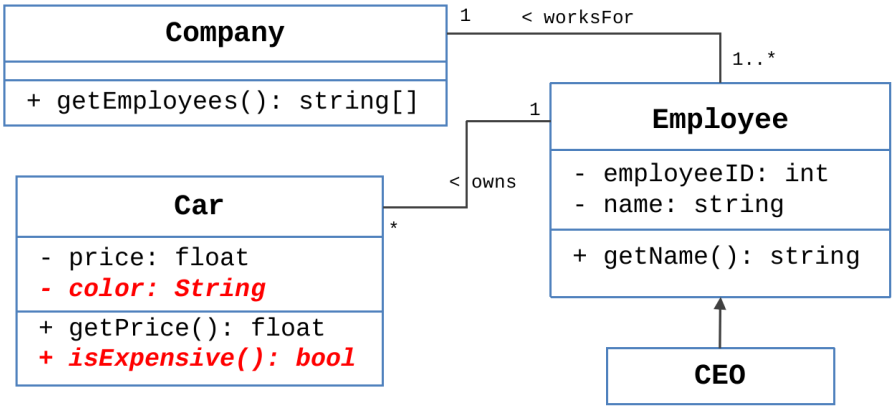

Figure 4.3: *Schema example with different kinds of associations and enrichments highlighted. Company class is registered associated to a set of Employees with at least one CEO. Employees may own some Cars, but it is assumed that a car cannot be owned by different employees.*

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.

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, and assuming that all 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, thus 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, if the implementation of class *Employee* has an attribute referring to his Company and Company has an attribute referring to its Employees, Company 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, which checks that there is no class clash in the namespace and deploys registered classes to Data Service (further detailed in section 4.2.3).

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 its beneficiary (command 4).

4.1.2 Enrichments

In line with the previous example, now the user wants to enrich class *Car* to include a new attribute for its 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's 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 4.2.3).

4.1.3 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 its beneficiary to execute class methods on all the objects registered within the offered dataset. However, create/delete permission (*createPerm* in figure 3.1) is configurable to further control whether the beneficiary of the data contract may create/delete objects in the dataset or not.

4.1.4 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 4.2.7). 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 persistent object referenced by current stub instance.

In the case of *makePersistent* method (further detailed in section 4.2.1), 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 an 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 it refers to a persistent object, 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 application context). To this end, stub methods are provided with one of the method implementations available according to users' model contracts. Since there may be several implementations available from different contracts,

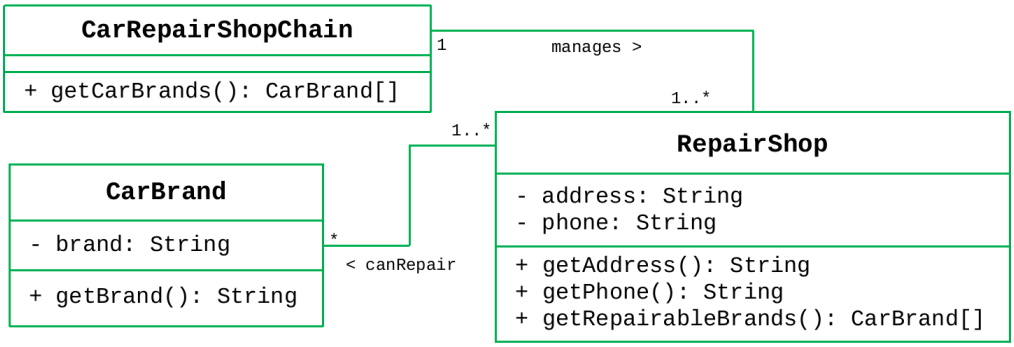

Figure 4.4: *Schema of the chain of car repair shops.*

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*: *serialize* and *deserialize*. The serialize method prepares a byte-array for the binary representation of 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.

4.1.5 Data sharing and integration examples

Following the company schema presented in section 4.1.1, 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 schema of the car repair chain is illustrated in figure 4.4. *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 schemas 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. Therefore, the developer could be eventually tempted to copy repair shops information locally, creating another *RepairShop* class for the company with objects performing a periodic synchronization with the chain Data Service to check for updates.

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 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 *address* and *phone* 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 schemas 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 4.5).

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

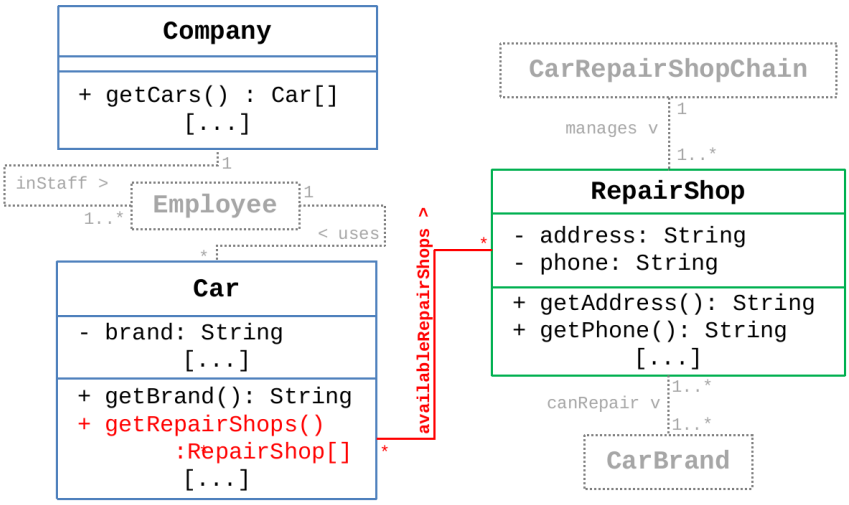

Figure 4.5: *Schema of the data integration between the company and the chain of car repair shops. Classes and associations in dashes means that are hidden for the non-proprietary of these entities. In red, the enrichments added to the company schema to get repair shops when needed.*

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 company 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 schemas, letting the chain to adapt *Car* class to its needs and adding new functionality accessible for the company.

4.2 Execution environment for data processing

This section presents the pillars of the proposed integration of programming and data models, which is related to the contribution **C3** and that is finally validated through next chapter 5. To this end, it is detailed how the user-level applications interact with *dataClay* to manage object persistence and stub method execution on existing objects, as

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

well as the interrelation between the different backends of the Data Service on behalf of the distributed execution environment.

In particular, 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. In this context, the user beneficiary of such a model contract has downloaded the corresponding stub classes and develops two applications linked with them:

- AppInsertCompanyData (figure 4.6): creates and stores the data of the employees in a company.

- AppGetCompanyEmployees (figure 4.7): 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 4.2.4. Throughout this section it is assumed that the application is already authenticated and has a session identifier for its execution 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 4.7: *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.*

4.2.1 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 session management in section 4.2.4). As objects are stored in the assigned DS backends (further detailed in section 4.2.3), they send the corresponding object metadata to Logic Module (LM).

LM keeps an up-to-date metadata repository, equivalent to a 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.

4.2.2 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 4.1.4) 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 [10] 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 4.2.3). 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, in Java the JVM heap memory acts as an object cache and objects are present there until the Java 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.

4.2.3 Execution classes - object handling

Execution classes are used in DS backends analogously to user's 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.

4.2.4 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, a session-based authentication mechanism has been implemented 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 this method:

```
dataClay.initSession(user, pass, datasets, datasetStore)
```

The *initSession* 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

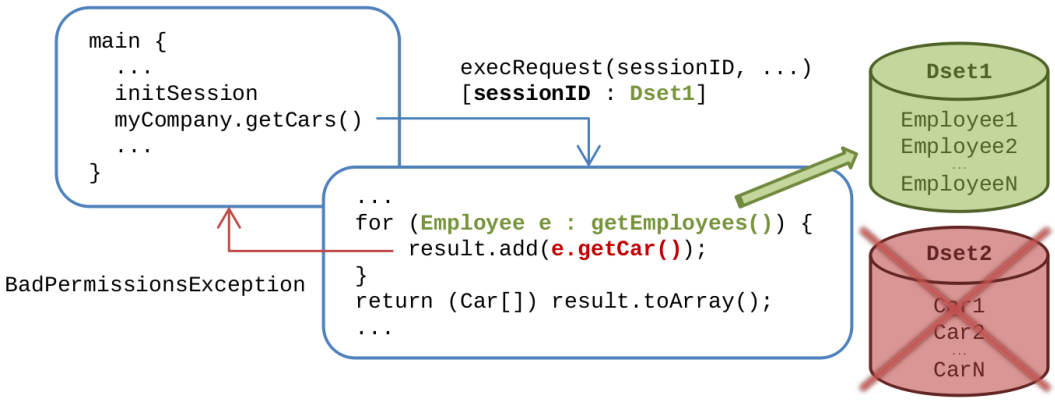

Figure 4.8: *Session management example where a shared method produces an exception due to bad dataset permissions from current session.*

session will be registered in the indicated dataset (*datasetStore*). 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.

As introduced in section 4.2.2, session identifiers are propagated through all the subsequent execution requests concerned along a workflow. An application that executes a method on an object stored in dataset *Dset1*, which eventually calls a second method on an object stored in dataset *Dset2*; sends its session information traversing all concerned data service nodes along the workflow. Therefore, as illustrated in figure 4.8, it might happen that the session has no permissions on *Dset2* so it will not be able to finalize its workflow since the second method would fail accessing objects in *Dset2* (raising an exception that can be handled as exposed in section 4.2.10). In this way, data is always protected regardless the permissions on schemas defined by current model contracts.

4.2.5 Check accessibility

The propagation of the session along the overall execution of any method request ensures that data access policies are never compromised. However, as exposed in section 1.2.3, referential constraints, derived attributes and user-defined active behaviors such as triggers and materialized views; require special attention in order to maintain data integrity involving objects from different datasets with potentially different access permissions for each consumer.

As introduced in section 3.3, the solution entails that the system provides the proper mechanism to allow schema developers to decide whether to ensure data integrity or not. In this regard, the proposed solution is based on the encapsulation through methods offered by the OO data model in conjunction with a particular system functionality that takes current data contracts into account.

For better understanding, an example is proposed based on previous *Company-Employee-Car* schema. Now, an object *Car1* of class *Car* has a reference to an object *Employee1* of class *Employee* if *Employee1* is using *Car1*. Assuming that *Car1* belongs to dataset *D1* and *Employee1* belongs to dataset *D2*, then a user with access permissions to *D1* and *D2* is enabled to navigate through the reference and eventually produce execution requests on object *Employee1*. On the contrary, another user with no access to *D2* cannot access *Employee1* thus cannot produce execution requests on that object. However, what about the reference?

In this case, the reference is actually an attribute in class *Car*, e.g. attribute *usedBy* of type *Employee*. In this case, given the encapsulation through methods, what is actually provided is a *getter* method such as *getUsedBy()* accessible through stub or execution classes and returning the reference to corresponding employee object. Therefore, to maintain data integrity is required a mechanism to check the current session before trying to access the pointed object. In *dataClay*, this is provided as a special method so-called *isAccessible()*, which is offered from the global class *DataClayObject* (like *makePersistent*) and takes the current session into account as any other method. In the example, the *getUsedBy* method would call *usedBy.isAccessible()* before returning the reference, and the Data Service backend executing the action would check the object metadata with the OID to indicate whether the current session has proper permissions to access the employee object or not. In this way, the schema developer might decide to return a null value, or a reference to an empty *Employee* with no information, throwing an exception, etc. In short, the developer chooses exactly how to maintain data integrity along his schema.

In case that object *Employee1* has been removed from the system, the reference is no longer valid, and *isAccessible* method will throw an exception that can be handled using a *try-catch* block. The developer has full control in this case, and can decide to set reference to null the first time *getUsedBy* is executed after the employee object is deleted.

In view of the above, the whole solution can be analogously applied for derived attributes, since developers enclose the algorithm to compute the derived value of the

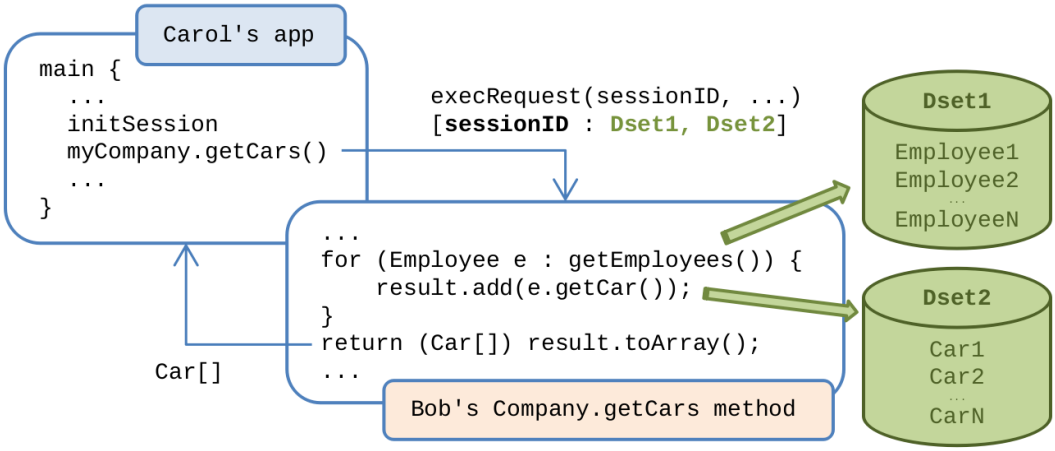

Figure 4.9: *Impersonation example. Application is enabled to execute getCars() through getEmployees() although having no model contract for Employee class.*

attribute on its *getter* method. Therefore, they can also decide what to return depending on the accessibility of the objects involved according to consumers' sessions.

This solution, in conjunction with the proposal for active behaviors explained in section 4.3, materializes the contribution **C2**.

4.2.6 Impersonation

A particular case of schema sharing is that users might code any class method on the basis of any of their accessible operations, either from their own classes or from their current model contracts with other users. Figure 4.9 (based on figure 4.8) illustrates a case for *Company-Employee-Car* schema presented in previous sections, and assumes that Alice has coded *Car* class and shares it with Bob and Carol. Then Bob codes *Company* and *Employee* classes but only shares *Company* with Carol. In this context, Carol has no access to *Employee* class, but with the model contracts with Bob and Alice she should be able to execute method *getCars()* although internally it uses *Emloyee* class.

To this end, *dataClay* implements impersonation on the execution of class methods. Given that data access is eventually protected as exposed in sections 4.2.4 and 4.2.5, *dataClay* allows Carol's application to use any of her accessible methods (first nesting level). The application is enabled to execute *getCars()* and, from then on, it will remain agnostic of any of the nested inner calls. That is, *getCars()* is executed as Bob programmed it, with the particularity that Carol's session identifier is the one used along all the workflow levels. Therefore, if Carol does not have a data contract for *Dset2*, access to car objects will be forbidden regardless of whether Bob has access to them or not.

4.2.7 Interoperability

Interoperability between different languages further improves the materialization of the contributions **C1** and **C3** by adding an extra layer of schema and data sharing between the programming languages supported.

With the abstraction of the conceptual schema, *dataClay* allows to retrieve stub classes in any of the supported languages regardless of 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 current implementation. Consequently, the retrievable stub methods for a language different than that of the corresponding class, are those having compatible parameters and return values.

As described in section 4.1.4, 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.

4.2.8 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.

4.2.9 Memory swapping

There are some situations where a method requires an amount of memory that exceeds the current available quantity. For instance, the execution of *AppInsertCompanyData* (figure 4.6) for Wal-Mart Stores (with 2,200,000 employees in 2016), could incur an *OutOfMemory* exception produced from the JVM due to the excessive amount of objects inserted in the *employees* linked list.

This problem is particularly prevalent in a shared environment like *dataClay*, where not only singular methods might require an undue amount of memory but the execution of too many methods concurrently might surpass it. Furthermore, the intervention of the language Garbage Collector (GC), already described in section 4.2.2, cannot help in this situation since indeed the language GC cannot collect anything at all in this case. Therefore, it can be said that this is an irresolvable problem that can occur at any moment

when required data does not fit in memory.

However, *dataClay* helps to mitigate this inconvenience in two different ways.

First, *dataClay* provides every object instantiating an execution class with the flag *isLoaded* that determines whether the object is actually in memory or not. This means that any particular instance of an execution class in the memory space of a DS backend, might or might not contain the actual data represented by this object. The object is actually loaded when it is accessed for the first time.

On the other hand, *dataClay* provides so-called *proxy references* that enable users to define their classes with internal structures that can be swapped to storage by the language GC in case of memory pressure.

For instance, assuming that previously presented application *AppInsertCompanyData* is turned into a method of a registered class, the collection of employees could be split into a collection of chunks, having a manageable number of employees per chunk. In this case, the programmer would define a maximum size per chunk, thus in case a new chunk is required to add an employee not fitting on existing ones, the collection would create it and register it as a proxy reference. The difference in this case is that the programmer uses a special method called *newProxy* instead of *makePersistent*. This *newProxy* method creates a proxy object on behalf of the actual chunk, making the collection to actually point to these proxy objects instead of chunks. These proxy objects only contain the OID of the represented chunks, but no memory references to chunks. Therefore, in case the language GC is triggered, chunks and their contained employees can be garbage collected (and removed from memory) because the collection object (and anything else) is not referring to them. When the collection object accesses any of the proxy objects, the execution environment in the Data Service node will check if the OID corresponding to its represented object, i.e. a chunk in this case, is present in the cache to actually access it. If this is not the case, the chunk is then loaded from the underlying storage.

4.2.10 Exception handling

One important feature of today Object Oriented programming languages is exception handling, a process of responding to the occurrence of anomalous or exceptional conditions requiring special treatment. Exception handling is provided by specialized programming language constructs that save the current state of execution in a predefined place and switch the execution to a specific subroutine known as an exception handler. The exception

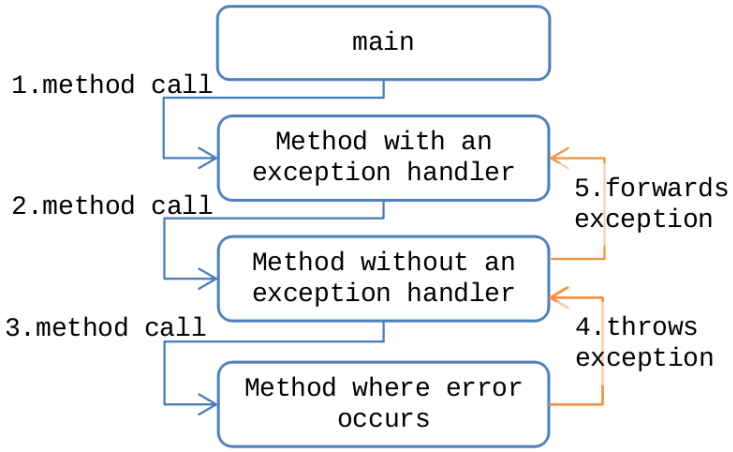

Figure 4.10: *Exception handling - call stack and exception propagation*

handler might be defined in the current method that raised the exception, or in any previous method in the call stack. This means that the exception is actually propagated until the runtime system finds a method of the call stack that contains a block of code that can handle the exception (typically a *catch* block).

Figure 4.10 illustrates an example of exception handling. An application starts its execution from the *main* method and produces three consecutive nested calls to different methods. The first method contains an exception handler, the second method does not, and the third method throws the exception. When the error is produced and the exception is thrown, the runtime system searches the stack call for the exception handler, forwarding the exception in the second method to eventually handle it in the first method.

Although in *dataClay* the runtime system is distributed along the DS backends, exception handling is currently supported for the Java programming language (Python exceptions are not supported yet). Developers can register Java classes with methods that throw exceptions, which can be propagated through the call stack traversing different DS backends or reaching a user-level application if necessary. That is, following the previous example, if the method with the exception handler started in a DS backend different than the one processing the method that throws the exception, this would be propagated from the latter to the former through any DS backend in the middle (e.g. the second method, which forwards the exception, is executed in another backend different than previous two). In the same way, if the exception handler is located in a method coded within the application the exception is also propagated up to this method.

To carry out this undertaking, current version of *dataClay* falls back on the Java

serialization mechanism which is implemented by the Java *Exception* class (i.e. Exception class implements the *Serializable* interface). Therefore, the exception can be forwarded externally by serializing and deserializing it properly to sent it through DS backends or up to the user-level application.

4.3 Active Behaviors through ECA subsystem

dataClay implements an event-driven subsystem, so-called ECA subsystem, to support Event-Condition-Action rules defined by the users. In this way, *dataClay* provides a mechanism to define triggers and materialized views, for instance, in the form of derived collections; which are well-known features in the state of the art (as exposed in sections 2.3.2 and 2.3.3). To begin with, details of the ECA subsystem are explained, and afterwards the example of section 3.3.3 is revisited to finally explain the solution proposed for integrity constraints (included in contribution **C2**).

4.3.1 ECA subsystem

The ECA subsystem consists of a service offered by the Logic Module that registers all the possible events and conducts the corresponding actions. In the current implementation, supported events are:

- Time event: a system event that enables the users to register a specific method to be executed on a certain date.
- New and remove events: events that are produced whenever a new object is created in or removed from the system.

The *Time Event* is basically a mechanism that enables the users to schedule a method to be triggered on a specific date. When the date is reached, methods are executed regularly by creating a session from the current available contracts of the user that registered the action. To this end, the client library offers the method:

```
dClayTool -executeOnDate date namespace class method
```

With respect to class events, every user class registered in the system exposes two kind of events: *NewInstance* and *RemovedInstance*. Given a certain class, *NewInstance* occurs when a persistent object of such a class is stored in the system. Analogously, *RemovedInstance* is produced when a certain persistent object is deleted from *dataClay*.

```
public class RedCars {
  // subscriptions
  public static ECANewInstance carNewInstance =
    new ECANewInstance(Car.class, isRedCar, addCar);
  public static ECARemovedInstance  carRemovedInstance =
    new ECARemovedInstance(Car.class, isRedCar, removeCar);

  public Set<Car> redCars; // collection of cars
  public RedCars() { // constructor
    redCars = new HashSet<Car>();
  }
  // static condition checker executed on class object
  public static boolean isRedCar(Car carRef) {
    return carReg.getColor() == "red";
  }
  public void addCar(Car carRef) {
    if (carRef.isAccessible()) {
     redCars.add(carRef);
    }
  }
  public void removeCar(Car carRef) {
    redCars.remove(carRef);
  }
}
```

Figure 4.11: *RedCars class with subscriptions to NewInstance and RemovedInstance events of class Car.*

In order to subscribe a class to any of these events exposed by any other class, *dataClay* offers two special class types that are used to define static attributes: *ECANewInstance* and *ECARemovedInstance*. These types are translated to subscriptions in the ECA subsystem when the class is registered in the Logic Module.

4.3.2 ECA example and integrity

Following the example in figure 3.2, *RedCars* class would be coded as illustrated in figure 4.11, having one static attribute per type: *carNewInstance* and *carRemovedInstance*. These attributes are statically initialized to indicate the methods to be executed as the *condition checker*, which is the same in both cases: *isRedCar*; and as the *actions*, which are *addCar* and *removeCar* accordingly.

Once the class has been registered, whenever an object is stored or removed from the system not only the object metadata is updated but also the ECA subsystem initiates the required notification processes. In particular, the ECA subsystem checks the recorded subscriptions and notifies the event to the corresponding classes offering the reference to the source object that produced the event. Following the example of figure 3.2, the ECA subsystem would notify *RedCars* of the creation of both the red car *carR* and the green car *carG*.

Specifically, both notifications would start with special execution requests to the *condition checker* defined in the subscriptions, i.e. *isRedCar*, passing the reference to source object as the parameter, *carR* and *carG* respectively. This execution request is special for two reasons. Firstly, because it does not require the information of any particular persistent object, thus it is defined as a static class method that can be executed just loading the corresponding class (i.e. targeting the class object). Secondly, because in case that this static *condition checker* is resolved to *true* (e.g. *isRedCar(carR)*), a set of regular execution requests to the defined *actions* (e.g. *addCar*) are then triggered targeting the objects instancing the subscribed class.

In particular, the *condition checker* method is forced to be a static class method and the ECA subsystem chooses a random DS backend to execute it, since classes are deployed to all DS backends. The selected DS backend tailors the subsequent *action* methods in case the *condition checker* returns a true value. Therefore in this case, the DS backend produces local and remote execution requests depending on the locations of the objects instancing the subscribed class (these are regular requests as those described in 4.2.2).

Resuming the example, the DS backend would manage the special execution request for *isRedCar(carR)* and *isRedCar(carG)*. In the former case, where *condition checker* is resolved to true, regular execution requests will be submitted to objects of class *RedCars* to perform the action *addCar(carR)*. That is, more than one object might be instancing *RedCars* class for different purposes, thus all them must be notified to execute the defined *action* of the ECA rule.

For security reasons, the *condition checker* can only perform *getter* actions on the object that produced the event, since it is executed globally only once to decide whether the action has to be performed or not. On the contrary, the *action* method is executed targeting every instance of *RedCars* thus can perform any programmable action.

Therefore *action* methods require a session, but they have not been registered by the executor as *Time Events*. In this context, it is decided that the session applied for the *action* method is defined by the contracts of the owner of the targeted object. That is, once a user creates an instance of *RedCars*, his particular contracts are considered for subsequent *action* methods produced on such an instance.

With regard to data integrity, this enables an analogous behavior as the one defined in section 4.2.5, which completes the materialization of contribution **C2**.

4.4 Summary

This chapter describes the design and implementation details of *dataClay* that are required to sustain and validate the logics behind contributions **C1** and **C2**, as well as exposing the core concepts for the contribution **C3** to be afterwards evaluated in chapter 5.

To begin with, the main *dataClay* components are introduced: user-level libraries, the *dClayTool* the Logic Module and the Data Service. User-level libraries in conjunction with stub classes and the *dClayTool* are intended for communicate applications with the system and for administration tasks. Logic Module handles object metadata such as object locations and *dataset* permissions defined by data contracts, as well as *namespaces* and information of user-defined schemas (classes) with their associated *model contracts* for stub generation. Last but not least, the Data Service is a distributed object store with the proper execution environment for near-data processing.

With a global picture of *dataClay* essentials, so-called *parceled control* is detailed as the solution proposed in *dataClay* for the decentralization of the schema administration, which has been identified as a mandatory requirement for a multi-provider data store. For the sake of clarity, the *dClayTool* serves to illustrate the management operations available for the users to register their schemas and datasets, along with the contracts and permissions to grant other users to use and access them. Along this section, the logics behind contribution **C1** are also pinpointed through so-called *enrichments*, which are used to extend any accessible schema with new attributes and methods.

Thereafter, the chapter describes all the implementation details necessary to understand contributions **C2** and **C3**.

Contribution **C2** is sustained by means of the session-based mechanisms and accessibility checking. Thanks to data encapsulation through class methods in conjunction with accessibility constraints to check which is the current executing session, enable providers to determine the actual behavior of their class methods depending on the particular visibility scope and permissions of the current executing session. Therefore, referential constraints, derived attributes or active behaviors (presented through *dataClay* ECA subsystem) become fully supported as it is shown with several examples and/or use cases.

In respect of contribution **C3**, this chapter further details the execution environment of *dataClay* describing the main mechanisms behind the (distributed) Data Service:

persistent data generation, execution request handling and object serialization, class loading for data processing (execution classes), management of concurrent execution requests, language interoperability, etc. With all that, the integration of the multi-provider data model of *dataClay* with a parallel programming model becomes feasible and it only requires its evaluation with a particular example entirely exposed in the next chapter.

Chapter 5

Performance evaluation

This chapter exposes different performance analyses to evaluate *dataClay* as well as the feasibility of the contributions proposed in this thesis, especially contribution **C3**.

It is worth noting that contribution **C2** has been left out of the scope of this chapter since data integrity constraints are coded within class methods as part of their behavior, which makes this contribution to be not provable or measurable in terms of any performance metric or comparison. In compensation, previous chapters illustrated different examples through the design and implementation details of the corresponding *dataClay* features, such as derived attributes or active behaviors with ECA subsystem. Analogously, contribution **C1** is also presented followed with examples such as data integration use cases, but in this case there is a possible impact on performance that can be evaluated by measuring the overhead in arbitrary class growth due to enrichments.

Furthermore, it is also remarkable that the iterative methodology used in this thesis applies to the development of *dataClay*, including all the functionality and features that support all the proposed contributions. In this sense, an extensive battery of approximately 1000 tests was implemented, including: unit tests, functional tests, integration tests and code coverage analysis. All tests are daily executed whenever the code changes.

Taking previous considerations into account, the following sections are then focused on:

- Validating that the mechanisms implemented to fulfill the defined requirements introduce tolerable or negligible overheads. This study is conducted by means of the Yahoo Cloud Serving Benchmark (YCSB [22]), a well-known benchmark in

recent literature [76] [77] [78] that allowed to compare *dataClay* with popular NoSQL solutions in terms of latencies and throughput.

- Evaluating the possible effects of schema enrichments using the same benchmark, thus validating the contribution **C1**. This study has been also conducted with YCSB using several implementations of the same schema each of them comprising different amount of enrichments.

- Validating the contribution **C3** by means of the integration of *dataClay* with the parallel programming model offered by the framework COMP Superscalar (COMPSs [74]), through the performance analyses of different data-intensive applications.

5.1 YCSB benchmark

This section proves that, despite the novel features presented along the thesis, *dataClay* can be compared in terms of I/O performance with other trendy databases. In the first place, a comparison between *dataClay*, Cassandra [34] (version 2) and MongoDB [35] (version 3) is shown, both in terms of throughput (operations per second) and latencies when performing read and update operations. Cassandra and MongoDB were chosen because nowadays they are two of the topmost popular NoSQL databases [79] and have become key technologies to resolve common storage issues in Big Data scenarios [80] [81]. The last part of this section presents an analysis on the potential overhead produced by enrichments, a key *dataClay* feature for effective data sharing.

5.1.1 Methodology

The described performance studies are conducted with YCSB, that allows to evaluate and compare DBMSs both in throughput (operations per second) and latencies when resolving common CRUD (create/insert, read, update, delete) requests. 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 (read, update, etc.) and access distribution (zipfian, uniform, etc.).

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. Analogously, a driver for *dataClay* has been implemented based on the source code of previous two. In all cases, every 10 seconds, YCSB outputs stats on global throughput in operations/second (aggregating throughputs from all threads) and the latency per operation in μseconds.

The tests has been executed in 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. One node was dedicated for the client running the YCSB workload executor, and three 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*: Logic Module (LM) and 3 Data Service (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* the internal level representation in the DS backends uses the PostgreSQL handler, i.e. objects are eventually stored in an underlying PostgreSQL database. 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.

Two main workloads were specified to analyze the performance outcome in basic data I/O operations: workload *WR* and workload *WU*. *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 the evaluated cases are: 1 single thread executing 1M operations (*1*1M*), 2 threads running 500K operations (*2*500K*), 4 threads with 250K

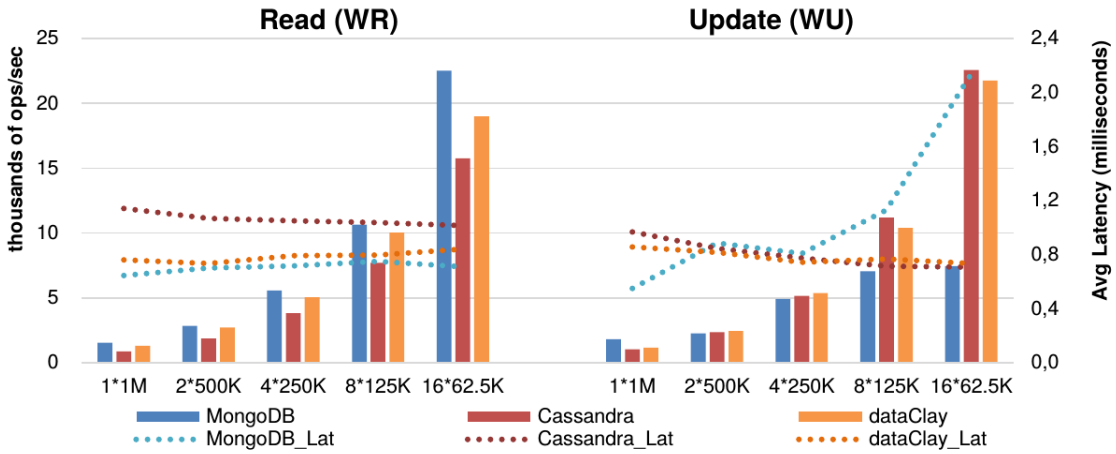

Figure 5.1: *Results of WR and WU workloads. In bars, throughput in thousands of ops/sec (left axis); in pointed lines, latencies in milliseconds (right axis). X-axis shows #threads * #ops per thread of the evaluated subcases.*

operations each (*4*250K*), 8 threads doing 125K (*8*125K*), and 16 threads performing 62,500 each (*16*62,5K*).

Regarding the access distribution, *Zipfian* is commonly used [82] since it represents a realistic behavior with some records being more popular than others. It is the default distribution in YCSB and the one used for the executed tests.

5.1.2 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 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 5.1 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 μseconds, *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 4.2.8). 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*.

5.1.3 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 5.2 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 ten byte-array attributes. Enrichments can be applied in several steps until the final class represents the record schema. In this regard, 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 been enriched 5 times with

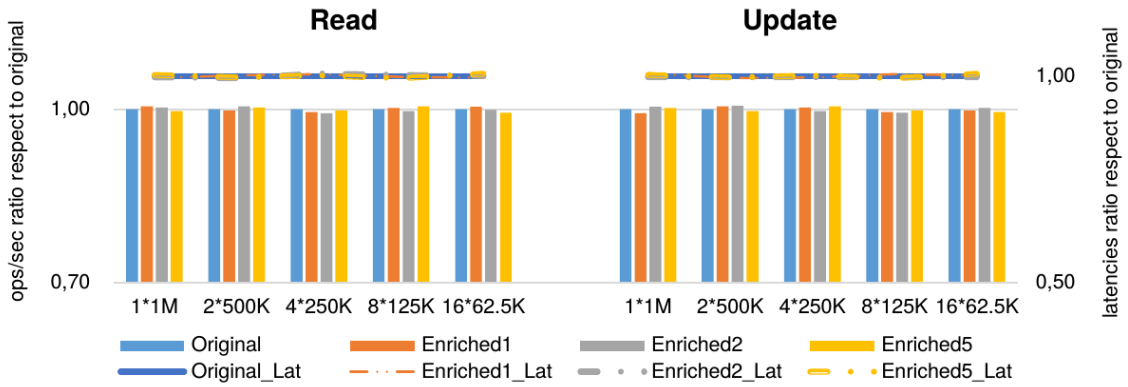

Figure 5.2: *dataClay with and without enrichments for WR (left) and WU (right) workloads. Bars represent the ratio of the throughput respect to the original dataClay execution (left axis), lines represent the ratio of the latencies (right axis). X-axis shows #threads * #ops per thread of the evaluated subcases.*

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 2.1±0.1 μseconds in Java and 1.1±0.1 μseconds in Python, which are negligible times unless there are hundreds of enrichments being applied simultaneously.

5.2 COMPSs and dataClay integration

This section shows the results after integrating *dataClay* with COMP Superscalar. As introduced in chapter 2), COMPSs is a framework that provides an easy-to-use programming model for parallel workflows and the runtime to orchestrate their execution. In COMPSs, the user only needs to define which methods of the application are tasks that can be executed in parallel, then the runtime conducts the workflow through the available computing nodes. In this way, **COMPSs can execute any conceivable**

parallel application, from embarrassingly parallel *MapReduce* workflows to the most complex workflows involving arbitrary levels of parallelism.

COMPSs uses a master-worker model to orchestrate the application tasks at runtime, considering the dependencies between them by analyzing their parameters and return values. The master typically runs on a single node (although it can share the node with a worker), and workers are deployed on the rest of the nodes available for the application execution. Workers are executed as independent services that remain waiting for input tasks coming from the master. Every worker is configured to use a set of the available resources on the node: a certain amount of CPUs, memory and disk, and specific network interfaces to communicate with the master.

In this context, the master process starts executing the application and whenever a method task is reached it is scheduled to a worker process which will be in charge to compute it. COMPSs maps tasks to workers by prioritizing free resources and data locality. This means that in case there are several free workers (not executing anything at the moment), the task is scheduled to the one that meets most of the input requirements. Input requirements can be in-memory objects or files already present in a specific worker if, for instance, some of the following cases are met:

- Two subsequent tasks executed in the same worker share the same input data, thus the second task might have them already present in the worker after the master covered the requirements of the first task.
- Two subsequent tasks are executed in the same worker, the first task producing the objects required by the second task.
- A task executed in a worker node requires some input data in the form of files that are already stored in such a worker node.

In order to validate the integration of *dataClay* with COMPSs, this section shows two applications where tackling impedance mismatch difficulties and hastening data transfers are crucial to significantly outperform their execution times. In particular, it is presented a weak scaling study on the execution of two well-known algorithms implemented in Java: Wordcount and K-means. Both applications are tailored to take the maximum advantage of COMPSs, and the obtained results are compared in terms of elapsed times to those produced when configuring COMPSs to use *dataClay* as the underlying data store. That is, when configured to use *dataClay*, COMPSs handle data as persistent objects and derives

the computation and communications to *dataClay* DS backends according to the execution environment exposed in chapter 4.

All the experiments for this COMPSs-*dataClay* evaluation were performed in *MareNostrum III* supercomputer [83] were COMPSs is installed and ready for its usage. Launched jobs were configured to request up to eight 16-CPU nodes with 32GB of DRAM each (2GB DRAM per CPU) interconnected with a FDR10 Infiniband network.

Furthermore, certain workloads are also examined in more detail in order to understand the internal behavior of both applications. To this end, some executions were repeated configuring COMPSs and *dataClay* to produce Paraver [84] formatted traces to be visualized (and shown throughout this section) using the Paraver tool, a performance analyzer included in the suite CEPBA-Tools currently maintained in the Barcelona Supercomputing Center. That is, COMPSs and *dataClay* are already instrumented to optionally produce Paraver traces.

5.2.1 Wordcount

Wordcount is an embarrassingly parallel algorithm that counts the appearances of all the different words within a set of files. The application parses the input files splitting their text lines into words and maintains a data structure to keep one counter per word. The application ends up producing a final output to present the final counters for each unique word.

The original application used for COMPSs is implemented as a *MapReduce* workflow. The *Map* stage comprises the parallelization of word counting in a per file basis (one task per file), and the *Reduce* stage aggregates all the partial results obtained from the *Map* tasks by following a binary tree strategy to combine them all. 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 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-defined 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.

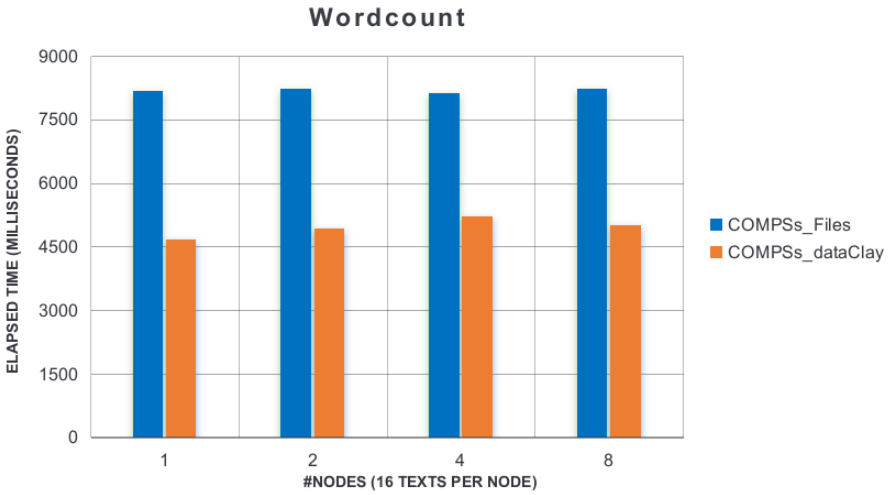

Figure 5.3: *Weak scaling study of the integration of COMPSs with dataClay with the Wordcount application. Y-axis represents the elapsed times in milliseconds. X-axis shows the number of nodes. Every node processes 16 texts of 256MB each.*

Figure 5.3 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 COMPSs *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 word-count) 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* COMPSs 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. In the regular version of COMPss (i.e. without *dataClay*), although files are cached and accessed in parallel (MareNostrum III uses GPFS [85]) every read operation incurs one I/O system call which makes the difference with *dataClay*.

This situation is further illustrated with figure 5.4, which presents two Paraver traces showing a Wordcount computation conducted by COMPSs for the case of 64 texts processed in parallel using 4 MareNostrum nodes. The first trace shows the behavior when the application is programmed on the basis of files, computing one file per COMPSs worker thread, i.e. one file per CPU. The second trace shows the same workload using *dataClay* as the underlying data store, thus processing *TextFile* objects instead of files,

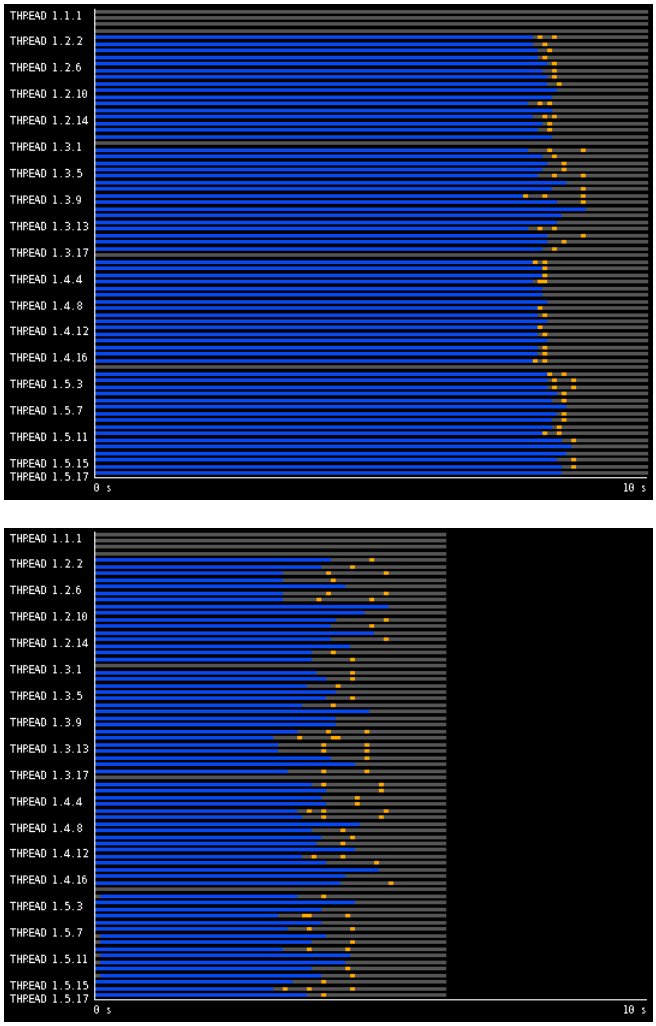

Figure 5.4: *Traces for the case of Wordcount execution on 64 files with 4 nodes. COMPSs using files first. COMPSs using dataClay second. Tasks of Map stage in blue. Tasks of Reduce stage in orange.*

one per worker thread, i.e. one object per CPU. The horizontal axis shows the total time of the trace, whereas the vertical axis shows some of the labels corresponding to threads. Blue boxes represent word-count tasks of the *Map* stage and its duration (one per text), whereas orange boxes show the tasks required to compute the *Reduce* stage.

The second trace is scaled to the same duration as the files-based trace (approximately 10 seconds), thus illustrating the time reduction using *dataClay* when comparing both traces, which is also revealed by the differences in the durations of blue word-count tasks.

5.2.2 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 [86] 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 the Wordcount study, K-means study focuses 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, the 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.

With COMPSs, the fragments are created directly in the workers 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 (if necessary).

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

Figure 5.5 illustrates the elapsed times (in seconds) of the K-means computation with

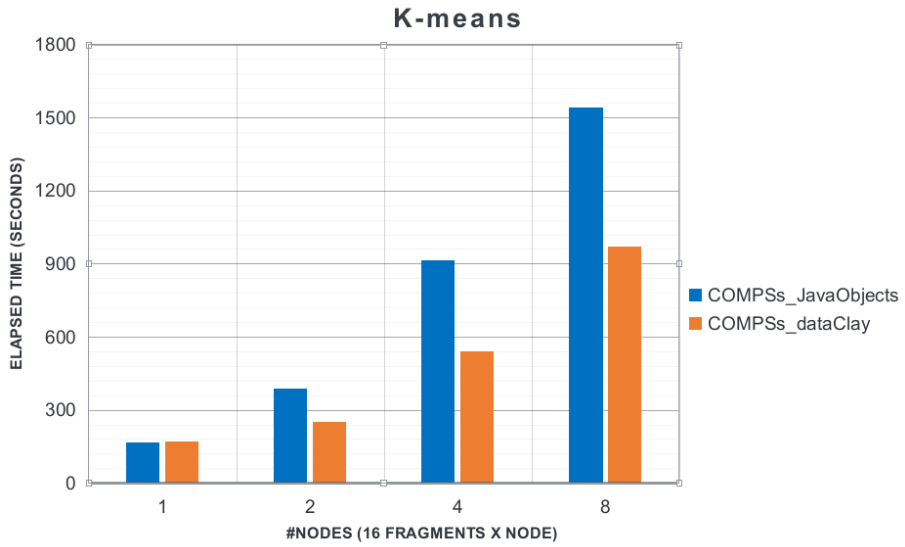

Figure 5.5: *Weak scaling study of the integration of COMPSs with dataClay with the K-means application. Y-axis represents the elapsed times in seconds. X-axis shows the number of nodes. 16 fragments of 12800 100-dimensional vectors are processed per node.*

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 the epsilon value was configured to be extremely low (0.0001), thus focusing 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, it is possible to outperform Java default serialization techniques used in the original COMPSs to transfer the partial results. The stub classes resulting from registered classes analyze 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* helps COMPSs to boost its performance with an improvement of up to a 41%.

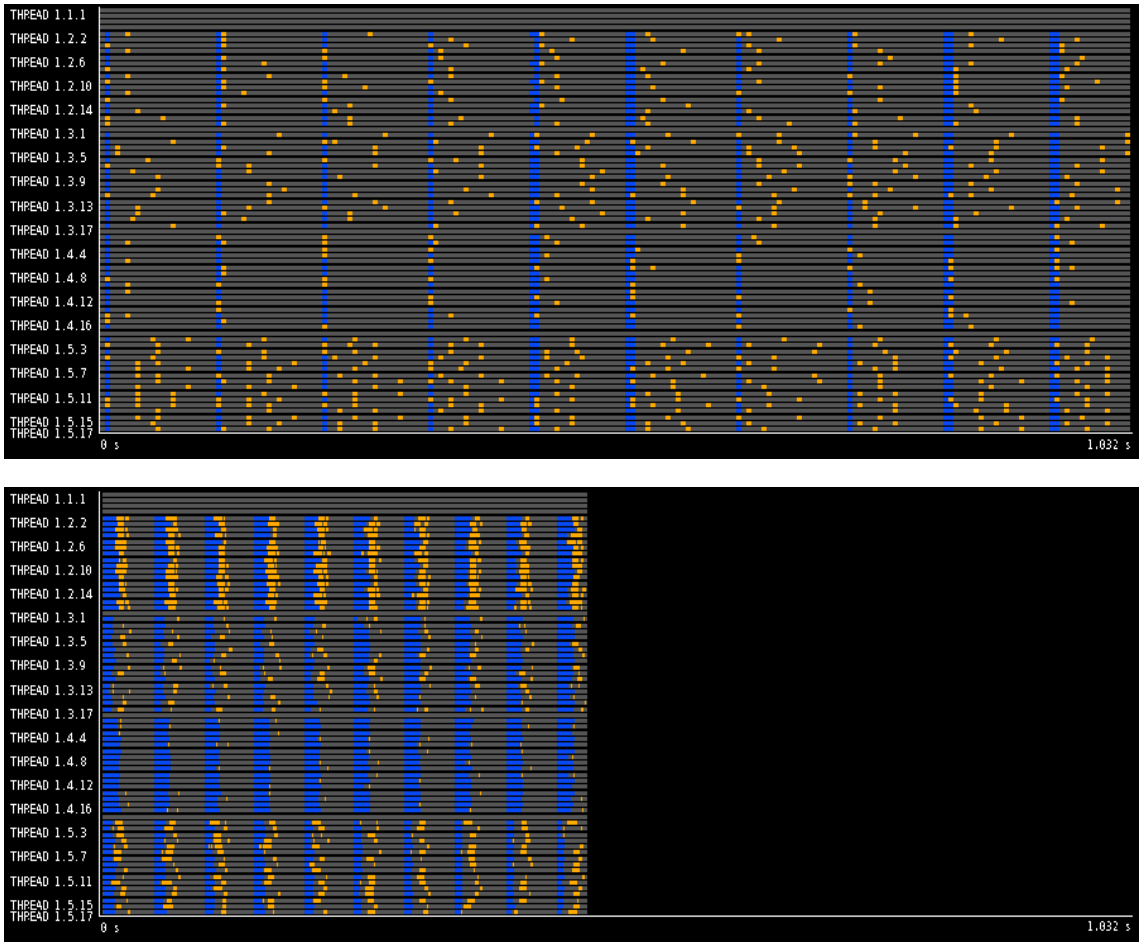

Figure 5.6: *Traces for the case of K-means execution on 64 fragments with 4 nodes. COMPSs not using dataClay first. COMPSs using dataClay second. Tasks of Map stage in blue. Tasks of Reduce stage in orange. X-axis shows the execution time in seconds. Y-axis shows thread ids.*

These results are further illustrated with Paraver traces. In the first place, figure 5.6 shows the 10 iterations for the K-means execution for the case of 64 fragments processed with 4 MareNostrum nodes. The first trace shows the behavior when fragments are distributed along the COMPSs workers as regular Java objects (i.e. not persistent). The second trace shows the behavior when fragments are persistent objects processed within *dataClay*. The horizontal axis shows the total time of the trace, whereas the vertical axis shows some of the labels corresponding to threads. Blue boxes represent clustering tasks of the *Map* stage and its duration (one per text), whereas orange boxes show the tasks required to compute the *Reduce* stage.

The second trace is scaled to the same duration as the first trace, thus revealing the reduction of time commented before. However, these traces show that tasks look longer when using *dataClay*. Indeed, using Paraver it can be determined that the average time to compute a blue clustering task of the *Map* stage within *dataClay* is about 116 seconds,

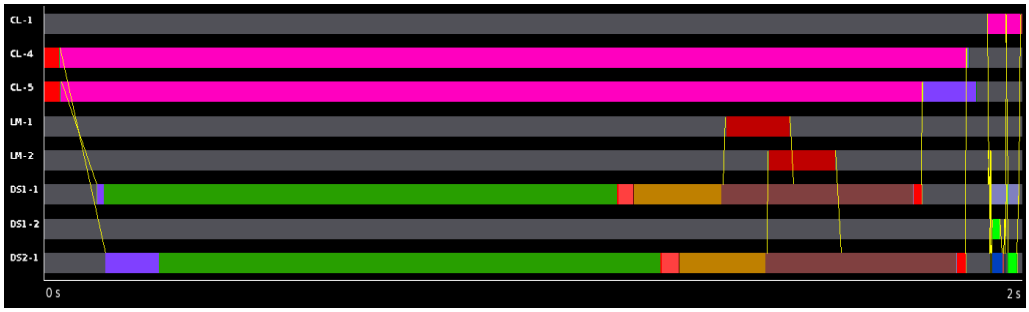

Figure 5.7: *2 fragments sample processed within dataClay. Clustering computation in green. Persistence of partial results afterwards (orange and brown) with the corresponding communications with the Logic Module that registers the objects (red). X-axis shows the execution time in seconds. Y-axis shows thread ids.*

whereas the same task executed from COMPSs workers using regular Java objects only requires 12 seconds. Analogously, orange reduce tasks require 26 seconds with *dataClay* and only 15 milliseconds without *dataClay*. Therefore, these differences demand to dig deeper in order to understand what is really happening.

The difference about an order of magnitude in clustering tasks can be explained by considering that *dataClay* performs session and permissions checks as stated in sections 4.2 and 4.2.4, but also because it stores the partial result for the subsequent *Reduce* stage. To show this behavior, a basic multi-threaded application was coded to illustrate the internal behavior of *dataClay* when executing one of the K-means iterations processing only 2 fragments located in two different DS backends. In particular, figure 5.7 shows a trace where vertical axis represent client (CL-*), Logic Module (LM-*) and Data Service threads of both DS backends (DS1,2-*). The most significant fact to be considered here is that, after the effective computation of the clustering task colored in green (including session and permission checks), the resulting objects are persisted in the underlying database of the corresponding DS backend and the metadata is registered to the Logic Module (orange and brown tasks of Data Service threads, and red tasks of the Logic Module threads).

On the other hand, and back to the main figure 5.6, the significant difference in the inner tasks of the *Reduce* stage (colored in orange) is because traces are shown at application/COMPSs level, thus the required communications in this stage are actually included in the total time of the tasks. Figure 5.8 illustrates this behavior showing the last iteration for the case of K-means execution on 64 fragments with 4 nodes. The first trace illustrates the communications (yellow lines) among the workers needed to perform the *Reduce* stage. The second trace is scaled with the same duration as the first trace to see the impact on the elapsed time, although communication lines cannot be shown since they

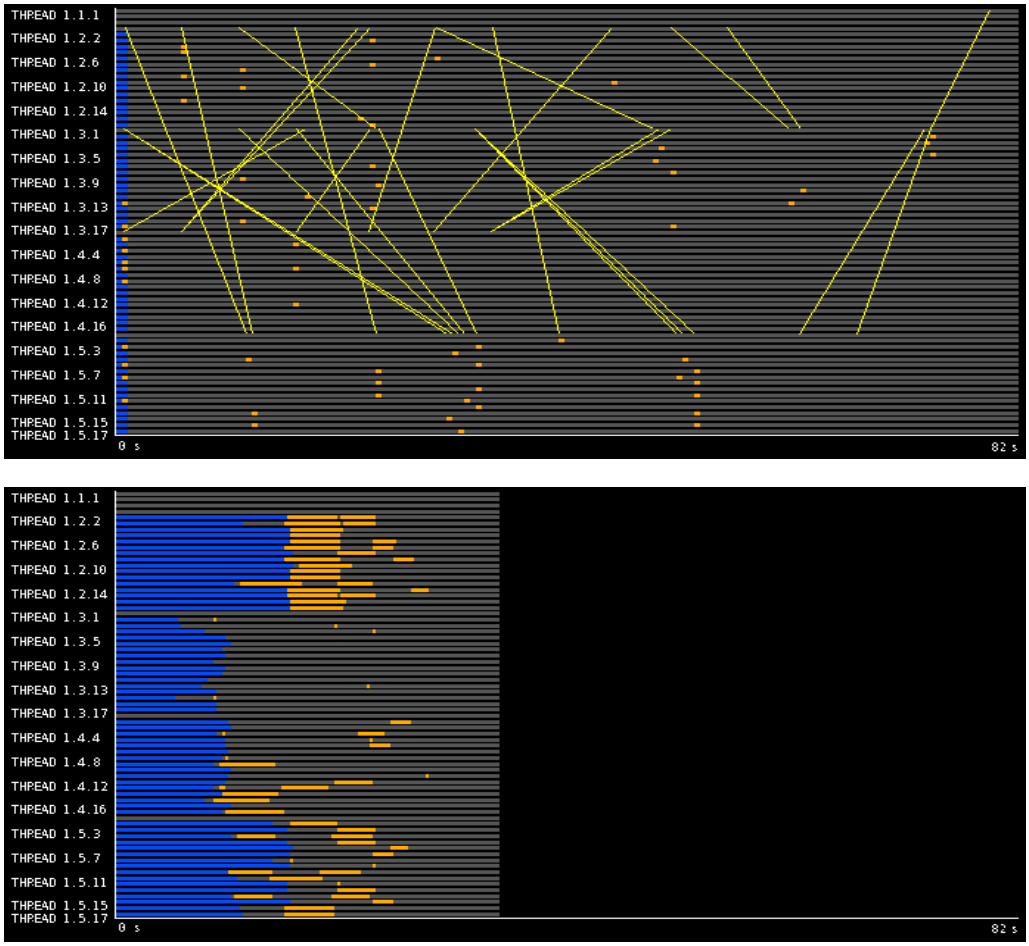

Figure 5.8: *Traces of the last iteration for the case of K-means execution on 64 fragments with 4 nodes. COMPSs not using dataClay first, including communication events. COMPSs using dataClay second (communications are included in reduce tasks). Tasks of Map stage in blue. Tasks of Reduce stage in orange. X-axis shows the execution time in seconds. Y-axis shows thread ids.*

are included in *Reduce* tasks computed within the *dataClay* environment. This explains the different durations of *Reduce* tasks when using *dataClay*, short *Reduce* tasks involve local partial results whereas long *Reduce* tasks involve partial results that are located in different nodes.

When COMPSs is not using *dataClay*, Paraver reveals that 98% of the elapsed time of K-means execution is dedicated to communications during the *Reduce* task as it is shown by the separation between subsequent iterations in main figure 5.6. On the contrary, using *dataClay* the serialization and communication mechanisms drastically reduced the inter-iteration gaps, compensating the costs of persisting partial results.

It is worth noting that the overhead due to Java serialization and reflection when not using *dataClay* could be mitigated if the data model used in the application implemented the *Externalizable* interface for all classes involved in data transferring. However, this

would compel the users to code their own serialization algorithms since *Externalizable* interface forces a class to implement *serialize* and *deserialize* methods in order to know how to deal with its object instances. With *dataClay* this is done transparently for the user thanks to having classes registered and thus avoiding reflection in both stub and execution classes.

5.3 Summary

This chapter presents different performance studies focused on the evaluation of *dataClay* and some of the contributions proposed in this thesis. Given that *dataClay* is a novel data store used to validate and evaluate the feasibility of the contributions proposed in this thesis, the first study evaluates *dataClay* in terms of throughput and latency comparing it with trendy NoSQL databases: MongoDB and Cassandra. To this end, a well-known benchmark called Yahoo Cloud Serving Benchmark (YCSB) is used to conduct concurrent execution requests performing basic persistence operations in different scenarios. On the one hand, in read-only workloads *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.

Once *dataClay* has been proven competitive, a second study with YCSB shows that *enrichments* (in relation to contribution **C1**) have no impact on the performance results. Actually, this was the expected behavior since *enrichments* are actually deployed as part of the code of any registered class (i.e. with no special treatment) as explained in previous chapters.

Finally, this chapter focuses on the contribution **C3** exposing the integration of the particular parallel programming model offered by COMPSs with the object-oriented data model of *dataClay*. Specifically, *dataClay* object-oriented data model is proven successful to tackle impedance mismatch difficulties, while improving data serialization for inter-node communications thanks to having the schema registered in the system. On the one hand, using *dataClay* instead of files, it is proven that COMPSs can execute a Wordcount application with up to 43% time reduction; and on the other hand, thanks to *dataClay* serialization techniques implicit in the generation of stub and execution classes, it is shown how COMPSs can be boosted up to a 41% in the K-means computation.

Chapter 6

Conclusions

This thesis addresses the gaps of current data services and database management systems in the area of data sharing for multi-provider scenarios. In particular, three novel contributions are proposed for the upcoming data stores aiming to effectively support inter-player data sharing as well as the efficient processing of data-intensive workflows. First contribution **C1** proposes novel techniques to maximize the flexibility to share and extend the accessible schema in such a multi-provider context. Second contribution **C2** defines a mechanism to ensure data integrity constraints within the framework of a decentralized administration. And last contribution **C3** exposes a novel integration of the application and data models in response to identified drawbacks and opportunities related to the execution of data-intensive applications.

In order to validate the proposed contributions, this thesis presents a distributed object-oriented data store, called *dataClay*, as a proof of concept to show how to fulfill all the requirements established in section 1.3. *dataClay* is based on the early ideas of logical data independence defined in the ANSI/SPARC architecture, but proposes the decentralization of the administration with the so-called **parceled control**. Parceled control is ensured by making data to be encapsulated and accessible only through methods with explicit granted access. Then, in conjunction with an orthogonal dataset control through data contracts and a session-based mechanism for the workflow execution, *dataClay* enables data providers to fully administrate their data, ensuring data integrity and deciding how they share it and with who.

In this context, contribution **C1** regarding the flexibility to share and extend data structures and functionality, is fulfilled by enabling the users to register and share their own

schemas and by allowing their evolution through so-called **enrichments**. Enrichments comprise the extension of 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 schemas according to the model contracts defined by the provider. To further facilitate this flexibility and data integration, data model impersonation enables the developers to design their class methods based on the scope of their accessible schemas and without concerning about any underlying functionality required. Although this makes that applications might implicitly use functionality that is not accessible for them, data still remains protected through schema-defined data integrity and the independent control of dataset accessibility reflected in the transparent propagation of session information.

In that connection, however, data integrity has proven to require special attention in a multi-provider context, where every consumer might have a different vision of the stored objects, and thus, a different vision of their relationships. In this sense, referential constraints, derived attributes, or in general any method coded on the basis of accessing other objects, requires that the system provides the developers with the proper mechanisms to define a correct behavior considering access permissions of current consumer. With *dataClay*, this is proven feasible by means of the encapsulation through methods, the session propagation and the provided system operations to check data accessibility. In this way, contribution **C2** is validated.

Finally, the contribution **C3** is validated through chapters 4 and 5.

To begin with, chapter 4 exposes the technical details of the distributed execution environment of *dataClay* including (among other features): a) how *dataClay* exploits data locality by taking account of current objects locations, b) how *dataClay* overcomes impedance mismatch issues with an Object Oriented data model for data representation, c) the support of interoperability between two of the most used high-level languages (Java and Python). In this way, *dataClay* meets the requirements related to the integration of application and data models.

Thereafter, chapter 5, starts validating *dataClay* in terms of throughput and latency in different scenarios and by comparing it 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. With the same benchmark, it is also shown that enrichments (in relation to contribution **C1**) have no effect on the performance results, since they are actually deployed as part of the code of any registered class (i.e. with no special treatment).

Once *dataClay* has been proven competitive, the contribution **C3** is definitely validated through the performance evaluation of the integration of *dataClay* with a parallel programming model (offered by COMP Superscalar). Specifically, using an OO data model is proven successful to tackle impedance mismatch difficulties, while improving data serialization for inter-node communications thanks to having the schema registered in the system. In particular, using *dataClay* instead of files, it is proven that COMPSs can execute a Wordcount application with up to 43% time reduction; and on the other hand, thanks to *dataClay* serialization techniques implicit in the generation of stub and execution classes, it is shown how COMPSs can be boosted up to a 41% in the K-means computation.

In view of the above, this thesis proves that all the hypothesis formulated in section 1.5 have been satisfied with the proposed contributions and through the design and implementation of *dataClay*. Therefore, future data stores for multi-provider ecosystems should consider these proposals in order to effectively support data sharing, data integration and efficient data processing, and thus fostering the collaboration among all the players.

Chapter 7

Future Work

This chapter presents a set of issues that remained out of the scope of this thesis but that are of vital importance for large-scale data sharing in a multi-provider context. Some of them are already *work-in-progress* by the *dataClay* team, nourishing research lines for the advent of new theses projects.

7.1 Metadata repository

Large-scale data sharing requires the decentralization of metadata repositories in order to avoid single points of failure and prevent potential bottlenecks. In *dataClay*, for instance, the Logic Module keeps all metadata about objects, sessions, and management operations related to parceled control. Therefore, although Data Service nodes maintain local metadata caches to mitigate this problem, a central repository still is an obvious bottleneck especially when dealing with workflows that create or delete objects constantly.

Different solutions have been proposed to tackle this issue, but still no implementation has been actually programmed. For instance, a distributed metadata service could be deployed among all the Data Service nodes. Every Data Service node could be in charge of the metadata of the objects stored in its database. Metadata could be replicated in several nodes and/or an eventually consistent replica of the metadata could be kept in the LM and updated periodically in an asynchronous fashion.

Object metadata could also be assigned to Data Service nodes by applying a hash function on the object identifiers (OIDs), thus producing a distributed hash table for metadata that could be inspired in some of the current implementations available in the literature [87] [88].

7.2 Global garbage collector

Working with large volumes of persistent data requires a non-trivial management of storage resources. In this regard, and analogously to language Garbage Collectors that help in the memory management of execution environments, an OO data store should implement some mechanism to detect and remove persistent objects that might be orphaned or isolated. The problem is that this task might become especially complex in multi-session distributed environments [89] [90] [91] where objects are interrelated at many levels and coupled to different data integrity constraints.

For example, in *dataClay* an object instantiating a collection (of other objects) could be made persistent and stored in a particular Data Service node possessing some elements. A particular alias could be assigned to this collection making it retrievable whenever is needed, but the contained objects might not have tagged with an alias but persisted with OIDs and referenced from the collection. Afterwards, at any time, this collection object could be deleted, but what happens with the contained elements?

If they are referenced from any other accessible object, they should not be deleted, but on the contrary a global *dataClay* Garbage Collector should be able to detect isolated objects (i.e. that cannot be accessible in any way) to garbage collect them and eventually remove them.

However, determining whether an object is being accessible at any time is not trivial. It is not only a matter of knowing the relationships between objects and whether they can be directly accessed (e.g. via aliases) or not, but also which of them are referenced from any execution thread among the runtime environment, or even worse, referenced from the environment of user-level applications. Therefore, information such as current opened sessions should be also considered in order to determine whether an object is being accessible or not, taking account of the accessible datasets, etc.

7.3 Replica management

Replica management is out of the scope of this thesis, but it is a common feature [92] [93] [94] for any distributed storage where exploiting replication can yield several benefits: enhancing parallelization with simultaneous access to different replicas of the same object (horizontal read scaling), fault tolerance in case of net splits or node failures with automatic forwarding to accessible replicas, etc.

In a multi-provider environment replica management might be very dependent of the requirements of each data model. This suggests that *dataClay* should not provide a specific replica management implementation but the tools or facilities necessary for the developers to easily implement replica management coupled to their data models.

For instance, depending on the consistency level, *dataClay* could support some typical features like: transactions for atomic updates, implementations of Paxos or Raft protocols for master-slave replication, a Domain Specific Language (DSL) to define a minimum number of replicas per object of a class or how they should be distributed, etc.

Nevertheless, replication could also be done automatically considering the current status of the system and/or the data patterns (e.g. learned from the kind of prefetching service stated before). In this sense, it could also be possible to apply clustering techniques to define groups of objects that could be replicated together depending on such patterns.

7.4 Iterators

In OO programming, the iterator pattern [95] is a well-known design pattern in which a special object, the iterator, is used to traverse a container getting access to its elements in a particular order and without exposing its underlying representation.

A smart usage of iterators in a distributed environment should enhance parallelism opportunities on certain executions. For instance, an application could iterate through the elements of a collection considering their actual location among the storage nodes. To this end, it is necessary that the data store provides developers with the required mechanisms to implement collections and iterator patterns enabled to work with the distributed data.

Furthermore, these mechanisms should be also compliant with the language native iterators, i.e. "iterator-like" objects in Python and Java classes implementing *Iterator* interface.

7.5 ECA subsystem

Event-driven architectures are of interest for the integration of execution environments with data stores, allowing active behaviors for non-interactive (or autonomous) functionality.

However, dealing with event-driven subsystems for large-scale data sharing requires special attention. For instance, current implementation of *dataClay* only supports two

kind of events at object persistence level: *NewInstance* and *RemovedInstance*. However, what if *UpdatedInstance* should be also supported?

For instance, following the examples in section 4.3, *RedCars* class could track not only the creation and removal of cars, but also the potential changes of their color. This means that not only the common create or delete methods (like *makePersistent* or *deletePersistent*) produce events, but also any *setter* method modifying the current object state. Consequently, scalable solutions should concern, for instance, about when an object is considered to be actually updated, e.g. when modifications are propagated to persistent storage.

Furthermore, scalable solutions should also consider distributing, not only the execution of *condition checkers* and *actions* as *dataClay* does, but also the *event* handling (which in *dataClay* is centralized through the Logic Module). Probably event management should be delegated to the backend nodes, for instance, implementing a multicast protocol to communicate an event produced in one DS backend to the rest of DS backends on the basis of P2P overlays [96].

7.6 Prefetching and placement

Data prefetching techniques are commonly applied at different levels of the I/O stack [97] [98] [99] in order to load certain data in advance (i.e. before it is actually requested) by learning data access patterns in conjunction with certain heuristics that help making the corresponding decisions.

In this regard, the performance of data-intensive workflows could be drastically improved by applying data prefetching techniques from the runtime environment. For instance, in *dataClay*, pre-loading objects from the underlying storage could take advantage of I/O bandwidths while preventing time waits due to I/O latencies when objects are retrieved on demand.

To this end, some mechanisms can be implemented considering the proposed registration of providers' schemas in the system.

On the one hand, although it is impossible to foresee the arbitrary code of user-level applications, knowing the underlying data schema might be useful to infer data patterns based either on the relationships between the classes or on the code of the registered methods.

On the other hand, an OO data store might also learn object-access patterns

dynamically from any method execution processed along the service backends. For instance, backends could record the order in which objects are accessed to create a global access pattern base of knowledge. In this way, the system could also provide the mechanisms to group and reorganize objects among backends to further exploit data locality.

Chapter 8

Result dissemination

This chapter presents the publications and collaborations related to this thesis, as well as the ongoing efforts on creating a start-up to exploit *dataClay* technology.

8.1 Publications

This thesis has three preceding publications and a recent paper under minor review:

J. Martí, A. Queralt, D. Gasull, and T. Cortes. "Living objects: towards flexible big data sharing." *Journal of Computer Science & Technology*. vol. 13. no. 2. pp. 56–63. 2013

J. Martí, D. Gasull, A. Queralt, and T. Cortes. "Towards DaaS 2.0: Enriching Data Models." in *Proceedings - 2013 IEEE 9th World Congress on Services, SERVICES 2013*. pp. 349–355. IEEE. IEEE. jun 2013

A. Queralt, J. Martí, H. Baars, A. Brinkmann, and T. Cortes. "Fusing storage and computing for the domain of business intelligence and analytics - Research opportunities." in *Proceedings of the Annual Hawaii International Conference on System Sciences*. vol. 2015-March. pp. 4752–4761. 2015

J. Martí, A. Queralt, D. Gasull, A. Barcelo, J. J. Costa, and T. Cortes. "dataclay: a distributed data store for effective inter-player data sharing." *Under Minor Review in Journal of Systems and Software*. 2016

The first one presented the concept of object encapsulation through so-called *self-contained objects*, i.e. functionality and security policies defined at method-level, which is the pillar for all the presented contributions. The encapsulation is required for

effective data sharing and schema evolution in multi-provider context (contribution **C1**); it is also required to enable developers to define data integrity constraints on their schemas according to contribution **C2**, and the OO data model is the basis of the integration proposed in contribution **C3**.

The second one exposed a first version of *dataClay* focusing on the actual implementation of *self-contained objects* and the necessary mechanisms to show how to actually share data models and to enable 3rd parties to enrich them. Therefore it was mainly focused on **the contribution C1 of this thesis**.

The third one proposes a revitalization of ideas from object-oriented databases applying to Business Intelligence and Analytics (BIA) solutions by using the concept of *self-contained objects*.

Last paper (under minor review) exposes *dataClay* on its most recent version with all the major concepts presented in this thesis.

8.2 Technology transfer

Nowadays the advisors and the author of this thesis are also working together on the transfer of *dataClay* technology from Barcelona Supercomputing Center to a start-up company that, in case of success, will try to exploit the *dataClay* platform in the business world. In order to facilitate this challenge, they started a process to patent the concepts **related to the contributions C1 and C2 of this thesis**.

The information of the PCT application is:

Title	Accessing data stored in a database system
Inventors	Antonio Cortés, Anna Queralt, Jonathan Martí, Daniel Gasull
Applicants	Barcelona Supercomputing Center ,
	Universitat Politècnica de Catalunya
PCT app number	PCT/EP2016/075420
Date of receipt	21 October 2016
Receiving Office	European Patent Office

8.3 Collaborations

Severo Ochoa. The Spanish Ministry of Economy and Competitiveness twice awarded Barcelona Supercomputing Center (BSC) the Severo Ochoa Centers of Excellence

accreditation in 2011 (grant SEV-2011-00067) and 2015 (grant SEV-2015-0493). Within this framework, BSC started a project that encourages all of its groups and departments to cooperate in interdisciplinaryresearch lines. In this project, the Storage System Research Group (in charge of *dataClay*) in collaboration with Grid Computing and Clusters Group (in charge of COMPSs) worked together towards the integration of COMPSs with *dataClay*, which is directly connected to the **contribution C3 of this thesis**.

Big Storage is an European Training Network (ETN) whose main goal is to train future data scientists in applying holistic and interdisciplinary approaches for taking advantage of a data-overwhelmed world. This requires HPC and Cloud infrastructures with a redefinition of storage architectures underpinning them, focusing on meeting highly ambitious performance and energy usage objectives. In this project, *dataClay* will be focused on the definition of new data prefetching strategies thanks to having both code and data stored together and pushing the exploitation of data locality to the limit.

NextGenIO is an European project that aims to define a new I/O stack for next generation Storage systems. These new standards will take account of new NVRAM technologies to exploit the access to non-volatile memories, faster than traditional storage units like hard disks. In this context, *dataClay* will be positioned to exploit this persistent layer closer to the application thus achieving better performance outcomes, which is connected with the contribution **C3**.

mF2C is an European project starting on January 2017. This project tries to fill the gaps between the use of Cloud and Fog Computing. In this project, contributions from the research on *dataClay* will be the cornerstone to manage the data between different devices in IoT environments such as smart cities.

Bibliography

[1] T. O'Reilly. "What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software." *Design*. vol. 65. pp. 17–37. 2007.

[2] Y. Demchenko, C. De Laat, and P. Membrey. "Defining architecture components of the Big Data Ecosystem." in *2014 International Conference on Collaboration Technologies and Systems, CTS 2014*. pp. 104–112. IEEE. may 2014.

[3] R. Battle and E. Benson. "Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)." *Web Semantics*. vol. 6. no. 1. pp. 61–69. 2008.

[4] C. Ireland, D. Bowers, M. Newton, and K. Waugh. "A classification of object-relational impedance mismatch." in *Proceedings - 2009 1st International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDA 2009*. pp. 36–43. IEEE. 2009.

[5] M. Artur. "Nvram as main storage of parallel file system.." *Journal of Computer Science & Control Systems*. vol. 9. no. 1. 2016.

[6] J. I. Han, Y. M. Kim, and J. Lee. "Achieving energy-efficiency with a next generation NVRAM-based SSD." in *International Conference on ICT Convergence*. pp. 563–568. 2014.

[7] C. Docan, M. Parashar, J. Cummings, and S. Klasky. "Moving the Code to the Data - Dynamic Code Deployment Using ActiveSpaces." in *2011 IEEE International Parallel & Distributed Processing Symposium*. pp. 758–769. IEEE. may 2011.

[8] C. Docan, M. Parashar, and S. Klasky. "DataSpaces: an interaction and coordination framework for coupled simulation workflows." *Cluster Computing*. vol. 15. pp. 163–181. jun 2012.

[9] M. J. Carey, N. Onose, and M. Petropoulos. "Data services." *Communications of the ACM*. vol. 55. pp. 86–97. jun 2012.

[10] C. J. Date. *An introduction to database systems*. Addison-Wesley Pub. Co. 3rd ed.. 1981.

[11] D. Maier, J. Stein, A. Otis, and A. Purdy. "Development of an object-oriented DBMS." *ACM SIGPLAN Notices*. vol. 21. no. 11. pp. 472–482. 1986.

[12] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. "The ObjectStore database system." *Communications of the ACM*. vol. 34. pp. 50–63. oct 1991.

[13] T. Inoue, A. Aikebaier, T. Enokido, and M. Takizawa. "Power consumption and processing models of servers in computation and storage based applications." *Mathematical and Computer Modelling*. vol. 58. no. 5-6. pp. 1475–1488. 2013.

[14] B. Welch and G. Noer. "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions." in *IEEE Symposium on Mass Storage Systems and Technologies*. pp. 1–12. IEEE. may 2013.

[15] T. Nuns, S. Duzellier, J. Bertrand, G. Hubert, V. Pouget, F. Darracq, J. P. David, and S. Soonckindt. "Evaluation of recent technologies of nonvolatile RAM." in *IEEE Transactions on Nuclear Science*. vol. 55. pp. 1982–1991. 2008.

[16] M. Zhou, X. Chen, Y. Liu, S. Li, G. Li, X. Li, and Z. Song. "Design and implementation of a random access file system for NVRAM." 2016.

[17] M. Romanus, S. Klasky, C.-S. Chang, I. Rodero, F. Zhang, T. Jin, Q. Sun, H. Bui, M. Parashar, J. Choi, S. Janhunen, and R. Hager. "Persistent Data Staging Services for Data Intensive In-situ Scientific Workflows." in *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing - DIDC '16*. (New York, New York, USA). pp. 37–44. ACM Press. 2016.

[18] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi. "Enabling in-situ execution of coupled scientific workflow on multi-core platform." in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012*. pp. 1352–1363. 2012.

[19] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. "An approach to persistent programming." *The Computer Journal*. vol. 26. pp. 360–365. 1983.

[20] A. M. Keller, R. Jensen, and S. Agarwal. "Persistence software: bridging object-oriented programming and relational databases." *ACM SIGMOD Record*. vol. 22. no. 2. pp. 523–528. 1993.

[21] P. Carbonnelle. "Pypl popularity of programming language." http://pypl.github.io/PYPL.html.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with ycsb." *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. pp. 143–154. 2010.

[23] C. W. Bachman. "Integrated data store." *DPMA Quarterly*. vol. 1. no. 2. pp. 10–30. 1965.

[24] E. F. Codd. "A relational model of data for large shared data banks." *Communications of the ACM*. vol. 26. no. 1. pp. 64–69. 1970.

[25] Oracle. "Oracle database." http://oracle.com/database. (accessed: 2015-12-3).

[26] IBM. "Ibm informix." http://ibm.com/software/data/informix/. (accessed: 2016-1-3).

[27] Actian. "Ingres." http://actian.com/products/operational-databases/ingres/. (accessed: 2016-3-8).

[28] IBM. "Db2." http://ibm.com/software/data/db2/. (accessed: 2016-1-3).

[29] M. Stonebraker, D. Moore, and P. Brown. *Object-Relational DBMSs: Tracking the Next Great Wave*. Morgan Kaufmann Publishers. 1998.

[30] H. Garcia-Molina and K. Salem. "Main Memory Database Systems: An Overview." *IEEE Transactions on Knowledge and Data Engineering*. vol. 4. no. 6. pp. 509–516. 1992.

[31] Jing Han, Haihong E, Guan Le, and Jian Du. "Survey on NoSQL database." in *2011 6th International Conference on Pervasive Computing and Applications*. pp. 363–366. IEEE. oct 2011.

[32] R. Cattell. "Scalable SQL and NoSQL data stores." *ACM SIGMOD Record*. vol. 39. p. 12. may 2011.

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

[34] A. S. Foundation. "Apache cassandra project." http://cassandra.apache.org. (accessed: 2015-10-2).

[35] M. Inc.. "Mongodb." http://mongodb.org. (accessed: 2015-11-7).

[36] C. Inc.. "Couchbase." http://www.couchbase.com. (accessed: 2016-2-17).

[37] K. Guo, W. Pan, M. Lu, X. Zhou, and J. Ma. "An effective and economical architecture for semantic-based heterogeneous multimedia big data retrieval." *Journal of Systems and Software*. vol. 102. pp. 207–216. 2015.

[38] G. Touya. "A Road Network Selection Process Based on Data Enrichment and Structure Detection." *Transactions in GIS*. vol. 14. pp. 595–614. oct 2010.

[39] R. C. F. Wong and C. H. C. Leung. "Automatic semantic annotation of real-world web images." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. vol. 30. no. 11. pp. 1933–1944. 2008.

[40] InterSystems. "Caché - high performance object database." http://intersystems.com/our-products/cache/cache-overview/. (accessed 2016-10-23).

[41] V. Corporation. "db4o (database for objects)." http://db4o.com. (accessed 2015-09-30).

[42] A. S. Foundation. "Hbase." http://hbase.apache.org. (accessed: 2016-1-5).

[43] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schröter, and G. Saake. "Javadaptor - flexible runtime updates of java applications." *Software: Practice and Experience*. vol. 43. no. 2. pp. 153–185. 2013.

[44] L. Pina, L. Veiga, and 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*. pp. 103–119. 2014.

[45] C. J. Date and H. Darwen. *A Guide To Sql Standard*. vol. 3. Addison-Wesley Reading. 1997.

[46] D. McCarthy and U. Dayal. "The architecture of an active database management system." in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data - SIGMOD '89*. vol. 18. (New York, New York, USA). pp. 215–224. ACM Press. 1989.

[47] B. M. Michelson. "Event-driven architecture overview." *Patricia Seybold Group*. vol. 2. 2006.

[48] D. Guinard and V. Trifa. "Towards the Web of Things : Web Mashups for Embedded Devices." *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009)*. pp. 1–8. 2009.

[49] E. Viara, E. Barillot, and G. Vaysseix. "The EYEDB OODBMS." in *Proceedings. IDEAS'99. International Database Engineering and Applications Symposium (Cat. No.PR00265)*. pp. 390–402. IEEE Comput. Soc. 1999.

[50] J. Juday. "Using ddl triggers to manage sql server 2005." *Database Journal,[Online]*. pp. 1–4. 2005.

[51] "mongo-triggers." https://github.com/iddogino/mongoTriggers. 2014. (accessed 2016-10-23).

[52] A. web services. "Dynamodb." http://aws.amazon.com/dynamodb/. (accessed: 2016-5-18).

[53] A. Gupta and I. S. Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications." *IEEE Data Engineering Bulletin*. vol. 18. pp. 3–18. 1995.

[54] M. A. Nascimento and G. Moerkotte. *Proceedings of the Thirtieth International Conference on Very Large Data Bases : Toronto, Canada, August 31-September 3, 2004*. Morgan Kaufmann Publishers. 2004.

[55] T. Rabl and H.-A. Jacobsen. “Materialized views in Cassandra.” 2014.

[56] B. Holt. *MapReduce Views in CouchDB*. USA: O’Reilly Media. 2011.

[57] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. “FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems.” in *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*. pp. 61–70. IEEE. oct 2015.

[58] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu. “Optimizing load balancing and data-locality with data-aware scheduling.” in *2014 IEEE International Conference on Big Data (Big Data)*. pp. 119–128. IEEE. oct 2014.

[59] S. Feuerstein and B. Pribyl. *Oracle PL/SQL programming*. O’Reilly & Associates. 1997.

[60] K. Henderson and Ken. *The guru’s guide to Transact-SQL*. Addison-Wesley. 2000.

[61] B. Momjian and Bruce. *PostgreSQL : introduction and concepts*. Addison-Wesley. 2001.

[62] J. Melton and Jim. *Understanding SQL’s stored procedures : a complete guide to SQL/PSM*. Morgan Kaufmann Publishers. 1998.

[63] N. Technology. “Neo4j.” http://neo4j.com. (accessed: 2016-1-14).

[64] R. Hat. “Hibernate.” http://hibernate.org. (accessed: 2016-1-14).

[65] DataNucleus. “Datanucleus.” http://datanucleus.org. (accessed: 2016-2-5).

[66] S. Sanfilippo. “Redis.” http://redis.io. (accessed 2016-10-23).

[67] X. Wu and A. L. N. Reddy. “SCMFS: A File System for Storage Class Memory.” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*. pp. 39:1 – 39:11. 2011.

[68] H. E. Bal, J. Maassen, R. V. Van Nieuwpoort, N. Drost, R. Kemp, T. Van Kessel, N. Palmer, G. Wrzesiska, T. Kielmann, K. Van Reeuwijk, F. J. Seinstra, C. J. H. Jacobs, and K. Verstoep. “Real-world distributed computer with Ibis.” *Computer*. vol. 43. no. 8. pp. 54–62. 2010.

[69] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. “Swift: A language for distributed parallel scripting.” *Parallel Computing*. vol. 37. no. 9. pp. 633–652. 2011.

[70] A. S. Foundation. “Apache hadoop.” http://hadoop.apache.org. (accessed: 2016-1-11).

[71] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets." in *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in Cloud Computing*. p. 10. 2010.

[72] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud.." *Nucleic acids research*. vol. 41. no. Web Server issue. 2013.

[73] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira Da Silva, M. Livny, and K. Wenger. "Pegasus, a workflow management system for science automation." *Future Generation Computer Systems*. vol. 46. pp. 17–35. 2015.

[74] E. Tejedor and 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*. pp. 185–193. IEEE. may 2008.

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

[76] C. Bazar and 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*. vol. V. no. 2. pp. 49–59. 2014.

[77] K. Bakshi. "Considerations for big data: Architecture and approach." in *IEEE Aerospace Conference Proceedings*. pp. 1–7. IEEE. mar 2012.

[78] J. Pokorny. "NoSQL databases: a step to database scalability in Web environment." *Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13*. vol. 9. no. September. pp. 14–22. 2013.

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

[80] T. Rabl and S. Gómez-Villamor. "Solving big data challenges for enterprise application performance management." *Proceedings of the VLDB Endowment*. vol. 5. pp. 1724–1735. 2012.

[81] Y.-S. Kang, I.-H. Park, J. Rhee, and Y.-H. Lee. "MongoDB-Based Repository Design for IoT-Generated RFID/Sensor Big Data." *IEEE Sensors Journal*. vol. 16. pp. 485–497. jan 2016.

[82] L. a. Adamic and B. a. Huberman. "Zipf's law and the internet." *Glottometrics*. vol. 3. pp. 143–150. 2002.

[83] B. S. Center. "Mare nostrum III." https://es.wikipedia.org/wiki/MareNostrum. (accessed 2016-10-23).

[84] V. Pillet, J. Labarta, T. Cortes, and S. Girona. "PARAVER: A Tool to Visualize and Analyze Parallel Code." *Proceedings of WoTUG-18: Transputer and occam Developments*. pp. 17–31. 1995.

[85] F. Schmuck and R. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." *Proceedings of the First USENIX Conference on File and Storage Technologies*. no. January. pp. 231–244. 2002.

[86] W. Zhao, H. Ma, and 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. pp. 674–679. Springer Berlin Heidelberg. 2009.

[87] M. F. Kaashoek and D. R. Karger. "Koorde: A Simple Degree-Optimal Distributed Hash Table." pp. 98–107. Springer Berlin Heidelberg. 2003.

[88] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table." in *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*. pp. 775–787. IEEE. may 2013.

[89] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. "A parallel, incremental, mostly concurrent garbage collector for servers." *ACM Transactions on Programming Languages and Systems*. vol. 27. pp. 1097–1146. nov 2005.

[90] L. Veiga and P. Ferreira. "Asynchronous complete distributed garbage collection." in *Proceedings - 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2005*. vol. 2005. pp. 24a–24a. IEEE. 2005.

[91] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. "STOPLESS : A Real-Time Garbage Collector for Multiprocessors." *System*. pp. 159–172. 2007.

[92] S. Goel and R. Buyya. "Data replication strategies in wide area distributed systems." *Enterprise Service Computing: From Concept to Deployment*. pp. 211–241. 2006.

[93] K. Ranganathan and I. Foster. "Identifying dynamic replication strategies for a high-performance data grid." *In Proc. of the International Grid Computing Workshop*. vol. 2242. pp. 75–86. 2001.

[94] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. "Data replication strategies in grid environments." in *Proceedings - 5th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP 2002*. pp. 378–383. 2002.

[95] J. Gibbons and B. C. D. S. Oliveira. "The essence of the Iterator pattern." *Journal of Functional Programming*. vol. 19. no. 3-4. p. 377. 2009.

[96] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. "A survey and comparison of peer-to-peer overlay network schemes." 2005.

[97] J. Griffioen and R. Appleton. "Reducing File System Latency using a Predictive Approach." *Proceedings of the Summer 1994 USENIX Technical Conference*. no. June. pp. 197–207. 1994.

[98] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch." *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. no. 20. p. 14. 2007.

[99] S. Vander Wiel and D. Lilja. "When caches aren't enough: data prefetching techniques." *Computer*. vol. 30. pp. 23–30. jul 1997.

[100] J. Martí, A. Queralt, D. Gasull, and T. Cortes. "Living objects: towards flexible big data sharing." *Journal of Computer Science & Technology*. vol. 13. no. 2. pp. 56–63. 2013.

[101] J. Martí, D. Gasull, A. Queralt, and T. Cortes. "Towards DaaS 2.0: Enriching Data Models." in *Proceedings - 2013 IEEE 9th World Congress on Services, SERVICES 2013*. pp. 349–355. IEEE. IEEE. jun 2013.

[102] A. Queralt, J. Martí, H. Baars, A. Brinkmann, and T. Cortes. "Fusing storage and computing for the domain of business intelligence and analytics - Research opportunities." in *Proceedings of the Annual Hawaii International Conference on System Sciences*. vol. 2015-March. pp. 4752–4761. 2015.

[103] J. Martí, A. Queralt, D. Gasull, A. Barcelo, J. J. Costa, and T. Cortes. "dataclay: a distributed data store for effective inter-player data sharing." *Under Minor Review in Journal of Systems and Software*. 2016.