PolarDBMS: Towards a Cost-Effective and Policy-Based Data Management in the Cloud

Ilir Fetai, Filip-M. Brinkmann and Heiko Schuldt

Department of Informatics and Mathematics
University of Basel Switzerland
{ilir.fetai|filip.brinkmann|heiko.schuldt}@unibas.ch

***Abstract*—The proliferation of Cloud computing has attracted a large variety of applications which are completely deployed on resources of Cloud providers. As data management is an essential part of these applications, Cloud providers have to deal with many different requirements for data management, depending on the characteristics and guarantees these applications are supposed to have. The objective of a Cloud provider is to support these diverse requirements with a basic set of customizable modules and protocols that can be (dynamically) combined. With the pay-as-you-go cost model of the Cloud, literally each user action and resource usage has a price tag attached to it. Thus, for the application providers, it is essential that the needs of their applications are provided in a cost-optimized manner. In this paper, we present the work in progress PolarDBMS, a flexible and dynamically adaptable system for managing data in the Cloud. PolarDBMS derives policies from application and service objectives. Based on these policies, it will automatically deploy the most efficient and cost-optimized set of modules and protocols and monitor their compliance. If necessary, the modules and/or their customization is changed dynamically at run-time. Several modules and protocols that have already been developed are presented. Additionally, we discuss the challenges that have to be met to fully implement PolarDBMS.**

I. Introduction

Clouds are very attractive environments for deploying different types of applications. They feature great advantages compared to traditional environments, mainly due to their highly available, scalable and elastic infrastructures based on a pay-as-you-go cost model.

Data management is an essential part of applications deployed in the Cloud. Applications have different requirements towards data management properties such as data availability, data consistency, or data preservation, to name just a few. These properties, in turn, might be realized with different guarantees like, for instance, different availability degrees or different consistency models. Such a guarantee can again have different implementations – e.g., [1], [2] for One-Copy-Serializability (1SR). Existing database management systems (DBMS) [3]–[6] usually provide only specific and especially fixed data management solutions for each of these properties, thus they are not able to dynamically change them at run-time.

Some existing DBMS (e.g., [3], [4]) provide weak consistency in order to increase scalability and availability (as a consequence of the CAP theorem [7]), whereas most traditional DBMSs provide strong data consistency and thus take limited scalability and availability into account [8]. This leads to overcustomized and earmarked DBMSs which focus on a very specific scenario without being able to adapt to diverging requirements of different scenarios. The client has no means of influencing the system's behaviour and is thus forced to either adapt his applications or build his own DBMS.

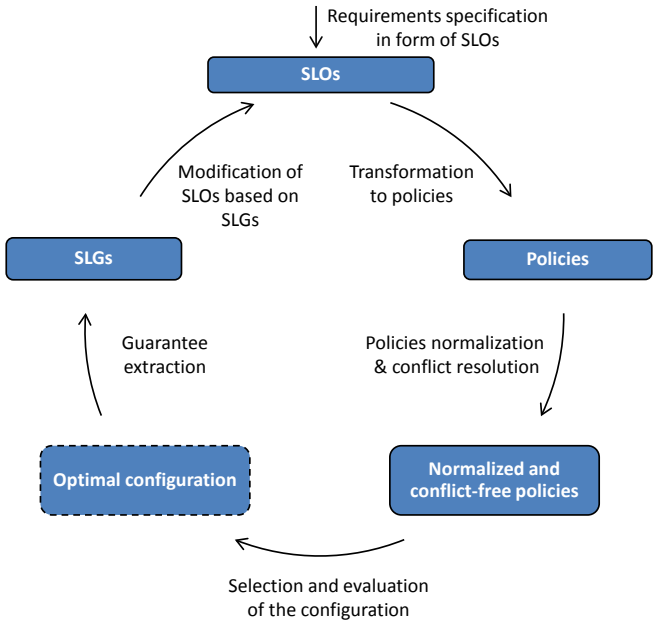

Fig. 1: Communication Cycle

In the Cloud, each action has a price tag associated with it. In general, the stronger the requirements a client demands, the higher are the costs that incur. By costs we mean both the overhead and the monetary costs generated for providing the guarantees. On one hand, cost can be reduced by selecting the best suited implementation which delivers the desired guarantees. On the other hand, the client should be able to reduce his cost by having the possibility to exactly specify what he wants (e.g., by requiring less than provided) – and also getting it. From the above we can conclude that there are three main aspects which lead to unnecessary costs. First, the inability of the client to specify exactly what it needs may lead to a situation, in which the client gets too much or too less – both generating unnecessary costs. In the first case, the client has to pay for the over-provisioning. In the second case, costs

are generated due to a degraded functionality and are expressed in form of penalty or business loss. Second, even in the case the client gets exactly what it needs, the desired guarantees may be delivered by different, possibly suboptimal, system implementations. Third, the Cloud provider being unable to deliver tailored services on a per client basis leads to the necessity of over-provisioning in order to satisfy all clients (leading to increased cost). Both aspects arise from the missing communication link between the involved parties, although the means to communicate are existing and well-established in form of Service-Level-Agreements (*SLAs*).

A. Communication Cycle

An optimal communication cycle between clients and Cloud providers in a business interaction is depicted in Figure 1. A client using a Cloud service specifies her requirements towards the service in form of an Service Level Objectives (SLOs). Table I contains a list of SLOs we use throughout this paper. For example, a client might define a lower bound of availability that has to be guaranteed by the service: $SLO : availability \geq 0.95$. The system providing the service transforms all SLOs of a client to internal objective representation called *policies*.

Since the service may be used by many different clients, their SLOs may lead to potentially conflicting policies. Thus, in a next step, the policy conflicts have to be resolved. At this stage the system has conflict-free policies, which form the basis of an evaluation process of different possible system configurations. Moreover, in a multi-tenant environment such as the Cloud, the evaluation process has to take SLOs (SLAs) of different clients into account. At the end of this process, the best possible configuration is chosen and transformed into Service Level Guarantees (*SLGs*). An SLG is a commitment of the provider on the fulfilment of an SLO: $SLG : availability \geq 0.99$. It might however be that the provided SLG does not fulfil the corresponding availability SLO ($SLO : availability \geq 0.95$), for example $SLG : availability \geq 0.9$. In that case, the client has the possibility to adapt its SLOs or decide to use another service provider. An adaptation of the SLOs leads to the restart of the entire cycle. Additionally, the cycle may be re-started later at any point in time and at any phase. For example, a change in the underlying infrastructure may lead to a reduced level of availability, which needs to be reflected in the corresponding SLGs. Again, the customer may decide to adapt its SLOs or change the service provider.

B. Modular DBMS

The communication cycle in Figure 1 defines a dynamic and generic negotiation process between clients and a provider, which gives the clients the possibility to precisely specify their objectives. For requirements regarding data management in the Cloud, the underlying DBMS has to be highly modular so as to allow the Cloud provider to host a large variety of different applications (with potentially heterogeneous client objectives) on top of the same infrastructure. The feasibility of modularizing data management functionality has been analysed in different works, like for example [9]. However, it is important to mention that the generic communication cycle in Figure 1 is architecture agnostic and applies to both modular and monolithic systems. The only difference is the problem of finding the optimal configuration (dashed box in Figure 1). In a monolithic system, the problem is restricted to optimally configuring it, while in the modular case, the problem space is extended to i.) finding the optimal combination of modules, ii.) configuring, and iii.) continuously monitoring and possibly replacing them.

In this paper, we present our work in progress system *PolarDBMS* (*Pol*icy-based and modul*ar DBMS* for the Cloud). PolarDBMS is based on a *modular* architecture. A module is a functional unit providing certain data management functionality (e.g., data replication, consensus protocols, atomic commitment, etc.).

The main objective of PolarDBMS is to automatically select (and, if necessary, dynamically adapt) the module combination and configuration that best provides the desired data management guarantees by incorporating the incurring costs, the client objectives and the capabilities of the underlying system.

This allows to individually combine the modules that best fit the clients' objectives and thus overcomes the drawbacks of monolithic DBMSs [10]. Obviously, modules are not independent from each other, and the disambiguation of client SLOs is part of the system design and deployment process. The separation of client objectives and policies from concrete mechanisms (modules) is in line with well established design principles in areas like operating system design [11], [12]. The use of a high level language allows clients to easily specify the requirements of their applications and shields details of the underlying DBMS. One goal is to dynamically remove or replace parts of PolarDBMS when objectives or system parameters change, thus to let the system incrementally evolve.

The contribution of this paper is twofold. First, we provide a complete analysis of an optimal negotiation cycle between clients and Cloud providers with the objective of providing cost-optimized services. The Cloud provider obtains a DBMS which is able to cope with different client requirements and thus increases profit by reducing the overhead (e.g., set-up, configuration, etc.). Additionally, we derive DBMS characteristics and behaviour out of SLAs. The knowledge

SLO	Description
availability	Defines the desired degree of availability.
maxBudget	Defines the maximum budget that can be spent for providing the desired guarantees.
upperBoundAccessLatency	Defines the desired upper bound of access latency.
minimizeTrxCost (inconsistency cost)	Based on the defined inconsistency cost, the policy leads to the choice of the consistency model which minimizes overall transaction cost.
maxTTL	Defines the maximum time-to-live for a data object.

TABLE I: Summary of SLOs

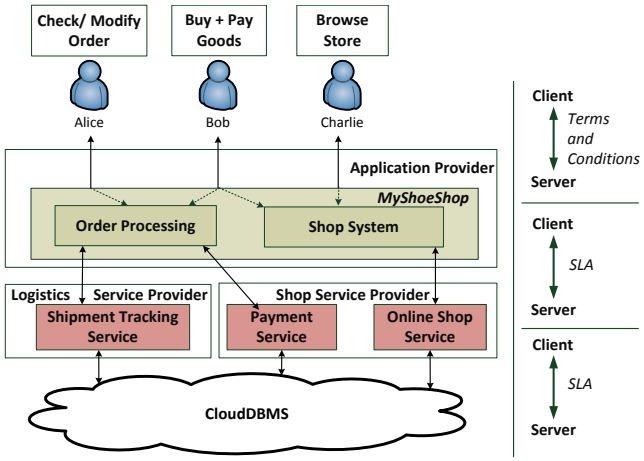

Fig. 2: Scenario

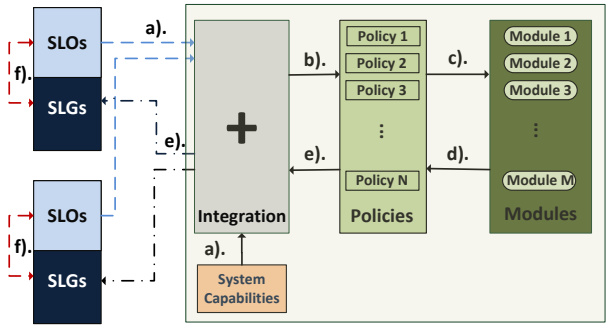

Fig. 3: Communication Flow

contained in SLAs combined with a modular DBMS, such as PolarDBMS, enables Cloud providers to offer tailored services to its clients. As a consequence, more clients can be attracted to move into the Cloud. Second, we provide an analysis of the most prominent data management properties for applications deployed in the Cloud, namely availability, consistency and archiving, together with a set of concrete modules we have already developed and evaluated.

The paper is organized as follows: Section II motivates the PolarDBMS idea. In Section III, we discuss and analyze in detail the different phases of the communication cycle, from the specification of the client objectives to the choice of the optimal PolarDBMS composition and configuration. Section IV describes concrete contributions achieved so far in the PolarDBMS context. Section V summarizes related work. Section VI lists main challenges towards a full PolarDBMS implementation and concludes.

II. Motivation

In what follows we will illustrate the policy cycle depicted in Figure 1 based on interactions between end customers and service providers in a concrete scenario. As depicted in Figure 2, various end customers use the `MyShoeShop` to search for and eventually buy shoes. While Charlie only browses the shop without intending to order any shoes, Bob does both. Alice, on the other hand, just changes the delivery address of a previous order. Inside the `shop system`, several subsystems are responsible for the different actions. While the `shop system` is responsible for displaying the product catalogue, the `order processing` system handles orders and shipping. The application provider who runs the shop does not host the infrastructure for her application. In fact, the latter relies on two different service providers: a provider specializing in logistics offers a `shipment tracking service` and another service provider hosts a `payment service`. Both services are used by the order processing system of the application provider. None of the service providers directly hosts a database. Instead, they both rely on (possibly different) cloud DBMSs (e.g., NoSQL and NewSQL systems [13]). Obviously, an action that is launched by an end user is, automatically and transparently to the client, broken down to several different organizations and systems. Since each of the participants offers and/or uses services, bilateral agreements exist which specify various aspects of the services. For example, the end users agree to the terms and conditions (ToC) of the online shop provider as soon as they place an order. The application provider and the service providers can rely on SLAs, consisting of SLOs and SLGs, to specify the details of the bilateral relationship. This applies also for the interaction between the service providers and the Cloud DBMS provider(s).

Furthermore, the aforementioned ToCs and SLAs are defined independently of each other and may thus be contradictory. This results in a complex set of dependencies leading to a non-transparent system w.r.t. the guarantees provided towards the end user. Furthermore, as today's businesses are rapidly evolving, a dynamic adaptation to changing requirements is necessary. The `MyShoeShop` provider in the scenario may, for instance, request different degrees of availability. She may require either an increased availability due to changed legal regulations or an expected peak load – or a reduced availability in order to save money, since higher availability implies higher cost. Today's Cloud services and applications come with a predefined and fixed set of characteristics. This leads to rigid systems that are difficult to adapt to changing business and technical needs. This urgently demands a dynamically adaptable system which, as a consequence, allows for a flexible interaction between SLOs and SLGs as depicted in the cycle in Figure 1. Furthermore, a means for mapping objectives and constraints to a system configuration is needed together with a mechanism for monitoring the satisfiability of both objectives and constraints.

III. POLAR DBMS

As described in the previous Section, today's information systems in the Cloud are intrinsically complex, with many different *actors* involved. Actors can be end-users and providers, but also specific systems or components. During an interaction, actors can play different roles: a *client* consumes services, whereas a *server* provides them. Hence, on the one hand it is of utmost importance that each client is able to specify her individual requirements towards the server. On the other hand, the server has specific capabilities and may constrain the way its resources are used. Hence, a formal description for both client requirements and server capabilities is needed. These formal descriptions are expressed via SLOs, which define the clients' requirements, and SLGs which represent the

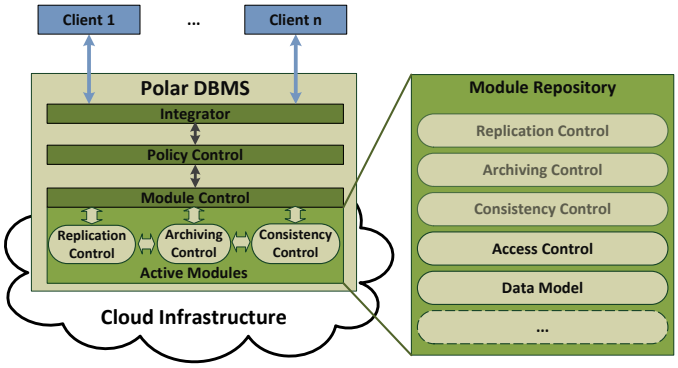

Fig. 4: PolarDBMS Architecture

corresponding guarantees given by the system. As depicted in Figure 3, different clients specify their SLOs and submit them to PolarDBMS. As part of phase a)., the submitted SLOs are integrated with the existing system capabilities. In a concrete example that runs on PolarDBMS, a client may define her desired data availability with the following SLO: $availability \geq 99.995\%$. Additionally, the client may require the data to be stored only in the EU. Let us however assume that the provider runs data centers only within the US. This is a sort of a business constraint, which, along with possible legal or other organizational constraints, defines the system capabilities. In the aforementioned example the integrator is, already at phase a.) (Figure 3), able to decide that the desired SLO cannot be satisfied. Assuming that the system capabilities do not prevent the satisfiability of the SLOs, the next step is to express the combined SLOs and system capabilities in form of *policies* – phase b.) in Figure 3. For example, the availability is mapped to a policy, which consists of various parameters, including the number of machines data has to be replicated to: $\#replicas$ (phase b)). High availability, thus high $\#replicas$ leads to higher costs. The relationship between $\#replicas$ and cost is reflected in another policy. This needs to be taken into account in case the client wants to limit the cost per billing period (SLO: $maxBudget \leq 1000\$$). This implies that the system already has information about its (estimated) runtime characteristics (such as average cost per replica). Initially, the system starts with default values derived from a cost model.

As already mentioned, the availability is also impacted by $\#replicas$. This parameter can be estimated without involving the capabilities of the concrete functional building blocks of PolarDBMS, which are represented by modules. However, the final decision on the fulfilment of the desired availability can only be made in collaboration with the underlying modules. This means that concrete objectives are provided to the modules as part of phase c). This step initiates the so called *negotiation process* between policies and modules. This negotiation process is actually an optimization problem with the goal of finding the optimal combination of modules together with their configurations (see dashed box in Figure 1). There may be multiple modules implementing different replication approaches leading to different availability levels. As part of the module selection, the best replication approach is chosen. Additionally, the replication approach may also be influenced by policies related to other data management properties such as the data consistency (CAP theorem [7]). This implies the necessity of finding the optimal module set.

The first step in the negotiation process (phase c.)) is the transformation of policies into suitable *module objectives*. These express functional and non-functional requirements towards the modules. In our example, the policies related to availability and cost are expressed by means of module objectives ($\#replicas$, $maxBudget$). In the next step, the main challenge is to find the optimal set of modules and their configuration, which deliver the best *module guarantees* w.r.t. the objectives. In our example, the modules will provide the expected budget ($expBudget$) they need for fulfilling the objectives. The guarantees are then transformed back to policies as part of phase d.). The validity of existing policies must not be violated by the new guarantees ($expBudget < maxBudget$). At this point, the negotiation process is finished and the policies are transformed into client SLGs (phase e.)) ($expAvailability$). The contrast between the originally defined SLOs and the offered SLGs (phase f.)) may lead to the client choosing another provider or to the adaptation of the SLOs, if e.g., $expAvailability < availability$. In the second case, the client may modify its desired availability requirement. This leads to the adjustment of the entire communication flow.

The different phases of the the communication flow depicted in Figure 3, are represented by the following architectural entities (Figure 4). i.) The *integrator* forms the interface between clients and policies. ii.) The *policy control* is responsible for the policy management. iii.) The selection and the configuration of the modules is carried out by the *module control*. Additionally, it is responsible for the management of the different available modules, namely active and inactive ones.

IV. Policy-based Data Management

In the following sections, we provide a more detailed discussion of selected data management properties, namely availability, consistency and preservation. This list is by far not complete; other essential properties used are briefly discussed in Section VI.

A. Data Availability

Data availability describes the requirement to ensure that data continues to be available even in case of server crashes or disastrous events. In many applications, a high degree of availability is necessary to guarantee that data is never lost (e.g., banking or insurance applications) and to be able to operate even if complete parts (data centers) of the entire infrastructure fail. Additionally, data availability is also used to describe data access performance, i.e., that data should be available for processing with a low or no latency at all ("always-on" experience [4]). According to Amazon, a slowdown of just one second per page request could cost them about $1.6 billion each year [14]. Similar results were reported by Google, which say that if their search results are slowed-down by just four tenths of a second they could loose at about 8 million searches per day [14]. Cloud infrastructures are usually built with the objective of providing a high degree of availability. The most common solution to high availability is to host several copies (*replicas*) of data at different data centers, to avoid that data gets lost in case of crashes when replicated only within one data center [15], [16].

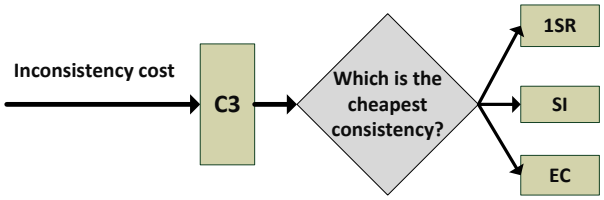

Fig. 5: C3

The best level of availability (and thus the optimal number of replicas) depends on the context of a concrete application and should be individually specified by means of client objectives, rather than pre-set by the Cloud provider. The same also applies to access latency to data, which also depends on the number (and placement) of replicas.

Based on the desired availability degree PolarDBMS will identify the required number of replicas and the best module able to optimally manage the replicas. As already described in Section III, the client objectives and the system capabilities need to be integrated. For example, the client may have requested an availability level of 99.999% which implies a maximum downtime of approx. 5 minutes/year, but the system is capable of providing only an availability level of 99.9% (maximum downtime of approx. 9 hours/year). Then the client objectives and the system capabilities have to be integrated and disambiguated, leading to an availability specification that can actually be enforced.

In addition to the availability requirement, a client may also specify an upper bound for the access latency to data ($upperBoundAccessLatency$). Assume that the client also expresses the maximum monetary budget for a specific period ($maxBudget$) which specifies that the requested availability level has to be provided within the budget constraints. In summary, PolarDBMS needs to take all client objectives and the system capabilities into account in order to select, customize and deploy the best suited modules and configure them optimally.

Depending on the consistency requirements of the data and the workload type (read-only, mostly read-only, or update), replication may also increase system performance by using the additional processing capabilities (e.g., parallelism, location-aware request processing, etc.). Replication can be implemented statically (using pre-defined numbers and locations of replicas) or dynamically where replicas are created, destroyed and/or re-located at runtime, based on an optimization model [17]. Current approaches however only take system parameters into account, such as distance between replicas, number of replicas, etc. and do not involve the client (and its individual requirements) into the decision process. PolarDBMS dynamically identifies the optimal number of replicas and their location to satisfy the availability and access latency requirements of the client and minimizes, at the same time, the costs that incur for data management.

B. Data Consistency

In every distributed system, there is a trade-off between data consistency, availability, and partition tolerance. According to the CAP-theorem, it is not possible to provide all three properties at the same time. As partition tolerance is a must in a networked system, this means that systems can additionally either guarantee consistency or provide a high degree of availability – but not both at the same time. Cloud infrastructures are built with virtually infinite capacity and scalability in mind. For this, new cloud DBMS solutions have been developed which only provide weak consistency [18], [19]. However, while this nicely serves several novel types of applications, it has turned out that it is very challenging to develop traditional applications on top of weakly consistent DBMSs [20]. The complexity arises from the necessity of compensating the missing consistency guarantees at application level.

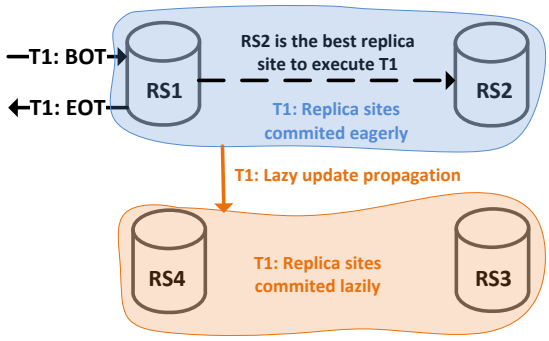

Fig. 6: SO-1SR

In a Cloud environment with its pay-as-you-go cost model where each resource and each action literally come with a price tag, the consideration of costs is an essential complement to the trade-off between availability and consistency. While strong consistency generates high infrastructure costs, weak consistency in turn may generate high penalty (inconsistency) costs for compensating access to stale data (e.g., book oversells) [21]–[23].

Data consistency in replicated system can be characterized by two properties: concurrency control (transaction isolation) and replica consistency. The former defines the level of isolation between concurrent transactions. The latter controls the allowed divergence of replicas in comparison to the most recent data (i.e., the site where the most recent update operation has been executed).

The stronger both properties are, the stronger is the consistency, the lower the scalability and the higher the costs that incur [8], [21]–[23]. One-Copy-Serializability (1SR) is the desired consistency level for replicated data. It provides a serializable execution of concurrent transactions and a one-copy view on replicated data. Existing approaches are committed to either support strong or weak consistency [2], [24]–[27], but not both. Moreover, they do not take dynamic adaptation and optimization into account, based on a specification of the desired system behaviour by means of objectives – especially not on the basis of the costs that incur.

In what follows we will describe data consistency modules we have developed and evaluated in the context of PolarDBMS.

1) C^3: Cost-Based Consistency Control: In our recent work [22], we have developed C^3, a modular consistency framework which is able to adjust data consistency (switch between different consistency models) at runtime with the goal

of minimizing overall costs. C^3 provides different consistency models (1SR, One-Copy Snapshot Isolation, and Eventual Consistency, see Figure 5) and is able to choose the best one based on the consistency and inconsistency costs. The operational costs are generated by the Cloud resources which need to be used for achieving a certain consistency level. The inconsistency costs are application specific costs and reflect the additional work which needs to be done in order to compensate the effects of inconsistent data (e.g., for compensating oversold books or tickets). The client can specify a concrete inconsistency cost value in the $minimizeTrxCost$ SLO and let the framework decide on the best consistency model or explicitly select a dedicated consistency model.

2) SO-1SR: Self-Optimizing 1SR Protocol: In addition to the cost-based selection of consistency model and protocol in C^3, also the behaviour of concrete protocols can be dynamically influenced by means of client SLOs and system capabilities (Figure 1). If the client has specified that she desires 1SR (e.g., by specifying a high inconsistency cost in the $minimizeTrxCost$ SLO), then the concrete 1SR protocol should take other client objectives (e.g., $upperBoundAccessLatency$) and system capabilities into account for further optimization. It is clear that with stronger consistency it is increasingly difficult to guarantee the desired response time of transactions. Hence, a from a list of modules providing 1SR guarantees the optimal one should be chosen. SO-1SR (self-optimizing 1SR depicted in Figure 6) [28] is our approach to provide efficient and cost-optimized 1SR guarantees. Essentially, SO-1SR is able to optimize different phases of the transaction life-cycle, namely transaction processing and commit separately yet not independently. It uses sophisticated load balancing techniques for optimizing transaction processing by taking the optimization overhead (network) and system capability into account (e.g., replica load and capacity). Additionally, SO-1SR reduces the (eager) commit overhead based on a dynamic commit strategy by taking the requirements of the transaction processing phase and system capabilities into account.

Both modules (C^3 and SO-1SR) decide on the best strategy based on policies derived out of SLOs. The evaluations conducted [22], [28] prove the feasibility of both approaches and show that they are able to fulfil client objectives and thus outperform traditional approaches.

C. Data Preservation

Clients often need to preserve data for extended periods of time. This necessity arises from various factors, like for instance legal regulations, data analysis and mining, data recovery, etc. This requirement is even more important when data is no longer managed under full control of (and in the systems managed by) the clients but in a Cloud environment. An archiving system in the Cloud should thus provide seamless, fully transparent, and policy-based preservation and access to archived data to the client.

During the lifetime of an object, an archiving system needs policy support in order to decide how the system shall handle the object (e.g., whether or not to materialize a version, when to delete it, etc.). In the archive, each data object may exist in different versions. The most straightforward approach to data archiving in the Cloud is to keep each and every version, potentially in several replicas (for reasons of availability). However, this generates extremely high data management costs, especially with data that are subject to high update rates. Most importantly, this is in sharp contrast to the overall objective of optimizing the overall costs that incur for data management in the Cloud.

The PolarDBMS approach to archiving does *not* require all versions to be materialized. Rather, it allows gaps in the version history. The idea is to keep log information for the versions that are not materialized (which usually consumes much less space than the actual object and is thus cheaper to achieve) so that they can be dynamically re-produced, if requested. The decision whether to keep the entire object in the archive or only log information is done dynamically, based on a cost model that takes the specified SLOs (in particular the available budget and the overall storage capacity) into account. At query time, PolarDBMS decides, based on its knowledge of the location of logs and data object versions, how to efficiently restore the desired data object(s).

Let us assume that a client provides a per-data object budget for archiving, $maxBudget = 0.05\$$. Further, let us assume that the budget SLO has been translated to an archiving duration of 8 years ($maxTTL = 8y$). However, according to legal regulations, accounting data must be archived for at least 10 years. In this case, these contradicting objectives must be resolved by creating a suitable policy, based on weights. In the concrete example, the legal regulation has a higher weight and must therefore override the budget objective. The new duration policy still needs to be supported with the available budget.

It is important to mention that existing policies defined in the context of other data management SLOs, e.g., data security, may also influence the archiving behavior (possibly with other weights). If, for example, the lifetime of a data object has reached its end or if the deletion of the archived object is explicitly requested, the data object needs to be removed from the system by also taking security (access authorization) policies into account.

Archiving is only one aspect of data preservation. Its main focus is the long-term storage of data and its efficient retrieval. However, especially in the context of scientific applications, not only the data, but also the processes (operations) that created the data are very important. This aspect is part of ongoing research in the area of data provenance and briefly described in Section VI.

V. Related Work

During the last years, many approaches and protocols for optimizing data management in distributed environments, especially in the Cloud, have been proposed. This includes approaches in the area of data consistency with the goal of optimizing existing data consistency models [2], or proposing new data consistency models [25]–[27], [29]–[31]. In both cases, scalability has been the driving force. However, approaches from either direction are limited to certain use cases and are not able to adapt to ever changing client requirements. Dealing with these dynamic requirements is one of the main objectives of PolarDBMS.

Existing traditional DBMSs have been built to support many applications which feature different characteristics and requirements — this is also known as "one size fits all" approach [32]. According to [33], traditional DBMS architectures do not work well for data warehouse applications (OLAP) as they were initially optimized for OLTP workloads. [32] lists possible reasons which led to the development of "one size fits all" DBMSs which are no longer applicable to the database market. Rather, it is foreseen that specialized DBMSs will evolve as a result of new application types and their specific requirements. PolarDBMS actually goes in the same direction and we do also argue that different application types, especially in the Cloud context, have widely different characteristics and requirements towards the underlying DBMSs. However, PolarDBMS follows the approach of having a thin DBMS framework, which will manage the concrete functionality implemented as modules instead of having many different and possibly specialized DBMSs. Additionally, PolarDBMS treats clients as first class citizens by giving them the possibility to directly influence the DBMS composition through a fully transparent communication cycle as depicted in Figure 3. Each different composition corresponds to what [32] calls a "specialized engine". However, our approach has the advantage of creating such "specialized engines" inside the same highly optimized and common system and code structure, without the necessity of reinventing the wheel over and over again.

OctopusDB [34] is a DBMS that allows to choose the suitable data model (row stores, column stores, etc.) depending on the application characteristics. OctopusDB fits very well in the overall PolarDBMS approach, however the choice of the best suited data model is only a subset of the objectives that PolarDBMS dynamically considers. [35] introduces Cloudy, a modular DBMS for the Cloud which also allows for dynamic optimization and customization. However, in contrast to PolarDBMS, it does not provide a means of specifying client requirements based on SLAs.

A modular DBMSs with the capability of exchanging modules at runtime has been proposed in [36]. However, their approach does not analyse on how clients can easily specify their requirements. Additionally, [36] targets the modularization of one specific DBMS type, namely relational databases. Similar to [36], PolarDBMS also follows the goal of being adaptable at runtime; however PolarDBMS provides a modular system that includes a wide variety of different data management properties and is thus not being limited to one DBMS type.

The necessity and the advantages of modular-based database systems has been thoroughly analysed in [9]. PolarDBMS also has a modular architecture. Our approach however does take a holistic view on the communication cycle incorporating the client, her requirements and the system capabilities into the process of finding the optimal module composition and configurations.

Recently, [37] has proposed an interesting DBMS development principle based on co-design of the DBMS and operating systems. The approach opens interesting opportunities for providing a highly optimized system by building the DBMS to fully exploit the underlying operating system and, vice versa, by building the operating system based on the DBMS requirements. Similarly, [38] has proposed a solution in which the entire DBMS functionality is fully implemented in the hardware.

In [39] the authors present a transactional "database-as-a-service" called Relational Cloud. The goal of the Relational Cloud is to reduce the configuration effort for users and operators by overcoming the following challenges: efficient multi-tenancy, elastic scalability and database privacy. The approach in [39] is able to adapt to workload changes by monitoring the query and data access patterns. Our PolarDBMS goes one step further by taking a holistic view on different data management aspects and not focusing only on the workload characteristics.

VI. Conclusion and Outlook

In this paper we have introduced the concepts of our work in progress system PolarDBMS, a novel DBMS following a modular architecture based on customizable modules that are dynamically adapted at run-time based on client objectives and system capabilities. Since one of the main optimization criteria for the selection, configuration, and combination of modules are the costs that incur at run-time due to the use of underlying resources, PolarDBMS is highly suitable for Cloud environments. Hence, PolarDBMS is expected to provide many advantages compared to existing DBMSs by allowing the client to specify her requirements in an easy and understandable way and by choosing, continuously monitoring and automatically adapting the best available modules for satisfying client requirements in a fully transparent way. The implementation of PolarDBMS and its modules is an ongoing effort, with main focus on data availability (dynamic replication), data consistency (concurrency control and replica consistency), and data preservation (archiving). However, there are many challenges that have to be met for reaching the goal of providing an adaptable, policy-based DBMS. The main challenges we have identified so far are summarized below.

Policy related challenges: One of the main challenges is to extract and derive policies out of existing information such as SLAs, ToCs, etc.) – and, vice versa, to derive SLGs out of a set of policies in an automatic or at least a semi-automatic way. The goals of this transformation process are twofold: first, it must lead to well-defined policy and SLO/SLG languages. Second, it must be able to disambiguate contradictory and incomplete policies. Moreover, the users need to be supported by suitable user interfaces that help in the process of creating and handling SLOs and SLGs. As already described in Section III, the definition of the integrator component is necessary, which will generate integrated and unambiguous policies. A policy mechanism should also answer questions on the behaviour of the system if the client has not provided all necessary SLOs, thus leading to incomplete policies.

System design and architecture: The modular architecture of PolarDBMS implies the necessity of a clear API for the cooperation between the PolarDBMS framework and its modules. A module repository is needed for the proper module management. It should support a formal description of the behaviour of each of the modules. Even if modules providing similar behaviour may coexist, PolarDBMS should be able to choose the "right" one for satisfying the client objectives.

Existing models and formal languages, like the Web Service Modelling Ontology and Web Service Modelling Lan-

guage [40], can be used and if necessary extended to formally describe module capabilities. With regards to the module selection a.k.a resource matching, different existing approaches [41], [42] can be used in the context of PolarDBMS.

Operation and Maintenance: PolarDBMS aims at supporting the dynamic reconfiguration at runtime without or with a low system interruption. This imposes new challenges w.r.t. to the management and operation of modules.

Analysis and integration of additional data management properties: As part of this work, we have proposed solutions for data availability, consistency, and preservation. However, data management has also other very important properties, like for example data security, data models, data integration, etc. The analysis and the integration of these additional properties increases the problem space and introduces additional dependencies that need to be considered. Additionally, inside the same data management property, different aspects need to be analyzed. For example, data preservation, in addition to archiving, includes also data provenance [43] as a concrete functionality. The workload type of applications (OLTP, OLAP or mixed) has a strong impact on the choice of the best data model (row-stores, column-stores, etc.). Hence, PolarDBMS should be able to dynamically adapt its data model by taking workload type and other characteristics of applications into account.

REFERENCES

[1] P. A. Bernstein and N. Goodman, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 596–615, Dec. 1984.

[2] B. Kemme and G. Alonso, "Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication," in *Proc. VLDB*, 2000, pp. 134–143.

[3] "Amazon S3," http://aws.amazon.com/s3/, accessed: 06-2013.

[4] G. DeCandia *et al.*, "Dynamo: Amazon's highly available Key-Value Store," in *Proc. ACM SIGOPS*, 2007, pp. 205–220.

[5] "MySQL," http://www.mysql.com/, accessed: 11-2013.

[6] "PostgreSQL," http://www.postgresql.org/, accessed: 11-2013.

[7] E. A. Brewer, "Towards Robust Distributed Systems," in *Proc. PODC*, 2000.

[8] J. Gray *et al.*, "The dangers of replication and a solution," in *Proc. SIGMOD*, 1996.

[9] K. R. Dittrich and A. Geppert, Eds., *Component database systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[10] A. Geppert and K. R. Dittrich, "Component database systems," K. R. Dittrich and A. Geppert, Eds., 2001.

[11] M. S. Miller and E. K. Drexler, "Markets and computation: Agoric open systems," in *The Ecology of Computation*, B. A. Huberman, Ed. North-Holland, Amsterdam, 1988.

[12] W. Wulf *et al.*, "HYDRA: the Kernel of a Multiprocessor Operating System," *Commun. ACM*, 1974.

[13] "NewSQL," http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext, accessed: 11-2013.

[14] "How one second could cost Amazon $1.6 billion in sales," http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, accessed: 06-2013.

[15] "Amazon EC2 Goes Down, Taking With It Reddit, Foursquare And Quora," http://techcrunch.com/2011/ 04/21/amazon-ec2-goes-down-taking-with-it-reddit- foursquare-and-quora/, Accessed: 06-2013.

[16] "Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region," http://aws.amazon.com/de/message/65648/, accessed: 06-2013.

[17] N. Bonvin, T. G. Papaioannou, and K. Aberer, "A Self-organized, Fault-tolerant and Scalable Replication Scheme for Cloud Storage," in *Proc. SoCC*, 2010.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Proc. OSDI '06*.

[19] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *In Proc. VLDB*, 2008.

[20] D. Agrawal, A. Abbadi, H. Mahmoud, F. Nawab, and K. Salem, "Managing Geo-replicated Data in Multi-datacenters," in *Databases in Networked Information Systems*. Springer, 2013, pp. 23–43.

[21] T. Kraska *et al.*, "Consistency rationing in the cloud: pay only when it matters," in *Proc. VLDB*, 2009, pp. 253–264.

[22] I. Fetai and H. Schuldt, "Cost-based data consistency in a data-as-a-service cloud environment," in *Proc. CLOUD*, 2012, pp. 526–533.

[23] ——, "Cost-based data consistency in a data-as-a-service cloud environment," University of Basel, Switzerland, CS Technical Report CS-2012-001, Feb. 2012, available at http://informatik.unibas.ch/research/publications_tec_report.html.

[24] W. Vogels, "Eventually consistent," *Communications of the ACM*, pp. 40–44, 2009.

[25] H. Jung, H. Han, A. Fekete, and U. Röhm, "Serializable snapshot isolation for replicated databases in high-update scenarios," *Proc. VLDB*, 2011.

[26] K. Daudjee and K. Salem, "Lazy Database Replication with Snapshot Isolation," in *Proc. VLDB*, 2006, pp. 715–726.

[27] S. Elnikety, W. Zwaenepoel, and F. Pedone, "Database replication using generalized snapshot isolation," in *Proc. SRDS*, 2005, pp. 73–84.

[28] I. Fetai and H. Schuldt, "SO-1SR: towards a self-optimizing one-copy serializability protocol for data management in the cloud," in *Proc. CloudDB*, San Francisco, California, USA, 2013.

[29] M. A. Bornea *et al.*, "One-copy serializability with snapshot isolation under the hood," in *ICDE*, 2011.

[30] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable Isolation for Snapshot Databases," in *Proc. SIGMOD*, 2008.

[31] P. A. Bernstein and S. Das, "Rethinking eventual consistency," in *SIGMOD Conference*, 2013, pp. 923–928.

[32] M. Stonebraker, "One Size Fits All: An Idea Whose Time has Come and Gone," in *Proc. ICDE*, 2005, pp. 2–11.

[33] C. D. French, ""One size fits all" Database Architectures do not work for DSS," in *Proc. SIGMOD*, 1995, pp. 449–450.

[34] J. Dittrich and A. Jindal, "Towards a one size fits all database architecture," in *CIDR*, 2011, pp. 195–198.

[35] D. Kossmann *et al.*, "Cloudy: a modular cloud storage system," *Proc. VLDB Endow.*, pp. 1533–1536, 2010.

[36] F. Irmert, M. Daum, and K. Meyer-Wegener, "A new Approach to Modular Database Systems," in *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management*, ser. SETMDM '08, 2008.

[37] J. Giceva *et al.*, "Cod: Database / operating system co-design," in *CIDR*, 2013.

[38] R. Johnson and I. Pandis, "The bionic DBMS is coming, but what will it look like?" in *CIDR*, 2013.

[39] C. Curino *et al.*, "Relational cloud: a database service for the cloud," in *CIDR*, 2011.

[40] D. Roman *et al.*, "Web service modeling ontology," *Appl. Ontol.*, 2005.

[41] H. Tangmunarunkit, S. Decker, and C. Kesselman, "Ontology-based resource matching in the grid - the grid meets the semantic web," in *Proc. ISWC, Sanibel-Captiva Islands*, 2003.

[42] S. Grimm, "Intersection-based matchmaking for semantic web service discovery," in *ICIW*, 2007.

[43] P. Buneman, S. Khanna, and W.-C. Tan, "Data provenance: Some basic issues," in *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*. Springer, 2000, pp. 87–93.